Penser en C++

Volume 1
par Bruce Eckel

Date de publication : 27/08/2008

Dernière mise à jour : 27/08/2008

Ce cours est une traduction du livre Thinking in C++de Bruce Eckel, dont l'original est disponible ici : [lien].

0 - Préface
0.1 - Quoi de neuf dans cette seconde édition ?
0.1.1 - Qu'y a-t-il dans le Volume 2 de ce livre ?
0.1.2 - Comment récupérer le Volume 2 ?
0.2 - Prérequis
0.3 - Apprendre le C++
0.4 - Buts
0.5 - Chapitres
0.6 - Exercices
0.6.1 - Solutions des exercices
0.7 - Le Code Source
0.8 - Normes du langage
0.8.1 - Support du langage
0.9 - Le CD ROM du livre
0.10 - CD ROMs, conférences, et consultations
0.11 - Erreurs
0.12 - A propos de la couverture
0.13 - Conception et production du livre
0.14 - Remerciements
1 - Introduction sur les Objets
1.1 - Les bienfaits de l'abstraction
1.2 - Un objet dispose d'une interface
1.3 - L'implémentation cachée
1.4 - Réutilisation de l'implémentation
1.5 - Héritage : réutilisation de l'interface
1.5.1 - Les relations est-un vs. est-comme-un
1.6 - Polymorphisme : des objets interchangeables
1.7 - Créer et détruire les objets
1.8 - Traitement des exceptions : gérer les erreurs
1.9 - Analyse et conception
1.9.1 - Phase 0 : Faire un plan
1.9.2 - Phase 1 : Que construit-on ?
1.9.3 - Phase 2 : Comment allons-nous le construire ?
1.9.4 - Phase 3 : Construire le coeur du système
1.9.5 - Phase 4 : Itérer sur les cas d'utilisation
1.9.6 - Phase 5 : Evolution
1.9.7 - Les plans sont payants
1.10 - Extreme programming
1.10.1 - Commencer par écrire les tests
1.10.2 - Programmation en binôme
1.11 - Les raisons du succès du C++
1.11.1 - Un meilleur C
1.11.2 - Vous êtes déjà sur la courbe d'apprentissage.
1.11.3 - Efficacité
1.11.4 - Les systèmes sont plus faciles à exprimer et à comprendre
1.11.5 - Puissance maximale grâce aux bibliothèques
1.11.6 - Réutilisation des sources avec les templates
1.11.7 - Traitement des erreurs
1.11.8 - Mise en oeuvre de gros projets
1.12 - Stratégies de transition
1.12.1 - Les grandes lignes
1.12.2 - Ecueils de la gestion
1.13 - Résumé
2 - Construire et utiliser les objets

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

2.1 - Le processus de traduction du langage
2.1.1 - Les interpréteurs
2.1.2 - Les compilateurs
2.1.3 - Le processus de compilation
2.2 - Outils de compilation séparée
2.2.1 - Déclarations vs. définitions
2.2.2 - Edition des liens
2.2.3 - Utilisation des bibliothèques
2.3 - Votre premier programme C++
2.3.1 - Utilisation de la classe iostream
2.3.2 - Espaces de noms
2.3.3 - Principes fondamentaux de structure de programme
2.3.4 - "Bonjour tout le monde !"
2.3.5 - Lancer le compilateur
2.4 - Plus sur les flux d'entrée-sortie
2.4.1 - Concaténation de tableaux de caractères
2.4.2 - Lire les entrées
2.4.3 - Appeler d'autres programmes
2.5 - Introduction aux chaînes de caractères
2.6 - Lire et écrire des fichiers
2.7 - Intoduction à la classe vector
2.8 - Résumé
2.9 - Exercices
3 - Le C de C++
3.1 - Création de fonctions
3.1.1 - Valeurs de retour des fonctions
3.1.2 - Utilisation de la bibliothèque de fonctions du C
3.1.3 - Créer vos propres bibliothèques avec le bibliothécaire
3.2 - Contrôle de l'exécution
3.2.1 - Vrai et faux
3.2.2 - if-else
3.2.3 - while
3.2.4 - do-while
3.2.5 - for
3.2.6 - Les mots clé break et continue
3.2.7 - switch
3.2.8 - Du bon et du mauvais usage du goto
3.2.9 - Récursion
3.3 - Introduction aux operateurs
3.3.1 - Priorité
3.3.2 - Auto incrémentation et décrémentation
3.4 - Introduction aux types de données
3.4.1 - Types intégrés de base
3.4.2 - bool, true, & false
3.4.3 - Spécificateurs
3.4.4 - Introduction aux pointeurs
3.4.5 - Modification d'objets extérieurs
3.4.6 - Introduction aux références en C++
3.4.7 - Pointeurs et références comme modificateurs
3.5 - Portée des variables
3.6 - Définir des variables "à la volée"
3.6 - Définir l'allocation mémoire
3.6.1 - Variables globales
3.6.2 - Variables locales
3.6.3 - static

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

3.6.4 - extern
3.6.5 - Constantes
3.6.6 - volatile
3.7 - Operateurs et leurs usages
3.7.1 - L'affectation
3.7.2 - Opérateurs mathématiques
3.7.3 - Opérateurs relationnels
3.7.4 - Opérateurs logiques
3.7.5 - Opérateurs bit à bit
3.7.6 - Opérateurs de décalage
3.7.7 - Opérateurs unaires
3.7.8 - L'opérateur ternaire
3.7.9 - L'opérateur virgule
3.7.10 - Piège classique quand on utilise les opérateurs
3.7.11 - Opérateurs de transtypage
3.7.12 - Transtypage C++ explicite
3.7.13 - sizeof – Un opérateur par lui même
3.7.14 - Le mot clef asm
3.7.15 - Opérateurs explicites
3.8 - Création de type composite
3.8.1 - Alias de noms avec typedef
3.8.2 - Combiner des variables avec des struct
3.8.3 - Eclaircir les programmes avec des enum
3.8.4 - Economiser de la mémoire avec union
3.8.5 - Tableaux
3.9 - Conseils de déboguage
3.9.1 - Drapeaux de déboguage
3.9.2 - Transformer des variables et des expressions en chaînes de caractère
3.9.3 - la macro C assert( )
3.10 - Adresses de fonctions
3.10.1 - Définir un pointeur de fonction
3.10.2 - Déclarations complexes & définitions
3.10.3 - Utiliser un pointeur de fonction
3.10.4 - Tableau de pointeurs de fonction
3.11 - Make: gestion de la compilation séparée
3.11.1 - Les actions du Make
3.11.2 - Les makefiles de ce livre
3.11.3 - Un exemple de makefile
3.12 - Résumé
3.13 - Exercices
4 - Abstraction des données
4.1 - Une petite bibliothèque dans le style C
4.1.1 - Allocation dynamique de mémoire
4.1.2 - Mauvaises conjectures
4.2 - Qu'est-ce qui ne va pas?
4.3 - L'objet de base
4.4 - Qu'est-ce qu'un objet?
4.5 - Typage de données abstraites
4.6 - Détails sur les objest
4.7 - L'étiquette d'un fichier d'en-tête
4.7.1 - L'importance des fichiers d'en-tête
4.7.2 - Le problème des déclarations multiples
4.7.3 - Les directives #define, #ifdef et #endif du préprocesseur
4.7.4 - Un standard pour les fichiers d'en-tête
4.7.5 - Les espaces de nommage dans les fichiers d'en-tête

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

4.7.6 - Utiliser des fichiers d'en-tête dans des projets
4.8 - Structures imbriquées
4.8.1 - Résolution de portée globale
4.9 - Résumé
4.10 - Exercices
5 - Cacher l'implémentation
5.1 - Fixer des limites
5.2 - Le contrôle d'accès en C++
5.2.1 - protected
5.3 - L'amitié
5.3.1 - Amis emboîtés
5.3.2 - Est-ce pur ?
5.4 - Organisation physique d'un objet
5.5 - La classe
5.5.1 - Modifier Stash pour employer le contrôle d'accès
5.5.2 - Modifier Stack pour employer le contrôle d'accès
5.6 - Manipuler les classes
5.6.1 - Dissimuler l'implémentation
5.6.2 - Réduire la recompilation
5.7 - Résumé
5.8 - Exercices
6 - Initialisation & Nettoyage
6.1 - Initialisation garantie avec le constructeur
6.2 - Garantir le nettoyage avec le destructeur
6.3 - Elimination de la définition de bloc
6.3.1 - les boucles
6.3.2 - Allocation de mémoire
6.4 - Stash avec constructueur et destructeur
6.5 - Stack avec des constructeurs & des destructeurs
6.6 - Initialisation d'aggrégat
6.7 - Les constructeurs par défaut
6.8 - Résumé
6.9 - Exercices
7 - Fonctions surchargée et arguments par défaut
7.1 - Plus sur les décorations de nom
7.1.1 - Valeur de retour surchargée :
7.1.2 - Edition de liens sécurisée
7.2 - Exemple de surchargement
7.3 - unions
7.4 - Les arguments par défaut
7.4.1 - Paramètre fictif
7.5 - Choix entre surcharge et arguments par défaut
7.6 - Résumé
7.7 - Exercices
8 - Constantes
8.1 - Substitution de valeurs
8.2 - Les pointeurs
8.2.1 - Pointeur vers const
8.2.2 - Pointeur const
8.2.3 - Assignation et vérification de type
8.3 - Arguments d'une fonction & valeurs retournées
8.3.1 - Passer par valeur const
8.3.2 - Retour par valeur const
8.3.3 - Passer et retourner des adresses
8.4 - Classes

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

8.4.1 - const dans les classes
8.4.2 - Constantes de compilation dans les classes
8.4.3 - objets cont & fonctions membres
8.5 - volatile
8.6 - Résumé
8.7 - Exercices
9 - Fonctions inlines
9.1 - Ecueils du préprocesseurs
9.1.1 - Les macros et l'accès
9.2 - Fonctions inline
9.2.1 - Les inline dans les classes
9.2.2 - Fonctions d'accès
9.3 - Stash & Stack avec les inlines
9.4 - Les inline & le compilateur
9.4.1 - Limitations
9.4.2 - Déclarations aval
9.4.3 - Activités cachées dans les constructeurs et les destructeurs
9.5 - Réduire le fouillis
9.6 - Caractéristiques supplémentaires du préprocesseur
9.6.1 - Collage de jeton
9.7 - Vérification d'erreurs améliorée
9.8 - Résumé
9.9 - Exercices
10 - Contrôle du nom
10.1 - Eléments statiques issus du C
10.1.1 - Variables statiques à l'intérieur des fonctions
10.1.2 - Contrôle de l'édition de liens
10.1.3 - Autre spécificateurs de classe de stockage
10.2 - Les namespaces
10.2.1 - Créer un espace de nommage
10.2.2 - Utiliser un espace de nommage
10.2.3 - L'utilisation des espace de nommage
10.3 - Membres statiques en C++
10.3.1 - Définir le stockage pour les données membres statiques
10.3.2 - Classes imbriquées et locales
10.3.3 - Fonctions membres statiques
10.4 - Dépendance de l'initialisation statique
10.4.1 - Que faire
10.5 - Spécification alternative des conventions de liens
10.6 - Sommaire
10.7 - Exercices
11 - Références & le constructeur de copie
11.1 - Les pointeurs en C++
11.2 - Les références en C++
11.2.1 - Les références dans les fonctions
11.2.2 - Indications sur le passage d'arguments
11.3 - Le constructeur par recopie
11.3.1 - Passer & renvoyer par valeur
11.3.2 - Construction par recopie
11.3.3 - Constructeur par recopie par défaut
11.3.4 - Alternatives à la construction par recopie
11.4 - Pointeurs sur membre
11.4.1 - Fonctions
11.5 - Résumé
11.6 - Exercices

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

12 - Surcharges d'opérateurs
12.1 - Soyez avertis et rassurés
12.2 - Syntaxe
12.3 - Opérateurs surchargeables
12.3.1 - Opérateurs unaires
12.3.2 - Opérateurs binaires
12.3.3 - Arguments & valeurs de retour
12.3.4 - opérateurs inhabituels
12.3.5 - Opérateurs que vous ne pouvez pas surcharger
12.4 - Opérateurs non membres
12.4.1 - Conseils élémentaires
12.5 - Surcharge de l'affectation
12.5.1 - Comportement de operator=
12.6 - Conversion de type automatique
12.6.1 - Conversion par constructeur
12.6.2 - Conversion par opérateur
12.6.3 - Exemple de conversion de type
12.6.4 - Les pièges de la conversion de type automatique
12.7 - Résumé
12.8 - Exercices
13 - Création d'Objets Dynamiques
13.1 - Création d'objets
13.1.1 - L'approche du C au tas
13.1.2 - l'operateur new
13.1.3 - l'opérateur delete
13.1.4 - Un exemple simple
13.1.5 - Le surcoût du gestionnaire de mémoire
13.2 - Exemples précédents revus
13.2.1 - detruire un void* est probablement une erreur
13.2.2 - La responsabilité du nettoyage avec les pointeurs
13.2.3 - Stash pour des pointeurs
13.3 - new & delete pour les tableaux
13.3.1 - Rendre un pointeur plus semblable à un tableau
13.4 - Manquer d'espace de stockage
13.5 - Surcharger new & delete
13.5.1 - La surcharge globale de new & delete
13.5.2 - Surcharger new & delete pour une classe
13.5.3 - Surcharger new & delete pour les tableaux
13.5.4 - Appels au constructeur
13.5.5 - new & delete de placement
13.6 - Résumé
13.7 - Exercices
14 - Héritage & composition
14.1 - Syntaxe de la composition
14.2 - Syntaxe de l'héritage
14.3 - La liste d'initialisation du construteur
14.3.1 - Initialisation d'un objet membre
14.3.2 - Types prédéfinis dans la liste d'initialisation
14.4 - Combiner composition & héritage
14.4.1 - Ordre des appels des constructeurs & et des destructeurs
14.5 - Masquage de nom
14.6 - Fonctions qui ne s'héritent pas automatiquement
14.6.1 - Héritage et fonctions membres statiques
14.7 - Choisir entre composition et héritage
14.7.1 - Sous-typer

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

14.7.2 - héritage privé
14.8 - protected
14.8.1 - héritage protégé
14.9 - Surcharge d'opérateur & héritage
14.10 - Héritage multiple
14.11 - Développement incrémental
14.12 - Transtypage ascendant
14.12.1 - Pourquoi "ascendant" ?
14.12.2 - Le transtypage ascendant et le constructeur de copie
14.12.3 - Composition vs. héritage (révisé)
14.12.4 - Transtypage ascendant de pointeur & de reference
14.12.5 - Une crise
14.13 - Résumé
14.14 - Exercices
15 - Polymorphisme & Fonctions Virtuelles
15.1 - Evolution des programmeurs C++
15.2 - Transtypage ascendant (upcasting)
15.3 - Le problème
15.3.1 - Liaison d'appel de fonction
15.4 - Fonctions virtuelles
15.4.1 - Extensibilité
15.5 - Comment le C++ implémente la liaison tardive
15.5.1 - Stocker l'information de type
15.5.2 - Représenter les fonntions virtuelles
15.5.3 - Sous le capot
15.5.4 - Installer le vpointeur
15.5.5 - Les objets sont différents
15.6 - Pourquoi les fonctions virtuelles ?
15.7 - Classes de base abstraites et fonctions virtuelles pures
15.7.1 - Définitions virtuelles pures
15.8 - L'héritage et la VTABLE
15.8.1 - Découpage d'objets en tranches
15.9 - Surcharge & redéfinition
15.9.1 - Type de retour covariant
15.10 - Fonctions virtuelles & constructeurs
15.10.1 - Ordre des appels au constructeur
15.10.2 - Comportement des fonctions virtuelles dans les constructeurs
15.11 - Destructeurs et destructeurs virtuels
15.11.1 - Destructeurs virtuels purs
15.11.2 - Les virtuels dans les destructeurs
15.11.3 - Créer une hiérarchie basée sur objet
15.12 - Surcharge d'opérateur
15.13 - Transtypage descendant
15.14 - Résumé
15.15 - Exercices
16 - Introduction aux Templates
16.1 - Les conteneurs
16.1.1 - Le besoin de conteneurs
16.2 - Survol des templates
16.2.1 - La solution template
16.3 - Syntaxe des templates
16.3.1 - Définitions de fonctions non inline
16.3.2 - IntStack comme template
16.3.3 - Constantes dans les templates
16.4 - Stack et Stash comme templates

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

16.4.1 - Pointeur Stash modélisé
16.5 - Activer et désactiver la possession
16.6 - Stocker des objets par valeur
16.7 - Présentation des itérateurs
16.7.1 - Stack avec itérateurs
16.7.2 - PStash avec les iterateurs
16.8 - Pourquoi les itérateurs ?
16.8.1 - Les templates de fonction
16.9 - Résumé
16.10 - Exercices
17 - A: Le style de codage
18 - B: Directives de programmation
19 - C: Lectures recommandées
19.1 - C
19.2 - C++ en général
19.2.1 - Ma propre liste de livres
19.3 - Profondeurs et recoins
19.4 - Analyse & conception
20 - Copyright et traduction
20.1 - Pour la version anglaise :
20.2 - Pour la version française :
20.2.1 - Equipe de traduction :
20.2.2 - Relecteurs
20.2.3 - Mise en place du projet

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

0 - Préface

Comme n'importe quel langage humain, le C++ permet d'exprimer des concepts. S'il est réussi, ce support d'expression sera plus simple et plus flexible que les solutions alternatives, qui au fur et à mesure que les problèmes s'amplifient, deviennent plus complexes.

On ne peut pas simplement considérer le C++ comme un ensemble de fonctionnalités - certaines fonctionnalités n'ayant pas de sens prises à part. On ne peut utiliser la somme des parties que si l'on pense *conception*, et non simplement code. Et pour comprendre le C++ de cette façon, il faut comprendre les problèmes liés au C et à la programmation en général. Ce livre traite des problèmes de programmation, pourquoi ce sont des problèmes, et l'approche que le C++ a prise pour résoudre de tels problèmes. Ainsi, le groupe de fonctionnalités que je traite dans chaque chapitre sera organisé selon la façon que j'ai de voir comment le langage résout un type particulier de problème. De cette manière j'espère vous amener, au fur et à mesure, depuis la compréhension du C jusqu'au point où la mentalité C++ devient une seconde nature.

Du début à la fin, j'adopterai l'attitude selon laquelle vous voulez construire un modèle dans votre tête qui vous permettra de comprendre le langage jusque dans ses moindres détails - si vous avez affaire à un puzzle, vous serez capable de l'assembler selon votre modèle et d'en déduire la réponse. J'essayerai de vous transmettre les idées qui ont réarrangé mon cerveau pour me faire commencer à «penser en C++.»

0.1 - Quoi de neuf dans cette seconde édition ?

Ce livre est une réécriture complète de la première édition pour refléter tous les changements introduits dans le langage C++ par la finalisation du standard C++, et pour refléter également ce que j'ai appris depuis l'écriture de la première édition. L'ensemble du texte présent dans la première édition a été examiné et réécrit, parfois en supprimant de vieux exemples, souvent en modifiant les exemples existants et en en ajoutant de nouveaux, et en ajoutant beaucoup de nouveaux exercices. Un réarrangement et un nouveau classement significatif du document ont eu lieu afin de répercuter la disponibilité de meilleurs outils et ma meilleure appréhension de la façon dont les gens apprennent le C++. Un nouveau chapitre a été ajouté, rapide introduction aux concepts du C et aux fonctionnalités de base du C++, pour ceux qui n'ont pas l'expérience du C pour aborder le reste du livre. Le CD ROM relié au dos du livre contient une conférence qui est une introduction encore plus douce aux concepts du C nécessaires pour comprendre le C++ (ou le Java). Elle a été créée par Chuck Allison pour ma société (MindView, Inc.), et est appelée "Penser en C : Bases pour Java et C++." Elle vous présente les aspects du C nécessaires pour passer au C++ ou à Java, en laissant de côté les parties difficiles auxquelles les programmeurs C sont confrontés dans leur travail de tous les jours, mais que les langages C++ et Java vous évitent d'utiliser (voire éliminent, dans le cas de Java).

Donc la réponse courte à la question "Qu'est-ce qui est nouveau dans la deuxième édition ?" est : ce qui n'est pas nouveau a été réécrit, parfois à tel point que vous ne reconnaitrez pas les exemples et le contenu d'origine.

0.1.1 - Qu'y a-t-il dans le Volume 2 de ce livre ?

L'achèvement du standard C++ a également ajouté un certain nombre de nouvelles bibliothèques importantes, comme les chaînes de caractères (**string**) et les conteneurs et algorithmes de la bibliothèque standard du C++, aussi bien que la nouvelle complexité des templates. Ces nouveautés et d'autres sujets plus avancés ont été relégués au volume 2 de ce livre, y compris des problèmes tels l'héritage multiple, la gestion des exceptions, les modèles de conception, et les sujets sur les systèmes stables de compilation et de débogage.

0.1.2 - Comment récupérer le Volume 2 ?

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Tout comme le livre que vous tenez actuellement, *Penser en C++, Volume 2*est en téléchargement dans sa totalité sur mon site Web *www.BruceEckel.com*. Vous pouvez trouver la date d'impression prévue du Volume 2 sur le site.

Le site Web contient également le code source pour les deux livres, avec des mises à jour et des informations sur d'autres conférences-sur-CD ROM proposées par MindView, Inc., des conférences publiques, et des formations, consultations, tutelles, et visites internes.

0.2 - Prérequis

Dans la première édition de ce livre, je suis parti du principe que quelqu'un d'autre vous avait enseigné le C et que vous aviez au moins la capacité de le lire confortablement. Mon objectif premier était de simplifier ce que je trouvais difficile : le langage C++. Dans cette édition j'ai ajouté un chapitre qui est une rapide introduction au C, présent avec le CD du séminaire *Penser en C*, mais je considère toujours que vous avez déjà une certaine expérience en programmation. En outre, de même que vous apprenez beaucoup de nouveaux mots intuitivement en les voyant dans le contexte d'un roman, il est possible d'apprendre beaucoup sur C à partir du contexte dans lequel il est utilisé dans le reste du livre.

0.3 - Apprendre le C++

J'ai suivi le même cheminement vers le C++ que les lecteurs de ce livre: avec une attidude très pragmatique envers la programmation, très "détails pratiques". Pire, mon passé et mon expérience étaient tournés vers la programmation embarquée niveau matériel, dans laquelle le C était souvent considéré comme un langage de haut-niveau et d'une surpuissance inefficace pour déplacer des bits. J'ai découvert plus tard que je n'étais pas vraiment un très bon programmeur C, cachant mon ignorance des structures, **malloc( )**et **free( )**, **setjmp( )**et **longjmp( )**, et autres concepts "sophistiqués", m'éloignant rapidement avec honte quand les conversations abordaient ces sujets, au lieu de tendre vers de nouvelles connaissances.

Quand j'ai commencé ma lutte pour apprendre le C++, le seul livre décent était l'autoproclamé "Guide de l'expert, Bjarne Stroustrup, The C++ Programming Language, Addison-Wesley, 1986 (première édition).", de Bjarne Stroustrup, et j'étais donc livré à moi-même pour simplifier les concepts de base. Ceci a eu comme conséquence mon premier livre sur le C++, Using C++, Osborne/McGraw-Hill 1989.ce qui était essentiellement une extraction cérébrale de mon expérience. Il a été conçu comme un guide pour le lecteur afin d'introduire les programmeurs en même temps au C et au C++. Les deux éditions Using C++ et C++ Inside & Out, Osborne/McGraw-Hill 1993.du livre ont entraîné une réaction enthousiaste.

Pratiquement en même temps que la sortie de *Using C++*, j'ai commencé à enseigner le langage dans des conférences et des présentations. Enseigner le C++ (et plus tard le Java) est devenu ma profession ; j'ai vu des têtes penchées, des visages livides, et des expressions embarrassées dans les assistances partout dans le monde depuis 1989. Alors que je commençais à donner des formations internes à de plus petits groupes de personnes, j'ai découvert une chose pendant les exercices. Mêmes ces personnes qui souriaient et acquiesçaient étaient perdues devant de nombreux problèmes. J'ai découvert, en créant et présidant pendant de nombreuses années les sujets C++ et Java à la conférence sur le développement logiciel, que les intervenants (moi compris) avaient tendance à présenter au public type trop de sujets trop rapidement. Ainsi en définitive, du fait de la diversité des niveaux de l'assistance et de la manière dont j'ai présenté le contenu, je finissais par perdre une partie du public. Peut-être est-ce trop demander, mais comme je suis une de ces personnes réfractaire aux cours traditionnels (et pour beaucoup, je crois, une telle résistance est générée par l'ennui), j'ai voulu essayer de garder tout le monde dans le rythme.

Pendant quelques temps, j'ai créé un certain nombre de présentations différentes dans un ordre assez court. Ainsi, j'ai fini par apprendre par expérience et itération (une technique qui fonctionne également bien dans la conception de programmes C++). Par la suite, j'ai développé un cours en utilisant tout ce que j'avais appris de mon expérience d'enseignement. Il aborde le problème de l'apprentissage par des étapes distinctes, faciles à digérer, et pour

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

impliquer l'auditoire (situation d'apprentissage idéale), des exercices suivent chacune des présentations. Vous pouvez découvrir mes conférences publiques sur *www.BruceEckel.com*, et vous pouvez également vous y renseigner sur les conférences que j'ai publié en CD ROM.

La première édition de ce livre s'est développée dans un cheminement de deux ans, et le contenu de ce livre a été testé en situation dans beaucoup de formes dans de nombreuses conférences. Le retour que j'ai perçu de chaque conférence m'a aidé à modifier et recentrer le sujet jusqu'à ce que je sente qu'il fonctionne bien comme outil didactique. Mais ce n'est pas simplement une documentation de conférence ; j'ai essayé d'entasser autant d'informations que possible dans ces pages, et de la structurer pour vous emmener vers le sujet suivant. Plus que tout autre chose, le livre est conçu pour servir le lecteur solitaire qui lutte avec un nouveau langage de programmation.

0.4 - Buts

Dans ce livre je me suis donné comme buts :

1. Présenter le cours pas à pas afin que le lecteur assimile chaque concept avant d'aller plus loin.
2. Utiliser des exemples qui soient aussi simples et courts que possible. Souvent, cela me détournera des problèmes « du monde réel », mais j'ai remarqué que les débutants sont généralement plus satisfaits de comprendre chaque détail d'un exemple qu'ils ne sont impressionnés par la portée du problème qu'il résout. Il y a également une limite à la taille du code qui peut être assimilé dans une situation de cours magistral, limite qu'il faut impérativement ne pas dépasser. A ce sujet je reçois parfois des critiques pour avoir utilisé des « exemples jouets », et je les accepte volontiers, avec le prétexte que ce que je présente est utile, pédagogiquement parlant.
3. Enchaîner soigneusement la présentation des fonctionnalités afin que l'on ne rencontre jamais quoi que ce soit qui n'ait jamais été exposé. Bien entendu, ce n'est pas toujours possible, et, dans de telles situations, je donnerai une brève description en introduction.
4. Montrer ce que je pense être important concernant la compréhension du langage, plutôt qu'exposer tout mon savoir. Je crois que l'information est fortement hiérarchisée, qu'il est avéré que 95 % des programmeurs n'ont pas besoin de tout connaître, et que cela ne ferait que les embrouiller et accroître leur impression de complexité du langage. Pour prendre un exemple en C, en connaissant par coeur le tableau de priorité des opérateurs (ce qui n'est pas mon cas), il est possible d'écrire un code astucieux. Mais si vous y réfléchissez un instant, ceci risque de dérouter le lecteur et/ou le mainteneur de ce code. Il est donc préférable d'oublier la priorité des opérateurs, et d'utiliser des parenthèses lorsque les choses ne sont pas claires. Une attitude similaire sera adoptée avec certaines informations du langage C++, qui je pense sont plus importantes pour les réalisateurs de compilateurs que pour les programmeurs.
5. Maintenir chaque section assez concentrée de telle manière que le temps de lecture - et le temps entre les exercices - soit raisonnable. Non seulement cela maintient l'attention et l'implication des auditeurs lors d'un séminaire, mais cela donne au lecteur une plus grande impression de travail bien fait.
6. Munir les lecteurs de bases solides afin que leurs connaissances soient suffisantes pour suivre un cours ou lire un livre plus difficiles (en particulier, le volume 2 de ce livre).
7. J'ai essayé de n'utiliser aucune version particulière d'un quelconque revendeur C++ parce que, pour apprendre un langage, je ne pense pas que les détails d'une implémentation particulière soient aussi importants que le langage lui-même. La documentation fournie par les revendeurs au sujet de leurs propres spécificités d'implémentation est la plus part du temps suffisante.

0.5 - Chapitres

Le C++ est un langage dans lequel des fonctionnalités nouvelles et différentes ont été mises en place à partir d'une syntaxe existante (de ce fait, il est considéré comme un langage de programmation orienté objet *hybride*). Au fur et à mesure que des gens dépassaient la phase d'apprentissage, nous avons commencé à comprendre la manière dont les développeurs franchissaient les étapes des fonctionnalités du langage C++. Puisque cela semblait être la progression naturelle d'un esprit entraîné aux langages procéduraux, j'ai décidé de comprendre et de suivre

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

moi-même ce cheminement et d'accélérer le processus en exprimant et en répondant aux questions qui me sont venues alors que j'apprenais ce langage, ou qui sont venues de ceux à qui j'apprenais ce langage.

J'ai conçu ce cours en gardant une chose à l'esprit : améliorer le processus d'apprentissage du C++. Les réactions de mes auditoires m'ont aidé à comprendre quelles parties étaient difficiles et nécessitaient des éclaircissements particuliers. Dans les domaines dans lesquels j'ai été ambitieux et où j'ai inclus trop de fonctionnalités à la fois, j'ai été amené à comprendre - à travers la présentation de ces connaissances - que si vous incluez beaucoup de nouvelles fonctionnalités, vous devez toutes les expliquer, et cela génère facilement la confusion des étudiants. En conséquence, j'ai essayé d'introduire le moins de fonctionnalités possibles à la fois ; idéalement, un concept principal par chapitre seulement.

L'objectif est alors d'enseigner un unique concept pour chaque chapitre, ou un petit groupe de concepts associés, de telle façon qu'aucune fonctionnalité additionnelle ne soit requise. De cette façon, vous pouvez digérer chaque partie en fonction de vos connaissances actuelles avant de passer à la suivante. Pour cela, j'ai conservé des fonctionnalités du C plus longtemps que je ne l'aurais souhaité. L'avantage, c'est que vous ne serez pas perturbés par la découverte de fonctionnalités C++ utilisées avant d'être expliquées, et votre introduction à ce langage sera douce et reflétera la manière dont vous l'auriez assimilé si vous deviez vous débrouiller seul.

Voici une rapide description des chapitres de cet ouvrage :

Chapitre 1 : Introduction à l'objet. Quand les projets sont devenus trop importants et complexes à maintenir simplement, la "crise du logiciel" est née, avec des programmeurs disant "Nous ne pouvons pas terminer les projets, et si nous le pouvons, ils sont trop chers !". Cela a entraîné un certain nombre de réponses, qui sont développées dans ce chapitre avec les idées de la programmation orientée objet (POO) et comment elle tente de résoudre la crise du logiciel. Ce chapitre vous entraîne à travers les concepts et fonctionnalités de base de la POO et introduit également les processus d'analyse et de conception. Par ailleurs, vous découvrirez les avantages et inconvénients de l'adoption du langage, ainsi que des suggestions pour entrer dans le monde du C++.

Chapitre 2 : Créer et utiliser des objets. Ce chapitre explique le processus de création de programmes utilisant des compilateurs et des bibliothèques. Il introduit le premier programme C++ du livre et montre comment les programmes sont construits et compilés. Des bibliothèques objet de base disponibles dans le Standard C++ sont alors introduites. Lorsque vous aurez terminé ce chapitre, vous aurez une bonne compréhension de ce que signifie écrire un programme C++ utilisant les bibliothèques objet disponibles immédiatement.

Chapitre 3 : Le C dans C++. Ce chapitre est une vue d'ensemble dense des fonctionnalités du C qui sont utilisées en C++, ainsi que d'un certain nombre de fonctionnalités de base disponibles uniquement en C++. Il présente également "make", un utilitaire commun dans le monde du développement logiciel, qui est utilisé pour compiler tous les exemples de ce livre (le code source du livre, qui est disponible sur *www.BruceEckel.com*, contient un makefile pour chaque chapitre). Le chapitre 3 suppose que vous ayez de solides connaissances dans des langages de programmation procéduraux comme le Pascal, le C, ou même du Basic (tant que vous avez écrit beaucoup de code dans ce langage, particulièrement des fonctions). Si vous trouvez ce chapitre un peu trop dense, vous devriez commencer par la conférence *Penser en C*disponible sur le CD livré avec ce livre (et également disponible sur *www.BruceEckel.com*).

Chapitre 4 : Abstraction des données. La plupart des fonctionnalités en C++ tournent autour de la possibilité de créer de nouveaux types de données. Non seulement cela permet une meilleure organisation du code, mais aussi cela prépare les fondations pour des capacités plus puissantes de POO. Vous verrez comme cette idée est facilitée par le simple fait de mettre des fonctions à l'intérieur de structures, les détails sur la façon de le faire, et le type de code que cela entraîne. Vous apprendrez aussi la meilleure manière d'organiser votre code en fichiers d'en-tête et fichiers d'implémentation.

Chapitre 5 : Masquer l'implémentation. Vous pouvez décider que certaines données ou fonctions de votre structure sont inaccessible à l'utilisateur de ce nouveau type en les rendant **privées**. Cela signifie que vous pouvez

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

séparer l'implémentation sous-jacente de l'interface que le programmeur client peut voir, et ainsi permettre de modifier facilement cette implémentation sans pour autant affecter le code du client. Le mot-clé **class**est également présenté comme un moyen spécialisé de description d'un nouveau type de données, et la signification du mot "objet" est démystifiée (c'est une variable spécialisée).

Chapitre 6 : Initialisation et nettoyage. Une des erreurs C les plus classiques provient de la non initialisation des variables. Le *constructeur*du C++ vous permet de garantir que les variables de votre nouveau type de données ("les objets de votre classe") seront toujours initialisées correctement. Si vos objets nécessitent aussi une certaine forme de nettoyage, vous pouvez garantir que ce nettoyage aura toujours lieu à l'aide du *destructeur*du C++.

Chapitre 7 : Surcharge de fonctions et arguments par défaut. C++ est prévu pour vous aider à construire des projets volumineux, complexes. Tout en le faisant, vous pouvez introduire des bibliothèques multiples qui utilisent les mêmes noms de fonctions, et vous pouvez aussi choisir d'utiliser le même nom avec différentes significations dans une bibliothèque unique. C++ le facilite avec la *surcharge de fonction*, qui vous permet de réutiliser le même nom de fonction tant que les listes d'arguments sont différentes. Les arguments par défaut vous permettent d'appeler la même fonction de différentes façons en fournissant automatiquement des valeurs par défauts pour certains de vos arguments.

Chapitre 8 : Constantes. Ce chapitre couvre les mots-clés **const**et **volatile**, qui ont des significations supplémentaires en C++, particulièrement à l'intérieur d'une classe. Vous apprendrez ce que signifie appliquer **const**à la définition d'un pointeur. Ce chapitre vous expliquera aussi comment la signification de **const**varie quand il est utilisé à l'intérieur ou à l'extérieur des classes, et comment créer des constantes dans les classes à la compilation.

Chapitre 9 : Fonctions 'inline'. Les macros du préprocesseur éliminent les coûts d'appel de fonction, mais le préprocesseur élimine aussi la vérification du type C++ valable. La fonction 'inline' apporte tous les avantages d'une macro du préprocesseur plus tous ceux d'un véritable appel de fonction. Ce chapitre explore complètement l'implémentation et l'utilisation des fonctions inline.

Chapitre 10 : Contrôle des noms. Créer des noms est une activité fondamentale en programmation, et quand un projet devient important, le nombre de noms peut devenir envahissant. Le C++ permet un grand contrôle des noms pour leur création, leur visibilité, leur placement de stockage, et leurs liens. Ce chapitre vous montre comment les noms sont contrôlés en C++ en utilisant deux techniques. Tout d'abord, le mot-clé **static**est utilisé pour contrôler la visibilité et les liens, et nous étudions sa signification particulière avec les classes. Une technique bien plus utile pour contrôler les noms de portée globale est la fonctionnalité d' **espace de nommage**du C++, qui vous permet de séparer l'espace de nom global en régions distinctes.

Chapitre 11 : Références et constructeur par copie. Les pointeurs C++ fonctionnent comme les pointeurs C avec l'avantage supplémentaire d'une vérification du type plus forte en C++. Le C++ fournit également une autre possibilité de manipulation des adresses : de l'Algol et du Pascal, le C++ relève la **référence**, qui laisse le compilateur gérer la manipulation des adresses alors que vous utilisez la notation normale. Vous rencontrerez aussi le constructeur par copie, qui contrôle la manière dont les objets sont passés par valeur en argument ou en retour de fonction. En conclusion, le pointeur-vers-membre du C++ sera éclairci.

Chapitre 12 : Surcharge des opérateurs. Cette fonctionnalité est parfois appelée le "sucre syntaxique" ; elle vous permet d'adoucir la syntaxe d'utilisation de votre type en autorisant des opérateurs comme des appels de fonctions. Dans ce chapitre vous apprendrez que la surcharge des opérateurs est simplement un type différent d'appel de fonction et vous apprendrez à écrire le votre, en prenant en considération les utilisations parfois déroutantes des arguments, des types de retour, et la décision de faire d'un opérateur soit un membre soit un ami.

Chapitre 13 : Création dynamique des objets. Combien d'avions un système de gestion du trafic aérien devra-t-il contrôler ? Combien de formes un système de DAO nécessitera-t-il ? Dans le problème général de la programmation, vous ne pouvez pas connaître la quantité, la durée de vie, ou même le type des objets requis par

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

votre programme en fonctionnement. Dans ce chapitre, vous apprendrez comment les **new**et les **delete**du C++ répondent d'une manière élégante à ce problème en créant sans risque des objets sur le tas. Vous verrez aussi comment **new**et **delete**peuvent être surchargés de multiples façons, vous permettant de contrôler l'allocation et la libération de la mémoire.

Chapitre 14 : Héritage et composition. L'abstraction des données vous permet de créer de nouveaux types de toutes pièces, mais avec la composition et l'héritage, vous pouvez créer de nouveaux types à partir de types existants. Avec la composition, vous assemblez un nouveau type en utilisant d'autres types comme des pièces, et avec l'héritage, vous créez une version plus spécifique d'un type existant. Dans ce chapitre, vous apprendrez la syntaxe, la redéfinition des fonctions, et l'importance de la construction et de la destruction dans l'héritage et la composition.

Chapitre 15 : Polymorphisme et fonctions virtuelles. Tout seul, vous pourriez prendre neuf mois pour découvrir et comprendre cette pierre angulaire de la POO. A travers des exemples courts et simples, vous verrez comment créer une famille de types avec l'héritage, et manipuler les objets de cette famille à travers leur classe de base commune. Le mot-clé **virtual**vous permet de traiter de façon générique tous les objets de cette famille, ce qui signifie que la majeure partie de votre code ne reposera pas sur des informations spécifiques des types. Cela rend votre programme extensible, et la construction des programmes et la maintenance du code sont plus simples et moins coûteuses.

Chapitre 16 : Introduction aux modèles. L'héritage et la composition vous permettent de réutiliser le code des objets, mais cela ne résout pas tous vos besoins de réutilisation. Les modèles permettent de réutiliser le code *source*en fournissant au compilateur un moyen de substituer des noms de types dans le corps d'une classe ou d'une fonction. Cela aide à l'utilisation de bibliothèques de *classes conteneurs*, qui sont des outils importants pour le développement rapide et robuste de programmes orientés objet (la bibliothèque standard C++ inclus une bibliothèque significative de classes conteneurs). Ce chapitre vous donne des bases complètes sur ce sujet essentiel.

Des sujets supplémentaires (et plus avancés) sont disponibles dans le tome 2 de ce livre, qui peut être téléchargé sur le site *www.BruceEckel.com*.

0.6 - Exercices

Je me suis aperçu que des exercices sont très utiles lors d'un séminaire pour consolider les connaissances des étudiants, on en trouvera donc un ensemble à la fin de chaque chapitre. Le nombre d'exercices a été considérablement augmenté par rapport à celui de la première édition.

Un grand nombre d'entre eux sont assez simples de sorte qu'ils peuvent être réalisés en un temps raisonnable dans le contexte d'une salle de classe, pendant que l'instructeur vérifie que tous les étudiants ont assimilé le sujet de la leçon. Quelques exercices sont plus pointus, afin d'éviter l'ennui chez les étudiants expérimentés. La majeure partie des exercices sont conçus pour être réalisés rapidement, ainsi que pour tester et perfectionner les connaissances plutôt que de représenter des défis majeurs. (Je présume que vous les trouverez par vous-même - ou plutôt qu'ils vous trouveront).

0.6.1 - Solutions des exercices

Les solutions des exercices se trouvent dans le document électronique *The Thinking in C++ Annotated Solution Guide*, disponible pour un faible coût sur *http://www.BruceEckel.com*.

0.7 - Le Code Source

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Le code source de ce livre est disponible en freeware sous copyright, via le site Web *http://www.BruceEckel.com*. Le copyright vous empêche de réutiliser le code sans autorisation dans un médium imprimé, mais vous accorde le droit de l'employer dans beaucoup d'autres situations (voir ci-dessous).

Le code est disponible sous forme de fichier zippé, conçu pour être extrait sous toutes les plateformes disposant d'un utilitaire “zip” (la plupart le sont - vous pouvez effectuer une recherche sur l'Internet afin de trouver une version pour votre plateforme si vous n'avez rien de préinstallé). Dans le répertoire de départ où vous avez décompacté le code vous trouverez la mention de copyright suivante :

```
                    //:! :Copyright.txt
Copyright (c) 2000, Bruce Eckel
Source code file from the book "Thinking in C++"
All rights reserved EXCEPT as allowed by the
following statements: You can freely use this file
for your own work (personal or commercial),
including modifications and distribution in
executable form only. Permission is granted to use
this file in classroom situations, including its
use in presentation materials, as long as the book
"Thinking in C++" is cited as the source.
Except in classroom situations, you cannot copy
and distribute this code; instead, the sole
distribution point is http://www.BruceEckel.com
(and official mirror sites) where it is
available for free. You cannot remove this
copyright and notice. You cannot distribute
modified versions of the source code in this
package. You cannot use this file in printed
media without the express permission of the
author. Bruce Eckel makes no representation about
the suitability of this software for any purpose.
It is provided "as is" without express or implied
warranty of any kind, including any implied
warranty of merchantability, fitness for a
particular purpose, or non-infringement. The entire
risk as to the quality and performance of the
software is with you. Bruce Eckel and the
publisher shall not be liable for any damages
suffered by you or any third party as a result of
using or distributing this software. In no event
will Bruce Eckel or the publisher be liable for
any lost revenue, profit, or data, or for direct,
indirect, special, consequential, incidental, or
punitive damages, however caused and regardless of
the theory of liability, arising out of the use of
or inability to use software, even if Bruce Eckel
and the publisher have been advised of the
possibility of such damages. Should the software
prove defective, you assume the cost of all
necessary servicing, repair, or correction. If you
think you've found an error, please submit the
correction using the form you will find at
www.BruceEckel.com. (Please use the same
form for non-code errors found in the book.)
///:~
```

Vous êtes autorisés à utiliser le code pour vos projets ainsi qu'à des fins d'éducation à la condition de conserver l'indication de copyright.

0.8 - Normes du langage

Dans ce livre, pour me référer à la conformité au standard C ISO, je parlerai généralement simplement de 'C'. Je ferai une distinction uniquement s'il est nécessaire de différencier le C Standard de versions du C plus anciennes, pré-Standard.

Alors que j'écris ce livre, le comité de normalisation du C++ a terminé de travailler sur le langage. Ainsi, j'utiliserai

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

le terme *C++ standard*pour me référer au langage standardisé. Si je me réfère simplement à C++ vous devez considérer que je parle du C++ standard.

Il y a une certaine confusion à propos du nom exact du comité de normalisation du C++, et du nom de la norme elle-même. Steve Clamage, président du comité, l'a clarifié :

Il y a deux comités de normalisation du C++ : le comité J16 NCITS (anciennement X3) et le comité ISO JTC1/SC22/WG14. L'ANSI a chargé le NCITS de créer des comités techniques pour développer des normes nationales américaines

Le J16 a été chargé en 1989 de créer une norme américaine pour le C++. Vers 1991, le WG14 a été chargé de créer un standard international. Le projet du J16 a été convertis en projet de "Type I" (international) et subordonné à l'effort de standardisation de l'ISO.

Les deux comités se réunissent en même temps au même endroit, et la voix du J16 constitue le vote américain au WG14. Le WG14 délègue le travail technique au J16. Le WG14 vote sur le travail technique du J16.

Le standard C++ a été créé à l'origine comme un standard ISO. L'ANSI a plus tard choisi (comme le recommandait le J16) d'adopter le C++ standard ISO comme norme américaine pour le C++.

Ainsi, l'"ISO" est la manière exacte de se référer au standard C++.

0.8.1 - Support du langage

Votre compilateur peut ne pas supporter toutes les fonctionnalités traitées dans ce livre, particulièrement si vous n'en avez pas la version la plus récente. Implémenter un langage comme le C++ est une tâche Herculéenne, et vous pouvez vous attendre à ce que les fonctionnalités apparaissent par petits bouts plutôt que toutes en même temps. Mais si vous essayez un des exemples du livre et obtenez beaucoup d'erreurs du compilateur, ce n'est pas nécessairement un bug du code ou du compilateur ; c'est peut-être simplement que ça n'a pas encore été implémenté sur votre propre compilateur.

0.9 - Le CD ROM du livre

Le contenu principal du CD ROM fourni à la fin de ce livre est une "conférence sur CD ROM" intitulée *Penser en C : Bases pour Java et C++*par Chuck Allison (édité par MindView, Inc., et aussi disponible sans limitation sur *www.BruceEckel.com*). Il contient de nombreuses heures de conférences audio et des présentations, et peut être consulté sur la plupart des ordinateurs si vous avez un lecteur de CD ROM et une carte son.

L'objectif de *Penser en C*est de vous emmener prudemment à travers les principes fondamentaux du langage C. Il se concentre sur les connaissances nécessaires pour vous permettre de passer aux langages C++ ou Java, au lieu d'essayer de faire de vous un expert de tous les points sombres du C. (Une des raisons d'utiliser un langage de haut niveau comme le C++ ou le Java est justement d'éviter plusieurs de ces points sombres.) Il contient aussi des exercices et des réponses guidées. Gardez à l'esprit que parce que le chapitre 3 de ce livre surpasse le CD *Penser en C*, le CD ne se substitue pas à ce chapitre, mais devrait plutôt être utilisé comme une préparation à ce livre.

Merci de noter que le CD ROM est basé sur les navigateurs, vous devez donc avoir un navigateur internet installé avant de l'utiliser.

0.10 - CD ROMs, conférences, et consultations

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Des conférences sur CD ROM sont prévues pour couvrir les volumes 1 et 2 de ce livre. Elles comprennent de nombreuses heures de mes conférences audio qui accompagnent des présentations qui couvrent un domaine sélectionné de chaque chapitre du livre. Elles peuvent être consultées sur la plupart des ordinateurs si vous avez un lecteur de CD ROM et une carte son. Ces CDs peuvent être achetés sur *www.BruceEckel.com*, où vous trouverez plus d'informations et d'échantillons de conférences.

Ma société, MindView, Inc., propose des conférences publiques de formation pratique basées sur le contenu de ce livre et également sur des sujets avancés. Le domaine sélectionné de chaque chapitre représente une leçon, qui est suivie par des exercices assistés, de manière à ce que chaque étudiant reçoit une attention personnalisée. Nous assurons également des formations internes, des consultations, des tutelles et des analyses de design et de code. Les informations et les formulaires d'inscription aux prochaines conférences et toute autre information de contact peuvent être trouvées sur *www.BruceEckel.com*.

Je suis parfois disponible pour de la consultation de conception, de l'évaluation de projet et des analyses de code. Quand j'ai commencé à écrire au sujet des ordinateurs, ma principale motivation était d'augmenter mes activités de consultant, parce que je trouve que le conseil est stimulant, éducatif, et une de mes expériences professionnelles les plus agréables. Ainsi, je ferai de mon mieux pour vous loger dans mon agenda, ou pour vous fournir un de mes associés (qui sont des personnes que je connais bien et en qui j'ai confiance, et souvent des personnes qui co-développent et présentent les conférences avec moi).

0.11 - Erreurs

Peu importe le nombre d'astuces utilisées par un écrivain pour détecter les erreurs, certaines vont toujours s'introduire dans le texte et sauteront aux yeux d'un lecteur différent. Si vous découvrez quelque chose qui vous semble être une erreur, utilisez, s'il-vous-plaît, le formulaire de corrections que vous trouverez sur le site www.BruceEckel.com. Votre aide est appréciée.

0.12 - A propos de la couverture

La première édition de ce livre avait mon visage sur la couverture, mais j'ai souhaité dès le début avoir pour la seconde édition une couverture qui serait plus une oeuvre d'art, comme la couverture de *Penser en Java*. Pour certaines raisons, le C++ me semble suggérer l'Art Déco avec ses courbes simples et ses chromes brossés. J'avais à l'esprit quelque chose comme ces affiches de bateaux et d'avions aux longs corps étendus.

Mon ami Daniel Will-Harris (*www.Will-Harris.com*), que j'ai rencontré pour la première fois dans la chorale junior du lycée, a continué pour devenir un excellent dessinateur et auteur. Il a fait pratiquement tous mes dessins, y compris la couverture pour la première édition de ce livre. Pendant le processus de conception de couverture, Daniel, mécontent des progrès que nous faisions, continuait à demander "Comment cela relie-t-il les personnes aux ordinateurs ?" nous étions coincés.

Sur un caprice, sans idée particulière à l'esprit, il m'a demandé de mettre mon visage sur le scanner. Daniel a fait "autotracer" le scan de mon visage par un de ses logiciels graphiques (Corel Xara, son favori). Comme il le décrit, "Autotracer est la manière de l'ordinateur de transformer une image en lignes et courbes qui y ressemblent vraiment." Il a alors joué avec ça jusqu'à ce qu'il obtienne un résultat qui ressemble à une carte topographique de mon visage, une image qui pourrait être la façon dont un ordinateur pourrait voir les personnes.

J'ai pris cette image et l'ai photocopiée sur du papier aquarelle (certains copieurs couleur peuvent manipuler des papiers épais), et j'ai alors commencé à faire de nombreuses expériences en ajoutant de l'aquarelle à l'image. Nous avons choisi nos préférées, et Daniel les a de nouveau scannées et arrangées sur la couverture, ajoutant le texte et les autres composants graphiques. Le processus complet s'est déroulé sur plusieurs mois, principalement à cause du temps qu'il m'a fallu pour les aquarelles. Mais je l'ai particulièrement apprécié parce que j'ai pu participer à l'art sur la couverture, et parce que ça m'a encouragé à faire plus d'aquarelles (ce qu'ils disent à propos

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

de la pratique est réellement vrai).

0.13 - Conception et production du livre

Le design intérieur du livre a été conçu par Daniel Will-Harris, qui avait l'habitude de jouer avec des lettres décalcomanies au lycée tandis qu'il attendait l'invention des ordinateurs et de la microédition. Cependant, j'ai produit la version imprimable moi-même, ainsi les erreurs de composition sont miennes. Microsoft Word versions 8 et 9 pour Windows ont été utilisé pour écrire ce livre et créer la version imprimable, ainsi que pour générer la table des matières et l'index. (J'ai créé un serveur d'automation COM en Python, appelé depuis les macros VBA de Word, pour m'aider dans l'indexation.) Python (voir *www.Python.org*) a été employé pour créer certains des outils pour vérifier le code, et aurait été employé pour l'outil d'extraction de code si je l'avais découvert plus tôt.

J'ai créé les diagrammes avec Visio - merci à Visio Corporation pour avoir créé cet outil très utile.

La police du corps est Georgia et celle des titres est Verdana. La version imprimable finale a été produite avec Adobe Acrobat 4 et imprimée directement à partir de ce fichier - merci beaucoup à Adobe pour avoir créé un outil qui autorise l'envoi par mail des documents d'impression, cela permettant de faire en une journée de multiples révisions plutôt que de devoir compter sur mon imprimante laser et les services express de nuit. (Nous avons tout d'abord essayé le procédé d'Acrobat sur *Thinking in Java*, et j'ai pu uploader la version finale de ce livre à l'imprimeur aux Etats-Unis depuis l'Afrique du Sud.)

La version HTML a été créée en exportant le document WORD en RTF, puis en utilisant RTF2HTML (voir *http://www.sunpack.com/RTF/*) pour faire la majeure partie du travail de conversion HTML. (merci à Chris Hector pour avoir conçu un outil aussi utile et particulièrement fiable.) Les fichiers résultants ont été nettoyés grâce à un programme que j'ai mis au point en mélangeant plusieurs programme Python, et les WMF ont été convertis en GIF avec JASC PaintShop Pro 6 et son outil de traitement des conversions (merci à JASC pour avoir résolu tant de problèmes pour moi avec leur excellent produit). La coloration syntaxique a été ajoutée via un script Perl, contribution généreuse de Zafir Anjum.

0.14 - Remerciements

Tout d'abord merci à tout ceux sur Internet qui ont soumis leurs corrections et leurs suggestions; vous avez été très utiles en améliorant la qualité de ce livre, et je n'aurais pas pu le faire sans vous. Remerciements tout particulier à John Cook.

Les idées et la compréhensibilité dans ce livre sont le fait de beaucoup de sources : des amis comme Chuck Allison, Andrea Provaglio, Dan Saks, Scott Meyers, Charles Petzold, et Michael Wilk; des pionniers du langage comme Bjarne Stroustrup, Andrew Koenig, et Rob Murray; des membres du Comité de Standardisation du C++ comme Nathan Myers (qui a été particulièrement salutaire et généreux de sa perspicacité), Bill Plauger, Reg Charney, Tom Penello, Tom Plum, Sam Druker, et Uwe Steinmueller; des personnes qui sont intervenues pendant ma présentation à la Conférence sur le Développement Logiciel; et souvent des étudiants qui, pendant mes séminaires, ont posé les questions que j'avais besoin d'entendre afin de rendre les choses plus claires.

Un énorme merci à mon amie Gen Kiyooka, dont la compagnie - Digigami - m'a fournie un serveur Web.

Mon ami Richard Hale Shaw et moi avons enseigné le C++ ensemble; la perspicacité de Richard et son soutient ont été très utiles (ainsi que celle de Kim). Merci également à KoAnn Vikoren, Eric Faurot, Jennifer Jessup, Tara Arrowood, Marco Pardi, Nicole Freeman, Barbara Hanscome, Regina Ridley, Alex Dunne, et le reste des membres de l'équipe de MFI.

Un remerciement spécial à tous mes professeurs et tous mes étudiants (qui sont également mes professeurs).

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Et pour les auteurs préférés, mes profondes appréciation et sympathie pour vos efforts : John Irving, Neal Stephenson, Robertson Davies (vous nous manquez beaucoup), Tom Robbins, William Gibson, Richard Bach, Carlos Castaneda, et Gene Wolfe.

A Guido van Rossum, pour avoir inventé Python et l'avoir offert au monde. Vous avez enrichi ma vie avec votre contribution.

Merci à tout le personnel de Prentice Hall : Alan Apt, Ana Terry, Scott Disanno, Toni Holm, et ma rédactrice électronique (???) Stephanie English. Pour le marketing, Bryan Gambrel et Jennie Burger.

Sonda Donovan qui m'a aidé pour la réalisation du CD Rom. Daniel Will-Harris (bien sûr) qui a créé la sérigraphie du disque lui-même.

A toutes les grandes gens de Crested Butte, merci pour en avoir fait un lieu magique, spécialement à Al Smith (créateur du merveilleux Camp4 Coffee Garden), mes voisins Dave et Erika, Marsha de la librairie de la place Heg, Pat et John de Teocalli Tamale, Sam de la boulangerie-café et Tiller pour son aide avec les recherches audio. Et à toutes les personnes géniales qui traînent à Camp4 et rendent mes matinées intéressantes.

La troupe d'amis supporters inclut, mais n'est pas limitée à eux, Zack Urlocker, Andrew Binstock, Neil Rubenking, Kraig Brockschmidt, Steve Sinofsky, JD Hildebrandt, Brian McElhinney, Brinkley Barr, Larry O'Brien, Bill Gates au *Midnight Engineering Magazine*, Larry Constantine, Lucy Lockwood, Tom Keffer, Dan Putterman, Gene Wang, Dave Mayer, David Intersimone, Claire Sawyers, les Italiens(Andrea Provaglio, Rossella Gioia, Laura Fallai, Marco et Lella Cantu, Corrado, Ilsa and Christina Giustozzi), Chris et Laura Strand (et Parker), les Almquist, Brad Jerbic, Marilyn Cvitanic, les Mabry, les Haflinger, les Pollock, Peter Vinci, les Robbin, les Moelter, Dave Stoner, Laurie Adams, les Cranston, Larry Fogg, Mike and Karen Sequeira, Gary Entsminger et Allison Brody, Kevin, Sonda, et Ella Donovan, Chester et Shannon Andersen, Joe Lordi, Dave et Brenda Bartlett, les Rentschler, Lynn et Todd, et leurs familles. Et bien sûr Maman et Papa.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

1 - Introduction sur les Objets

La genèse de la révolution informatique fut dans l'invention d'une machine. La genèse de nos langage de programmation tend donc à ressembler à cette machine.

Mais les ordinateurs ne sont pas tant des machines que des outils d'amplification de l'esprit(« des vélos pour le cerveau », comme aime à le répéter Steve Jobs) et un nouveau moyen d'expression. Ainsi, ces outils commencent à ressembler moins à des machines et plus à des parties de notre cerveau ou d'autres moyens d'expressions telles que l'écriture, la peinture, la sculpture ou la réalisation de films. La Programmation Orientée Objet fait partie de cette tendance de l'utilisation de l'ordinateur en tant que moyen d'expression.

Ce chapitre présente les concepts de base de la programmation orientée objet (POO), ainsi qu'un survol des méthodes de développement de la POO. Ce chapitre et ce livre présupposent que vous avez déjà expérimenté un langage de programmation procédural, qui ne soit pas forcément le C. Si vous pensez que vous avez besoin de plus de pratique dans la programmation et/ou la syntaxe du C avant de commencer ce livre, vous devriez explorer le CD ROM fourni avec le livre, Thinking in C: Foundations for C++ and Java, également disponible sur *www.BruceEckel.com*.

Ce chapitre tient plus de la culture générale. Beaucoup de personnes ne veulent pas se lancer dans la programmation orientée objet sans en comprendre d'abord les tenants et les aboutissants. C'est pourquoi beaucoup de concepts seront introduits ici afin de vous donner un solide aperçu de la POO. Au contraire, certaines personnes ne saisissent les concepts généraux qu'après en avoir compris une partie des mécanismes; ces gens-là se sentent embourbés et perdus s'ils n'ont pas un bout de code à se mettre sous la dent. Si vous faites partie de cette catégorie de personnes et êtes impatients d'attaquer les spécificités du langage, vous pouvez sauter ce chapitre - cela ne vous gênera pas pour l'écriture de programmes ou l'apprentissage du langage. Mais vous voudrez peut-être y revenir plus tard pour approfondir vos connaissances sur les objets, les comprendre et assimiler la conception objet.

1.1 - Les bienfaits de l'abstraction

Tous les langages de programmation fournissent des abstractions. On peut dire que la complexité des problèmes que vous êtes capable de résoudre est directement proportionnelle au type et à la qualité d'abstraction. Par « type », il faut comprendre « Que tentez-vous d'abstraire ? ». Le langage assembleur est une petite abstraction de la machine sous-jacente. Beaucoup de langages « impératifs » (tels que Fortran, BASIC, et C) sont des abstractions du langage assembleur. Ces langages sont de nettes améliorations par rapport à l'assembleur, mais leur abstraction première requiert que vous réfléchissiez en termes de structure de l'ordinateur plutôt qu'à la structure du problème que vous essayez de résoudre. Le programmeur doit établir l'association entre le modèle de la machine (dans « l'espace solution », qui est le lieu où vous modélisez le problème, tel que l'ordinateur) et le modèle du problème à résoudre (dans « l'espace problème », qui est l'endroit où se trouve le problème). Les efforts requis pour réaliser cette association, et le fait qu'elle est étrangère au langage de programmation, produit des programmes difficiles à écrire et à entretenir, ce qui a mené à la création de l'industrie du « Génie Logiciel».

L'alternative à la modélisation de la machine est de modéliser le problème que vous tentez de résoudre. Les premiers langages tels que LISP ou APL choisirent une vue particulière du monde (« Tous les problèmes se ramènent à des listes » ou « Tous les problèmes sont algorithmiques »). PROLOG convertit tous les problèmes en chaînes de décision. Des langages ont été créés pour la programmation par contrainte, ou pour la programmation ne manipulant que des symboles graphiques (ces derniers se sont révélés être trop restrictifs). Chacune de ces approches est une bonne solution pour la classe particulière de problèmes qu'ils ont a résoudre, mais devient plus délicate dès lors que vous les sortez de leur domaine.

L'approche orientée objet va un pas plus loin en fournissant des outils au programmeur pour représenter des éléments dans l'espace problème. Cette représentation est assez générale pour que le programmeur ne soit

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

restreint à aucun type particulier de problème. Nous nous référons aux éléments dans l'espace problème et leur représentation dans l'espace solution en tant qu'« objets ». (Bien sûr, vous aurez aussi besoin d'autres objets qui n'ont pas leur analogue dans l'espace problème). L'idée est qu'on permet au programme de s'adapter au fond du problème en ajoutant de nouveaux types d'objets , de façon à ce que, quand vous lisez le code décrivant la solution, vous lisez aussi quelque chose qui décrit le problème. C'est un langage d'abstraction plus flexible et puissant que tout ce que nous avons eu jusqu'à présent. Ainsi, la POO vous permet de décrire le problème selon les termes du problème plutôt que selon les termes de la machine sur laquelle la solution sera exécutée. Cependant, il y a toujours une connexion à l'ordinateur. Chaque objet ressemble à un mini-ordinateur ; il a un état, et il a des opérations que vous pouvez lui demander d'exécuter. Cependant, cela ne semble pas être une si mauvaise analogie avec les objets du monde réel - ils ont tous des caractéristiques et des comportements.

Des concepteurs de langage ont décrété que la programmation orientée objet en elle-même n'était pas adéquate pour résoudre facilement tous les problèmes de programmation, et recommandent la combinaison d'approches variées dans des langages de programmation *multiparadigmes*.

Alan Kay a résumé les cinq caractéristiques de base de Smalltalk, le premier véritable langage de programmation orienté objet et l'un des langages sur lequel est basé C++. Ces caractéristiques représentent une approche pure de la programmation orientée objet :

1. **Toute chose est un objet**. Pensez à un objet comme à une variable améliorée : il stocke des données, mais vous pouvez « effectuer des requêtes » sur cet objet, lui demander de faire des opérations sur lui-même. En théorie, vous pouvez prendre n'importe quel composant conceptuel du problème que vous essayez de résoudre (un chien, un immeuble, un service administratif, etc...) et le représenter en tant qu'objet dans le programme.
2. **Un programme est un groupe d'objets s'indiquant quoi faire en envoyant des messages**. Pour qu'un objet effectue une requête, vous « envoyez un message » à cet objet. Plus concrètement, vous pouvez penser à un message comme à un appel de fonction appartenant à un objet particulier.
3. **Chaque objet a sa propre mémoire composée d'autres objets**. Autrement dit, vous créez un nouveau type d'objet en créant un paquetage contenant des objets déjà existants. Ainsi, vous pouvez créer un programme dont la complexité est cachée derrière la simplicité des objets.
4. **Chaque objet a un type**. Dans le jargon, chaque objet est une *instance*d'une *classe*, où « classe » est synonyme de « type ». La caractéristique distinctive la plus importante d'une classe est : « Quels messages pouvez-vous lui envoyer ? ».
5. **Tous les objets d'un type particulier peuvent recevoir les mêmes messages**. C'est une caractéristique lourde de signification, comme vous le verrez plus tard. Parce qu'un objet de type « cercle » est également un objet de type « forme », un cercle garanti d'accepter les messages de forme. Cela signifie que vous pouvez écrire du code qui parle aux formes et qui sera automatiquement accepté par tout ce qui correspond à la description d'une forme. Cette *substituabilité*est l'un des concepts les plus puissants de la POO.

1.2 - Un objet dispose d'une interface

Aristote fut probablement le premier à commencer une étude approfondie du concept de *type*; il parle de « la classe des poissons et la classe des oiseaux ». L'idée que tous les objets, tout en étant uniques, appartiennent à une classe d'objets qui ont des caractéristiques et des comportements en commun fut utilisée directement dans le premier langage orienté objet, Simula-67, avec son mot clef fondamental **class**qui introduit un nouveau type dans un programme.

Simula, comme son nom l'indique, a été conçu pour développer des simulations telles que "le problème du guichet de banque" Vous trouverez une implémentation intéressante de ce problème dans le Volume 2 de ce livre, disponible sur www.BruceEckel.com.Dans celle-ci, vous avez un ensemble de guichetiers, de clients, de comptes, de transactions et de devises - un tas « d'objets ». Des objets semblables, leur état durant l'exécution du programme mis à part, sont groupés ensemble en tant que « classes d'objets » et c'est de là que vient le mot clef *class*. Créer des types de données abstraits (des classes) est un concept fondamental dans la programmation

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

orientée objet. On utilise les types de données abstraits de manière quasi identique aux types de données prédéfinis. On peut créer des variables d'un type particulier (appelés *objets*ou *instances*dans le jargon orienté objet) et manipuler ces variables (ce qu'on appelle *envoyer des messages*ou des *requêtes*; on envoie un message et l'objet se débrouille pour le traiter). Les membres (éléments) d'une même classe partagent des caractéristiques communes : chaque compte dispose d'un solde, chaque guichetier peut accepter un dépôt, etc. Cependant, chaque élément a son propre état, chaque compte a un solde différent, chaque guichetier a un nom. Ainsi, les guichetiers, clients, comptes, transactions, etc. peuvent tous être représentés par une unique entité au sein du programme. Cette entité est l'objet, et chaque objet appartient à une classe particulière qui définit ses caractéristiques et ses comportements.

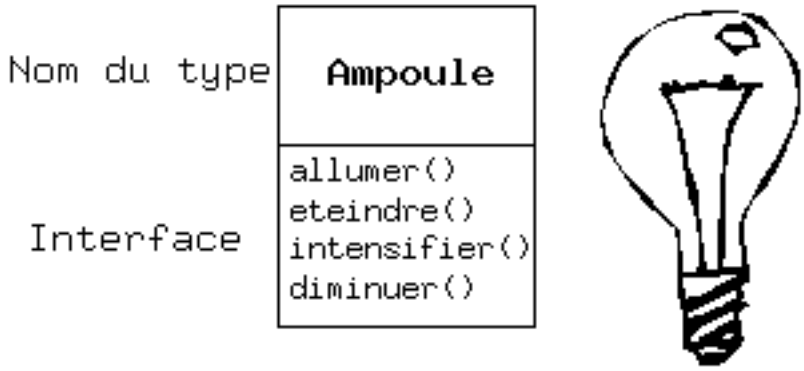
Donc, comme la programmation orientée objet consiste en la création de nouveaux types de données, quasiment tous les langages orientés objet utilisent le mot clef « class ». Quand vous voyez le mot « type » pensez « classe » et inversement Certains font une distinction, au motif que le type détermine l'interface tandis que la classe est une implémentation particulière de cette interface.

Comme une classe décrit un ensemble d'objets partageant des caractéristiques (données) et des comportements (fonctionnalités) communs, une classe est réellement un type de données. En effet, un nombre en virgule flottante par exemple, dispose d'un ensemble de caractéristiques et de comportements. La différence est qu'un programmeur définit une classe pour représenter un problème au lieu d'être forcé d'utiliser un type de données conçu pour représenter une unité de stockage de l'ordinateur. Le langage de programmation est étendu en ajoutant de nouveaux types de données spécifiques à nos besoins. Le système de programmation accepte la nouvelle classe et lui donne toute l'attention et le contrôle de type qu'il fournit aux types prédéfinis.

L'approche orientée objet n'est pas limitée aux simulations. Que vous pensiez ou non que tout programme n'est qu'une simulation du système qu'on représente, l'utilisation des techniques de la POO peut facilement réduire un ensemble de problèmes à une solution simple.

Une fois qu'une classe est créée, on peut créer autant d'objets de cette classe qu'on veut et les manipuler comme s'ils étaient les éléments du problème qu'on tente de résoudre. En fait, l'une des difficultés de la programmation orientée objet est de créer une bijection entre les éléments de l'espace problème et les éléments de l'espace solution.

Mais comment utiliser un objet? Il faut pouvoir lui demander d'exécuter une requête, telle que terminer une transaction, dessiner quelque chose à l'écran, ou allumer un interrupteur. Et chaque objet ne peut traiter que certaines requêtes. Les requêtes qu'un objet est capable de traiter sont définies par son *interface*, et son type est ce qui détermine son interface. Prenons l'exemple d'une ampoule électrique :




```
Ampoule amp;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
amp.allumer();
```

L'interface précise *quelles*opérations on peut effectuer sur un objet particulier. Cependant, il doit exister du code quelque part pour satisfaire cette requête. Ceci, avec les données cachées, constitue l' *implémentation*. Du point de vue de la programmation procédurale, ce n'est pas si compliqué. Un type dispose d'une fonction associée à chaque requête possible, et quand on effectue une requête particulière sur un objet, cette fonction est appelée. Ce mécanisme est souvent résumé en disant qu'on « envoie un message » (fait une requête) à un objet, et l'objet se débrouille pour l'interpréter (il exécute le code associé).

Ici, le nom du type / de la classe est **Ampoule**, le nom de l'objet **Ampoule**créé est **amp**, et on peut demander à un objet **Ampoule**de s'allumer, de s'éteindre, d'intensifier ou de diminuer sa luminosité. Un objet **Ampoule**est créé en déclarant un nom (**amp**) pour cet objet. Pour envoyer un message à cet objet, il suffit de spécifier le nom de l'objet suivi de la requête avec un point entre les deux. Du point de vue de l'utilisateur d'une classe prédéfinie, c'est à peu prés tout ce qu'il faut savoir pour programmer avec des objets.

L'illustration ci-dessus reprend le formalisme *Unified Modeling Language*(UML). Chaque classe est représentée par une boîte, avec le nom du type dans la partie supérieure, les données membres qu'on décide de décrire dans la partie du milieu et les *fonctions membres*(les fonctions appartenant à cet objet qui reçoivent les messages envoyés à cet objet) dans la partie du bas de la boîte. Souvent on ne montre dans les diagrammes UML que le nom de la classe et les fonctions publiques, et la partie du milieu n'existe donc pas. Si seul le nom de la classe nous intéresse, alors la portion du bas n'a pas besoin d'être montrée non plus.

1.3 - L'implémentation cachée

Il est utile de diviser le terrain de jeu en créateurs de classe (ceux qui créent les nouveaux types de données) et *programmeurs clients* Je suis redevable de mon ami Scott Meyers pour cette formulation.(ceux qui utilisent ces types de données dans leurs applications). Le but des programmeurs clients est de se monter une boîte à outils pleine de classes réutilisables pour le développement rapide d'applications (RAD, Rapid Application Development en anglais). Les créateurs de classes, eux, se focalisent sur la construction d'une classe qui n'expose que le nécessaire aux programmeurs clients et cache tout le reste. Pourquoi cela ? Parce que si c'est caché, le programmeur client ne peut l'utiliser, et le créateur de la classe peut changer la portion cachée comme il l'entend sans se préoccuper de l'impact que cela pourrait avoir chez les utilisateurs de sa classe. La portion cachée correspond en général aux données de l'objet qui pourraient facilement être corrompues par un programmeur client négligent ou mal informé. Ainsi, cacher l'implémentation réduit considérablement les bugs.

Le concept d'implémentation cachée ne saurait être trop loué : dans chaque relation il est important de fixer des frontières respectées par toutes les parties concernées. Quand on crée une bibliothèque, on établit une relation avec un programmeur client, programmeur qui crée une application (ou une bibliothèque plus conséquente) en utilisant notre bibliothèque.

Si tous les membres d'une classe sont accessibles pour tout le monde, alors le programmeur client peut faire ce qu'il veut avec cette classe et il n'y a aucun moyen de faire respecter certaines règles. Même s'il est vraiment préférable que l'utilisateur de la classe ne manipule pas directement certains membres de la classe, sans contrôle d'accès il n'y a aucun moyen de l'empêcher : tout est exposé à tout le monde.

La raison première du contrôle d'accès est donc d'empêcher les programmeurs clients de toucher à certaines portions auxquelles ils ne devraient pas avoir accès - les parties qui sont nécessaires pour les manipulations internes du type de données mais n'appartiennent pas à l'interface dont les utilisateurs ont besoin pour résoudre leur problème. C'est en réalité un service rendu aux utilisateurs car ils peuvent voir facilement ce qui est important pour leurs besoins et ce qu'ils peuvent ignorer.

La seconde raison d'être du contrôle d'accès est de permettre au concepteur de la bibliothèque de changer le

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

fonctionnement interne de la classe sans se soucier des effets que cela peut avoir sur les programmeurs clients. Par exemple, on peut implémenter une classe particulière d'une manière simpliste afin d'accélérer le développement, et se rendre compte plus tard qu'on a besoin de la réécrire afin de gagner en performances. Si l'interface et l'implémentation sont clairement séparées et protégées, cela peut être réalisé facilement et nécessite simplement une réédition des liens par l'utilisateur.

Le C++ utilise trois mots clefs pour fixer des limites au sein d'une classe : **public**, **private**et **protected**. Leur signification et leur utilisation est relativement explicite. Ces *spécificateurs d'accès*déterminent qui peut utiliser les définitions qui suivent. **public**veut dire que les définitions suivantes sont disponibles pour tout le monde. Le mot clef **private**, au contraire, veut dire que personne, le créateur de la classe et les fonctions internes de ce type mis à part, ne peut accéder à ces définitions. **private**est un mur de briques entre le créateur de la classe et le programmeur client. Si quelqu'un tente d'accéder à un membre défini comme **private**, ils récupèreront une erreur lors de la compilation. **protected**se comporte tout comme **private**, en moins restrictif : une classe dérivée a accès aux membres **protected**, mais pas aux membres **private**. L'héritage sera introduit bientôt.

1.4 - Réutilisation de l'implémentation

Une fois qu'une classe a été créée et testée, elle devrait (idéalement) représenter une partie de code utile. Il s'avère que cette réutilisabilité n'est pas aussi facile à obtenir que cela ; cela demande de l'expérience et de la perspicacité pour produire une bonne conception. Mais une fois bien conçue, cette classe ne demande qu'à être réutilisée. La réutilisation de code est l'un des plus grands avantages que les langages orientés objets fournissent.

La manière la plus simple de réutiliser une classe est d'utiliser directement un objet de cette classe, mais on peut aussi placer un objet de cette classe à l'intérieur d'une nouvelle classe. On appelle cela « créer un objet membre ». La nouvelle classe peut être constituée de n'importe quel nombre d'objets d'autres types, selon la combinaison nécessaire pour que la nouvelle classe puisse réaliser ce pour quoi elle a été conçue. Parce que la nouvelle classe est composée à partir de classes existantes, ce concept est appelé *composition*(ou, plus généralement, *agrégation*). On se réfère souvent à la composition comme à une relation « possède-un », comme dans « une voiture possède un moteur ».

(Le diagramme UML ci-dessus indique la composition avec le losange rempli, qui indique qu'il y a un moteur dans une voiture. J'utiliserai une forme plus simple : juste une ligne, sans le losange, pour indiquer une association. C'est un niveau de détails généralement suffisant pour la plupart des diagrammes, et l'on n'a pas besoin de se soucier si l'on utilise l'agrégation ou la composition.)

La composition s'accompagne d'une grande flexibilité : les objets membres de la nouvelle classe sont généralement privés, ce qui les rend inaccessibles aux programmeurs clients de la classe. Cela permet de modifier ces membres sans perturber le code des clients existants. On peut aussi changer les objets membres lors la phase d'exécution, pour changer dynamiquement le comportement du programme. L'héritage, décrit juste après, ne dispose pas de cette flexibilité car le compilateur doit placer des restrictions lors de la compilation sur les classes créées avec héritage.

Parce que la notion d'héritage est très importante au sein de la programmation orientée objet, elle est trop souvent accentuée, et le programmeur novice pourrait croire que l'héritage doit être utilisé partout. Cela mène à des conceptions ultra compliquées et cauchemardesques. La composition est la première approche à examiner lorsqu'on crée une nouvelle classe, car elle est plus simple et plus flexible. Le design de la classe en sera plus

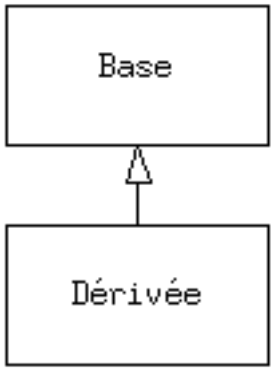
Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

propre. Avec de l'expérience, les endroits où utiliser l'héritage deviendront raisonnablement évidents.

1.5 - Héritage : réutilisation de l'interface

L'idée d'objet en elle-même est un outil efficace. Elle permet de fournir des données et des fonctionnalités liées entre elles par *concept*, afin de représenter une idée de l'espace problème plutôt que d'être forcé d'utiliser les idiomes internes de la machine. Ces concepts sont exprimés en tant qu'unité fondamentale dans le langage de programmation en utilisant le mot clef *class*.

Il serait toutefois dommage, après s'être donné beaucoup de mal pour créer une classe, de devoir en créer une toute nouvelle qui aurait des fonctionnalités similaires. Ce serait mieux si on pouvait prendre la classe existante, la cloner, et faire des ajouts ou des modifications à ce clone. C'est ce que l' *héritage*permet de faire, avec la restriction suivante : si la classe originale (aussi appelée classe de *base*, *superclasse*ou classe *parent*) est changée, le « clone » modifié (appelé classe *dérivée*, *héritée*, *enfant*ou *sous-classe*) répercutera aussi ces changements.



(La flèche dans le diagramme UML ci-dessus pointe de la classe dérivée vers la classe de base. Comme vous le verrez, il peut y avoir plus d'une classe dérivée.)

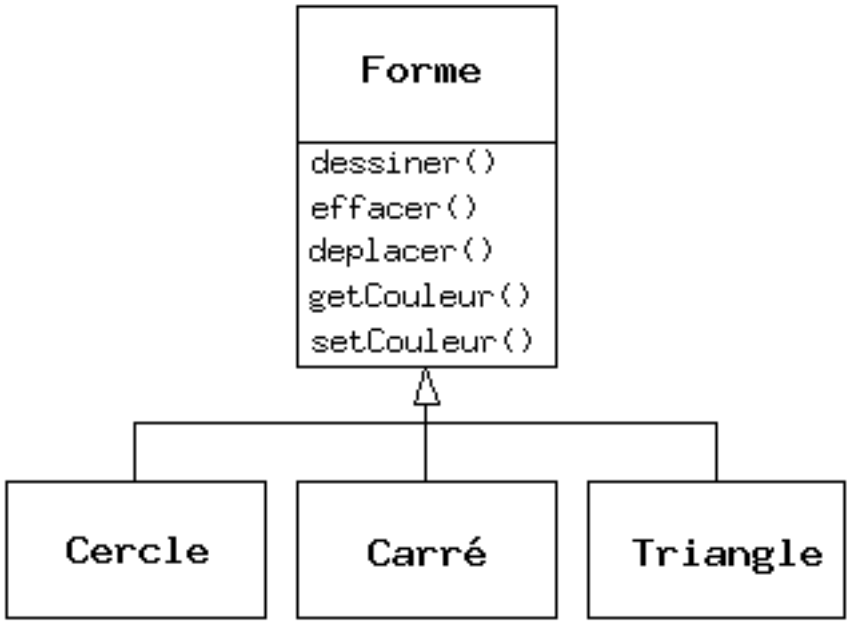
Un type fait plus que décrire des contraintes sur un ensemble d'objets ; il a aussi des relations avec d'autres types. Deux types peuvent avoir des caractéristiques et des comportements en commun, mais l'un des deux peut avoir plus de caractéristiques que l'autre et peut aussi réagir à plus de messages (ou y réagir de manière différente). L'héritage exprime cette similarité entre les types en introduisant le concept de types de base et de types dérivés. Un type de base contient toutes les caractéristiques et comportements partagés entre les types dérivés. Un type de base est créé pour représenter le coeur de certains objets du système. De ce type de base, on dérive d'autres types pour exprimer les différentes manières existantes pour réaliser ce coeur.

Prenons l'exemple d'une machine de recyclage qui trie les détritus. Le type de base serait « détritus », caractérisé par un poids, une valeur, etc. et peut être concassé, fondu, ou décomposé. A partir de ce type de base, sont dérivés des types de détritus plus spécifiques qui peuvent avoir des caractéristiques supplémentaires (une bouteille a une couleur) ou des actions additionnelles (une canette peut être découpée, un container d'acier est magnétique). De plus, des comportements peuvent être différents (la valeur du papier dépend de son type et de son état général). En utilisant l'héritage, on peut bâtir une hiérarchie qui exprime le problème avec ses propres termes.

Un autre exemple classique : les « formes géométriques », utilisées entre autres dans les systèmes d'aide à la conception ou dans les jeux vidéo. Le type de base est la « forme géométrique », et chaque forme a une taille, une couleur, une position, etc. Chaque forme peut être dessinée, effacée, déplacée, peinte, etc. A partir de ce type de base, des types spécifiques sont dérivés (hérités) : des cercles, des carrés, des triangles et autres, chacun avec

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

des caractéristiques et des comportements supplémentaires (certaines figures peuvent être inversées par exemple). Certains comportements peuvent être différents, par exemple quand on veut calculer l'aire de la forme. La hiérarchie des types révèle à la fois les similarités et les différences entre les formes.



Représenter la solution avec les mêmes termes que ceux du problème est extraordinairement bénéfique car on n'a pas besoin de modèles intermédiaires pour passer de la description du problème à la description de la solution. Avec les objets, la hiérarchie de types est le modèle primaire, on passe donc du système dans le monde réel directement au système du code. En fait, l'une des difficultés à laquelle les gens se trouvent confrontés lors de la conception orientée objet est que c'est trop simple de passer du début à la fin. Les esprits habitués à des solutions compliquées sont toujours stupéfaits par cette simplicité.

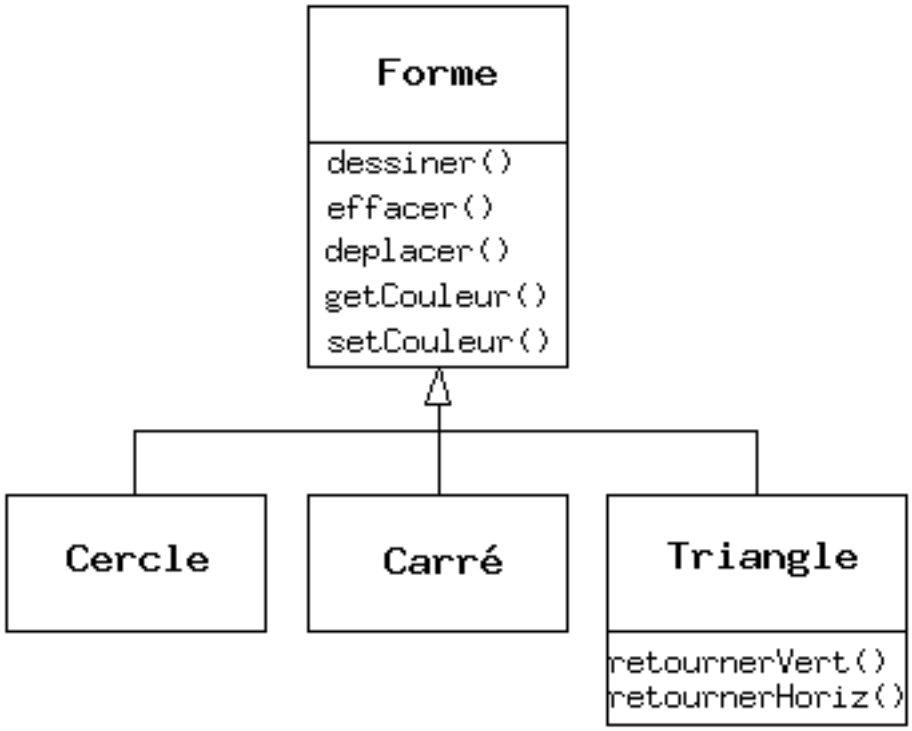
Quand on hérite d'un certain type, on crée un nouveau type. Ce nouveau type non seulement contient tous les membres du type existant (bien que les membres *private*soient cachés et inaccessibles), mais plus important, il duplique aussi l'interface de la classe de la base. Autrement dit, tous les messages acceptés par les objets de la classe de base seront acceptés par les objets de la classe dérivée. Comme on connaît le type de la classe par les messages qu'on peut lui envoyer, cela veut dire que la classe dérivée *est du même type que la classe de base*. Dans l'exemple précédent, « un cercle est une forme ». Cette équivalence de type via l'héritage est l'une des notions fondamentales dans la compréhension de la programmation orientée objet.

Comme la classe de base et la classe dérivée ont toutes les deux la même interface, certaines implémentations accompagnent cette interface. C'est à dire qu'il doit y avoir du code à exécuter quand un objet reçoit un message particulier. Si on ne fait qu'hériter une classe sans rien lui rajouter, les méthodes de l'interface de la classe de base sont importées dans la classe dérivée. Cela veut dire que les objets de la classe dérivée n'ont pas seulement le même type, ils ont aussi le même comportement, ce qui n'est pas particulièrement intéressant.

Il y a deux façons de différencier la nouvelle classe dérivée de la classe de base originale. La première est relativement directe : il suffit d'ajouter de nouvelles fonctions à la classe dérivée. Ces nouvelles fonctions ne font pas partie de la classe parent. Cela veut dire que la classe de base n'était pas assez complète pour ce qu'on voulait en faire, on a donc ajouté de nouvelles fonctions. Cet usage simple de l'héritage se révèle souvent être une solution idéale. Cependant, il faut tout de même vérifier s'il ne serait pas souhaitable d'intégrer ces fonctions dans la classe de base qui pourrait aussi en avoir l'usage. Ce processus de découverte et d'itération dans la conception

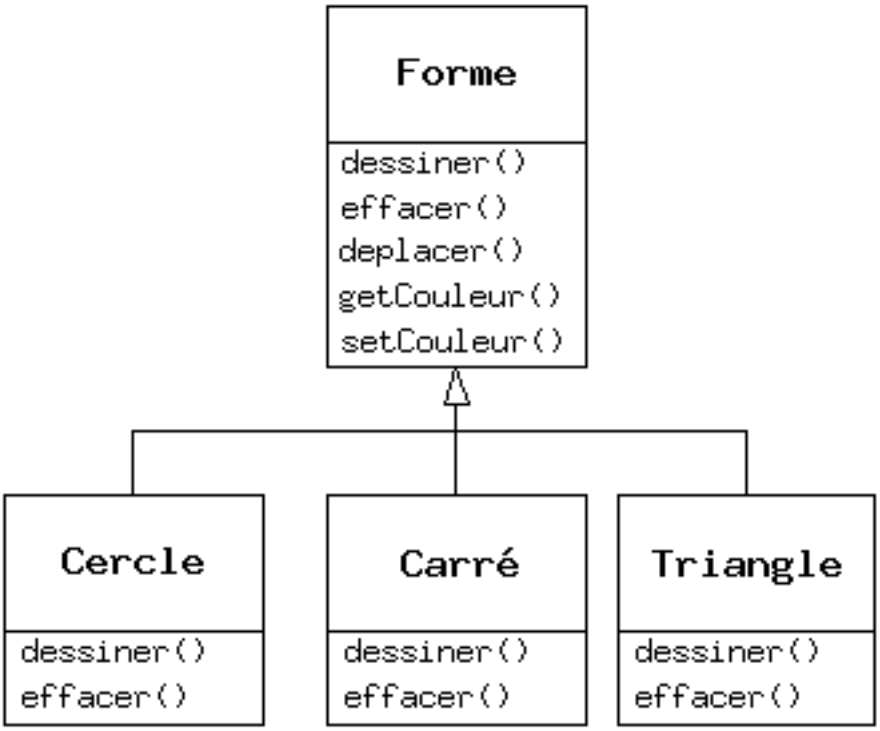
Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

est fréquent dans la programmation orientée objet.



Bien que l'héritage puisse parfois impliquer (spécialement en Java, où le mot clef qui indique l'héritage est *extends*) que de nouvelles fonctions vont être ajoutées à l'interface, ce n'est pas toujours vrai. La seconde et plus importante manière de différencier la nouvelle classe est de *changer*le comportement d'une des fonctions existantes de la classe de base. Cela s'appelle *redéfinir*cette fonction.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/



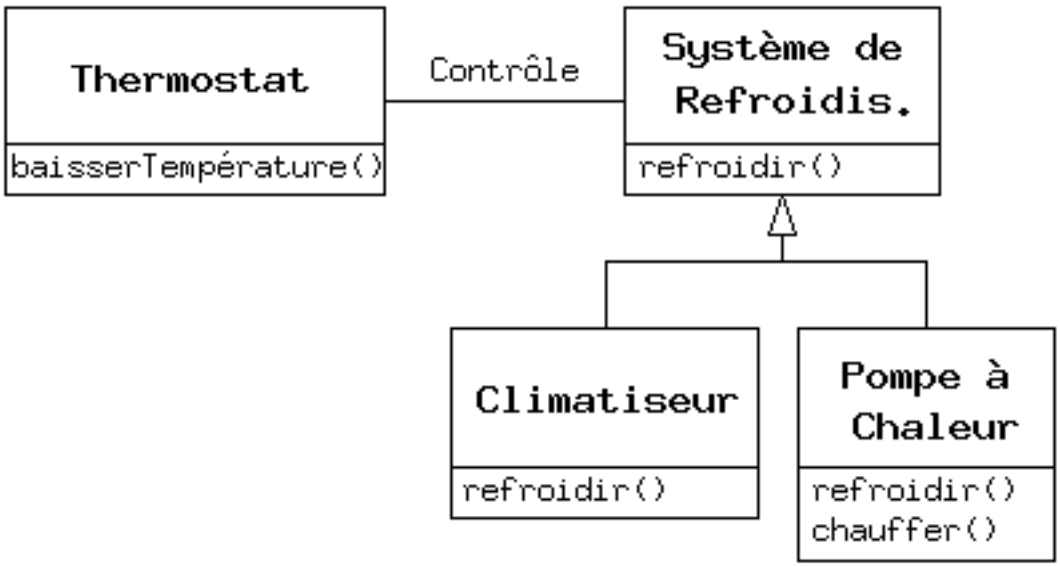
Pour redéfinir une fonction, il suffit de créer une nouvelle définition pour la fonction dans la classe dérivée. C'est comme dire : « j'utilise la même interface ici, mais je la traite d'une manière différente dans ce nouveau type ».

1.5.1 - Les relations est-un vs. est-comme-un

Un certain débat est récurrent à propos de l'héritage : l'héritage ne devrait-il pas *seulement*redéfinir les fonctions de la classe de base (et ne pas ajouter de nouvelles fonctions membres qui ne font pas partie de la superclasse) ? Cela voudrait dire que le type dérivé serait *exactement*le même que celui de la classe de base puisqu'il aurait exactement la même interface. Avec comme conséquence logique le fait qu'on puisse exactement substituer un objet de la classe dérivée à un objet de la classe de base. On fait souvent référence à cette *substitution pure*sous le nom de *principe de substitution*. Dans un sens, c'est la manière idéale de traiter l'héritage. La relation entre la classe de base et la classe dérivée dans ce cas est une relation *est-un*, parce qu'on peut dire « un cercle *est une*forme ». Un test pour l'héritage est de déterminer si la relation est-un entre les deux classes considérées a un sens.

Mais parfois il est nécessaire d'ajouter de nouveaux éléments à l'interface d'un type dérivé, et donc étendre l'interface et créer un nouveau type. Le nouveau type peut toujours être substitué au type de base, mais la substitution n'est plus parfaite parce que les nouvelles fonctions ne sont pas accessibles à partir de la classe parent. On appelle cette relation une relation *est-comme-un*- le nouveau type dispose de l'interface de l'ancien type mais il contient aussi d'autres fonctions, on ne peut donc pas réellement dire qu'ils soient exactement identiques. Prenons le cas d'un système de climatisation. Supposons que notre maison dispose des tuyaux et des systèmes de contrôle pour le refroidissement, autrement dit, elle dispose d'une interface qui nous permet de contrôler le refroidissement. Imaginons que le système de climatisation tombe en panne et qu'on le remplace par une pompe à chaleur, qui peut à la fois chauffer et refroidir. La pompe à chaleur *est-comme-un*système de climatisation, mais il peut faire plus de choses. Parce que le système de contrôle n'a été conçu que pour contrôler le refroidissement, il en est restreint à ne communiquer qu'avec la partie refroidissement du nouvel objet. L'interface du nouvel objet a été étendue, mais le système existant ne connaît rien qui ne soit dans l'interface originale.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/



Bien sûr, quand on voit cette modélisation, il est clair que la classe de base « Système de refroidissement » n'est pas assez générale, et devrait être renommée en « Système de contrôle de température » afin de pouvoir inclure le chauffage - auquel cas le principe de substitution marcherait. Cependant, le diagramme ci-dessus est un exemple de ce qui peut arriver dans le monde réel.

Quand on considère le principe de substitution, il est tentant de se dire que cette approche (la substitution pure) est la seule manière correcte de modéliser, et de fait *c'est*appréciable si la conception fonctionne ainsi. Mais dans certains cas il est tout aussi clair qu'il faut ajouter de nouvelles fonctions à l'interface d'une classe dérivée. En examinant le problème, les deux cas deviennent relativement évidents.

1.6 - Polymorphisme : des objets interchangeables

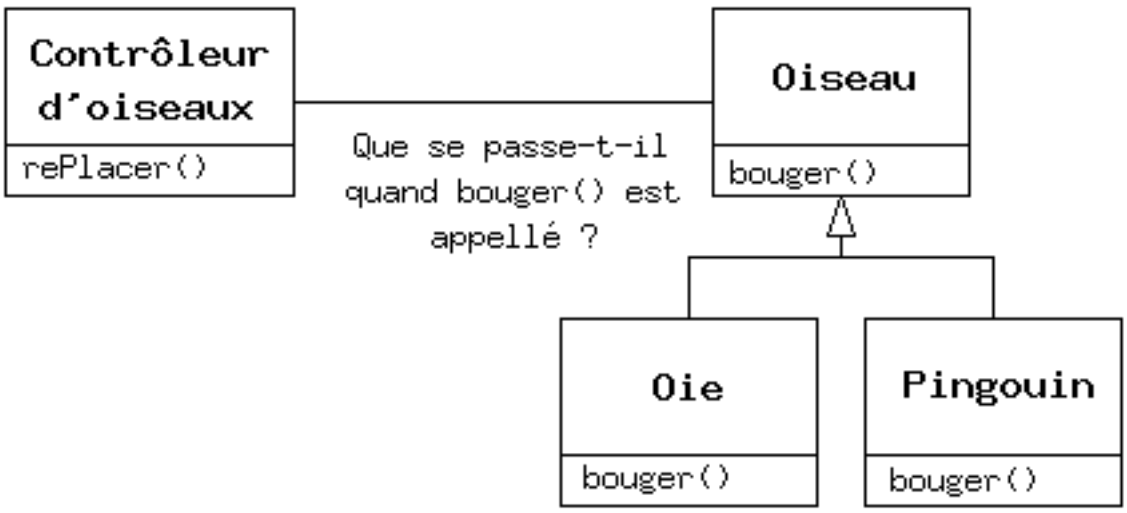
Lorsqu'on manipule des hiérarchies de types, il arrive souvent qu'on veuille traiter un objet en n'utilisant pas son type spécifique, mais en tant qu'objet de son type de base. Cela permet d'écrire du code indépendant des types spécifiques. Dans l'exemple de la forme géométrique, les fonctions manipulent des formes génériques sans se soucier de savoir si ce sont des cercles, des carrés, des triangles, et ainsi de suite. Toutes les formes peuvent être dessinées, effacées, et déplacées, donc ces fonctions envoient simplement un message à un objet forme, elles ne se soucient pas de la manière dont l'objet traite le message.

Un tel code n'est pas affecté par l'addition de nouveaux types, et ajouter de nouveaux types est la façon la plus commune d'étendre un programme orienté objet pour traiter de nouvelles situations. Par exemple, on peut dériver un nouveau type de forme appelé pentagone sans modifier les fonctions qui traitent des formes génériques. Cette capacité à étendre facilement un programme en dérivant de nouveaux sous-types est importante car elle améliore considérablement la conception tout en réduisant le coût de maintenance.

Un problème se pose cependant en voulant traiter les types dérivés comme leur type de base générique (les cercles comme des formes géométriques, les vélos comme des véhicules, les cormorans comme des oiseaux, etc.). Si une fonction demande à une forme générique de se dessiner, ou à un véhicule générique de tourner, ou à un oiseau générique de se déplacer, le compilateur ne peut savoir précisément, lors de la phase de compilation, quelle portion de code sera exécutée. C'est d'ailleurs le point crucial : quand le message est envoyé, le programmeur ne *veut*pas savoir quelle portion de code sera exécutée ; la fonction dessiner peut être appliquée aussi bien à un cercle qu'à un carré ou un triangle, et l'objet va exécuter le bon code suivant son type spécifique. Si on n'a pas besoin de savoir quelle portion de code est exécutée, alors le code exécuté lorsqu'on ajoute un nouveau

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

sous-type peut être différent sans exiger de modification dans l'appel de la fonction. Le compilateur ne peut donc précisément savoir quelle partie de code sera exécutée, donc que va-t-il faire ? Par exemple, dans le diagramme suivant, l'objet **Contrôleur d'oiseaux**travaille seulement avec des objets **Oiseaux**génériques, et ne sait pas de quel type ils sont. C'est pratique du point de vue de **Contrôleur d'oiseaux**, car il n'a pas besoin d'écrire du code spécifique pour déterminer le type exact d' **Oiseau**avec lequel il travaille, ou le comportement de cet **Oiseau**. Comment se fait-il donc que, lorsque **bouger()**est appelé tout en ignorant le type spécifique de l' **Oiseau**, on obtienne le bon comportement (une **Oie**court, vole ou nage, et un **Pingouin**court ou nage) ?



La réponse constitue l'astuce fondamentale de la programmation orientée objet : le compilateur ne peut faire un appel de fonction au sens traditionnel du terme. Un appel de fonction généré par un compilateur non orienté objet crée ce qu'on appelle une *association prédéfinie*, un terme que vous n'avez sans doute jamais entendu auparavant car vous ne pensiez pas qu'on puisse faire autrement. En d'autres termes, le compilateur génère un appel à un nom de fonction spécifique, et l'éditeur de liens résout cet appel à l'adresse absolue du code à exécuter. En POO, le programme ne peut déterminer l'adresse du code avant la phase d'exécution, un autre mécanisme est donc nécessaire quand un message est envoyé à un objet générique.

Pour résoudre ce problème, les langages orientés objet utilisent le concept d' *association tardive*. Quand un objet reçoit un message, le code appelé n'est pas déterminé avant l'exécution. Le compilateur s'assure que la fonction existe et vérifie le type des arguments et de la valeur de retour (un langage omettant ces vérifications est dit *faiblement typé*), mais il ne sait pas exactement quel est le code à exécuter.

Pour créer une association tardive, le compilateur C++ insère une portion spéciale de code en lieu et place de l'appel absolu. Ce code calcule l'adresse du corps de la fonction, en utilisant des informations stockées dans l'objet (ce mécanisme est couvert plus en détails dans le Chapitre 15). Ainsi, chaque objet peut se comporter différemment suivant le contenu de cette portion spéciale de code. Quand un objet reçoit un message, l'objet sait quoi faire de ce message.

On déclare qu'on veut une fonction qui ait la flexibilité des propriétés de l'association tardive en utilisant le mot-clé **virtual**. On n'a pas besoin de comprendre les mécanismes de **virtual**pour l'utiliser, mais sans lui on ne peut pas faire de la programmation orientée objet en C++. En C++, on doit se souvenir d'ajouter le mot-clé **virtual**parce que, par défaut, les fonctions membre *ne sont pas*liées dynamiquement. Les fonctions virtuelles permettent d'exprimer des différences de comportement entre des classes de la même famille. Ces différences sont ce qui engendre un comportement polymorphe.

Reprenons l'exemple de la forme géométrique. Le diagramme de la hiérarchie des classes (toutes basées sur la même interface) se trouve plus haut dans le chapitre. Pour illustrer le polymorphisme, écrivons un bout de code qui

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

ignore les détails spécifiques du type et parle uniquement à la classe de base. Ce code est *déconnecté*des informations spécifiques au type, donc plus facile à écrire et à comprendre. Et si un nouveau type - un **Hexagone**, par exemple - est ajouté grâce à l'héritage, le code continuera de fonctionner aussi bien pour ce nouveau type de **Forme**qu'il le faisait avec les types existants. Le programme est donc extensible.

Si nous écrivons une fonction en C++ (comme vous allez bientôt apprendre à le faire) :

```
                    void faireQuelqueChose(Forme &f) {
  f.effacer();
  // ...
  f.dessiner();
}
```

Cette fonction s'adresse à n'importe quelle **Forme**, elle est donc indépendante du type spécifique de l'objet qu'elle dessine et efface (le ' **&**' signifie «Prends l'adresse de l'objet qui est passé à **faireQuelqueChose()**» mais ce n'est pas important que vous compreniez les détails de cela pour l'instant). Si nous utilisons cette fonction **faireQuelqueChose()**dans une autre partie du programme :

```
                    Cercle c;
Triangle t;
Ligne l;
faireQuelqueChose(c);
faireQuelqueChose(t);
faireQuelqueChose(l);
```

Les appels à **faireQuelqueChose()**fonctionnent correctement, sans se préoccuper du type exact de l'objet.

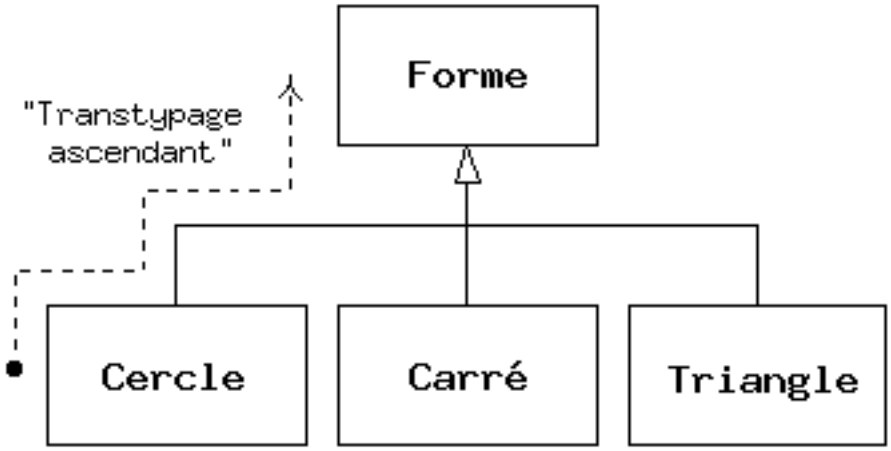
En fait c'est une manière de faire très élégante. Considérons la ligne :

```
                    faireQuelqueChose(c);
```

Un **Cercle**est ici passé à une fonction qui attend une **Forme**. Comme un **Cercle** *est*une **Forme**, il peut être traité comme tel par **faireQuelqueChose()**. C'est à dire qu'un **Cercle**peut accepter tous les messages que **faireQuelqueChose()**pourrait envoyer à une forme. C'est donc une façon parfaitement logique et sûre de procéder.

Traiter un type dérivé comme s'il était son type de base est appelé *transtypage ascendant*, *surtypage*ou *généralisation*(upcasting). L'adjectif ascendant vient du fait que dans un diagramme d'héritage typique, le type de base est représenté en haut, les classes dérivées s'y rattachant par le bas. Ainsi, changer un type vers son type de base revient à remonter dans le diagramme d'héritage : transtypage « ascendant ».

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/



Un programme orienté objet contient obligatoirement des transtypages ascendants, car c'est de cette manière que le type spécifique de l'objet peut être délibérément ignoré. Examinons le code de **faireQuelqueChose()**:

```
                    f.effacer();
// ...
f.dessiner();
```

Remarquez qu'il ne dit pas « Si tu es un **Cercle**, fais ceci, si tu es un **Carré**, fais cela, etc. ». Ce genre de code qui vérifie tous les types possibles que peut prendre une **Forme**est confus et il faut le changer à chaque extension de la classe **Forme**. Ici, il suffit de dire : « Tu es une forme géométrique, je sais que tu peux te **dessiner()**et t' **effacer()**, alors fais-le et occupe-toi des détails spécifiques ».

Ce qui est impressionnant dans le code de **faireQuelqueChose()**, c'est que tout fonctionne comme on le souhaite. Appeler **dessiner()**pour un **Cercle**exécute une portion de code différente de celle exécutée lorsqu'on appelle **dessiner()**pour un **Carré**ou une **Ligne**, mais lorsque le message **dessiner()**est envoyé à une **Forme**anonyme, on obtient le comportement idoine basé sur le type réel de la **Forme**. C'est impressionnant dans la mesure où le compilateur C++ ne sait pas à quel type d'objet il a affaire lors de la compilation du code de **faireQuelqueChose()**. On serait en droit de s'attendre à un appel aux versions **effacer()**et **dessiner()**de **Forme**, et non celles des classes spécifiques **Cercle**, **Carré**et **Ligne**. Mais quand on envoie un message à un objet, il fera ce qu'il a à faire, même quand la généralisation est impliquée. C'est ce qu'implique le polymorphisme. Le compilateur et le système d'exécution s'occupent des détails, et c'est tout ce que vous avez besoin de savoir en plus de savoir comment modéliser avec. Si une fonction membre est **virtual**, alors quand on envoie un message à un objet, l'objet le traite correctement, même quand le transtypage ascendant est mis en jeu.

1.7 - Créer et détruire les objets

Techniquement, le domaine de la POO est celui de l'abstraction du typage des données, de l'héritage et du polymorphisme, mais d'autres questions peuvent être au moins aussi importantes.

La façon dont les objets sont créés et détruits est particulièrement importante. Où sont les données d'un objet et comment la durée de vie de l'objet est-elle gérée ? Différents langages de programmation utiliseront ici différentes philosophies. L'approche du C++ est de privilégier le contrôle de l'efficacité, alors le choix est laissé au programmeur. Pour maximiser la vitesse, le stockage et la durée de vie peuvent être déterminés à l'écriture du programme en plaçant les objets sur la pile ou dans un espace de stockage statique. La pile est une région de la mémoire utilisée directement par le microprocesseur pour stocker les données durant l'exécution du programme. Les variables sur la pile sont souvent qualifiées de variables *automatiques*ou *de portée*. La zone de stockage statique est simplement une zone fixée de la mémoire allouée avant le début de l'exécution du programme.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

L'utilisation de la pile ou des zones de stockage statiques met la priorité sur la vitesse d'allocation et de libération, ce qui est peut être avantageux dans certaines situations. Cependant vous sacrifiez la flexibilité parce que vous êtes obligés de connaître la quantité exacte, la durée de vie et le type des objets au moment où vous écrivez le programme. Si vous tentez de résoudre un problème beaucoup plus général, comme de la conception assistée par ordinateur, de la gestion d'entrepôt ou du contrôle de trafic aérien, c'est trop restrictif.

La seconde approche est de créer des objets dynamiquement dans un emplacement mémoire appelé le *tas*. Dans cette approche vous ne connaissez pas, avant l'exécution, le nombre d'objets dont vous aurez besoin, leur durée de vie ou leur type exact. Ces décisions seront prises en leur temps pendant l'exécution. Si vous avez besoin d'un nouvel objet vous le créez simplement sur le tas lorsque vous en avez besoin en utilisant le mot-clé **new**. Lorsque vous en avez fini avec le stockage, vous devez libérer la mémoire en utilisant le mot-clé **delete**.

Parce que le stockage est géré dynamiquement pendant l'exécution, la durée nécessaire à l'allocation sur le tas est significativement plus longue que celle pour allouer sur la pile. (Créer sur la pile consite souvent en une seule instruction du microprocesseur pour abaisser le pointeur de la pile, et une autre pour l'élever à nouveau.) L'approche dynamique donne généralement l'impression que les objets tendent à se complexifier, or les frais supplémentaires pour le stockage et la libération n'ont pas un impact important sur la création d'un objet. En plus, la plus grande flexibilité offerte est essentielle à la résolution de problèmes généraux de programmation.

Il y a une autre question, cependant, c'est la durée de vie d'un objet. Si vous créez un objet sur la pile ou dans un espace de stockage statique, le compilateur détermine la durée de l'objet et peu automatiquement le détruire. Cependant, si vous le créez sur le tas, le compilateur ne connaît pas sa durée de vie. En C++, le programmeur est obligé de déterminer par programme le moment où l'objet est détruit, et effectue cette destruction en utilisant le mot-clé **delete**. Comme une alternative, l'environnement peut offrir un dispositif appelé *garbage collector*(ramasse-miettes) qui détermine automatiquement quand un objet n'est plus utilisé et le détruit. Bien sûr, écrire un programme utilisant un *garbage collector*est plus pratique, mais cela requiert que toutes les applications tolèrent l'existence de ce collecteur et les frais inhérents à la collecte des déchets. Ceci ne satisfait pas les conditions de conception du langage C++ et ainsi il n'en est pas fait mention, mais les collecteurs tiers existent en C++.

1.8 - Traitement des exceptions : gérer les erreurs

Depuis les débuts des langages de programmation, le traitement des erreurs s'est révélé l'un des problèmes les plus ardus. Parce qu'il est difficile de concevoir un bon mécanisme de gestion des erreurs, beaucoup de langages ignorent ce problème et le délèguent aux concepteurs de bibliothèques qui fournissent des mécanismes de demi-mesure qui fonctionnent dans beaucoup de situations mais peuvent être facilement contournés, généralement en les ignorant. L'une des faiblesses de la plupart des mécanismes d'erreur est qu'ils reposent sur la vigilance du programmeur à suivre des conventions non imposées par le langage. Si les programmeurs ne sont pas assez vigilants, ce qui est souvent le cas s'ils sont pressés, ces mécanismes peuvent facilement être oubliés.

La *gestion des exceptions*intègre la gestion des erreurs directement au niveau du langage de programmation et parfois même au niveau du système d'exploitation. Une exception est un objet qui est « émis » depuis l'endroit où l'erreur est apparue et peut être intercepté par un *gestionnaire d'exception*conçu pour gérer ce type particulier d'erreur. C'est comme si la gestion des exceptions était un chemin d'exécution parallèle à suivre quand les choses se gâtent. Et parce qu'elle utilise un chemin d'exécution séparé, elle n'interfère pas avec le code s'exécutant normalement. Cela rend le code plus simple à écrire car on n'a pas à vérifier constamment si des erreurs sont survenues. De plus, une exception émise n'est pas comme une valeur de retour d'une fonction signalant une erreur ou un drapeau positionné par une fonction pour indiquer une erreur - ils peuvent être ignorés. Une exception ne peut pas être ignorée, on a donc l'assurance qu'elle sera traitée quelque part. Enfin, les exceptions permettent de revenir d'une mauvaise situation assez facilement. Plutôt que de terminer un programme, il est souvent possible de remettre les choses en place et de restaurer son exécution, ce qui produit des systèmes plus robustes.

Il est bon de noter que le traitement des exceptions n'est pas une caractéristique orientée objet, bien que dans les

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

langages OO une exception soit normalement représentée par un objet. Le traitement des exceptions existait avant les langages orientés objet.

La gestion des exceptions est simplement introduite et utilisée de manière superficielle dans ce volume - le Volume 2 (disponible sur *www.BruceEckel.com*) traite en profondeur la gestion des exceptions.

1.9 - Analyse et conception

Le paradigme de la POO constitue une approche nouvelle et différente de la programmation et beaucoup de personnes rencontrent des difficultés pour appréhender leur premier projet orienté objet. Une fois compris que tout est supposé être un objet, et au fur et à mesure qu'on se met à penser dans un style plus orienté objet, on commence à créer de « bonnes » conceptions qui s'appuient sur tous les avantages que la POO offre.

Une *méthode*(ou *méthodologie*) est un ensemble de processus et d'heuristiques utilisés pour réduire la complexité d'un problème. Beaucoup de méthodes orientées objet ont été formulées depuis l'apparition de la POO. Cette section vous donne un aperçu de ce que vous essayez d'accomplir en utilisant une méthode.

Spécialement en POO, une méthodologie s'appuie sur un certain nombre d'expériences, il est donc important de comprendre quel problème la méthode tente de résoudre avant d'en adopter une. Ceci est particulièrement vrai avec le C++, qui a été conçu pour réduire la complexité (comparé au C) dans l'écriture d'un programme. Cette philosophie supprime le besoin de méthodologies toujours plus complexes. Au contraire, des méthodologies plus simples peuvent se révéler tout à fait suffisantes avec le C++ pour une classe de problèmes plus large que ce qu'elles pourraient traiter avec des langages procéduraux.

Il est important de réaliser que le terme « méthodologie » est trompeur et promet trop de choses. Tout ce qui est mis en oeuvre quand on conçoit et réalise un programme est une méthode. Ca peut être une méthode personnelle, et on peut ne pas en être conscient, mais c'est une démarche qu'on suit au fur et à mesure de l'avancement du projet. Si cette méthode est efficace, elle ne nécessitera sans doute que quelques petites adaptations pour fonctionner avec le C++. Si vous n'êtes pas satisfait de votre productivité ou du résultat obtenu, vous serez peut-être tenté d'adopter une méthode plus formelle, ou d'en composer une à partir de plusieurs méthodes formelles.

Au fur et à mesure que le projet avance, le plus important est de ne pas se perdre, ce qui est malheureusement très facile. La plupart des méthodes d'analyse et de conception sont conçues pour résoudre même les problèmes les plus gros. Il faut donc bien être conscient que la plupart des projets ne rentrant pas dans cette catégorie, on peut arriver à une bonne analyse et conception avec juste une petite partie de ce qu'une méthode recommande Un excellent exemple de cela est UML Distilled, de Martin Fowler (Addison-Wesley 2000), qui réduit le processus UML parfois surdimensionné à un sous ensemble utilisable.. Une méthode de conception, même limitée, met sur la voie bien mieux que si on commence à coder directement.

Il est aussi facile de rester coincé et tomber dans la « paralysie analytique » où on se dit qu'on ne peut passer à la phase suivante car on n'a pas traqué le moindre petit détail de la phase courante. Il faut bien se dire que quelle que soit la profondeur de l'analyse, certains aspects d'un problème ne se révéleront qu'en phase de conception, et d'autres en phase de réalisation, voire même pas avant que le programme ne soit achevé et exécuté. A cause de ceci, il est crucial d'avancer relativement rapidement dans l'analyse et la conception, et d'implémenter un test du système proposé.

Il est bon de développer un peu ce point. A cause des déboires rencontrés avec les langages procéduraux, il est louable qu'une équipe veuille avancer avec précautions et comprendre tous les détails avant de passer à la conception et l'implémentation. Il est certain que lors de la création d'une base de données, il est capital de comprendre à fond les besoins du client. Mais la conception d'une base de données fait partie d'une classe de problèmes bien définie et bien comprise ; dans ce genre de programmes, la structure de la base de données *est*le

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

problème à résoudre. Les problèmes traités dans ce chapitre font partie de la classe de problèmes « joker » (invention personnelle), dans laquelle la solution n'est pas une simple reformulation d'une solution déjà éprouvée de nombreuses fois, mais implique un ou plusieurs « facteurs joker » - des éléments pour lesquels il n'existe aucune solution préétablie connue, et qui nécessitent de pousser les recherches. Ma règle d'or pour estimer de tels projets : s'il y a plus d'un joker, ne pas essayer de prédire combien de temps cela va prendre ou combien cela va coûter tant que l'on n'a pas créé un prototype fonctionnel. Il y a trop de degrés de liberté.. Tenter d'analyser à fond un problème joker avant de passer à la conception et l'implémentation mène à la paralysie analytique parce qu'on ne dispose pas d'assez d'informations pour résoudre ce type de problèmes durant la phase d'analyse. Résoudre ce genre de problèmes requiert de répéter le cycle complet, et cela demande de prendre certains risques (ce qui est sensé, car on essaie de faire quelque chose de nouveau et les revenus potentiels en sont plus élevés). On pourrait croire que le risque est augmenté par cette ruée vers une première implémentation, mais elle peut réduire le risque dans un projet joker car on peut tout de suite se rendre compte si telle approche du problème est viable ou non. Le développement d'un produit s'apparente à de la gestion de risque.

Souvent cela se traduit par « construire un prototype qu'il va falloir jeter ». Avec la POO, on peut encore avoir à en jeter *une partie*, mais comme le code est encapsulé dans des classes, on aura inévitablement produit durant la première itération quelques classes qui valent la peine d'être conservées, et développé des idées sur la conception du système. Ainsi, une première passe rapide sur un problème fournit non seulement des informations critiques pour la prochaine itération d'analyse, de conception et d'implémentation, mais elle produit aussi une base du code pour cette itération.

Ceci dit, si on cherche une méthode qui contient de nombreux détails et suggère de nombreuses étapes et documents, il est toujours difficile de savoir où s'arrêter. Il faut garder à l'esprit ce qu'on essaye de découvrir :

1 Quels sont les objets ? (Comment partitionner le projet en ses composants élémentaires ?)
2 Quelles en sont les interfaces ? (Quels sont les messages qu'on a besoin d'envoyer à chaque objet ?)

Si on arrive à trouver quels sont les objets et leur interface, alors on peut commencer à coder. On pourra avoir besoin d'autres descriptions et documents, mais on ne peut pas faire avec moins que ça.

Le développement peut être décomposé en cinq phases, et une phase 0 qui est juste l'engagement initial à respecter une structure de base.

1.9.1 - Phase 0 : Faire un plan

Il faut d'abord décider quelles étapes on va suivre dans le développement. Cela semble simple (en fait, *tout*semble simple) et malgré tout les gens ne prennent cette décision qu'après avoir commencé à coder. Si le plan se résume à « retroussons nos manches et codons », alors ça ne pose pas de problèmes (quelquefois c'est une approche valable quand on a affaire à un problème bien connu). Mais il faut néanmoins accepter que ce soit le plan.

On peut aussi décider dans cette phase qu'une structure additionnelle est nécessaire. Certains programmeurs aiment travailler en « mode vacances » sans structure imposée sur le processus de développement de leur travail : « Ce sera fait lorsque ce sera fait ». Cela peut être séduisant un moment, mais disposer de quelques jalons aide à se concentrer et focalise les efforts sur ces jalons au lieu d'être obnubilé par le but unique de « finir le projet ». De plus, cela divise le projet en parties plus petites, ce qui le rend moins redoutable (sans compter que les jalons offrent des opportunités de fête).

Quand j'ai commencé à étudier la structure des histoires (afin de pouvoir un jour écrire un roman), j'étais réticent au début à l'idée de structure, trouvant que quand j'écrivais, je laissais juste la plume courir sur le papier. Mais j'ai réalisé plus tard que quand j'écris à propos des ordinateurs, la structure est suffisamment claire pour que je n'y réfléchisse pas trop. Mais je structure tout de même mon travail, bien que ce soit inconsciemment dans ma tête. Donc même si on pense que le plan est juste de commencer à coder, on passe tout de même par les phases

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

successives en se posant certaines questions et en y répondant.

L'exposé de la mission

Tout système qu'on construit, quelle que soit sa complexité, a un but, un besoin fondamental qu'il satisfait. Si on peut voir au delà de l'interface utilisateur, des détails spécifiques au matériel - ou au système -, des algorithmes de codage et des problèmes d'efficacité, on arrive finalement au coeur du problème, simple et nu. Comme le soi-disant *concept fondamental*d'un film hollywoodien, on peut le décrire en une ou deux phrases. Cette description pure est le point de départ.

Le concept fondamental est assez important car il donne le ton du projet ; c'est l'exposé de la mission. Ce ne sera pas nécessairement le premier jet qui sera le bon (on peut être dans une phase ultérieure du projet avant qu'il ne soit complètement clair), mais il faut continuer d'essayer jusqu'à ce que ça sonne bien. Par exemple, dans un système de contrôle de trafic aérien, on peut commencer avec un concept fondamental basé sur le système qu'on construit : « Le programme tour de contrôle garde la trace d'un avion ». Mais cela n'est plus valable quand le système se réduit à un petit aérodrome, avec un seul contrôleur ou même aucun. Un modèle plus utile ne décrira pas tant la solution qu'on crée que le problème : « Des avions arrivent, déchargent, partent en révision, rechargent et repartent ».

1.9.2 - Phase 1 : Que construit-on ?

Dans la génération précédente de conception de programmes (*conception procédurale*), cela s'appelait « l' *analyse des besoins*et les *spécifications du système*». C'étaient des endroits où on se perdait facilement, avec des documents au nom intimidant qui pouvaient occulter le projet. Leurs intentions étaient bonnes, pourtant. L'analyse des besoins consiste à « faire une liste des indicateurs qu'on utilisera pour savoir quand le travail sera terminé et le client satisfait ». Les spécifications du système consistent en « une description de ce que le programme fera (sans ce préoccuper du *comment*) pour satisfaire les besoins ». L'analyse des besoins est un contrat entre le développeur et le client (même si le client travaille dans la même entreprise ou se trouve être un objet ou un autre système). Les spécifications du système sont une exploration générale du problème, et en un sens permettent de savoir s'il peut être résolu et en combien de temps. Comme ils requièrent des consensus entre les intervenants sur le projet (et parce qu'ils changent au cours du temps), il vaut mieux les garder aussi bruts que possible - idéalement en tant que listes et diagrammes - pour ne pas perdre de temps. Il peut y avoir d'autres contraintes qui demandent de produire de gros documents, mais en gardant les documents initiaux petits et concis, cela permet de les créer en quelques sessions de brainstorming avec un leader qui affine la description dynamiquement. Cela permet d'impliquer tous les acteurs du projet, et encourage la participation de toute l'équipe. Plus important encore, cela permet de lancer un projet dans l'enthousiasme.

Il est nécessaire de rester concentré sur ce qu'on essaye d'accomplir dans cette phase : déterminer ce que le système est supposé faire. L'outil le plus utile pour cela est une collection de ce qu'on appelle « cas d'utilisation ». Les cas d'utilisation identifient les caractéristiques clefs du système qui vont révéler certaines des classes fondamentales qu'on utilisera. Ce sont essentiellement des réponses descriptives à des questions comme Merci à James H Jarrett pour son aide.:

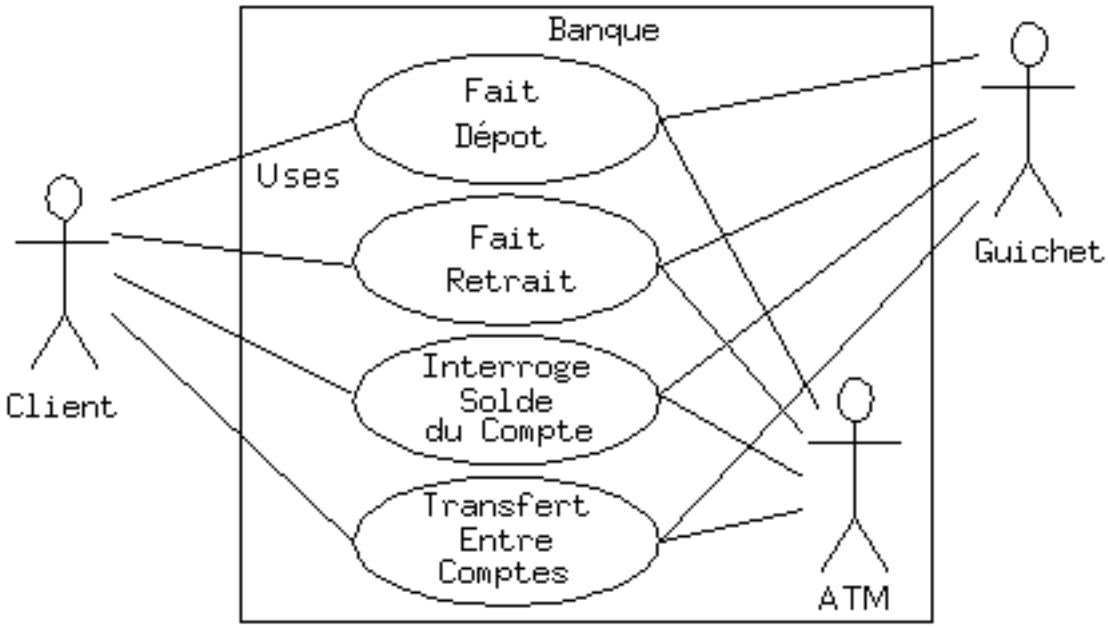
- « Qui utilisera le système ? »
- « Que peuvent faire ces personnes avec le système ? »
- « Comment tel acteur fait-il cela avec le système ? »
- « Comment cela pourrait-il fonctionner si quelqu'un d'autre faisait cela, ou si le même acteur avait un objectif différent ? » (pour trouver les variations)
- « Quels problèmes peuvent apparaître quand on fait cela avec le système ? » (pour trouver les exceptions)

Si on conçoit un guichet automatique, par exemple, le cas d'utilisation pour un aspect particulier des fonctionnalités du système est capable de décrire ce que le guichet fait dans chaque situation possible. Chacune de ces situations est appelée un *scénario*, et un cas d'utilisation peut être considéré comme une collection de scénarios. On peut

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

penser à un scénario comme à une question qui commence par « Qu'est-ce que le système fait si... ? ». Par exemple, « Qu'est que le guichet fait si un client vient de déposer un chèque dans 24 heures et qu'il n'y a pas assez dans le compte sans le chèque pour fournir le retrait demandé ? ».

Les diagrammes de cas d'utilisations sont voulus simples pour ne pas se perdre prématurément dans les détails de l'implémentation du système :



Chaque bonhomme représente un « acteur », typiquement une personne ou une autre sorte d'agent (cela peut même être un autre système informatique, comme c'est le cas avec « ATM »). La boîte représente les limites du système. Les ellipses représentent les cas d'utilisation, qui sont les descriptions des actions qui peuvent être réalisées avec le système. Les lignes entre les acteurs et les cas d'utilisation représentent les interactions.

Tant que le système est perçu ainsi par l'utilisateur, son implémentation n'est pas importante.

Un cas d'utilisation n'a pas besoin d'être complexe, même si le système sous-jacent l'est. Il est seulement destiné à montrer le système tel qu'il apparaît à l'utilisateur. Par exemple:

Les cas d'utilisation produisent les spécifications des besoins en déterminant toutes les interactions que l'utilisateur peut avoir avec le système. Il faut trouver un ensemble complet de cas d'utilisations du système, et cela terminé on se retrouve avec le coeur de ce que le système est censé faire. La beauté des cas d'utilisation est qu'ils ramènent

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

toujours aux points essentiels et empêchent de se disperser dans des discussions non essentielles à la réalisation du travail à faire. Autrement dit, si on dispose d'un ensemble complet de cas d'utilisation on peut décrire le système et passer à la phase suivante. Tout ne sera pas parfaitement clair dès le premier jet, mais ça ne fait rien. Tout se décantera avec le temps, et si on cherche à obtenir des spécifications du système parfaites à ce point on se retrouvera coincé.

Si on est bloqué, on peut lancer cette phase en utilisant un outil d'approximation grossier : décrire le système en quelques paragraphes et chercher les noms et les verbes. Les noms suggèrent les acteurs, le contexte des cas d'utilisation ou les objets manipulés dans les cas d'utilisation. Les verbes suggèrent les interactions entre les acteurs et les cas d'utilisation, et spécifient les étapes à l'intérieur des cas d'utilisation. On verra aussi que les noms et les verbes produisent des objets et des messages durant la phase de design (on peut noter que les cas d'utilisation décrivent les interactions entre les sous-systèmes, donc la technique « des noms et des verbes » ne peut être utilisée qu'en tant qu'outil de brainstorming car il ne fournit pas les cas d'utilisation Plus d'information sur les cas d'utilisation peuvent être trouvées dans Applying Use Cases par Schneider & Winters (Addison-Wesley 1998) et Use Case Driven Object Modeling with UML par Rosenberg (Addison-Wesley 1999).).

La frontière entre un cas d'utilisation et un acteur peut révéler l'existence d'une interface utilisateur, mais ne définit pas cette interface utilisateur. Pour une méthode de définition et de création d'interfaces utilisateur, se référer à *Software for Use*de Larry Constantine et Lucy Lockwood, (Addison Wesley Longman, 1999) ou sur *www.ForUse.com*.

Bien que cela tienne plus de l'art obscur, à ce point un calendrier de base est important. On dispose maintenant d'une vue d'ensemble de ce qu'on construit et on peut donc se faire une idée du temps nécessaire à sa réalisation. Un grand nombre de facteurs entre en jeu ici. Si on surestime le temps de travail, l'entreprise peut décider d'abandonner le projet (et utiliser leurs ressources sur quelque chose de plus raisonnable - ce qui est une *bonne*chose). Ou un directeur peut avoir déjà décidé du temps que le projet devrait prendre et voudra influencer les estimations. Mais il vaut mieux proposer un calendrier honnête et prendre les décisions importantes au début. Beaucoup de techniques pour obtenir des calendriers précis ont été proposées (de même que pour prédire l'évolution de la bourse), mais la meilleure approche est probablement de se baser sur son expérience et son intuition. Proposer une estimation du temps nécessaire pour réaliser le système, puis doubler cette estimation et ajouter 10 pour cent. L'estimation initiale est probablement correcte, on *peut*obtenir un système fonctionnel avec ce temps. Le doublement transforme le délai en quelque chose de décent, et les 10 pour cent permettront de poser le vernis final et de traiter les détails. Mon estimation personnelle sur ce point a changé tardivement. Doubler et ajouter 10 pour cent vous donnera une estimation raisonnablement précise (en supposant qu'il n'y ait pas trop de jokers), mais vous devez toujours travailler assidûment pour terminer dans les temps. Si vous voulez prendre du bon temps et vous amuser durant le processus, le multiplicateur correct est plus proche de trois ou quatre, je pense.. Peu importe comment on l'explique et les gémissements obtenus quand on révèle un tel planning, il semble juste que ça fonctionne de cette façon.

1.9.3 - Phase 2 : Comment allons-nous le construire ?

Dans cette phase on doit fournir une conception qui décrive ce à quoi les classes ressemblent et comment elles interagissent. Un bon outil pour déterminer les classes et les interactions est la méthode des cartes *Classes-Responsabilités-Collaboration*(CRC). L'un des avantages de cet outil est sa simplicité : on prend des cartes vierges et on écrit dessus au fur et à mesure. Chaque carte représente une classe, et sur la carte on écrit :

1. Le nom de la classe. Il est important que le nom de cette classe reflète l'essence de ce que la classe fait, afin que sa compréhension soit immédiate.
2. Les « responsabilités » de la classe : ce qu'elle doit faire. Typiquement, cela peut être résumé par le nom des fonctions membres (puisque ces noms doivent être explicites dans une bonne conception), mais cela n'empêche pas de compléter par d'autres notes. Pour s'aider, on peut se placer du point de vue d'un programmeur fainéant : quels objets voudrait-on voir apparaître pour résoudre le problème ?
3. Les « collaborations » de la classe : avec quelles classes interagit-elle ? « Interagir » est intentionnellement

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

évasif, il peut se référer à une agrégation ou indiquer qu'un autre objet existant va travailler pour le compte d'un objet de la classe. Les collaborations doivent aussi prendre en compte l'audience de cette classe. Par exemple, si on crée une classe **Pétard**, qui va l'observer, un **Chimiste**ou un **Spectateur**? Le premier voudra connaître la composition chimique, tandis que le deuxième sera préoccupé par les couleurs et le bruit produits quand il explose.

On pourrait se dire que les cartes devraient être plus grandes à cause de toutes les informations qu'on aimerait mettre dessus, mais il vaut mieux les garder les plus petites possibles, non seulement pour concevoir de petites classes, mais aussi pour éviter de plonger trop tôt dans les détails. Si on ne peut pas mettre toutes les informations nécessaires à propos d'une classe sur une petite carte, la classe est trop complexe (soit le niveau de détails est trop élevé, soit il faut créer plus d'une classe). La classe idéale doit être comprise en un coup d'oeil. L'objectif des cartes CRC est de fournir un premier jet de la conception afin de saisir le plan général pour pouvoir ensuite affiner cette conception.

L'un des avantages des cartes CRC réside dans la communication. Il vaut mieux les réaliser en groupe, sans ordinateur. Chacun prend le rôle d'une ou plusieurs classes (qui au début n'ont pas de nom ni d'information associée). Il suffit alors de dérouler une simulation impliquant un scénario à la fois, et décider quels messages sont envoyés aux différents objets pour satisfaire chaque scénario. Au fur et à mesure du processus, on découvre quelles sont les classes nécessaires, leurs responsabilités et collaborations, et on peut remplir les cartes. Quand tous les scénarios ont été couverts, on devrait disposer d'une bonne approximation de la conception.

Avant d'utiliser les cartes CRC, la meilleure conception initiale que j'ai fourni sur un projet fut obtenue en dessinant des objets sur un tableau devant une équipe qui n'avait jamais participé à un projet de POO auparavant. Nous avons discuté de la communication entre ces objets, effacé et remplacé certains d'entre eux par d'autres objets. De fait, je recréais la méthode des cartes CRC au tableau. L'équipe (qui connaissait ce que le projet était censé faire) a effectivement créé la conception ; et de ce fait ils la contrôlaient. Je me contentais de guider le processus en posant les bonnes questions, proposant quelques hypothèses et utilisant les réponses de l'équipe pour modifier ces hypothèses. La beauté de la chose fut que l'équipe a appris à bâtir une conception orientée objet non en potassant des exemples abstraits, mais en travaillant sur la conception qui les intéressait au moment présent : celle de leur projet.

Une fois qu'on dispose d'un ensemble de cartes CRC, on peut vouloir une description plus formelle de la conception en utilisant l'UML. Pour débuter, je recommande encore une fois UML Distilled.. L'utilisation de l'UML n'est pas une obligation, mais cela peut être utile, surtout si on veut afficher au mur un diagramme auquel tout le monde puisse se référer, ce qui est une bonne idée. Une alternative à l'UML est une description textuelle des objets et de leur interface, ou suivant le langage de programmation, le code Python lui-même (www.Python.org) est souvent utilisé sous forme de « pseudo code exécutable »..

L'UML fournit aussi une notation pour décrire le modèle dynamique du système. C'est pratique dans les cas où les états de transition d'un système ou d'un sous-système sont suffisamment importants pour nécessiter leurs propres diagrammes (dans un système de contrôle par exemple). On peut aussi décrire les structures de données, pour les systèmes ou sous-systèmes dans lesquels les données sont le facteur dominant (comme une base de données).

On sait que la phase 2 est terminée quand on dispose de la description des objets et de leur interface. Ou du moins de la majorité d'entre eux - il y en a toujours quelques-uns qu'on ne découvre qu'en phase 3. Mais cela ne fait rien. La préoccupation principale est de découvrir tous les objets. Il est plus agréable de les découvrir le plus tôt possible mais la POO est assez souple pour pouvoir s'adapter si on en découvre de nouveaux par la suite. En fait, la conception d'un objet se fait en cinq étapes.

Les cinq étapes de la conception d'un objet

La conception d'un objet n'est pas limitée à la phase de codage du programme. En fait, la conception d'un objet passe par une suite d'étapes. Garder cela à l'esprit permet d'éviter de prétendre à la perfection immédiate. On

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

réalise que la compréhension de ce que fait un objet et de ce à quoi il doit ressembler se fait progressivement. Ceci s'applique d'ailleurs aussi à la conception de nombreux types de programmes ; le modèle d'un type de programme n'émerge qu'après s'être confronté encore et encore au problème (les *Design Patterns*sont traités dans le Volume 2). Les objets aussi ne se révèlent à la compréhension qu'après un long processus.

1. Découverte de l'objet. Cette étape se situe durant l'analyse initiale du programme. Les objets peuvent être découverts en cherchant les facteurs extérieurs et les frontières, la duplication d'éléments dans le système, et les plus petites unités conceptuelles. Certains objets sont évidents si on dispose d'un ensemble de bibliothèques de classes. La ressemblance entre les classes peut suggérer des classes de base et l'héritage peut en être déduit immédiatement, ou plus tard dans la phase de conception.

2. Assemblage des objets. Lors de la construction d'un objet, on peut découvrir le besoin de nouveaux membres qui n'était pas apparu durant l'étape de découverte. Les besoins internes d'un objet peuvent requérir d'autres classes pour les supporter.

3. Construction du système. Une fois de plus, un objet peut révéler des besoins supplémentaires durant cette étape. Au fur et à mesure de l'avancement du projet, les objets évoluent. Les besoins de la communication et de l'interconnexion avec les autres objets du système peuvent changer les besoins des classes ou demander de nouvelles classes. Par exemple, on peut découvrir le besoin de classes d'utilitaires, telles que des listes chaînées, qui contiennent peu ou pas d'information et sont juste là pour aider les autres classes.

4. Extension du système. Si on ajoute de nouvelles fonctionnalités au système, on peut se rendre compte que sa conception ne facilite pas l'extension du système. Avec cette nouvelle information, on peut restructurer certaines parties du système, éventuellement en ajoutant de nouvelles classes ou de nouvelles hiérarchies de classes.

5. Réutilisation des objets. Ceci est le test final pour une classe. Si quelqu'un tente de réutiliser une classe dans une situation entièrement différente, il y découvrira certainement des imperfections. La modification de la classe pour s'adapter à de nouveaux programmes va en révéler les principes généraux, jusqu'à l'obtention d'un type vraiment réutilisable. Cependant, il ne faut pas s'attendre à ce que tous les objets d'un système soient réutilisables - il est tout à fait légitime que la majorité des objets soient spécifiques au système. Les classes réutilisables sont moins fréquentes, et doivent traiter de problèmes plus génériques pour être réutilisables.

Indications quant au développement des objets

Ces étapes suggèrent quelques règles de base concernant le développement des classes :

1. Quand un problème spécifique génère une classe, la laisser grandir et mûrir durant la résolution d'autres problèmes.
2. Se rappeler que la conception du système consiste principalement à découvrir les classes dont on a besoin (et leurs interfaces). Si on dispose déjà de ces classes, le projet ne devrait pas être compliqué.
3. Ne pas vouloir tout savoir dès le début ; compléter ses connaissances au fur et à mesure de l'avancement du projet. La connaissance viendra de toutes façons tôt ou tard.
4. Commencer à programmer ; obtenir un prototype qui marche afin de pouvoir approuver la conception ou au contraire la dénoncer. Ne pas avoir peur de se retrouver avec du code-spaghetti à la procédurale - les classes partitionnent le problème et aident à contrôler l'anarchie. Les mauvaises classes n'affectent pas les classes bien conçues.
5. Toujours rester le plus simple possible. De petits objets propres avec une utilité apparente sont toujours mieux conçus que ceux disposant de grosses interfaces compliquées. Quand une décision doit être prise, utiliser l'approche du « rasoir d'Occam » : choisir la solution la plus simple, car les classes simples sont presque toujours les meilleures. Commencer petit et simple, et étendre l'interface de la classe quand on la comprend mieux, mais au fil du temps il devient difficile d'enlever des éléments d'une classe.

1.9.4 - Phase 3 : Construire le coeur du système

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Ceci est la conversion initiale de la conception brute en portion de code compilable et exécutable qui peut être testée, et surtout qui va permettre d'approuver ou d'invalider l'architecture retenue. Ce n'est pas un processus qui se fait en une passe, mais plutôt le début d'une série d'étapes qui vont construire le système au fur et à mesure comme le montre la phase 4.

Le but ici est de trouver le coeur de l'architecture du système qui a besoin d'être implémenté afin de générer un système fonctionnel, sans se soucier de l'état de complétion du système dans cette passe initiale. Il s'agit ici de créer un cadre sur lequel on va pouvoir s'appuyer pour les itérations suivantes. On réalise aussi la première des nombreuses intégrations et phases de tests, et on donne les premiers retours aux clients sur ce à quoi leur système ressemblera et son état d'avancement. Idéalement, on découvre quelques-uns des risques critiques. Des changements ou des améliorations sur l'architecture originelle seront probablement découverts - des choses qu'on n'aurait pas découvert avant l'implémentation du système.

Une partie de la construction du système consiste à confronter le système avec l'analyse des besoins et les spécifications du système (quelle que soit la forme sous laquelle ils existent). Il faut s'assurer en effet que les tests vérifient les besoins et les cas d'utilisations. Quand le coeur du système est stable, on peut passer à la suite et ajouter des fonctionnalités supplémentaires.

1.9.5 - Phase 4 : Itérer sur les cas d'utilisation

Une fois que le cadre de base fonctionne, chaque fonctionnalité ajoutée est un petit projet en elle-même. On ajoute une fonctionnalité durant une *itération*, période relativement courte du développement.

Combien de temps dure une itération ? Idéalement, chaque itération dure entre une et trois semaines (ceci peut varier suivant le langage d'implémentation choisi). A la fin de cette période, on dispose d'un système intégré et testé avec plus de fonctionnalités que celles dont il disposait auparavant. Mais ce qu'il est intéressant de noter, c'est qu'un simple cas d'utilisation constitue la base d'une itération. Chaque cas d'utilisation est un ensemble de fonctionnalités qu'on ajoute au système toutes en même temps, durant une itération. Non seulement cela permet de se faire une meilleure idée de ce que recouvre ce cas d'utilisation, mais cela permet de le valider, puisqu'il n'est pas abandonné après l'analyse et la conception, mais sert au contraire tout au long du processus de création.

Les répétitions s'arrêtent quand on dispose d'un système comportant toutes les fonctionnalités souhaitées ou qu'une date limite arrive et que le client se contente de la version courante (se rappeler que les commanditaires dirigent l'industrie du logiciel). Puisque le processus est itératif, on dispose de nombreuses opportunités pour délivrer une version intermédiaire au lieu qu'un produit final ; les projets open-source travaillent uniquement dans un environnement itératif avec de nombreux retours, ce qui précisément les rend si productifs.

Un processus de développement itératif est intéressant pour de nombreuses raisons. Cela permet de révéler et de résoudre des risques critiques très tôt, les clients ont de nombreuses opportunités pour changer d'avis, la satisfaction des programmeurs est plus élevée, et le projet peut être piloté avec plus de précision. Mais un bénéfice additionnel particulièrement important est le retour aux commanditaires du projet, qui peuvent voir grâce à l'état courant du produit où le projet en est. Ceci peut réduire ou éliminer le besoin de réunions soporifiques sur le projet, et améliore la confiance et le support des commanditaires.

1.9.6 - Phase 5 : Evolution

Cette phase du cycle de développement a traditionnellement été appelée « maintenance », un terme fourre-tout qui peut tout vouloir dire depuis « faire marcher le produit comme il était supposé le faire dès le début » à « ajouter de nouvelles fonctionnalités que le client a oublié de mentionner » au plus traditionnel « corriger les bugs qui apparaissent » et « ajouter de nouvelles fonctionnalités quand le besoin s'en fait sentir ». Le terme « maintenance » a été la cause de si nombreux malentendus qu'il en est arrivé à prendre un sens péjoratif, en partie parce qu'il suggère qu'on a fourni un programme parfait et que tout ce qu'on a besoin de faire est d'en changer quelques

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

parties, le graisser et l'empêcher de rouiller. Il existe peut-être un meilleur terme pour décrire ce qu'il en est réellement.

J'utiliserai plutôt le terme *évolution* Au moins un aspect de l'évolution est traité dans le livre Refactoring: improving the design of existing code de Martin Fowler (Addison-Wesley 1999). Soyez prévenu que ce livre utilise exclusivement des exemples en Java.. C'est à dire, « Tout ne sera pas parfait dès le premier jet, il faut se laisser la latitude d'apprendre et de revenir en arrière pour faire des modifications ». De nombreux changements seront peut-être nécessaires au fur et à mesure que l'appréhension et la compréhension du problème augmentent. Si on continue d'évoluer ainsi jusqu'au bout, l'élégance obtenue sera payante, à la fois à court et long terme. L'évolution permet de passer d'un bon à un excellent programme, et clarifie les points restés obscurs durant la première passe. C'est aussi dans cette phase que les classes passent d'un statut d'utilité limitée au système à ressources réutilisables.

Ici, « jusqu'au bout » ne veut pas simplement dire que le programme fonctionne suivant les exigences et les cas d'utilisation. Cela veut aussi dire que la structure interne du code présente une logique d'organisation et semble bien s'assembler, sans abus de syntaxe, d'objets surdimensionnés ou de code inutilement exposé. De plus, il faut s'assurer que la structure du programme puisse s'adapter aux changements qui vont inévitablement arriver pendant sa durée vie, et que ces changements puissent se faire aisément et proprement. Ceci n'est pas une petite caractéristique. Il faut comprendre non seulement ce qu'on construit, mais aussi comment le programme va évoluer (ce que j'appelle le *vecteur changement* Ce terme est exploré dans le chapitre Design Patterns du Volume 2.). Heureusement, les langages de programmation orientés objet sont particulièrement adaptés à ce genre de modifications continuelles - les frontières créées par les objets sont ce qui empêche la structure du programme de s'effondrer. Ils permettent aussi de faire des changements - même ceux qui seraient considérés comme sévères dans un programme procédural - sans causer de ravages dans l'ensemble du code. En fait le support de l'évolution pourrait bien être le bénéfice le plus important de la programmation orientée objet.

Avec l'évolution, on crée quelque chose qui approche ce qu l'on croit avoir construit, on le compare avec les exigences et on repère les endroits où ça coince. On peut alors revenir en arrière et corriger ça en remodélisant et réimplémentant les portions du programme qui ne fonctionnaient pas correctement Ceci est comparable au « prototypage rapide » où l'on était censé construire rapidement une version brouillon permettant de se faire une bonne idée du système, puis jeter ce prototype et tout reconstruire proprement. Le problème avec le prototypage rapide est que les gens ne jettent pas le prototype, mais au contraire s'en servent comme base. Combiné avec le manque de structure de la programmation procédurale, ceci a souvent produit des systèmes fouillis coûteux à maintenir.. De fait, on peut avoir besoin de résoudre le problème ou un de ses aspects un certain nombre de fois avant de trouver la bonne solution (une étude de *Design Patterns*, dépeinte dans le Volume 2, s'avère généralement utile ici).

Il faut aussi évoluer quand on construit un système, que l'on voit qu'il remplit les exigences et que l'on découvre finalement que ce n'était pas ce que l'on voulait. Quand on se rend compte après avoir vu le système en action qu'on essayait de résoudre un autre problème. Si on pense que ce genre d'évolution est à prendre en considération, alors on se doit de construire une première version aussi rapidement que possible afin de déterminer au plus tôt si c'est réellement ce qu'on veut.

La chose la plus importante à retenir est que par défaut - par définition, plutôt - si on modifie une classe alors ses classes parentes et dérivées continueront de fonctionner. Il ne faut pas craindre les modifications (surtout si on dispose d'un ensemble de tests qui permettent de vérifier les modifications apportées). Les modifications ne vont pas nécessairement casser le programme, et tout changement apporté sera limité aux sous-classes et / ou aux collaborateurs spécifiques de la classe qu'on change.

1.9.7 - Les plans sont payants

Bien sûr on ne bâtirait pas une maison sans une multitude de plans dessinés avec attention. Si on construit un pont ou une niche pour chien, les plans ne seront pas aussi élaborés mais on démarre avec quelques esquisses

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

pour se guider. Le développement de logiciels a connu les extrêmes. Longtemps les gens ont travaillé sans structure, mais on a commencé à assister à l'effondrement de gros projets. En réaction, on en est arrivé à des méthodologies comprenant un luxe de structure et de détails, destinées justement à ces gros projets. Ces méthodologies étaient trop intimidantes pour qu'on les utilise - on avait l'impression de passer son temps à écrire des documents et aucun moment à coder (ce qui était souvent le cas). J'espère que ce que je vous ai montré ici suggère un juste milieu. Utilisez une approche qui corresponde à vos besoins (et votre personnalité). Même s'il est minimal, la présence d'un plan vous apportera beaucoup dans la gestion de votre projet. Rappelez-vous que selon la plupart des estimations, plus de 50 pour cent des projets échouent (certaines estimations vont jusqu'à 70 pour cent).

En suivant un plan - de préférence un qui soit simple et concis - et en produisant une modélisation de la structure avant de commencer à coder, vous découvrirez que les choses s'arrangent bien mieux que si on se lance comme ça dans l'écriture, et vous en retirerez aussi une plus grande satisfaction. Suivant mon expérience, arriver à une solution élégante procure une satisfaction à un niveau entièrement différent ; cela ressemble plus à de l'art qu'à de la technologie. Et l'élégance est toujours payante, ce n'est pas une vaine poursuite. Non seulement on obtient un programme plus facile à construire et déboguer, mais qui est aussi plus facile à comprendre et maintenir, et c'est là que sa valeur financière réside.

1.10 - Extreme programming

J'ai étudié à différentes reprises les techniques d'analyse et de conception depuis que je suis sorti de l'école. Le concept de *Extreme programming*(XP) est le plus radical et divertissant que j'ai vu. Il est rapporté dans *Extreme Programming Explained*de Kent Beck (Addison-Wesley 2000) et sur le web à *www.xprogramming.com*.

XP est à la fois une philosophie à propos de la programmation et un ensemble de règles de base. Certaines de ces règles sont reprises dans d'autres méthodologies récentes, mais les deux contributions les plus importantes et novatrices, sont à mon sens « commencer par écrire les tests » et « programmation en binôme ». Bien qu'il soutienne et argumente l'ensemble de la théorie, Beck insiste sur le fait que l'adoption de ces deux pratiques améliore grandement la productivité et la fiabilité.

1.10.1 - Commencer par écrire les tests

Les tests ont traditionnellement été relégués à la dernière partie d'un projet, une fois que « tout marche, mais c'est juste pour s'en assurer ». Ils ne sont généralement pas prioritaires et les gens qui se spécialisent dedans ne sont pas reconnus à leur juste valeur et se sont souvent vus cantonnés dans un sous-sol, loin des « véritables programmeurs ». Les équipes de test ont réagi en conséquence, allant jusqu'à porter des vêtements de deuil et glousser joyeusement quand ils trouvaient des erreurs (pour être honnête, j'ai eu moi aussi ce genre de sentiments lorsque je mettais des compilateurs C++ en faute).

XP révolutionne complètement le concept du test en lui donnant une priorité aussi importante (ou même plus forte) que le code. En fait, les tests sont écrits avant le code qui est testé, et les tests restent tout le temps avec le code. Ces tests doivent être exécutés avec succès à chaque nouvelle intégration dans le projet (ce qui peut arriver plus d'une fois par jour).

Ecrire les tests d'abord a deux effets extrêmement importants.

Premièrement, cela nécessite une définition claire de l'interface d'une classe. J'ai souvent suggéré que les gens « imaginent la classe parfaite qui résolve un problème particulier » comme outil pour concevoir le système. La stratégie de test de XP va plus loin - elle spécifie exactement ce à quoi la classe doit ressembler pour le client de cette classe, et comment la classe doit se comporter, dans des termes non ambigus. On peut écrire tout ce qu'on veut, ou créer tous les diagrammes décrivant comment une classe devrait se comporter et ce à quoi elle ressemble, mais rien n'est aussi réel qu'une batterie de tests. Le premier est une liste de voeux, mais les tests sont

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

un contrat certifié par un compilateur et un programme qui marche. Il est difficile d'imaginer une description plus concrète d'une classe que les tests.

En créant les tests, on est forcé de penser complètement la classe et souvent on découvre des fonctionnalités nécessaires qui ont pu être manquées lors de l'utilisation des diagrammes UML, des cartes CRC, des cas d'utilisation, etc.

Le deuxième effet important dans l'écriture des tests en premier vient du fait qu'on peut lancer les tests à chaque nouvelle version du logiciel. Cela permet d'obtenir l'autre moitié des tests réalisés par le compilateur. Si on regarde l'évolution des langages de programmation de ce point de vue, on se rend compte que les améliorations réelles dans la technologie ont en fait tourné autour du test. Les langages assembleur vérifiaient uniquement la syntaxe, puis le C a imposé des restrictions sémantiques, et cela permettait d'éviter certains types d'erreurs. Les langages orientés objet imposent encore plus de restrictions sémantiques, qui sont quand on y pense des formes de test. « Est-ce que ce type de données est utilisé correctement ? Est-ce que cette fonction est appelée correctement ? » sont le genre de tests effectués par le compilateur ou le système d'exécution. On a pu voir le résultat d'avoir ces tests dans le langage même : les gens ont été capables de construire des systèmes plus complexes, et de les faire marcher, et ce en moins de temps et d'efforts. Je me suis souvent demandé pourquoi, mais maintenant je réalise que c'est grâce aux tests : si on fait quelque chose de faux, le filet de sécurité des tests intégré au langage prévient qu'il y a un problème et montre même où il réside.

Mais les tests intégrés permis par la conception du langage ne peuvent aller plus loin. A partir d'un certain point, il est de notre responsabilité de produire une suite complète de tests (en coopération avec le compilateur et le système d'exécution) qui vérifie tout le programme. Et, de même qu'il est agréable d'avoir un compilateur qui vérifie ce qu'on code, ne serait-il pas préférable que ces tests soient présents depuis le début ? C'est pourquoi on les écrit en premier, et qu'on les exécute automatiquement à chaque nouvelle version du système. Les tests deviennent une extension du filet de sécurité fourni par le langage.

L'utilisation de langages de programmation de plus en plus puissants m'a permis de tenter plus de choses audacieuses, parce que je sais que le langage m'empêchera de perdre mon temps à chasser les bugs. La stratégie de tests de XP réalise la même chose pour l'ensemble du projet. Et parce qu'on sait que les tests vont révéler tous les problèmes introduits (et on ajoute de nouveaux tests au fur et à mesure qu'on les imagine), on peut faire de gros changements sans se soucier de mettre le projet complet en déroute. Ceci est une approche particulièrement puissante.

1.10.2 - Programmation en binôme

La programmation en binôme va à l'encontre de l'individualisme farouche endoctriné, depuis l'école (où on réussit ou échoue suivant nos mérites personnels, et où travailler avec ses voisins est considéré comme « tricher ») et jusqu'aux médias, en particulier les films hollywoodiens dans lequel le héros se bat contre la conformité stupide. Bien que ce soit une perspective plutôt américaine, les histoires d'Hollywood sont présentes partout.Les programmeurs aussi sont considérés comme des parangons d'individualisme - des « codeurs cowboys » comme aime à le dire Larry Constantine. Et XP, qui se bat lui-même contre la pensée conventionnelle, soutient que le code devrait être écrit avec deux personnes par station de travail. Et cela devrait être fait dans un endroit regroupant plusieurs stations de travail, sans les barrières dont raffolent les spécialistes de l'aménagement de bureau. En fait, Beck dit que la première tâche nécessaire pour implémenter XP est de venir avec des tournevis et d'enlever tout ce qui se trouve dans le passage Y compris (et plus particulièrement) le système PA (Public Address : commutateur téléphonique privé). J'ai un jour travaillé dans une entreprise qui insistait sur le fait de diffuser tous les appels téléphoniques reçus aux administrateurs, et cela interrompait constamment notre productivité (mais les managers ne pouvaient pas envisager un service aussi important que le PA sans s'étouffer). Finalement, j'ai discrètement coupé les fils du haut parleur.(Cela nécessite un responsable qui puisse absorber la colère des responsables de l'équipement).

Dans la programmation en binôme, une personne produit le code tandis que l'autre y réfléchit. Le penseur garde la

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

conception générale à l'esprit, pas seulement la description du problème en cours, mais aussi les règles de XP à portée de main. Si deux personnes travaillent, il y a moins de chance que l'une d'entre elles s'en aille en disant « Je ne veux pas commencer en écrivant les tests », par exemple. Et si le codeur reste bloqué, ils peuvent changer leurs places. Si les deux restent bloqués, leurs songeries peuvent être remarquées par quelqu'un d'autre dans la zone de travail qui peut venir les aider. Travailler en binôme permet de garder une bonne productivité et de rester sur la bonne pente. Probablement plus important, cela rend la programmation beaucoup plus sociable et amusante.

J'ai commencé à utiliser la programmation en binôme durant les périodes d'exercice dans certains de mes séminaires et il semblerait que cela enrichisse l'expérience personnelle de chacun.

1.11 - Les raisons du succès du C++

Une des raisons pour laquelle le C++ connait autant de succès est qu'il n'a pas pour unique vocation d'apporter au C une approche orientée objet (malgré le fait qu'il ait été conçu dans ce sens), mais il a aussi pour but de résoudre beaucoup d'autres problèmes auxquels font face les développeurs actuels, spécialement ceux qui ont beaucoup investi dans le C. Traditionnellement, les langages orientés objets souffrent du fait que vous devez abandonner tout ce que vous savez déjà et repartir de zéro avec un nouvel ensemble de concepts et une nouvelle syntaxe, argumentant qu'il est bénéfique à long terme de se débarrasser de tout les vieux bagages des langages procéduraux. Cela peut être vrai à long terme. Mais à court terme, beaucoup de ces vieux bagages ont encore de la valeur. L'élément ayant le plus de valeur peut ne pas être le code de base (qui, avec les outils adéquats, peut être traduit), mais plutôt dans la *base spirituelle*existante. Si vous êtes un développeur C fonctionnel et que vous devez abandonner toutes vos connaissances dans ce langage afin d'en adopter un nouveau, vous devenez immédiatement moins productif pendant plusieurs mois, le temps que votre esprit s'adapte à ce nouveau paradigme. Si l'on considère que vous pouvez conserver vos connaissances en C et les étendre, vous pouvez continuer d'être productif avec vos connaissances actuelles pendant que vous vous tournez vers la POO. Comme tout le monde a son propre modèle mental de programmation, ce changement est suffisamment handicapant pour ne pas avoir à ajouter des coûts de démarrage avec un nouveau modèle de langage. Donc, la raison du succès du C++, en résumé, est économique: Passer à un langage orienté objet a toujours un coût, mais le C++ peut coûter moins. Je dis “peut” car, au vu de la complexité du C++, il peut être en réalité moins coûteux de passer au Java. Mais la décision du choix du langage dépend de nombreux facteurs, et, dans ce livre, je supposerai que vous avez choisi le C++.

Le but du C++ est d'augmenter la productivité. Cette productivité peux intervenir en de nombreux points, mais le langage est construit pour vous aider autant que possible, tout en vous gênant le moins possible avec des règles arbitraires ou des conditions à utiliser dans un environnement particulier. Le C++ est conçu pour être pratique; les décisions de conception du langage C++ ont été basées sur l'offre d'un maximum de bénéfice au développeur (au moins, d'un point de vue par rapport au C).

1.11.1 - Un meilleur C

Vous êtes tout de suite gagnant, même si vous continuez à écrire du code C, car le C++ à comblé de nombreuses lacunes du langage C et offre une meilleure vérification de types ainsi qu'une meilleur analyse du temps de compilation. Vous êtes forcé de déclarer des fonctions ce qui permet au compilateur de vérifier leurs utilisations. La nécessité du préprocesseur a virtuellement été eliminée concernant la substitution de valeurs et les macros, ce qui enlève bon nombre de bugs difficile à trouver. Le C++ possède un dispositif appellé *référence*qui apporte des facilités pour l'utilisation des adresses concernant les arguments des fonctions et les valeurs de retour. La prise en charge des noms est améliorée avec une fonctionnalité appelée *surcharge de fonction*, qui vous permet d'utiliser le même nom pour différentes fonction. Un autre dispositif nommé *espace de nommage*améliore également le contrôle des noms. Il existe de nombreuses autres fonctionnalités moins importantes qui améliorent la sécurité du C.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

1.11.2 - Vous êtes déjà sur la courbe d'apprentissage.

Le problème avec l'apprentissage d'un nouveau langage est la productivité. Aucune entreprise ne peut se permettre de perdre subitement un ingénieur productif parce qu'il apprend un nouveau langage. Le C++ est une extension du C, il n'y a pas de nouvelle syntaxe ni de nouveau modèle de programmation. Il vous permet de continuer de créer du code utile, appliquant graduellement les nouvelles fonctionnalités au fur et à mesure que vous les apprenez et les comprenez. C'est certainement une des plus importantes raisons du succès du C++.

En plus de cela, tout votre code C reste viable en C++, cependant, parce que le compilateur C++ est plus pointilleux, vous trouverez toujours des erreurs C cachées lors de votre compilation en C++.

1.11.3 - Efficacité

Parfois il est plus approprié de négliger la vitesse d'exécution pour gagner en productivité. Un modèle financier, par exemple, peut être utile pour une courte période de temps, aussi il est plus important de créer le modèle rapidement plutôt que de l'exécuter rapidement. Cependant, la plupart des applications nécessitent un certain degré d'efficacité, aussi, le C++ se range toujours du coté d'une plus grande efficacité. Comme les développeurs ont tendance à avoir un esprit d'efficacité, c'est également une manière de s'assurer qu'ils ne pourront pas arguer que le langage est trop lourd et trop lent. Un grand nombre de fonctionnalités en C++ ont pour objectif de vous permettre de personnaliser les performances lorsque le code généré n'est pas suffisamment efficace.

Non seulement vous avez le même contrôle bas niveau qu'en C (et la possibilité d'écrire directement en assembleur dans un programme C++), mais s'il faut croire ce que l'on dit la rapidité d'un programme orienté objet en C++ tend à être dans les ±10% par rapport à un programme écrit en C, et souvent plus près. Cependant, lisez les articles de Dan Saks dans le C/C++ User's Journal à propos d'importantes recherches concernant les performances des bibliothèques C++.La conception produite pour un programme orienté objet peut être réellement plus efficace que sa contrepartie en C.

1.11.4 - Les systèmes sont plus faciles à exprimer et à comprendre

Les classes conçues pour s'adapter au problème ont tendance à mieux l'exprimer. Cela signifie que quand vous écrivez du code, vous décrivez votre solution dans les termes de l'espace du problème(“Mettre un tore dans le casier”) plutôt que de la décrire dans les termes de la machine (“Activer le bit du circuit integré qui va déclancher la fermeture du relais”). Vous manipulez des concepts de haut niveau et pouvez faire beaucoup plus avec une seule ligne de code.

L'autre bénéfice de cette facilité d'expression est la maintenance, qui (si l'on en croit les rapports) a un énorme coût dans la vie d'un programme. Si un programme est plus simple à comprendre, alors il est plus facile de le maintenir. Cela peut également réduire les coûts de création et de maintenance de la documentation.

1.11.5 - Puissance maximale grâce aux bibliothèques

La façon la plus rapide de créer un programme est d'utiliser du code déjà écrit : une bibliothèque. Un des buts fondamentaux du C++ est de faciliter l'emploi des bibliothèques. Ce but est obtenu en convertissant ces bibliothèques en nouveaux types de données (classes), et utiliser une bibliothèque revient à ajouter de nouveaux types au langage. Comme le compilateur C++ s'occupe de l'interfaçage avec la bibliothèque - garantissant une initialisation et un nettoyage propres, et s'assurant que les fonctions sont appelées correctement - on peut se concentrer sur ce qu'on attend de la bibliothèque, et non sur les moyens de le faire.

Parce que les noms peuvent être isolés dans une portion de votre programme via les espace de nom C++, vous

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

pouvez utiliser autant de bibliothèques que vous le désirez sans les conflits de noms que vous encourez en C.

1.11.6 - Réutilisation des sources avec les templates

Il existe une classe significative de types qui exigent une modification du code source avant de pouvoir les réutiliser efficacement. Le dispositif de template en C++ opère une modification du code source automatiquement, ce qui en fait un outil particulièrement performant pour réutiliser les codes des bibliothèques. Un type conçu avec l'utilisation de *templates*va fonctionner avec moins d'efforts avec beaucoup d'autres types. Les templates sont spécialement intéressants car ils cachent au développeur client la complexité de cette partie de code réutilisée au développeur client.

1.11.7 - Traitement des erreurs

L'une des difficultés du C est la gestion des erreurs, problème connu et largement ignoré - on compte souvent sur la chance. Si on construit un programme gros et complexe, il n'y a rien de pire que de trouver une erreur enfouie quelque part sans qu'on sache d'où elle vient. Le *traitement des exceptions*du C++ (introduit dans ce Volume, et traité de manière complète dans le Volume 2, téléchargeable depuis *www.BruceEckel.com*) est une façon de garantir qu'une erreur a été remarquée, et que quelque chose est mis en oeuvre pour la traiter.

1.11.8 - Mise en oeuvre de gros projets

Beaucoup de langages traditionnels imposent des limitations internes sur la taille des programmes et leur complexité. BASIC, par exemple, peut s'avérer intéressant pour mettre en oeuvre rapidement des solutions pour certains types de problèmes ; mais si le programme dépasse quelques pages de long ou s'aventure en dehors du domaine du langage, cela revient à tenter de nager dans un liquide encore plus visqueux. C, lui aussi, possède ces limitations. Par exemple, quand un programme dépasse peut-être 50.000 lignes de code, les conflits de noms commencent à devenir un problème - concrètement, vous êtes à court de noms de fonctions et de variables. Un autre problème particulièrement grave est le nombre de lacunes du langage C - les erreurs disséminées au sein d'un gros programme peuvent être extrêmement difficiles à localiser.

Aucune limite ne prévient que le cadre du langage est dépassé, et même s'il en existait, elle serait probablement ignorée. On devrait se dire : « Mon programme BASIC devient trop gros, je vais le réécrire en C ! », mais à la place on tente de glisser quelques lignes supplémentaires pour implémenter cette nouvelle fonctionnalité. Le coût total continue donc à augmenter.

C++ est conçu pour aider à *programmer en grand*, c'est à dire, qu'il supprime les frontières de complexité qui séparent les petits programmes et les grands. Vous n'avez certainement pas besoin d'utiliser la POO, les templates, les espaces de noms, et autres gestionnaire d'exception quand vous programmez un programme du style "Hello World", cependant ces dispositifs sont là quand vous en avez besoin. De plus, le compilateur est intransigeant quant il s'agit de jeter dehors les erreurs productrices de bugs pour les petits et gros programmes.

1.12 - Stratégies de transition

Si vous investissez dans la POO, votre prochaine question est probablement "Comment puis-je faire en sorte que mes responsables/collègues/départements/pairs commencent à utiliser des objets?". Demandez-vous comment vous - un programmeur indépendant - voudriez commencer à apprendre à utiliser un nouveau langage et une nouvelle vision de la programmation. Vous avez déjà fait cela auparavant. Premièrement, il y a l'éducation et les exemples ; ensuite arrivent les projets d'essai qui vous donnent les bases sans faire des choses trop déroutantes. Ensuite, vient un projet du "monde réel", qui fait vraiment quelque chose d'utile. Au travers de vos premiers projets, vous continuez votre éducation en lisant, en posant des questions aux experts et en échangeant des petites astuces entre amis. C'est l'approche que de nombreux programmeurs expérimentés suggèrent pour passer du C

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

au C++. Convertir une entreprise entière apportera bien sûr une certaine dynamique de groupe, mais cela aidera à chaque étape de se rappeler comment une seule personne le ferait.

1.12.1 - Les grandes lignes

Vous trouverez ici les grandes lignes à prendre en compte lors de votre transition vers la POO et le C++ :

1. L'entraînement

La première étape est une forme d'éducation. Rappelez vous de l'investissement de l'entreprise dans du code C, et essayez de ne pas sombrer dans le désarroi pendant six à neuf mois alors que tout le monde cherche à comprendre le fonctionnement de l'héritage multiple. Prenez un petit groupe pour l'endoctrinement, de préférence composé de personnes curieuses, qui travaillent bien ensemble, et sont capables de créer leur propre réseau de soutien tout en apprenant le C++.

Une approche alternative qui est parfois suggérée est la formation de tous les niveaux de la société d'un seul coup, comprenant aussi bien les cours de stratégie pour les directeurs que les cours de conception et programmation pour les chefs de projets. Cette méthode est spécialement bonne pour les plus petites entreprises souhaitant opérer des changements majeurs dans la manière de faire les choses, ou au niveau de la division pour les plus grosses entreprises. Puisque le coût est plus élevé, cependant, certains choisiront de commencer avec un entraînement au niveau du projet, la création d'un projet pilote (peut-être avec un mentor extérieur), et laisseront l'équipe du projet devenir les formateurs du reste de l'entreprise.

2. Projets à faibles risques

Essayez tout d'abord un projet à faibles risques et tenez compte des erreurs. Une fois que vous avez acquis une certaine expérience, vous pouvez soit commencer d'autres projets avec les membres de cette première équipe soit utiliser les membres de l'équipe comme supports techniques de la POO. Ce premier projet ne peut pas fonctionner correctement la première fois, ainsi il ne devrait pas être critique pour la compagnie. Il devrait être simple, d'un seul bloc, et instructif ; ceci signifie qu'il devrait impliquer de créer les classes qui seront significatives pour les autres programmeurs de l'entreprise quand ils commenceront à leur tour à apprendre le C++.

3. Le modèle du succès

Cherchez des exemples d'une bonne conception orientée objet plutôt que commencer à zéro. Il y a une bonne probabilité que quelqu'un ait déjà résolu votre problème, et s'ils ne l'ont pas tout à fait résolu vous pouvez probablement appliquer ce que vous avez appris sur l'abstraction pour modifier une conception existante pour adapter à vos besoins. C'est le concept général des *modèles de conception*(*design patterns*), couverts par le Volume 2.

4. Utiliser des bibliothèques de classes existantes

La principale motivation économique pour passer à la POO est la facilité d'utilisation du code existant sous forme de bibliothèques de classe (en particulier, les bibliothèques du Standard C++, qui sont détaillées dans le Volume 2 de ce livre). Le cycle de développement d'application le plus court s'ensuivra quand vous n'aurez rien d'autre à écrire que **main( )**, en créant et en utilisant des objets de bibliothèques disponibles immédiatement. Cependant, certains nouveaux programmeurs ne le comprennent pas, ignorent les bibliothèques de classe existantes, ou, par fascination du langage, désirent écrire des classes qui existent peut-être déjà. Votre succès avec la POO et le C++ sera optimisé si vous faites un effort de recherche et de réutilisation du code d'autres personnes rapidement dans le processus de transition.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

5. Ne réécrivez pas du code existant en C++

Bien que *compiler*votre code C avec un compilateur C++ a habituellement des avantages (parfois énormes) en trouvant des problèmes dans l'ancien code, la meilleure façon de passer votre temps n'est généralement pas de prendre le code existant, fonctionnel, et de le réécrire en C++. (Si vous devez le transformer en objets, vous pouvez "envelopper" le code C dans des classes C++.) Il y a des avantages importants, particulièrement si le code est prévu pour être réutilisé. Mais il y a des chances que vous ne voyiez pas les augmentations spectaculaires de la productivité que vous espériez dans vos premiers projets à moins que ces projets n'en soient de nouveaux. Le C++ et la POO brillent mieux en prenant un projet de la conception à la réalisation.

1.12.2 - Ecueils de la gestion

Si vous êtes directeur, votre travail est d'acquérir des ressources pour votre équipe, de franchir les obstacles sur la route du succès de votre équipe, et en général d'essayer de fournir l'environnement le plus productif et le plus agréable possible, ainsi votre équipe est-elle plus susceptible de réaliser ces miracles que vous demandez toujours. Passer au C++ rentre dans chacune de ces trois catégories, et ce serait fantastique si ça ne vous coûtait également rien. Bien que se convertir au C++ peut être meilleur marché - selon vos contraintes En raison de ses améliorations de productivité, le langage Java devrait également être pris en compte ici.- que les alternatives de POO pour une équipe de programmeurs en langage C (et probablement pour des programmeurs dans d'autres langages procéduraux), ce n'est pas gratuit, et il y a des obstacles dont vous devez être conscient avant de tenter de vendre le passage à C++ au sein de votre entreprise et de commencer la transition elle-même.

Coûts de démarrage

Le coût du passage au C++ est plus que simplement l'acquisition des compilateurs C++ (le compilateur GNU C++, un des meilleurs, est gratuit). Vos coûts à moyen et long terme seront réduits au minimum si vous investissez dans la formation (et probablement la tutelle pour votre premier projet) et aussi si vous identifiez et achetez les bibliothèques de classe qui résolvent votre problème plutôt que d'essayer de construire ces bibliothèques par vous-même. Ce sont des coûts financiers bruts qui doivent être pris en compte dans une proposition réaliste. En outre, il y a les coûts cachés dans la perte de productivité lors de l'apprentissage d'un nouveau langage et probablement un nouvel environnement de programmation. La formation et la tutelle peuvent certainement les réduire, mais les membres d'équipe doivent gagner leurs propres luttes pour comprendre la nouvelle technologie. Pendant ce processus ils feront plus d'erreurs (c'est une caractéristique, parce que les erreurs reconnues permettent d'apprendre plus rapidement) et seront moins productifs. Même à ce moment là, avec certains types de problèmes de programmation, les bonnes classes, et le bon environnement de développement, il est possible d'être plus productif en apprenant le C++ (même en considérant que vous faites plus d'erreurs et écrivez moins de lignes de code par jour) qu'en étant resté en C.

Questions de performance

Une question habituelle est, "La POO ne rend-t-elle pas automatiquement mes programmes beaucoup plus volumineux et plus lents ?" La réponse est, "Ca dépend." La plupart des langages traditionnels de POO ont été conçus pour une expérimentation et un prototypage rapide à l'esprit plutôt qu'une cure d'amaigrissement. Ainsi, elles ont pratiquement garanti une augmentation significative de taille et une diminution de la vitesse. Cependant, le C++ est conçu avec la programmation de production à l'esprit. Quand votre accent se porte sur le prototypage rapide, vous pouvez rassembler des composants aussi rapidement que possible tout en ignorant les questions d'efficacité. Si vous utilisez des bibliothèques tierces, elles sont généralement déjà optimisées par leurs fournisseurs ; de toutes façons ce n'est pas un problème si vous êtes en mode de développement rapide. Quand vous avez un système que vous appréciez, s'il est petit et assez rapide, alors c'est bon. Sinon, vous commencez à l'ajuster avec un outil d'analyse, regardant d'abord les accélérations qui peuvent être obtenues avec des applications simples des fonctionnalités intégrées du C++. Si cela n'aide pas, vous recherchez les modifications qui peuvent être faites dans l'implémentation sous-jacente de telle façon qu'aucun code qui utilise une classe particulière n'ait à être changé. C'est seulement si rien d'autre ne résout le problème que vous devez modifier la

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

conception. Le fait que la performance soit si critique dans cette partie de la conception est un indicateur qu'elle doit faire partie des critères principaux de conception. Vous avez l'avantage de le trouver précocement en utilisant le développement rapide.

Comme cité précédemment, le nombre qui est le plus souvent donné pour la différence en taille et en vitesse entre le C et le C++ est ±10%, et souvent beaucoup plus proche de l'égalité. Vous pourriez même obtenir une amélioration significative de taille et de vitesse en utilisant le C++ plutôt que le C parce que la conception que vous faites en C++ pourrait être tout à fait différente de celle que vous feriez en C.

Les comparaisons de taille et de vitesse entre le C et le C++ sont affaire d'opinion et de ouï-dire plutôt que de mesures incontestables, et il est probable qu'elles le restent. Indépendamment du nombre de personnes qui proposent qu'une entreprise teste le même projet en utilisant le C et le C++, aucune société n'est susceptible de gaspiller de l'argent de cette façon à moins qu'elle soit très grande et intéressée par de tels projets de recherche. Même dans ce cas, il semble que l'argent pourrait être mieux dépensé. Presque universellement, les programmeurs qui sont passés du C (ou d'un autre langage procédural) au C++ (ou à un autre langage de POO) ont eu une expérience personnelle de grande accélération dans leur productivité de programmation, et c'est l'argument le plus incontestable que vous pouvez trouver.

Erreurs courantes de conception

Quand vous engagez votre équipe dans la POO et le C++, les programmeurs passeront classiquement par une série d'erreurs communes de conception. Ceci se produit souvent du fait des faibles remontées des experts pendant la conception et l'implémentation des premiers projets, parce qu'aucun expert n'existe encore au sein de l'entreprise et qu'il peut y avoir de la résistance à l'engagement de consultants. Il est facile de s'apercevoir que vous mettez en oeuvre la POO trop tôt dans le cycle et prenez une mauvaise direction. Une évidence pour quelqu'un d'expérimenté dans le langage peut être un sujet de grande débat interne pour un débutant. Une grande part de ce traumatisme peut être évitée en employant un expert extérieur expérimenté pour la formation et la tutelle.

D'autre part, le fait qu'il soit facile de faire ces erreurs de conception montre l'inconvénient principal du C++ : sa compatibilité ascendante avec le C (naturellement, c'est également sa principale force). Pour accomplir l'exploit de pouvoir compiler du code C, le langage a dû faire quelques compromis, qui ont eu comme conséquence un certain nombre de "coins sombres". Ils existent, et constituent une grande partie de la courbe d'apprentissage du langage. Dans ce livre et le volume suivant (et dans d'autres livres ; voir l'annexe C), j'essaierai d'indiquer la plupart des pièges que vous êtes susceptibles de rencontrer en travaillant en C++. Vous devriez toujours être conscient qu'il y a quelques trous dans le filet de sécurité.

1.13 - Résumé

Ce chapitre tente de vous donner un aperçu des sujets couverts par la programmation orientée objet et le C++ (les raisons qui font que la POO est particulière, de même que le C++), les concepts des méthodologies de la POO, et finalement le genre de problèmes que vous rencontrerez quand vous migrerez dans votre entreprise à la programmation orientée objet et le C++.

La POO et le C++ ne sont pas forcément destinés à tout le monde. Il est important d'évaluer ses besoins et décider si le C++ satisfera au mieux ces besoins, ou si un autre système de programmation ne conviendrait pas mieux (celui qu'on utilise actuellement y compris). Si on connaît ses besoins futurs et qu'ils impliquent des contraintes spécifiques non satisfaites par le C++, alors on se doit d'étudier les alternatives existantes. En particulier, je recommande de jeter un oeil à Java (http://java.sun.com) et à Python (http://www.Python.org).Et même si finalement le C++ est retenu, on saura au moins quelles étaient les options et les raisons de ce choix.

On sait à quoi ressemble un programme procédural : des définitions de données et des appels de fonctions. Pour

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

trouver le sens d'un tel programme il faut se plonger dans la chaîne des appels de fonctions et des concepts de bas niveau pour se représenter le modèle du programme. C'est la raison pour laquelle on a besoin de représentations intermédiaires quand on conçoit des programmes procéduraux - par nature, ces programmes tendent à être confus car le code utilise des termes plus orientés vers la machine que vers le problème qu'on tente de résoudre.

Parce que le C++ introduit de nombreux nouveaux concepts au langage C, on pourrait se dire que la fonction **main()**dans un programme C++ sera bien plus compliquée que son équivalent dans un programme C. On sera agréablement surpris de constater qu'un programme C++ bien écrit est généralement beaucoup plus simple et facile à comprendre que son équivalent en C. On n'y voit que les définitions des objets qui représentent les concepts de l'espace problème (plutôt que leur représentation dans l'espace machine) et les messages envoyés à ces objets pour représenter les activités dans cet espace. L'un des avantages de la POO est qu'avec un programme bien conçu, il est facile de comprendre le code en le lisant. De plus, il y a généralement moins de code, car beaucoup de problèmes sont résolus en réutilisant du code existant dans des bibliothèques.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

2 - Construire et utiliser les objets

Ce chapitre va introduire suffisamment de syntaxe C++ et de concepts de programmation pour vous permettre d'écrire et de lancer des programmes simples orientés objet. Dans le chapitre suivant nous verrons en détail la syntaxe de base du C et du C++.

En lisant ce chapitre en premier, vous acquerrez une idée générale sur ce qu'est la programmation avec les objets en C++, et vous découvrirez également quelques-unes des raisons de l'enthousiasme entourant ce langage. Cela devrait être suffisant pour que vous puissiez aborder le chapitre 3, qui est un peu plus consistant du fait qu'il contient beaucoup de détails sur le langage C.

Le type de données personnalisé, ou *classe*, est ce qui distingue le C++ des langages de programmation procéduraux traditionnels. Une classe est un nouveau type de données que vous ou un tiers créez pour résoudre un type particulier de problème. Une fois qu'une classe est créée, n'importe qui peut l'employer sans connaître les détails de son fonctionnement, ou comment les classes sont construites. Ce chapitre traite des classes comme s'il s'agissait simplement d'autres types de données intégrés, disponible à l'usage dans les programmes.

Les classes qu'un tiers a créé sont typiquement empaquetées dans une bibliothèque. Ce chapitre utilise plusieurs des bibliothèques de classes disponibles avec toutes les implémentations C++. Une bibliothèque standard particulièrement importante, iostreams, vous permet (entre autres choses) de lire dans des fichiers et au clavier, et d'écrire dans des fichiers ou sur l'affichage. Vous verrez également la très utile classe **string**, et le conteneur **vector**de la bibliothèque standard du C++. Vers la fin de ce chapitre, vous verrez à quel point il est facile d'utiliser une bibliothèque prédéfinie de classes.

Afin de créer votre premier programme, vous devez comprendre les outils utilisés pour construire des applications.

2.1 - Le processus de traduction du langage

Tous les langages informatiques sont traduits à partir de quelque chose d'aisé à comprendre pour un humain (le *code source*) en quelque chose qui peut être exécuté sur un ordinateur (les *instructions machine*). Traditionnellement, les traducteurs se scindent en deux classes : les *interpréteurs*et les *compilateurs*.

2.1.1 - Les interpréteurs

Un interpréteur traduit le code source en activités (lesquelles peuvent être constituées de groupes d'instructions machine) et exécute immédiatement ces activités. Le BASIC, par exemple, a été un langage interprété très populaire. Traditionnellement, les interpréteurs BASIC traduisent et exécutent une ligne à la fois, puis oublient que la ligne a été traduite. Cela fait qu'ils sont lents, puisqu'ils doivent retraduire tout le code répété. Le BASIC a été également compilé, pour la rapidité. Des interpréteurs plus modernes, comme ceux du langage Python, traduisent le programme entier dans un langage intermédiaire qui est alors exécuté par un interpréteur beaucoup plus rapide La frontière entre les compilateurs et les interpréteurs tend à se brouiller, particulièrement avec Python, qui possède la puissance d'un langage compilé et beaucoup de ses fonctionnalités, mais la rapidité de modification d'un langage interprété.

Les interpréteurs ont beaucoup d'avantages. La transition entre l'écriture du code et son exécution est presque immédiate, et le code source est toujours disponible ainsi l'interpréteur peut être beaucoup plus spécifique quand une erreur se produit. Les avantages souvent cités pour les interpréteurs sont la facilité d'interaction et la vitesse de développement (mais pas nécessairement d'exécution) des programmes.

Les langages interprétés ont souvent de graves limitations lors de la réalisation de grands projets (Python semble être une exception en cela). L'interpréteur (ou une version réduite) doit toujours être en mémoire pour exécuter le

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

code, et même l'interpréteur le plus rapide introduira d'inacceptables restrictions de vitesse. La plupart des interpréteurs requièrent que la totalité du code source soit passé à l'interpréteur en une fois. Non seulement cela introduit une limitation spatiale, mais cela entraîne également plus de bogues difficiles à résoudre si le langage ne fournit pas de facilités pour localiser les effets des différentes parties du code.

2.1.2 - Les compilateurs

Un compilateur traduit le code source directement en langage assembleur ou en instructions machine. L'éventuel produit fini est un fichier ou des fichiers contenant le code machine. C'est un processus complexe, nécessitant généralement plusieurs étapes. La transition entre l'écriture du code et son exécution est significativement plus longue avec un compilateur.

En fonction de la perspicacité du créateur du compilateur, les programmes générés par un compilateur tendent à utiliser beaucoup moins d'espace pour s'exécuter, et s'exécutent beaucoup plus rapidement. Bien que la taille et la vitesse soient probablement les raisons les plus citées d'utilisation des compilateurs, dans nombre de situations ce ne sont pas les raisons les plus importantes. Certains langages (comme le C) sont conçus pour autoriser la compilation séparée de certaines parties du programme. Ces parties sont éventuellement combinées en un programme *exécutable*final par un outil appelé *éditeur de liens*(*linker*). Ce processus est appelé *compilation séparée*.

La compilation séparée a moult avantages. Un programme qui, en prenant tout en une fois, excède les limites du compilateur ou de l'environnement de compilation peut être compilé par morceaux. Les programmes peuvent être construits et testés morceau par morceau. Une fois qu'un morceau fonctionne, il peut être sauvegardé et traité comme un module. Les collections de morceaux testés et validés peuvent être combinées en *bibliothèques*pour être utilisés par d'autres programmeurs. Pendant que chaque morceau est créé, la complexité des autres morceaux est cachée. Tous ces dispositifs permettent la création de programmes volumineux Python est encore une exception, car il permet également la compilation séparée..

Les dispositifs de débogage des compilateurs se sont sensiblement améliorés ces derniers temps. Les compilateurs de première génération généraient seulement du code machine, et le programmeur insérait des rapports d'impression pour voir ce qui se passait. Ce n'est pas toujours efficace. Les compilateurs modernes peuvent insérer des informations à propos du code source dans le programme exécutable. Ces informations sont utilisées par de puissants *débogueurs de haut-niveau*pour montrer exactement ce qui se passe en traçant la progression dans le code source.

Quelques compilateurs abordent le problème de la vitesse de compilation en exécutant une *compilation en mémoire*. La plupart des compilateurs fonctionnent avec des fichiers, les lisant et les écrivant à chaque étape du processus de compilation. Les compilateurs résidents gardent le programme de compilation dans la RAM. Pour les petits programmes, cela peut sembler aussi réactif qu'un interpréteur.

2.1.3 - Le processus de compilation

Pour programmer en C et C++ vous avez besoin de comprendre les étapes et les outils du processus de compilation. Certains langages (le C et le C++, en particulier) débutent la compilation en exécutant un *préprocesseur*sur le code source. Le préprocesseur est un programme simple qui remplace les modèles dans le code source par d'autres modèles que le programmeur a défini (en utilisant les *directives du préprocesseur*). Les directives du préprocesseur sont utilisées pour limiter les frappes et augmenter la lisibilité du code. (Plus loin dans le livre vous apprendrez comment la conception du C++ est faite pour décourager une grande partie de l'utilisation du préprocesseur, puisqu'elle peut causer les bogues subtils.) Le code prétraité est souvent écrit dans un fichier intermédiaire.

Les compilateurs travaillent souvent en deux temps. La première passe *décompose*le code prétraité. Le

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

compilateur sépare le code source en petites unités et l'organise dans une structure appelé *arbre*. Dans l'expression " **A + B**" les éléments " **A**", " **+**", et " **B**" sont des feuilles de l'arbre de décomposition.

Un *optimisateur global*est parfois utilisé entre la première et la deuxième passe pour produire un code plus petit et plus rapide.

Dans la seconde passe, le *générateur de code*parcourt l'arbre de décomposition et génère soit du code en langage assembleur soit du code machine pour les noeuds de l'arbre. Si le générateur de code produit du code assembleur, l'assembleur doit être exécuté. Le résultat final dans les deux cas est un module objet (un fichier dont l'extension est typiquement **.o**ou **.obj**). Un *optimiseur à lucarne*(*peephole optimizer*) est parfois utilisé dans la deuxième passe pour rechercher des morceaux de code contenant des instructions de langage assembleur redondantes.

L'utilisation du mot "objet" pour décrire les morceaux du code machine est un artefact regrettable. Le mot fût employé avant que la programmation orientée objet ne se soit généralisée. "Objet" est utilisé dans le même sens que "but" lorsqu'on parle de compilation, alors qu'en programmation orientée objet cela désigne "une chose avec une frontière".

L' *éditeur de liens*combine une liste de modules objets en un programme exécutable qui peut être chargé et lancé par le système d'exploitation. Quand une fonction d'un module objet fait référence à une fonction ou une variable d'un autre module objet, l'éditeur de liens résout ces références ; cela assure que toutes les fonctions et les données externes dont vous déclarez l'existence pendant la compilation existent. L'éditeur de liens ajoute également un module objet spécial pour accomplir les activités du démarrage.

L'éditeur de liens peut faire des recherches dans des fichiers spéciaux appelés *bibliothèques*afin de résoudre toutes les références. Une bibliothèque contient une collection de modules objets dans un fichier unique. Une bibliothèque est créée et maintenue par un programme appelé *bibliothécaire*(*librarian*).

Vérification statique du type

Le compilateur exécute la *vérification de type*pendant la première passe. La vérification de type teste l'utilisation appropriée des arguments dans les fonctions et empêche beaucoup de sortes d'erreurs de programmation. Puisque la vérification de type se produit pendant la compilation et non à l'exécution du programme, elle est appelé *vérification statique du type*.

Certains langages orientés objet (notamment Java) font des vérifications de type pendant l'exécution (*vérification dynamique du type*). Si elle est combinée à la vérification statique du type, la vérification dynamique du type est plus puissante que la vérification statique seule. Cependant, cela ajoute également un coût supplémentaire à l'exécution du programme.

Le C++ utilise la vérification statique de type car le langage ne peut assumer aucun support d'exécution particulier en cas de mauvaises opérations. La vérification statique du type notifie au programmeur les mauvaises utilisations de types pendant la compilation, et ainsi maximise la vitesse d'exécution. En apprenant le C++, vous verrez que la plupart des décisions de conception du langage favorisent ce genre de rapidité, programmation axée sur la production pour laquelle le langage C est célèbre.

Vous pouvez désactiver la vérification statique du type en C++. Vous pouvez également mettre en oeuvre votre propre vérification dynamique de type - vous avez seulement besoin d'écrire le code.

2.2 - Outils de compilation séparée

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

La compilation séparée est particulièrement importante dans le développement de grands projets. En C et C++, un programme peut être créé par petits morceaux maniables, testés indépendamment. L'outil primordial pour séparer un programme en morceaux est la capacité de créer des sous-routines ou des sous-programmes nommés. En C et C++, un sous-programme est appelé *fonction*, et les fonctions sont les parties du code qui peuvent être placées dans différents fichiers, permettant une compilation séparée. Autrement dit, la fonction est l'unité atomique du code, puisque vous ne pouvez pas avoir une partie d'une fonction dans un fichier et une autre partie dans un fichier différent ; la fonction entière doit être placée dans un fichier unique (mais les fichiers peuvent contenir plus d'une fonction).

Quand vous appelez une fonction, vous lui passez typiquement des *arguments*, qui sont des valeurs que la fonction utilise pendant son exécution. Quand une fonction se termine, vous récupérez typiquement une *valeur de retour*, une valeur que la fonction vous retourne comme résultat. Il est aussi possible d'écrire des fonctions qui ne prennent aucun argument et qui ne retournent aucune valeur.

Pour créer un programme avec plusieurs fichiers, les fonctions d'un fichier doivent accéder à des fonctions et à des données d'autres fichiers. Lorsqu'il compile un fichier, le compilateur C ou C++ doit connaître les fonctions et données des autres fichiers, en particulier leurs noms et leur emploi correct. Le compilateur s'assure que les fonctions et les données sont employées correctement. Ce processus "d'indiquer au compilateur" les noms des fonctions et des données externes et ce à quoi elles ressemblent est appelé *déclaration*. Une fois que vous avez déclaré une fonction ou variable, le compilateur sait comment vérifier le code pour s'assurer qu'elle est employée correctement.

2.2.1 - Déclarations vs. définitions

Il est important de comprendre la différence entre *déclarations*et *définitions*, parce que ces termes seront utilisés précisément partout dans le livre. Par essence, tous les programmes C et C++ exigent des déclarations. Avant que vous puissiez écrire votre premier programme, vous devez comprendre la manière convenable d'écrire une déclaration.

Une *déclaration*introduit un nom - un identifiant - pour le compilateur. Elle indique au compilateur "Cette fonction ou cette variable existe quelque part, et voici à quoi elle devrait ressembler." Une *définition*, d'un autre côté, dit : "Faire cette variable ici" ou "Faire cette fonction ici". Elle alloue de l'espace pour le nom. Ce principe s'applique aux variables comme aux fonction ; dans tous les cas, le compilateur alloue de l'espace au moment de la définition. Pour une variable, le compilateur détermine sa taille et entraîne la réservation de l'espace en mémoire pour contenir les données de cette variable. Pour une fonction, le compilateur génère le code, qui finit par occuper de l'espace en mémoire.

Vous pouvez déclarer une variable ou une fonction dans beaucoup d'endroits différents, mais il doit y avoir seulement une définition en C et C++ (ceci s'appelle parfois l'ODR : *one-definition rule*). Quand l'éditeur de liens unit tous les modules objets, il se plaindra généralement s'il trouve plus d'une définition pour la même fonction ou variable.

Une définition peut également être une déclaration. Si le compilateur n'a pas vu le nom **x**avant, et que vous définissez **int x;**, le compilateur voit le nom comme une déclaration et alloue son espace de stockage en une seule fois.

Syntaxe de la déclaration de fonction

Une déclaration de fonction en C et C++ donne le nom de fonction, le type des paramètres passés à la fonction, et la valeur de retour de la fonction. Par exemple, voici une déclaration pour une fonction appelée **func1( )**qui prend deux arguments entiers (les nombres entiers sont annoncés en C/C++ avec le mot-clé **int**) et retourne un entier :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
int func1(int,int);
```

Le premier mot-clé que vous voyez est la valeur de retour elle-même **int**. Les paramètres sont entourés de parenthèses après le nom de la fonction, dans l'ordre dans lequel ils sont utilisés. Le point virgule indique la fin d'une instruction; dans l'exemple, il indique au compilateur "c'est tout - il n'y a pas de définition de fonction ici !"

Les déclarations du C et du C++ essaient d'imiter la forme d'utilisation de l'élément. Par exemple, si **a**est un autre entier la fonction ci-dessus peut être utilisée de cette façon :

```
a = func1(2,3);
```

Puisque **func1**( ) retourne un entier, le compilateur C ou C++ vérifiera l'utilisation de **func1( )**pour s'assurer que **a**peut accepter la valeur de retour de la fonction et que les arguments sont appropriés.

Les arguments dans les déclarations de fonction peuvent avoir des noms. le compilateur ignore les noms, mais ils peuvent être utiles en tant que dispositifs mnémoniques pour l'utilisateur. Par exemple, nous pouvons déclarer **func1( )**d'une façon différente qui a la même signification :

```
int func1(int taille, int largeur);
```

Un piège

Il y a une différence significative entre le C et le C++ pour les fonctions dont la liste d'arguments est vide. En C, la déclaration :

```
int func2();
```

signifie "une fonction avec n'importe quel nombre et type d'arguments." Cela empêche la vérification du type, alors qu'en C++ cela signifie "une fonction sans argument."

Définitions de fonction

Les définitions de fonction ressemblent aux déclarations de fonction sauf qu'elles ont un corps. Un corps est un ensemble d'instructions entouré d'accolades. Les accolades annoncent le début et la fin d'un bloc de code. Pour donner une définition à **func1( )**qui soit un corps vide (un corps ne contenant aucun code), écrivez :

```
int func1(int taille, int largeur) { }
```

Notez que dans la définition de fonction, les accolades remplacent le point-virgule. Puisque les accolades entourent une instruction ou un groupe d'instructions, vous n'avez pas besoin d'un point-virgule. Notez aussi que les paramètres dans la définition de fonction doivent avoir des noms si vous voulez les utiliser dans le corps de la fonction (comme ils ne sont jamais utilisés dans l'exemple, c'est optionnel).

Syntaxe de la déclaration de variable

La signification attribuée à l'expression "déclaration de variable" a historiquement été déroutante et contradictoire, et il est important que vous compreniez la définition correcte, ainsi vous pouvez lire le code correctement. Une

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

déclaration de variable indique au compilateur à quoi une variable ressemble. Elle dit, "Je sais que tu n'as pas vu ce nom avant, mais je promets qu'il existe quelque part, et que c'est une variable du type X."

Dans une déclaration de fonction, vous donnez un type (la valeur de retour), le nom de la fonction, la liste des arguments, et un point-virgule. C'est suffisant pour que le compilateur comprenne que c'est une déclaration et ce à quoi la fonction devrait ressembler. de la même manière, une déclaration de variable pourrait être un type suivi d'un nom. Par exemple :

```
int a;
```

pourrait déclarer la variable **a**comme un entier, en utilisant la logique ci-dessus. Voilà le conflit : il y a assez d'information dans le code ci-dessus pour que le compilateur crée l'espace pour un entier appelé **a**, et c'est ce qui se produit. Pour résoudre ce dilemme, un mot-clé était nécessaire en C et C++ pour dire "ceci est seulement une déclaration ; elle a été définie ailleurs." Le mot-clé est **extern**. Il peut signifier que la définition est **extern**e au fichier, ou que la définition a lieu plus tard dans le fichier.

Déclarer une variable sans la définir signifie utiliser le mot-clé **extern**avant une description de la variable, comme ceci :

```
extern int a;
```

externpeut aussi s'appliquer aux déclarations de fonctions. Pour **func1( )**, ça ressemble à :

```
extern int func1(int taille, int largeur);
```

Cette instruction est équivalente aux précédentes déclarations de **func1( )**. Puisqu'il n'y a pas de corps de fonction, le compilateur doit la traiter comme une déclaration de fonction plutôt qu'une définition de fonction. Le mot-clé **extern**est donc superflu et optionnel pour les déclarations de fonctions. Il est vraisemblablement regrettable que les concepteurs du C n'aient pas requis l'utilisation d' **extern**pour les déclarations de fonctions ; cela aurait été plus cohérent et moins déroutant (mais cela aurait nécessité plus de frappes, ce qui explique probablement la décision).

Voici quelques exemples de déclarations :

```
//: C02:Declare.cpp
// Exemples de déclaration & définition
extern int i; // Déclaration sans définition
extern float f(float); // Déclaration de fonction

float b;  // Déclaration & définition
float f(float a) {  // Définition
  return a + 1.0;
}

int i; // Définition
int h(int x) { // Déclaration & définition
  return x + 1;
}

int main() {
  b = 1.0;
  i = 2;
  f(b);
  h(i);
} ///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Dans les déclarations de fonctions, les identifiants d'argument sont optionnels. Dans les définitions, ils sont requis (les identifiants sont requis uniquement en C, pas en C++).

Inclusion d'en-têtes

La plupart des bibliothèques contiennent un nombre significatif de fonctions et de variables. Pour économiser le travail et assurer la cohérence quand sont faites des déclarations externes pour ces éléments, le C et le C++ utilisent un dispositif appelé le *fichier d'en-tête*. Un fichier d'en-tête est un fichier contenant les déclarations externes d'une bibliothèque; il a par convention une extension de nom de fichier "h", comme **headerfile.h**. (Vous pourrez également voir certains codes plus anciens utilisant des extensions différentes, comme **.hxx**ou **.hpp**, mais cela devient rare.)

Le programmeur qui crée la bibliothèque fournis le fichier d'en-tête. Pour déclarer les fonctions et variables externes de la bibliothèque, l'utilisateur inclus simplement le fichier d'en-tête. Pour inclure un fichier d'en-tête, utilisez la directive du préprocesseur **#include**. Elle demande au préprocesseur d'ouvrir le fichier d'en-tête cité et d'insérer son contenu là où l'instruction **#include**apparaît. Un **#include**peut citer un fichier de deux façons : entre équerres (**< >**) ou entre guillemets doubles.

Les noms de fichiers entre équerres, comme :

```
#include <entete>
```

font rechercher le fichier par le préprocesseur d'une manière propre à votre implémentation, mais typiquement il y a une sorte de "chemin de recherche des inclusions" que vous spécifiez dans votre environnement ou en ligne de commandes du compilateur. Le mécanisme de définition du chemin de recherche varie selon les machines, les systèmes d'exploitation, et les implémentations du C++, et peut nécessiter quelque investigation de votre part.

Les noms de fichiers entre guillemets doubles, comme :

```
#include "local.h"
```

indiquent au préprocesseur de chercher le fichier (selon les spécifications) "d'une manière définie par l'implémentation." Ceci signifie typiquement de chercher le fichier relativement au répertoire courant. Si le fichier n'est pas trouvé, alors la directive d'inclusion est retraitée comme s'ils s'agissait d'équerres et non de guillemets.

Pour inclure le fichier d'en-tête iostream, vous écrivez :

```
#include <iostream>
```

Le préprocesseur trouvera le fichier d'en-tête iostream (souvent dans un sous-répertoire appelé "include") et l'insérera.

Format d'inclusion du Standard C++

Alors que le C++ évoluait, les différents fournisseurs de compilateurs ont choisi différentes extensions pour les noms de fichiers. En outre, les divers systèmes d'exploitation ont différentes contraintes sur les noms de fichiers, en particulier la taille du nom. Ces sujets ont entraîné des problèmes de portabilité du code source. Pour arrondir les angles, le standard utilise un format qui permet des noms de fichiers plus longs que les huit caractères notoires et élimine l'extension. Par exemple, au lieu du vieux style d'inclusion **iostream.h**, qui ressemble à :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
#include <iostream.h>
```

vous pouvez maintenant écrire :

```
#include <iostream>
```

L'interprète peut mettre en application les instructions d'inclusion d'une façon qui convient aux besoins de ces compilateur et système d'exploitation particuliers, si nécessaire en tronquant le nom et en ajoutant une extension. Bien sûr, vous pouvez aussi copier les en-têtes donnés par le fournisseur de votre compilateur dans ceux sans extensions si vous voulez utiliser ce style avant que le fournisseur ne le supporte.

Les bibliothèques héritées du C sont toujours disponibles avec l'extension traditionnelle " **.h**". Cependant, vous pouvez aussi les utiliser avec le style d'inclusion plus moderne du C++ en ajoutant un " **c**" devant le nom. Ainsi :

```
#include <stdio.h>;
#include <stdlib.h>
```

devient :

```
#include <cstdio>
#include <cstdlib>
```

Et ainsi de suite, pour tous les en-têtes standards du C. Cela apporte une distinction agréable au lecteur, indiquant quand vous employez des bibliothèques C et non C++.

L'effet du nouveau format d'inclusion n'est pas identique à l'ancien : utiliser le **.h**vous donne la version plus ancienne, sans modèles, et omettre le **.h**vous donne la nouvelle version, avec modèles. Vous aurez généralement des problèmes si vous essayez d'entremêler les deux formes dans un même programme.

2.2.2 - Edition des liens

L'éditeur de liens rassemble les modules objets (qui utilisent souvent des extensions de nom de fichier comme **.o**ou **.obj**), générés par le compilateur, dans un programme exécutable que le système d'exploitation peut charger et démarrer. C'est la dernière phase du processus de compilation.

Les caractéristiques de l'éditeur de liens changent d'un système à l'autre. Généralement, vous donnez simplement à l'éditeur de liens les noms des modules objets et des bibliothèques que vous voulez lier ensemble, et le nom de l'exécutable, et il va fonctionner. Certains systèmes nécessitent d'appeler l'éditeur de liens vous-même. Avec la plupart des éditeurs C++, vous appelez l'éditeur de liens à travers le compilateur C++. Dans beaucoup de situations, l'éditeur de liens est appelé sans que vous le voyez.

Certains éditeurs de liens plus anciens ne chercheront pas les fichiers objets et les bibliothèques plus d'une fois, et ils cherchent dans la liste que vous leur donnez de gauche à droite. Ceci signifie que l'ordre des fichiers objets et des bibliothèques peut être important. Si vous avez un problème mystérieux qui n'apparaît pas avant l'édition des liens, une cause possible est l'ordre dans lequel les fichiers sont donnés à l'éditeur de liens.

2.2.3 - Utilisation des bibliothèques

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Maintenant que vous connaissez la terminologie de base, vous pouvez comprendre comment utiliser une bibliothèque :

1 Incluez le fichier d'en-tête de la bibliothèque.
2 Utilisez les fonctions et variables de la bibliothèque.
3 Liez la bibliothèque dans le programme exécutable.

Ces étapes s'appliquent également quand les modules objets ne sont pas combinés dans une bibliothèque. Inclure un fichier d'en-tête et lier les modules objets sont les étapes de base de la compilation séparée en C et C++.

Comment l'éditeur de liens cherche-t-il une bibliothèque ?

Quand vous faites une référence externe à une fonction ou variable en C ou C++, l'éditeur de liens, lorsqu'il rencontre cette référence, peut faire deux choses. S'il n'a pas encore rencontré la définition de la fonction ou variable, il ajoute l'identifiant à sa liste des "références non résolues". Si l'éditeur de liens a déjà rencontré la définition, la référence est résolue.

Si l'éditeur de liens ne peut pas trouver la définition dans la liste des modules objets, il cherche dans les bibliothèques. Les bibliothèques ont une sorte d'index de telle sorte que l'éditeur de liens n'a pas besoin de parcourir tous les modules objets de la bibliothèque - il regarde juste l'index. Quand l'éditeur de liens trouve une définition dans une bibliothèque, le module objet complet, et non seulement la définition de fonction, est lié dans le programme exécutable. Notez que la bibliothèque n'est pas liée dans son ensemble, seulement le module objet de la bibliothèque qui contient la définition que vous voulez (sinon les programmes seraient inutilement volumineux). Si vous voulez minimiser la taille du programme exécutable, vous pouvez imaginer mettre une seule fonction par fichier du code source quand vous construisez vos propres bibliothèques. Cela nécessite plus de rédaction. Je recommanderais l'utilisation de Perl ou Python pour automatiser cette tâche en tant qu'élément de votre processus de création des bibliothèques (voir www.Perl.org ou www.Python.org),mais ça peut être utile aux utilisateurs.

Comme l'éditeur de liens cherche les fichiers dans l'ordre dans lequel vous les listez, vous pouvez préempter l'utilisation d'une fonction de bibliothèque en insérant un fichier avec votre propre fonction, utilisant le même nom de fonction, dans la liste avant l'apparition du nom de la bibliothèque. Puisque l'éditeur de liens résoudra toutes les références à cette fonction en utilisant votre fonction avant de chercher dans la bibliothèque, votre fonction sera utilisée à la place de la fonction de la bibliothèque. Notez que cela peut aussi être un bogue, et que les espaces de nommage du C++ préviennent ce genre de choses.

Ajouts cachés

Quand un programme exécutable C ou C++ est créé, certains éléments sont secrètement liés. L'un d'eux est le module de démarrage, qui contient les routines d'initialisation qui doivent être lancées à chaque fois qu'un programme C ou C++ commence à s'exécuter. Ces routines mettent en place la pile et initialisent certaines variables du programme.

L'éditeur de liens cherche toujours dans la bibliothèque standard les versions compilées de toutes les fonctions "standard" appelées dans le programme. Puisque la librairie standard est toujours recherchée, vous pouvez utiliser tout ce qu'elle contient en incluant simplement le fichier d'en-tête approprié dans votre programme ; vous n'avez pas à indiquer de rechercher dans la bibliothèque standard. Les fonctions iostream, par exemple, sont dans la bibliothèque Standard du C++. Pour les utiliser, vous incluez juste le fichier d'en-tête **<iostream>**.

Si vous utilisez une bibliothèque complémentaire, vous devez explicitement ajouter le nom de la bibliothèque à la liste des fichiers donnés à l'éditeur de liens.

Utilisation des bibliothèques C

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Bien que vous écriviez du code en C++, on ne vous empêchera jamais d'utiliser des fonctions d'une bibliothèque C. En fait, la bibliothèque C complète est incluse par défaut dans le Standard C++. Une quantité énorme de travail a été effectuée pour vous dans ces fonctions, elles peuvent donc vous faire gagner beaucoup de temps.

Ce livre utilisera les fonctions de la bibliothèque Standard C++ (et donc aussi le standard C) par commodité, mais seules les fonctions de la bibliothèque *standard*seront utilisées, pour assurer la portabilité des programmes. Dans les quelques cas pour lesquels des fonctions de bibliothèques qui ne sont pas dans le standard C++ doivent être utilisées, nous ferons tous les efforts possibles pour utiliser des fonctions conformes POSIX. POSIX est une norme basée sur un effort de standardisation d'Unix qui inclut des fonctions qui dépassent la portée de la bibliothèque C++. Vous pouvez généralement espérer trouver des fonctions POSIX sur les plateformes Unix (en particulier, Linux), et souvent sous DOS/Windows. Par exemple, si vous utilisez le multithreading, vous partez d'autant mieux que vous utilisez la bibliothèque de thread POSIX parce que votre code sera alors plus facile à comprendre, porter et maintenir (et la bibliothèque de thread POSIX utilisera généralement simplement les possibilités de thread sous-jacentes du système d'exploitation, si elles existent).

2.3 - Votre premier programme C++

Vous en savez maintenant presque suffisamment sur les bases pour créer et compiler un programme. Le programme utilisera les classes iostream (flux d'entrée/sortie) du Standard C++. Elles lisent et écrivent dans des fichiers et l'entrée et la sortie "standard" (qui normalement reposent sur la console, mais peuvent être redirigés vers des fichiers ou des périphériques). Dans ce programme simple, un objet flux sera utilisé pour écrire un message à l'écran.

2.3.1 - Utilisation de la classe iostream

Pour déclarer les fonctions et données externes de la classe iostream, incluez le fichier d'en-tête avec l'instruction

```
#include <iostream>
```

Le premier programme utilise le concept de sortie standard, qui signifie "un endroit universel pour envoyer la sortie". Vous verrez d'autres exemples utilisant la sortie standard de différentes façons, mais ici elle ira simplement sur la console. La bibliothèque iostream définit automatiquement une variable (un objet) appelée **cout**qui accepte toutes les données liées à la sortie standard.

Pour envoyer des données à la sortie standard, vous utilisez l'opérateur **< <**. Les programmeurs C connaissent cet opérateur comme celui du "décalage des bits à gauche", ce qui sera décrit dans le prochain chapitre. Ca suffit pour dire qu'un décalage des bits à gauche n'a rien à voir avec la sortie. Cependant, le C++ permet la *surcharge*des opérateurs. Quand vous surchargez un opérateur, vous donnez un nouveau sens à cet opérateur quand il est utilisé avec un objet d'un type donné. Avec les objets iostream, l'opérateur **< <**signifie "envoyer à". Par exemple :

```
cout << "salut !";
```

envoie la chaîne de caractères "salut !" à l'objet appelé **cout**(qui est le raccourci de "console output" - sortie de la console)

Ca fait assez de surcharge d'opérateur pour commencer. Le chapitre 12 couvre la surcharge des opérateurs en détail.

2.3.2 - Espaces de noms

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Comme mentionné dans le chapitre 1, un des problèmes rencontré dans le langage C est que vous "épuisez les noms" pour les fonctions et les identifiants quand vos programmes atteignent une certaine taille. Bien sûr, vous n'épuisez pas vraiment les noms ; cependant, il devient difficile d'en trouver de nouveaux après un certain temps. Plus important, quand un programme atteint une certaine taille il est généralement coupé en morceaux, chacun étant construit et maintenu par une personne ou un groupe différent. Puisque le C n'a en réalité qu'un seul domaine où tous les identifiants et noms de fonction existent, cela signifie que tous les développeurs doivent faire attention à ne pas utiliser accidentellement les mêmes noms dans des situations où ils peuvent entrer en conflit. Cela devient rapidement fastidieux, chronophage, et, en fin de compte, cher.

Le Standard C++ contient un mécanisme pour éviter ces heurts : le mot-clé **namespace**. Chaque ensemble de définitions C++ d'une bibliothèque ou d'un programme est "enveloppé" dans un espace de nom, et si une autre définition a un nom identique, mais dans un espace de nom différent, alors il n'y a pas de conflit.

Les espaces de noms sont un outil pratique et utile, mais leur présence signifie que vous devez vous rendre compte de leur présence avant que vous ne puissiez écrire le moindre programme. Si vous incluez simplement un fichier d'en-tête et que vous utilisez des fonctions ou objets de cet en-tête, vous aurez probablement des erreurs bizarres quand vous essaierez de compiler le programme, dues au fait que le compilateur ne peut trouver aucune des déclarations des éléments dont vous avez justement inclus le fichier d'en-tête ! Après avoir vu ce message plusieurs fois, vous deviendrez familier avec sa signification (qui est "Vous avez inclus le fichier d'en-tête mais toutes les déclarations sont dans un espace de nom et vous n'avez pas signalé au compilateur que vous vouliez utiliser les déclarations de cet espace de nom").

Il y a un mot-clé qui vous permet de dire "Je veux utiliser les déclarations et/ou définitions de cet espace de nom". Ce mot-clé, de façon assez appropriée, est **using**(utiliser). Toutes les bibliothèques du Standard C++ sont enveloppées dans un espace de nom unique, **std**(pour "standard"). Puisque ce livre utilise presque exclusivement les bibliothèques standards, vous verrez la *directive using*dans presque tous les programmes :

```
using namespace std;
```

Cela signifie que vous voulez exposer tous les éléments de l'espace de nom appelé **std**. Après cette instruction, vous n'avez plus à vous préoccuper de l'appartenance à un espace de nom de votre composant particulier de bibliothèque, puisque la directive **using**rend cet espace de nom disponible tout au long du fichier où la directive **using**a été écrite.

Exposer tous les éléments d'un espace de nom après que quelqu'un ait pris la peine de les masquer peut paraître un peu contre-productif, et en fait vous devez faire attention à moins penser le faire (comme vous allez l'apprendre plus tard dans ce livre). Cependant, la directive **using**expose seulement ces noms pour le fichier en cours, donc ce n'est pas si drastique qu'on peut le croire à première vue. (Mais pensez-y à deux fois avant de le faire dans un fichier d'en-tête - *c'est*risqué.)

Il y a un rapport entre les espaces de nom et la façon dont les fichiers d'en-tête sont inclus. Avant que l'inclusion moderne des fichiers d'en-tête soit standardisée (sans la fin " **.h**", comme dans **<iostream>**), la méthode classique d'inclusion d'un fichier d'en-tête était avec le " **.h**", comme **<iostream.h>**. A ce moment là, les espaces de nom ne faisaient pas non plus partie du langage. Donc pour assurer la compatibilité ascendante avec le code existant, si vous dites

```
#include <iostream.h>
```

cela signifie

```
#include <iostream>
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
using namespace std;
```

Cependant, dans ce livre, le format d'inclusion standard sera utilisé (sans le " **.h**") et donc la directive **using**doit être explicite.

Pour l'instant, c'est tout ce que vous avez besoin de savoir sur les espaces de nom, mais dans le chapitre 10 le sujet est couvert plus en détail.

2.3.3 - Principes fondamentaux de structure de programme

Un programme C ou C++ est une collection de variables, de définitions de fonctions, et d'appels de fonctions. Quand le programme démarre, il exécute un code d'initialisation et appelle une fonction spéciale, " **main( )**". Vous mettez le code basique du programme dedans.

Comme mentionné plus tôt, une définition de fonction consiste en un type de valeur de retour (qui doit être spécifié en C++), un nom de fonction, une liste d'arguments entre parenthèses, et le code de la fonction contenu dans des accolades. Voici un échantillon de définition de fonction :

```
int fonction() {
  // Code de la fonction (ceci est un commentaire)
}
```

La fonction ci-dessus a une liste d'arguments vide et son corps contient uniquement un commentaire.

Il peut y avoir de nombreuses paires d'accolades dans une définition de fonction, mais il doit toujours y en avoir au moins une paire entourant le corps de la fonction. Puisque **main( )**est une fonction, elle doit respecter ces règles. En C++, **main( )**a toujours le type de valeur de retour **int**.

Le C et le C++ sont des langages de forme libre. A de rares exceptions près, le compilateur ignore les retours à la ligne et les espaces, il doit donc avoir une certaine manière de déterminer la fin d'une instruction. Les instructions sont délimitées par les points-virgules.

Les commentaires C commencent avec **/***et finissent avec ***/**. Ils peuvent comprendre des retours à la ligne. Le C++ utilise les commentaires du style C et a un type de commentaire supplémentaire : **//**. **//**commence un commentaire qui se termine avec un retour à la ligne. C'est plus pratique que **/* */**pour les commentaires d'une seule ligne, et c'est beaucoup utilisé dans ce livre.

2.3.4 - "Bonjour tout le monde !"

Et maintenant, enfin, le premier programme :

```
//: C02:Hello.cpp
// Dire bonjour en C++
#include <iostream> // Déclaration des flux
using namespace std;

int main() {
  cout << "Bonjour tout le monde ! J'ai "
       << 8 << " ans aujourd'hui !" << endl;
} ///:~
```

L'objet **cout**reçoit une série d'arguments à travers les opérateurs " **<<**". Il écrit ces arguments dans l'ordre

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

gauche-à-droite. La fonction de flux spéciale **endl**restitue la ligne et en crée une nouvelle. Avec les flux d'entrée/sortie, vous pouvez enchaîner une série d'arguments comme indiqué, ce qui rend la classe facile à utiliser.

En C, le texte entre guillemets doubles est classiquement appelé une "string" (chaîne de caractères). Cependant, la bibliothèque du Standard C++ contient une classe puissante appelée **string**pour manipuler le texte, et donc j'utiliserai le terme plus précis *tableau de caractères*pour le texte entre guillemets doubles.

Le compilateur crée un espace de stockage pour les tableaux de caractères et stocke l'équivalent ASCII de chaque caractère dans cet espace. Le compilateur termine automatiquement ce tableau de caractères par un espace supplémentaire contenant la valeur 0 pour indiquer la fin du tableau de caractères.

Dans un tableau de caractères, vous pouvez insérer des caractères spéciaux en utilisant des *séquences d'échappement*. Elles consistent en un antislash (****) suivi par un code spécial. Par exemple, **\n**signifie "nouvelle ligne". Le manuel de votre compilateur ou un guide C local donne l'ensemble complet des séquences d'échappement ; d'autres incluent **\t**(tabulation), ****(antislash), et **\b**(retour arrière).

Notez que l'instruction peut continuer sur plusieurs lignes, et que l'instruction complète se termine avec un point-virgule.

Les arguments tableau de caractère et entier constant sont mêlés ensemble dans l'instruction **cout**ci-dessus. Comme l'opérateur **<<**est surchargé avec diverses significations quand il est utilisé avec **cout**, vous pouvez envoyer à **cout**une gamme d'arguments différents et il "comprendra quoi faire avec le message".

Tout au long du livre vous noterez que la première ligne de chaque fichier sera un commentaire qui commence par les caractères qui annoncent un commentaire (classiquement ***//***), suivis de deux points, et la dernière ligne du listing se terminera avec un commentaire suivi par " ***//:~***". C'est une technique que j'utilise pour permettre une extraction simple de l'information des fichiers de code (le programme pour le faire peut être trouvé dans le Volume 2 de ce livre, sur *www.BruceEckel.com*). La première ligne contient aussi le nom et l'emplacement du fichier, il peut donc être cité dans le texte ou dans d'autres fichiers, et ainsi vous pouvez facilement le trouver dans le code source pour ce livre (qui est téléchargeable sur *www.BruceEckel.com*).

2.3.5 - Lancer le compilateur

Après avoir téléchargé et décompressé le code source du livre, trouvez le programme dans le sous-répertoire **CO2**. Lancez le compilateur avec **Hello.cpp**en argument. Pour de simples programmes d'un fichier comme celui-ci, la plupart des compilateurs mèneront le processus à terme. Par exemple, pour utiliser le compilateur C++ GNU (qui est disponible gratuitement sur internet), vous écrivez :

```
g++ Hello.cpp
```

Les autres compilateurs auront une syntaxe similaire ; consultez la documentation de votre compilateur pour les détails.

2.4 - Plus sur les flux d'entrée-sortie

Jusqu'ici vous avez seulement vu l'aspect le plus rudimentaire de la classe iostream. Le formatage de la sortie disponible avec les flux inclut également des fonctionnalités comme le formatage des nombres en notation décimale, octale et hexadécimale. Voici un autre exemple d'utilisation des flux :

```
//: C02:Stream2.cpp
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
// Plus de fonctionnalités des flux
#include <iostream>
using namespace std;

int main() {
  // Spécifier des formats avec des manipulateurs :
  cout << "un nombre en notation décimale : "
       << dec << 15 << endl;
  cout << "en octale : " << oct << 15 << endl;
  cout << "en hexadécimale : " << hex << 15 << endl;
  cout << "un nombre à virgule flottante : "
       << 3.14159 << endl;
  cout << "un caractère non imprimable (échap) : "
       << char(27) << endl;
} ///:~
```

Cet exemple montre la classe iostream imprimant des nombres en notation décimale, octale, et hexadécimale en utilisant des *manipulateurs*(qui n'écrivent rien, mais modifient l'état du flux en sortie). Le formatage des nombres à virgule flottante est automatiquement déterminé par le compilateur. En plus, chaque caractère peut être envoyé à un objet flux en utilisant un *cast*vers un **char**(un **char**est un type de donnée qui contient un caractère unique). Ce *cast*ressemble à un appel de fonction : **char( )**, avec le code ASCII du caractère. Dans le programme ci-dessus, le **char(27)**envoie un "échap" à **cout**.

2.4.1 - Concaténation de tableaux de caractères

Une fonctionnalité importante du processeur C est la *concaténation de tableaux de caractères*. Cette fonctionnalité est utilisée dans certains exemples de ce livre. Si deux tableaux de caractères entre guillemets sont adjacents, et qu'aucune ponctuation ne les sépare, le compilateur regroupera les tableaux de caractères ensemble dans un unique tableau de caractères. C'est particulièrement utile quand les listes de code ont des restrictions de largeur :

```
                        //: C02:Concat.cpp
// Concaténation de tableaux de caractères
#include <iostream>;
using namespace std;

int main() {
  cout << "C'est vraiment trop long pour être mis "
    "sur une seule ligne mais ça peut être séparé sans "
    "effet indésirable\ntant qu'il n'y a pas "
    "de ponctuation pour séparer les tableaux de caractères "
    "adjacents.\n";
} ///:~
```

A première vue, le code ci-dessus peut ressembler à une erreur puisqu'il n'y a pas le point-virgule familier à la fin de chaque ligne. Souvenez-vous que le C et le C++ sont des langages de forme libre, et bien que vous verrez habituellement un point-virgule à la fin de chaque ligne, le besoin actuel est d'un point virgule à la fin de chaque instruction, et il est possible qu'une instruction s'étende sur plusieurs lignes.

2.4.2 - Lire les entrées

Les classes de flux d'entrée-sortie offrent la possibilité de lire des entrées. L'objet utilisé pour l'entrée standard est **cin**(pour "console input" - entrée de la console). **cin**attend normalement une entrée sur la console, mais cette entrée peut être redirigée à partir d'autres sources. Un exemple de redirection est montré plus loin dans ce chapitre.

L'opérateur de flux d'entrée-sortie utilisé avec **cin**est **>>**. Cet opérateur attend le même type d'entrée que son argument. Par exemple, si vous donnez un argument entier, il attend un entier de la console. Voici un exemple :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                        //: C02:Numconv.cpp
// Convertit une notation décimale en octale et hexadécimale
#include <iostream>
using namespace std;

int main() {
  int number;
  cout << "Entrez un nombre décimal : ";
  cin >> number;
  cout << "Valeur en octal = 0"
       << oct << number << endl;
  cout << "Valeur en hexadécimal = 0x"
       << hex << number << endl;
} ///:~
```

Ce programme convertit un nombre tapé par l'utilisateur dans ses représentations octales et hexadécimales.

2.4.3 - Appeler d'autres programmes

Alors que la manière classique d'appeler un programme qui lit l'entrée standard et écrit sur la sortie standard est un script dans un shell Unix ou un fichier batch du DOS, tout programme peut être appelé à partir d'un programme C ou C++ en utilisant la fonction du Standard C **system( )**, qui est déclarée dans le fichier d'en-tête **<cstdlib>**:

```
                        //: C02:CallHello.cpp
// Appeler un autre programme
#include <cstdlib> // Déclare "system()"
using namespace std;

int main() {
  system("Hello");
} ///:~
```

Pour utiliser la fonction **system( )**, vous lui donnez un tableau de caractères que vous pouvez normalement taper en ligne de commandes du système d'exploitation. Il peut aussi comprendre des arguments de ligne de commande, et le tableau de caractères peut être construit à l'exécution (au lieu de simplement utiliser un tableau de caractères statique comme montré ci- dessus). La commande exécute et contrôle les retours du programme.

Ce programme vous montre à quel point il est facile d'utiliser des fonctions de la bibliothèque C ordinaire en C++ ; incluez simplement le fichier d'en-tête et appelez la fonction. Cette compatibilité ascendante du C au C++ est un grand avantage si vous apprenez le langage en commençant avec une expérience en C.

2.5 - Introduction aux chaînes de caractères

Bien qu'un tableau de caractères puisse être assez utile, il est assez limité. C'est simplement un groupe de caractères en mémoire, mais si vous voulez faire quelque chose avec vous devez gérer tous les moindres détails. Par exemple, la taille d'un tableau de caractères donné est fixe au moment de la compilation. Si vous avez un tableau de caractères et que vous voulez y ajouter quelques caractères supplémentaires, vous devrez comprendre énormément de fonctionnalités (incluant la gestion dynamique de la mémoire, la copie de tableau de caractères, et la concaténation) avant de pouvoir réaliser votre souhait. C'est exactement le genre de chose que l'on aime qu'un objet fasse pour nous.

La classe **string**du Standard C++ est conçue pour prendre en charge (et masquer) toutes les manipulations de bas niveau des tableaux de caractères qui étaient à la charge du developpeur C. Ces manipulations étaient à l'origine de perte de temps et source d'erreurs depuis les débuts du langage C. Ainsi, bien qu'un chapitre entier soit consacré à la classe **string**dans le Volume 2 de ce livre, les *chaînes de caractères*sont si importantes et elles rendent la vie tellement plus simple qu'elles seront introduites ici et utilisées régulièrement dans la première partie

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

du livre.

Pour utiliser les chaînes de caractères, vous incluez le fichier d'en-tête C++ **<string>**. La classe **string**est dans l'espace de nom **std**donc une directive **using**est nécessaire. Du fait de la surcharge des opérateurs, la syntaxe d'utilisation des chaînes de caractères est assez intuitive :

```
                    //: C02:HelloStrings.cpp
// Les bases de la classe string du Standard C++
#include <string>
#include <iostream>
using namespace std;

int main() {
  string s1, s2; // Chaînes de caractères vides
  string s3 = "Bonjour, Monde !"; // Initialisation
  string s4("J'ai"); // Egalement une initialisation
  s2 = "ans aujourd'hui"; // Affectation à une chaîne de caractères
  s1 = s3 + " " + s4; // Combinaison de chaînes de caractères
  s1 += " 8 "; // Ajout à une chaîne de caractères
  cout << s1 + s2 + " !" << endl;
} ///:~
```

Les deux premières chaînes de caractères, **s1**et **s2**, commencent vides, alors que **s3**et **s4**montrent deux méthodes équivalentes d'initialisation des objets **string**à partir de tableaux de caractères (vous pouvez aussi simplement initialiser des objets **string**à partir d'autres objets **string**).

Vous pouvez assigner une valeur à n'importe quel objet **string**en utilisant " **=**". Cela remplace le précédent contenu de la chaîne de caractères avec ce qui est du côté droit de l'opérateur, et vous n'avez pas à vous inquiéter de ce qui arrive au contenu précédent - c'est géré automatiquement pour vous. Pour combiner des chaînes de caractères, vous utilisez simplement l'opérateur " **+**", qui permet aussi de combiner des tableaux de caractères avec des chaînes de caractères. Si vous voulez ajouter soit une chaîne de caractère soit un tableau de caractère à une autre chaîne de caractères, vous pouvez utiliser l'opérateur **+=**. Enfin, notez que les flux d'entrée/sortie savent déjà quoi faire avec les chaînes de caractères, ainsi vous pouvez simplement envoyer une chaîne de caractères (ou une expression qui produit une chaîne de caractères, comme **s1 + s2 + " !"**) directement à **cout**pour l'afficher.

2.6 - Lire et écrire des fichiers

En C, le processus d'ouverture et de manipulation des fichiers nécessite beaucoup d'expérience dans le langage pour vous préparer à la complexité des opérations. Cependant, la bibliothèque de flux d'entrée-sortie du C++ fournit un moyen simple pour manipuler des fichiers, et donc cette fonctionnalité peut être introduite beaucoup plus tôt qu'elle ne le serait en C.

Pour ouvrir des fichiers en lecture et en écriture, vous devez inclure la bibliothèque **<fstream>**. Bien qu'elle inclue automatiquement **<iostream>**, il est généralement prudent d'inclure explicitement **<iostream>**si vous prévoyez d'utiliser **cin**, **cout**, etc.

Pour ouvrir un fichier en lecture, vous créez un objet **ifstream**, qui se comporte ensuite comme **cin**. Pour ouvrir un fichier en écriture, vous créez un objet **ofstream**, qui se comporte ensuite comme **cout**. Une fois le fichier ouvert, vous pouvez y lire ou écrire comme vous le feriez avec n'importe quel autre objet de flux d'entrée-sortie. C'est aussi simple que ça (c'est, bien entendu, son point fort).

Une des fonctions les plus utiles de la bibliothèque de flux d'entrée-sortie est **getline( )**, qui vous permet de lire une ligne (terminée par un retour chariot) dans un objet **string** . Il y a en fait certain nombre de variantes de getline( ), qui seront traitées complètement dans le chapitre sur les flux d'entrée-sortie du Volume 2.. Le premier argument est l'objet **ifstream**que vous lisez et le second l'objet **string**. Quand l'appel de fonction est terminé, l'objet **string**contiendra la ligne.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Voici un exemple simple qui copie le contenu d'un fichier dans un autre :

```
                    //: C02:Scopy.cpp
// Copie un fichier dans un autre, une ligne à la fois
#include <string>
#include <fstream>
using namespace std;

int main() {
  ifstream in("Scopy.cpp"); // Ouvre en lecture
  ofstream out("Scopy2.cpp"); // Ouvre en écriture
  string s;
  while(getline(in, s)) // Ecarte le caractère nouvelle ligne...
    out << s << "\n"; // ... on doit donc l'ajouter
} ///:~
```

Pour ouvrir les fichiers, vous passez juste aux objets **ifstream**et **ofstream**les noms de fichiers que vous voulez créer, comme vu ci-dessus.

Un nouveau concept est introduit ici, la boucle **while**. Bien que cela sera expliqué en détail dans le chapitre suivant, l'idée de base est que l'expression entre les parenthèses qui suivent le **while**contrôle l'exécution de l'instruction suivante (qui peut être aussi des instructions multiples, en les enveloppant dans des accolades). Tant que l'expression entre parenthèses (dans l'exemple, **getline(in, s)**) produit un résultat "vrai", l'instruction contrôlée par le **while**continuera à s'exécuter. Il s'avère que **getline( )**retournera une valeur qui peut être interprétée comme "vrai" si une autre ligne a été lue avec succès, et "faux" lorsque la fin de l'entrée est atteinte. Ainsi, la boucle **while**ci-dessus lit chaque ligne du fichier d'entrée et envoie chaque ligne au fichier de sortie.

getline( )lit les caractères de chaque ligne jusqu'à rencontrer une nouvelle ligne (le caractère de fin peut être changé, mais ça cas ne sera pas traité avant le chapitre sur les flux d'entrée-sortie du Volume 2). Cependant, elle ne prend pas en compte le retour chariot et ne stocke pas ce caractère dans l'objet string résultant. Ainsi, si nous voulons que le fichier de destination ressemble au fichier source, nous devons remettre le retour chariot dedans, comme montré précédemment.

Un autre exemple intéressant est de copier le fichier entier dans un unique objet **string**:

```
                    //: C02:FillString.cpp
// Lit un fichier en entier dans une seule chaîne de caractères
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream in("FillString.cpp");
  string s, line;
  while(getline(in, line))
    s += line + "\n";
  cout << s;
} ///:~
```

Du fait de la nature dynamique des chaînes de caractères, vous n'avez pas à vous inquiéter de la quantité de mémoire à allouer pour une **string**; vous pouvez simplement continuer à ajouter des choses et la chaîne de caractères continuera à s'étendre pour retenir tout ce que vous mettez dedans.

Une des choses agréables dans le fait de mettre un fichier en entier dans une chaîne de caractères est que la classe **string**a beaucoup de fonctions de recherche et de manipulation qui peuvent alors vous permettre de modifier le fichier comme une simple chaîne de caractères. Cependant, ceci a ses limites. Pour une chose, il est souvent pratique de traiter un fichier comme un ensemble de lignes plutôt que simplement comme un gros blob Binary large object : données binaires de grande taille.de texte. Par exemple, si vous voulez ajouter une

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

numérotation des lignes, c'est plus simple si vous avez chaque ligne dans un objet **string**différent. Pour accomplir cela, vous aurez besoin d'une autre approche.

2.7 - Intoduction à la classe vector

Avec les **string**nous avons pu remplir une chaîne de caractères sans savoir de quelle taille nous allions avoir besoin. Le problème avec la lecture de lignes dans des objets **string**individuels est que vous ne savez pas à l'avance de combien d'objets vous allez avoir besoin - vous ne le savez qu'après avoir lu le fichier en entier. Pour résoudre ce problème, nous avons besoin d'une sorte de support qui va automatiquement s'agrandir pour contenir autant d'objets **string**que nous aurons besoin d'y mettre.

En fait, pourquoi se limiter à des objets **string**? Il s'avère que ce genre de problème - ne pas connaître le nombres d'objets que vous allez avoir lorsque vous écrivez le programme - est fréquent. Et il semble que ce “conteneur” serait beaucoup plus utile s'il pouvait contenir *toute sorte d'objet !*Heureusement, la bibliothèque standard a une solution toute faite : les classes de conteneurs standards. Les classes de conteneur sont une des forces du standard C++.

Il y a souvent une petite confusion entre les conteneurs et les algorithmes dans la bibliothèque standard du C++, et l'entité connue sous le nom de STL. Standard Template Library (Bibliothèque de modèle standards) est le nom qu'Alex Stepanov (qui travaillait alors pour Hewlett-Packard) utilisa lorsqu'il présenta sa bibliothèque au Comité de Normalisation du C++ à la conférence de San Diego, Californie au printemps 1994. Le nom est resté, surtout après que HP ai décidé de la rendre disponible au libre téléchargement. Entre-temps,le comité l'a intégré dans la bibliothèque standard du C++, en y apportant un grand nombre de changements. Le développement de la STL continue au sein de Silicon Graphics (SGI; voir *http://www.sgi.com/Technology/STL*). Le STL de SGI diverge de la bibliothèque strandard du C++ sur un certain nombre de points subtils. Ainsi, bien que cela soit une idée faussement répandue, la bibliothèque standard du C++ n'inclut pas la STL. Cela peut prêter à confusion du fait que les conteneurs et les algorithmes de la bibliothèque standard du C++ ont la même racine (et souvent les mêmes noms) que la STL de SGI. Dans ce livre, je dirai “bibliothèque standard du C++” ou “conteneurs de la bibliothèque standard,” ou quelque chose de similaire et éviterai le terme “STL.”

Même si l'implémentation des conteneurs et des algorithmes de la bibliothèque standard du C++ utilisent des concepts avancés et que la couverture de cette dernière prenne deux grands chapitres dans le volume 2 de ce livre, cette bibliothèque peut également être efficace sans en savoir beaucoup à son sujet. Elle est si utile que le plus basique des conteneurs standards, le **vector**, est introduit dès ce chapitre et utilisé tout au long de ce livre. Vous constaterez que vous pouvez faire une quantité de choses énorme juste en employant les fonctionnalités de base du **vector**et sans vous inquiéter pour l'implémentation (encore une fois, un but important de la POO). Puisque vous apprendrez beaucoup plus à ce sujet et sur d'autres conteneurs quand vous atteindrez les chapitres sur la bibliothèque standard dans le volume 2, on pardonnera le fait que les programmes utilisant le **vector**dans la première partie de ce livre ne soient pas exactement ce qu'un programmeur expérimenté de C++ ferait. Vous vous rendrez compte que dans la plupart des cas, l'utilisation montrée ici est adéquate.

La classe **vector**est une *classe générique*, ce qui signifie qu'elle peut être appliquée efficacement à différents types. Ainsi, on peut créer un vecteur de formes, un vecteur de chats, un vecteur de chaînes de caractères, etc. Fondamentalement, avec une classe générique vous pouvez créer une “classe de n'importe quoi”. Pour dire au compilateur ce avec quoi la classe va travailler (dans ce cas, ce que le **vector**va contenir), vous mettez le nom du type désiré entre les caractères #<' et #>'. Ainsi un vecteur de chaînes de caractères se note **vector<string>**. Lorsque vous faites cela, vous obtenez un vecteur personnalisé pouvant contenir uniquement des objets string, et vous aurez un message d'erreur de la part du compilateur si vous essayez d'y mettre autre chose.

Puisque le vecteur exprime le concept de “conteneur,” il doit y avoir une manière d'y mettre des objets et d'en enlever. Pour ajouter un élément nouveau à la fin d'un **vector**, vous utilisez la fonction membre **push_back( ).**(Rappelez vous que, lorsqu'il s'agit d'une fonction membre, vous utilisez le # **.**' pour l'appeler à partir d'un objet particulier.) La raison du nom de cette fonction membre qui peut sembler verbeux # **push_back( )**au lieu de

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

quelque chose de plus simple comme “put”# est qu'il y d'autres conteneurs et fonctions membres pour mettre des éléments dans les conteneurs. Par exemple, il y a une fonction membre **insert( )**pour mettre quelque chose au milieu d'un conteneur. Le **vector**supporte cela mais son utilisation est plus compliquée et nous n'avons pas besoin de la découvrir avant le volume 2 de ce livre. Il y a aussi **push_front( )**(qui n'est pas une fonction membre de la classe **vector**) pour mettre des objets en tête. Il y a pas mal d'autres fonctions membre de **vector**et pas mal de conteneurs standards dans le bibliothèque standard du C++, mais vous seriez surpris de tout ce que vous pouvez faire en ne connaissant que quelques fonctionnalités simples.

Donc vous pouvez mettre des éléments dans un **vector**avec **push_back( )**, mais comment les récupérer par la suite ? Cette solution est plus intelligente et et élégante # la surcharge d'opérateur est utilisée pour faire ressembler le **vector**à un *tableau*. Le tableau (qui sera décrit plus en détail dans le prochain chapitre) est un type de données qui est disponible dans pratiquement tout langage de programmation il devait déjà vous être familier. Les tableaux sont des *agrégats*, ce qui signifie qu'ils consistent en un nombre d'éléments groupés entre eux. La caractéristique distinctive d'un tableau est que les éléments sont de même taille et sont arrangés pour être l'un après l'autre. Plus important, ces éléments peuvent être sélectionnés par “indexation”, ce qui veut dire que vous pouvez dire “je veux l'élément numéro n” et cet élément sera accessible, en général très rapidement. Bien qu'il y ait des exceptions dans les langages de programmation, l'indexation est normalement réalisée en utilisant les crochets, de cette façon si vous avez un tableau **a**et que vous voulez accéder au cinquième élément, vous écrivez **a[4]**(notez que l'indexation commence à zéro).

Cette notation d'indexation très compacte et puissante est incorporée dans la classe **vector**en utilisant la surcharge d'opérateur, tout comme pour # **<<**' et # **>>**' sont incorporés dans iostreams. Encore une fois, nous n'avons pas besoin de connaître les détails de l'implémentation de la surcharge # cela fera l'objet d'un prochain chapitre # mais c'est utile si vous avez l'impression qu'il y a de la magie dans l'air dans l'utilisation de **[ ]**avec le vecteur.

Avec cela à l'esprit, vous pouvez maintenant voir un programme utilisant la classe **vector**. Pour utiliser un **vector**, vous incluez le fichier d'en-tête **<vector>**:

```
                    //: C02:Fillvector.cpp
// Copie un fichier entier dans un vecteur de chaînes de caractères
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> v;
  ifstream in("Fillvector.cpp");
  string line;
  while(getline(in, line))
    v.push_back(line); // Ajoute la ligne à la fin
  // Ajoute les numéros de lignes:
  for(int i = 0; i < v.size(); i++)
    cout << i << ": " << v[i] << endl;
} ///:~
```

Une grande partie de ce programme est similaire au précédent; un fichier est ouvert et les lignes sont lues une par une dans des objets **string**. Cependant, ces objets **string**sont *poussés*à la fin du **vecteur v**. Une fois la boucle **while**terminée, le fichier entier réside en mémoire, dans **v**.

L'étape suivante du programme est ce que l'on appelle une boucle **for**. Elle est similaire à la boucle **while**à l'exception du fait qu'elle ajoute des possibilités de contrôle supplémentaires. Après le **for**, il y a une “expression de contrôle” entre parenthèses, tout comme pour la boucle **while**. Cependant, cette expression de contrôle est en trois parties : une partie qui initialise, une qui teste si l'on doit sortir de la boucle, et une qui change quelque chose, typiquement pour itérer sur une séquence d'éléments. Ce programme exhibe une boucle **for**telle qu'elle est communément utilisée : l'initialisation **int i = 0**crée un entier **i**utilisé comme un compteur de boucle et de valeur

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

initiale zéro. La portion de test dit que pour rester dans la boucle, **i**doit être inférieur au nombre d'éléments du vecteur **v**. (Cela est déterminé en utilisant la fonction membre **size( )**, que j'ai glissé ici, mais vous admettrez que sa signification est assez évidente.) La portion finale emploie une notation du C et du C++, l'opérateur d'"auto-incrémentation", pour ajouter une unité à la valeur de **i**. En effet, **i++**signifie "prend la valeur de **i**, ajoutes-y un, et mets le résultat dans **i**". Ainsi, l'effet global de la boucle **for**est de prendre une variable **i**et de l'incrémenter par pas de un jusqu'à la taille du vecteur moins un. Pour chaque valeur de **i**, le **cout**est exécuté et cela construit une ligne contenant la valeur de **i**(magiquement convertie en tableau de caractères par **cout**), deux-points et un espace, la ligne du fichier, et un retour à la ligne amené par **endl**. Lorsque vous compilerez et exécuterez ce programme, vous verrez que l'effet est d'ajouter une numérotation de ligne au fichier.

Du fait que l'opérateur # **>>**' fonctionne avec iostreams, vous pouvez facilement modifier le programme afin qu'il découpe l'entrée en mots au lieu de lignes :

```
                    //: C02:GetWords.cpp
// Break a file into whitespace-separated words
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> words;
  ifstream in("GetWords.cpp");
  string word;
  while(in >> word)
    words.push_back(word);
  for(int i = 0; i < words.size(); i++)
    cout << words[i] << endl;
} ///:~
```

L'expression

```
                    while(in >> word)
```

permet d'obtenir une entrée "mot" à "mot", et quand cette expression est évaluée à "faux" cela signifie que la fin du fichier a été atteinte. Naturellement, délimiter des mots par un espace est assez brut, mais cela est un exemple simple. Plus tard dans ce livre vous verrez des exemples plus sophistiqués qui vous permettront de découper l'entrée comme bon vous semble.

Pour démontrer la facilité d'utilisation d'un vecteur de n'importe quel type, voici un exemple qui crée un **vector<int>**:

```
                    //: C02:Intvector.cpp
// Creating a vector that holds integers
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> v;
  for(int i = 0; i < 10; i++)
    v.push_back(i);
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
  for(int i = 0; i < v.size(); i++)
    v[i] = v[i] * 10; // Assignment
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
} ///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Pour créer un **vector**pour contenir un certain type, vous avez juste à mettre ce type en temps que paramètre de template (entre les caractères **<**et **>**). Les templates et les bibliothèques de templates optimisées sont prévus pour être aussi faciles à employer.

Cet exemple va vous montrer un autre aspect essentiel de la classe **vector**. Dans l'expression

```
v[i] = v[i] * 10;
```

vous pouvez voir que le **vector**n'est pas limité seulement à y mettre des choses et à les récupérer. Vous êtes aussi abilités à *affecter à*(et donc à modifier) n'importe quel élément du vecteur, ceci en utilisant l'opérateur d'indexation. Cela signifie que la classe **vector**est un outil d'usage universel et flexible pour travailler avec une collection d'objets, et nous en ferons usage dans les chapitres à venir.

2.8 - Résumé

Le but de ce chapitre est de vous montrer à quel point la programmation orientée objet peut être facile # *si*un tiers a fait pour vous le travail de définition des objets. Dans ce cas incluez le fichier d'en-tête, créez des objets, et envoyez leur des messages. Si les types que vous utilisez sont performants et bien conçus, vous n'avez pas beaucoup plus de travail à faire et votre programme sera également performant.

Dans l'optique de montrer la facilité de la POO lorsqu'on utilise des bibliothèques de classes, ce chapitre a également introduit quelques uns des types les plus basiques et utiles de la bibliothèque standard du C++: la famille des iostreams (en particulier ceux qui lisent et écrivent sur la console et dans les fichiers), la classe **string**, et le template **vector**. Vous avez vu comme il est très simple de les utiliser et pouvez probablement imaginer ce que vous pouvez accomplir avec, mais il y a encore beaucoup plus à faire Si vous êtes particulièrement désireux de voir toutes les choses qui peuvent être faites avec ces derniers et d'autres composants de la bibliothèque standard, voir le volume 2 de ce livre chez www.BruceEckel.com, et également www.dinkumware.com.. Même si nous n'emploierons seulement qu'un sous-ensemble limité des fonctionnalités de ces outils dans la première partie de ce livre, ils fourniront néanmoins les bases de l'apprentissage d'un langage de bas niveau comme le C. Et tout en étant éducatif, l'apprentissage des aspects bas niveau du C prend du temps. En fin de compte, vous serez beaucoup plus productif si vous avez des objets pour contrôler les aspects bas niveau. Après tout, l' *objectif*de la POO est de cacher les détails ainsi vous pouvez #peindre avec une plus grande brosse#.

Cependant, pour autant que la POO essaye d'être de haut niveau, il y a quelques aspects fondamentaux du C que vous ne pouvez pas ignorer, et ceux-ci seront couverts par le prochain chapitre.

2.9 - Exercices

Les solutions des exercices suivants se trouvent dans le document électronique *The Thinking in C++ Annotated Solution Guide*, disponible pour un prix modique sur http://www.BruceEckel.com

1 Modifier **Hello.cpp**pour afficher vos nom et âge (ou pointure de chaussure, ou l'âge de votre chien, si cela vous convient mieux). Compilez et exécutez le programme.
2 En se basant sur **Stream2.cpp**et **Numconv.cpp**, créez un programme qui demande le rayon d'un cercle et affiche l'aire de ce dernier. Vous ne pouvez utiliser que l'opérateur ***pour élever le rayon au carré. N'essayez pas d'afficher la valeur octal ou hexadécimal (cela n'est possible qu'avec les nombres entiers).
3 Ecrivez un programme qui ouvre un fichier et compte les mots séparés par un espace dans ce fichier.
4 Ecrivez un programme qui compte le nombre d'occurrences d'un mot particulier dans un fichier (utiliser la classe **string**et l'opérateur **==**pour trouver le mot).

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

5 Changez **FillVector.cpp**pour qu'il imprime les lignes (inversées) de la dernière à la première.
6 Changez **FillVector.cpp**pour qu'il concatène tous les éléments du **vector**dans une simple chaîne avant de les afficher, mais n'essayez pas d'ajouter la numérotation.
7 Affichez un fichier ligne par ligne, qui attend que l'utilisateur appuie sur la touche “Entrée“ après chaque ligne.
8 Créez un **vector<float>**et y mettre 25 nombres flottants en utilisant une boucle **for**. Affichez le **vecteur**.
9 Créez trois objets **vector<float>**et remplissez les deux premiers comme dans l'exercice précédent. Ecrivez une boucle **for**qui additionne les éléments correspondants des deux premiers **vecteurs**et met le résultat dans l'élément correspondant du troisième **vecteur**. Affichez les trois **vecteurs**.
10 Créez un **vector<float>**et mettez-y 25 nombres flottants comme dans les exercices précédents. Maintenant élevez au carré chaque nombre et mettez le résultat au même emplacement dans le **vector**. Affichez le **vector**avant et après la multiplication.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

3 - Le C de C++

Puisque le C++ est basé sur le C, vous devez être familier avec la syntaxe du C pour programmer en C++, tout comme vous devez raisonnablement être à l'aise en algèbre pour entreprendre des calculs.

Si vous n'avez jamais vu de C avant, ce chapitre va vous donner une expérience convenable du style de C utilisé en C++. Si vous êtes familier avec le style de C décrit dans la première édition de Kernighan et Ritchie (souvent appelé K&R C), vous trouverez quelques nouvelles fonctionnalités différentes en C++ comme en C standard. Si vous êtes familier avec le C standard, vous devriez parcourir ce chapitre en recherchant les fonctionnalités particulières au C++. Notez qu'il y a des fonctionnalités fondamentales du C++ introduites ici qui sont des idées de base voisines des fonctionnalités du C ou souvent des modifications de la façon dont le C fait les choses. Les fonctionnalités plus sophistiquées du C++ ne seront pas introduites avant les chapitres suivants.

Ce chapitre est un passage en revue assez rapide des constructions du C et une introduction à quelques constructions de base du C++, en considérant que vous avez quelque expérience de programmation dans un autre langage. Une introduction plus douce au C se trouve dans le CD ROM relié au dos du livre, appelée *Penser en C : Bases pour Java et C++*par Chuck Allison (publiée par MindView, Inc., et également disponible sur www.MindView.net). C'est une conférence sur CD ROM avec pour objectif de vous emmener prudemment à travers les principes fondamentaux du langage C. Elle se concentre sur les connaissances nécessaires pour vous permettre de passer aux langages C++ ou Java, au lieu d'essayer de faire de vous un expert de tous les points d'ombre du C (une des raisons d'utiliser un langage de haut niveau comme le C++ ou le Java est justement d'éviter plusieurs de ces points sombres). Elle contient aussi des exercices et des réponses guidées. Gardez à l'esprit que parce que ce chapitre va au-delà du CD *Penser en C*, le CD ne se substitue pas à ce chapitre, mais devrait plutôt être utilisé comme une préparation pour ce chapitre et ce livre.

3.1 - Création de fonctions

En C d'avant la standardisation, vous pouviez appeler une fonction avec un nombre quelconque d'arguments sans que le compilateur ne se plaigne. Tout semblait bien se passer jusqu'à l'exécution du programme. Vous obteniez des résultats mystérieux (ou pire, un plantage du programme) sans raison. Le manque d'aide par rapport au passage d'arguments et les bugs énigmatiques qui en résultaient est probablement une des raisons pour laquelle le C a été appelé un « langage assembleur de haut niveau ». Les programmeurs en C pré-standard s'y adaptaient.

Le C et le C++ standards utilisent une fonctionnalité appelée *prototypage de fonction*. Avec le prototypage de fonction, vous devez utiliser une description des types des arguments lors de la déclaration et de la définition d'une fonction. Cette description est le « prototype ». Lorsque cette fonction est appelée, le compilateur se sert de ce prototype pour s'assurer que les bons arguments sont passés et que la valeur de retour est traitée correctement. Si le programmeur fait une erreur en appelant la fonction, le compilateur remarque cette erreur.

Dans ses grandes lignes, vous avez appris le prototypage de fonction (sans lui donner ce nom) au chapitre précédent, puisque la forme des déclaration de fonction en C++ nécessite un prototypage correct. Dans un prototype de fonction, la liste des arguments contient le type des arguments qui doivent être passés à la fonction et (de façon optionnelle pour la déclaration) les identifiants des arguments. L'ordre et le type des arguments doit correspondre dans la déclaration, la définition et l'appel de la fonction. Voici un exemple de prototype de fonction dans une déclaration :

```
int translate(float x, float y, float z);
```

Vous ne pouvez pas utiliser pas la même forme pour déclarer des variables dans les prototypes de fonction que pour les définitions de variables ordinaires. Vous ne pouvez donc pas écrire : **float x, y, z**. Vous devez indiquer le type de *chaque*argument. Dans une déclaration de fonction, la forme suivante est également acceptable :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
int translate(float, float, float);
```

Comme le compilateur ne fait rien d'autre que vérifier les types lorsqu'une fonction est appelée, les identifiants sont seulement mentionnés pour des raisons de clarté lorsque quelqu'un lit le code.

Dans une définition de fonction, les noms sont obligatoires car les arguments sont référencés dans la fonction :

```
int translate(float x, float y, float z) {
  x = y = z;
  // ...
}
```

Il s'avère que cette règle ne s'applique qu'en C. En C++, un argument peut être anonyme dans la liste des arguments d'une définition de fonction. Comme il est anonyme, vous ne pouvez bien sûr pas l'utiliser dans le corps de la fonction. Les arguments anonymes sont autorisés pour donner au programmeur un moyen de « réserver de la place dans la liste des arguments ». Quiconque utilise la fonction doit alors l'appeler avec les bons arguments. Cependant, le créateur de la fonction peut utiliser l'argument par la suite sans forcer de modification du code qui utilise cette fonction. Cette possibilité d'ignorer un argument dans la liste est aussi possible en laissant le nom, mais vous obtiendrez un message d'avertissement énervant à propos de la valeur non utilisée à chaque compilation de la fonction. Cet avertissement est éliminé en supprimant le nom.

Le C et le C++ ont deux autres moyens de déclarer une liste d'arguments. Une liste d'arguments vide peut être déclarée en C++ par **fonc( )**, ce qui indique au compilateur qu'il y a exactement zero argument. Notez bien que ceci indique une liste d'argument vide en C++ seulement. En C, cela indique « un nombre indéfini d'arguments » (ce qui est un « trou » en C, car dans ce cas le contrôle des types est impossible). En C et en C++, la déclaration **fonc(void);**signifie une liste d'arguments vide. Le mot clé **void**signifie, dans ce cas, « rien » (il peut aussi signifier « pas de type » dans le cas des pointeurs, comme il sera montré plus tard dans ce chapitre).

L'autre option pour la liste d'arguments est utilisée lorsque vous ne connaissez pas le nombre ou le type des arguments ; cela s'appelle une *liste variable d'arguments*. Cette « liste d'argument incertaine » est représentée par des points de suspension (**...**). Définir une fonction avec une liste variable d'arguments est bien plus compliqué que pour une fonction normale. Vous pouvez utiliser une liste d'argument variable pour une fonction avec un nombre fixe d'argument si (pour une raison) vous souhaitez désactiver la vérification d'erreur du prototype. Pour cette raison, vous devriez restreindre l'utilisation des liste variables d'arguments au C et les éviter en C++ (lequel, comme vous allez l'apprendre, propose de bien meilleurs alternatives). L'utilisation des listes variables d'arguments est décrite dans la section concernant la bibliothèque de votre guide sur le C.

3.1.1 - Valeurs de retour des fonctions

Un prototype de fonction en C++ doit spécifier le type de la valeur de retour de cette fonction (en C, si vous omettez le type de la valeur de retour, il vaut implicitement **int**). La spécification du type de retour précède le nom de la fonction. Pour spécifier qu'aucune valeur n'est retournée, il faut utiliser le mot clé **void**. Une erreur sera alors générée si vous essayez de retourner une valeur de cette fonction. Voici quelques prototypes de fonctions complets :

```
int f1(void); // Retourne un int, ne prend pas d'argument
int f2(); // Comme f1() en C++ mais pas en C standard !
float f3(float, int, char, double); // Retourne un float
void f4(void); // Ne prend pas d'argument, ne retourne rien
```

Pour retourner une valeur depuis une fonction, utilisez l'instruction **return**. **return**sort de la fonction et revient juste après l'appel de cette fonction. Si **return**a un argument, cet argument devient la valeur de retour de la fonction. Si

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

une fonction mentionne qu'elle renvoie un type particulier, chaque instruction **return**doit renvoyer ce type. Plusieurs instructions **return**peuvent figurer dans la définition d'une fonction :

```
                        //: C03:Return.cpp
// Utilisation de "return"
#include <iostream>
using namespace std;

char cfonc(int i) {
  if(i == 0)
    return 'a';
  if(i == 1)
    return 'g';
  if(i == 5)
    return 'z';
  return 'c';
}

int main() {
  cout << "Entrez un entier : ";
  int val;
  cin >> val;
  cout << cfonc(val) << endl;
} ///:~
```

Dans **cfonc( )**, le premier **if**qui est évalué à **true**sort de la fonction par l'instruction **return**. Notez que la déclaration de la fonction n'est pas nécessaire, car sa définition apparaît avec son utilisation dans **main( )**, et le compilateur connaît donc la fonction depuis cette définition.

3.1.2 - Utilisation de la bibliothèque de fonctions du C

Toutes les fonctions de la bibliothèque de fonctions du C sont disponibles lorsque vous programmez en C++. Étudiez attentivement la bibliothèque de fonctions avant de définir vos propres fonctions – il y a de grandes chances que quelqu'un ait déjà résolu votre problème, et y ait consacré plus de réflexion et de débogage.

Cependant, soyez attentifs : beaucoup de compilateurs proposent de grandes quantités de fonctions supplémentaires qui facilitent la vie et dont l'utilisation est tentante, mais qui ne font pas partie de la bibliothèque du C standard. Si vous êtes certains que vous n'aurez jamais à déplacer votre application vers une autre plateforme (et qui peut être certain de cela ?), allez-y – utilisez ces fonctions et simplifiez vous la vie. Si vous souhaitez que votre application soit portable, vous devez vous restreindre aux fonctions de la bibliothèque standard. Si des activités spéficiques à la plateforme sont nécessaires, essayez d'isoler ce code en un seul endroit afin qu'il puisse être changé facilement lors du portage sur une autre plateforme. En C++, les activités spécifiques à la plateforme sont souvent encapsulées dans une classe, ce qui est la solution idéale.

La recette pour utiliser une fonction d'une bibliothèque est la suivante : d'abord, trouvez la fonction dans votre référence de programmation (beaucoup de références de programmation ont un index des fonctions aussi bien par catégories qu'alphabétique). La description de la fonction devrait inclure une section qui montre la syntaxe du code. Le début de la section comporte en général au moins une ligne **#include**, vous montrant le fichier d'en-tête contenant le prototype de la fonction. Recopiez cette ligne **#include**dans votre fichier pour que la fonction y soit correctement déclarée. Vous pouvez maintenant appeler cette fonction de la manière qui est montrée dans la section présentant la syntaxe. Si vous faites une erreur, le compilateur la découvrira en comparant votre appel de fonction au prototype de la fonction dans l'en-tête et vous signifiera votre erreur. L'éditeur de liens parcourt implicitement la bibliothèque standard afin que la seule chose que vous ayez à faire soit d'inclure le fichier d'en-tête et d'appeler la fonction.

3.1.3 - Créer vos propres bibliothèques avec le bibliothécaire

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Vous pouvez rassembler vos propres fonctions dans une bibliothèque. La plupart des environnements de programmation sont fournis avec un bibliothécaire qui gère des groupes de modules objets. Chaque bibliothécaire a ses propres commandes, mais le principe général est le suivant : si vous souhaitez créer une bibliothèque, fabriquez un fichier d'en-tête contenant les prototypes de toutes les fonctions de votre bibliothèque. Mettez ce fichier d'en-tête quelque part dans le chemin de recheche du pré-processeur, soit dans le répertoire local (qui pourra alors être trouvé par **#include "en_tete"**) soit dans le répertoire d'inclusion (qui pourra alors être trouvé par **#include <en_tete>**). Prenez ensuite tous les modules objets et passez-les au bibliothécaire, en même temps qu'un nom pour la bibliothèque (la plupart des bibliothécaires attendent une extension habituelle, comme **.lib**ou **.a**). Placez la bibliothèque terminée avec les autres bibliothèques, afin que l'éditeur de liens puisse la trouver. Pour utiliser votre bibliothèque, vous aurez à ajouter quelque chose à la ligne de commande pour que l'éditeur de liens sache où chercher la bibliothèque contenant les fonctions que vous appelez. Vous trouverez tous les détails dans votre manuel local, car ils varient d'un système à l'autre.

3.2 - Contrôle de l'exécution

Cette section traite du contrôle de l'exécution en C++. Vous devez d'abord vous familiariser avec ces instructions avant de pouvoir lire et écrire en C ou C++.

Le C++ utilise toutes les structures de contrôle du C. Ces instructions comprennent **if-else**, **while**, **do-while**, **for**, et une instruction de sélection nommée **switch**. Le C++ autorise également l'infâme **goto**, qui sera proscrit dans cet ouvrage.

3.2.1 - Vrai et faux

Toutes les structures de contrôle se basent sur la vérité ou la non vérité d'une expression conditionnelle pour déterminer le chemin d'exécution. Un exemple d'expression conditionnelle est **A == B**. Elle utilise l'opérateur **==**pour voir si la variable **A**est équivalente à la variable **B**. L'expression génère un Booléen **true**ou **false**(ce sont des mots clé du C++ uniquement ; en C une expression est “vraie” si elle est évaluée comme étant différente de zéro). D'autres opérateurs conditionnels sont **>**, **<**, **>=**, etc. Les instructions conditionnelles seront abordées plus en détail plus loin dans ce chapitre.

3.2.2 - if-else

La structure de contrôle **if-else**peut exister sous deux formes : avec ou sans le **else**. Les deux formes sont :

```
                        if(expression)
    instruction
```

or

```
                        if(expression)
    instruction
else
    instruction
```

l'“expression” est évaluée à **true**ou **false**. Le terme “instruction” désigne soit une instruction seule terminée par un point virgule soit une instruction composée qui est un groupe d'instructions simples entourées d'accolades. Chaque fois que le terme “instruction” est utilisé, cela implique toujours qu'il s'agisse d'une instruction simple ou composée. Notez qu'une telle instruction peut être également un autre **if**, de façon qu'elles puissent être cascadées.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                          //: C03:Ifthen.cpp
// Demonstration des structures conditionnelles if et if-else
#include <iostream>
using namespace std;

int main() {
  int i;
  cout << "tapez un nombre puis 'Entrée" << endl;
  cin >> i;
  if(i > 5)
    cout << "Il est plus grand que 5" << endl;
  else
    if(i < 5)
      cout << "Il est plus petit que 5 " << endl;
    else
      cout << "Il est égal à " << endl;

  cout << "tapez un nombre puis 'Entrée" << endl;
  cin >> i;
  if(i < 10)
    if(i > 5)  // "if" est juste une autre instruction
      cout << "5 < i < 10" << endl;
    else
      cout << "i <= 5" << endl;
  else // Se réfère au "if(i < 10)"
    cout << "i >= 10" << endl;
} ///:~
```

Par convention le corps d'une structure de contrôle est indenté pour que le lecteur puisse déterminer aisément où elle commence et où elle se termine Remarquez que toutes les conventions ne s'accorde pas à indenter le code en ce sens. La guerre de religion entre les styles de formattage est incessante. Conférez l'appendice A pour une description du style utilisé dans ce livre..

3.2.3 - while

Les boucles **while**, **do-while,**et **for**. une instruction se répète jusqu'à ce que l'expression de contrôle soit évaluée à **false**. La forme d'une boucle **while**est

```
                          while(expression)
    instruction
```

L'expression est évaluée une fois à l'entrée dans la boucle puis réévaluée avant chaque itération sur l'instruction.

L'exemple suivant reste dans la boucle **while**jusqu'à ce que vous entriez le nombre secret ou faites un appui sur control-C.

```
                          //: C03:Guess.cpp
// Devinez un nombre (demontre le "while")
#include <iostream>
using namespace std;

int main() {
  int secret = 15;
  int guess = 0;
  // "!=" est l'opérateur conditionnel "différent de" :
  while(guess != secret) { // Instruction composée
    cout << "Devinez le nombre : ";
    cin >> guess;
  }
  cout << "Vous l'avez trouvé !" << endl;
} ///:~
```

L'expression conditionnelle du **while**n'est pas restreinte à un simple test comme dans l'exemple ci-dessus ; il peut

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

être aussi compliqué que vous le désirez tant qu'il produit un résutat **true**ou **false**. Vous verrez même du code dans lequel la boucle n'a aucun corps, juste un point virgule dénudé de tout effet :

```
                        while(/* Plein de choses ici */)
  ;
```

Dans un tel cas, le programmeur a écrit l'expression conditionnelle pour non seulement réaliser le test mais aussi pour faire le boulot.

3.2.4 - do-while

La construction d'un boucle **do-while**est

```
                        do
    instruction
 while(expression);
```

la boucle **do-while**est différente du while parce que l'instruction est exécutée au moins une fois, même si l'expression est évaluée à fausse dès la première fois. Dans une boucle **while**ordinaire, si l'expression conditionnelle est fausse à la première évaluation, l'instruction n'est jamais exécutée.

Si on utilise un **do-while**dans notre **Guess.cpp**, la variable **guess**n'a pas besoin d'une valeur initiale factice, puisqu'elle est initialisée par l'instruction **cin**avant le test :

```
                        //: C03:Guess2.cpp
// Le programme de devinette avec un do-while
#include <iostream>
using namespace std;

int main() {
  int secret = 15;
  int guess; // Pas besoin d'initialisation
  do {
    cout << "Devinez le nombre : ";
    cin >> guess; // L'initialisation s'effectue
  }   while(guess != secret);
  cout << "Vous l'avez trouvé!" << endl;
} ///:~
```

Pour des raisons diverses, la plupart des programmeurs tend à éviter l'utilisation du **do-while**et travaille simplement avec un **while**.

3.2.5 - for

Une boucle **for**permet de faire une initialisation avant la première itération. Ensuite elle effectue un test conditionnel et, à la fin de chaque itération, une forme de “saut”. La construction de la boucle **for**est :

```
                        for(initialisation; condition; saut)
    instruction
```

chacune des expressions *initialisation*, *condition,*ou *saut*peut être laissée vide. l' *initialisation*est exécutée une seule fois au tout début. La *condition*est testée avant chaque itération (si elle est évaluée à fausse au début, l'instruction ne s'exécutera jamais). A la fin de chaque boucle, le *saut*s'exécute.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Une boucle **for**est généralement utilisée pour “compter” des tâches:

```
                        //: C03:Charlist.cpp
// Affiche tous les caractères ASCII
// Demontre "for"
#include <iostream>
using namespace std;

int main() {
  for(int i = 0; i < 128; i = i + 1)
    if (i != 26)  // Caractère ANSI d'effacement de l'écran
      cout << " valeur : " << i
           << " caractère : "
           << char(i) // Conversion de type
           << endl;
} ///:~
```

Vous pouvez noter que la variable **i**n'est définie qu'a partir de là où elle est utilisée, plutôt qu'au début du block dénoté par l'accolade ouvrante ‘ **{**’. Cela change des langages procéduraux traditionnels (incluant le C), qui requièrent que toutes les variables soient définies au début du bloc. Ceci sera discuté plus loin dans ce chapitre.

3.2.6 - Les mots clé break et continue

Dans le corps de toutes les boucles **while**, **do-while,**ou **for**, il est possible de contrôler le déroulement de l'exécution en utilisant **break**et **continue**. **break**force la sortie de la boucle sans exécuter le reste des instructions de la boucle. **continue**arrête l'exécution de l'itération en cours et retourne au début de la boucle pour démarrer une nouvelle itération.

Pour illustrer **break**et **continue**, le programme suivant est un menu système tres simple :

```
                        //: C03:Menu.cpp
// Démonstration d'un simple menu système
// the use of "break" and "continue"
#include <iostream>
using namespace std;

int main() {
  char c; // Pour capturer la réponse
  while(true) {
    cout << "MENU PRINCIPAL :" << endl;
    cout << "g : gauche, d : droite, q : quitter -> ";
    cin >> c;
    if(c == 'q')
      break; // Out of "while(1)"
    if(c == 'g') {
      cout << "MENU DE GAUCHE :" << endl;
      cout << "sélectionnez a ou b : ";
      cin >> c;
      if(c == 'a') {
        cout << "vous avez choisi 'a'" << endl;
        continue; // Retour au menu principal
      }
      if(c == 'b') {
        cout << "vous avez choisi 'b'" << endl;
        continue; // Retour au menu principal
      }
      else {
        cout << "vous n'avez choisi ni a ni b !"
             << endl;
        continue; // Retour au menu principal
      }
    }
    if(c == 'd') {
      cout << "MENU DE DROITE:" << endl;
      cout << "sélectionnez c ou d : ";
      cin >> c;
      if(c == 'c') {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
          cout << "vous avez choisi 'c'" << endl;
          continue; // Retour au menu principal
        }
        if(c == 'd') {
          cout << "vous avez choisi 'd'" << endl;
          continue; // Retour au menu principal
        }
        else {
          cout << "vous n'avez choisi ni c ni d !"
               << endl;
          continue; // Retour au menu principal
        }
      }
      cout << "vous devez saisir g, d ou q !" << endl;
    }
    cout << "quitte le menu..." << endl;
  } ///:~
```

Si l'utilisateur sélectionne ‘q’ dans le menu principal, le mot clé **break**est utilisé pour quitter, sinon, le programme continue normalement son exécution indéfiniment. Après chaque sélection dans un sous-menu, le mot clé **continue**est utilisé pour remonter au début de la boucle while.

l'instruction **while(true)**est équivalente à dire “exécute cette boucle infiniment”. L'instruction **break**vous autorise à casser cette boucle sans fin quant l'utilisateur saisi un ‘q’.

3.2.7 - switch

Une instruction **switch**effectue un choix parmi une sélection de blocs de code basé sur la valeur d'une expression intégrale. Sa construction est de la forme :

```
                        switch(sélecteur) {
      case valeur-intégrale1 : instruction; break;
      case valeur-intégrale2 : instruction; break;
      case valeur-intégrale3 : instruction; break;
      case valeur-intégrale4 : instruction; break;
      case valeur-intégrale5 : instruction; break;
      (...)
      default: instruction;
  }
```

Le *sélecteur*est une expression qui produit une valeur entière. Le **switch**compare le résultat du *sélecteur*avec chaque *valeur entière*. si il trouve une valeur identique, l'instruction correspondante (simple ou composée) est exécutée. Si aucune correspondance n'est trouvée, l'instruction **default**est exécutée.

Vous remarquerez dans la définition ci-dessus que chaque **case**se termine avec un **break**, ce qui entraine l'exécution à sauter à la fin du corps du **switch**(l'accolade fermante qui complete le **switch**). Ceci est la manière conventionnelle de construire un **switch**, mais le **break**est facultatif. S'il est omis, votre **case**“s'étend” au suivant. Ainsi, le code du prochain **case**s'exécute jusqu'à ce qu'un **break**soit rencontré. Bien qu'un tel comportement ne soit généralement pas désiré, il peut être très utile à un programmeur expérimenté.

L'instruction **switch**est un moyen clair pour implémenter un aiguillage (i.e., sélectionner parmi un nombre de chemins d'exécution différents), mais elle requiert un sélecteur qui s'évalue en une valeur intégrale au moment de la compilation. Si vous voulez utiliser, par exemple, un objet **string**comme sélecteur, cela ne marchera pas dans une instruction **switch**. Pour un sélecteur de type **string**, vous devez utiliser à la place une série d'instructions **if**et le comparer à la **string**de l'expression conditionnelle.

L'exemple du menu précédent un particulièrement bon exemple pour utiliser un **switch**:

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                        //: C03:Menu2.cpp
// Un menu utilisant un switch
#include <iostream>
using namespace std;

int main() {
  bool quit = false;  // Flag pour quitter
  while(quit == false) {
    cout << "Sélectionnez a, b, c ou q pour quitter: ";
    char reponse;
    cin >> response;
    switch(reponse) {
      case 'a' : cout << "vous avez choisi 'a'" << endl;
                 break;
      case 'b' : cout << "vous avez choisi 'b'" << endl;
                 break;
      case 'c' : cout << "vous avez choisi 'c'" << endl;
                 break;
      case 'q' : cout << "quittte le menu" << endl;
                 quit = true;
                 break;
      default  : cout << "sélectionnez a,b,c ou q !"
                 << endl;
    }
  }
} ///:~
```

Le flag **quit**est un **bool**, raccourci pour “Booléen,” qui est un type que vous ne trouverez qu'en C++. Il ne peut prendre que les valeurs des mots clé **true**ou **false**. Sélectionner ‘q’ met le flag **quit**à **true**. A la prochaine évaluation du sélecteur, **quit == false**retourne **false**donc le corps de la boucle **while**ne s'exécute pas.

3.2.8 - Du bon et du mauvais usage du goto

Le mot-clé **goto**est supporté en C++, puisqu'il existe en C. Utiliser **goto**dénote souvent un style de programmation pauvre, et ca l'est réellement la plupart du temps. Chaque fois que vous utilisez **goto**, regardez votre code, et regardez s'il n'y a pas une autre manière de le faire. A de rares occasions, vous pouvez découvrir que le **goto**peut résoudre un problème qui ne peut être résolu autrement, mais encore, pensez y à deux fois. Voici un exemple qui pourrait faire un candidat plausible :

```
                        //: C03:gotoKeyword.cpp
// L'infâme goto est supporté en C++
#include <iostream>
using namespace std;

int main() {
  long val = 0;
  for(int i = 1; i < 1000; i++) {
    for(int j = 1; j < 100; j += 10) {
      val = i * j;
      if(val > 47000)
        goto bas;
        // Break serait remonté uniquement au 'for' extérieur
    }
  }
  bas: // une étiquette
  cout << val << endl;
} ///:~
```

Une alternative serait de définir un booléen qui serait testé dans la boucle **for**extérieure, qui le cas échéant exécuterait un **break**. Cependant, si vous avez plusieurs boucles **for**ou **while**imbriquées, cela pourrait devenir maladroit.

3.2.9 - Récursion

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

La récursion une technique de programmation interressante et quelque fois utile par laquelle vous appelez la fonction dans laquelle vous êtes. Bien sûr, si vous ne faites que cela, vous allez appeler la fonction jusqu'à ce qu'il n'y ait plus de mémoire, donc vous devez fournir une “issue de secours” aux appels récursifs. Dans l'exemple suivant, cette “issue de secours” est réalisée en disant simplement que la récursion ira jusqu'à ce que **cat**dépasse ‘Z’ : Merci à Kris C. Matson d'avoir suggéré ce sujet d'exercice.

```
                        //: C03:CatsInHats.cpp
// Simple demonstration de récursion
#include <iostream></iostream>
using namespace std;

void retirerChapeau(char cat) {
  for(char c = 'A'; c < cat; c++)
    cout << "  ";
  if(cat <= 'Z') {
    cout << "cat " << cat << endl;
    retirerChapeau(cat + 1); // appel récursif
  } else
    cout << "VOOM !!!" << endl;
}

int main() {
  retirerChapeau('A');
} ///:~
```

Dans **retirerChapeau( )**, vous pouvez voir que tant que **cat**est plus petit que ‘Z’, **retirerChapeau( )**sera appelé depuis *l'intérieur*de **retirerChapeau( )**, d'où la récursion. Chaque fois que **retirerChapeau( )**est appelé, sont paramètre est plus grand de un par rapport à la valeur actuelle de **cat**donc le paramètre continue d'augmenter.

La récursion est souvent utilisée pour résoudre des problèmes d'une complexité arbitraire, comme il n'y a pas de limite particulière de “taille” pour la solution – la fonction peut continuer sa récursion jusqu'à résolution du problème.

3.3 - Introduction aux operateurs

Vous pouvez penser aux opérateurs comme un type spécial de fonction (vous allez apprendre que le C++ traite la surchage d’opérateurs exactement de cette façon). Un opérateur prend un ou plusieurs arguments et retourne une nouvelle valeur. Les arguments sont sous une forme différente des appels de fonction ordinaires, mais le résultat est identique.

De part votre expérience de programmation précédente, vous devriez être habitué aux opérateurs qui ont été employés jusqu’ici. Les concepts de l'addition (**+**), de la soustraction et du moins unaire (**-**), de la multiplication (*), de la division (/), et de l'affectation (=) ont tous essentiellement la même signification dans n'importe quel langage de programmation. L'ensemble complet des opérateurs est détaillé plus tard dans ce chapitre.

3.3.1 - Priorité

La priorité d'opérateur définit l'ordre dans lequel une expression est évaluée quand plusieurs opérateurs différents sont présents. Le C et le C++ ont des règles spécifiques pour déterminer l'ordre d'évaluation. Le plus facile à retenir est que la multiplication et la division se produisent avant l'addition et soustraction. Si, après cela, une expression n'est pas claire pour vous, elle ne le sera probablement pas pour n'importe qui d'autre lisant le code, aussi, vous devriez utiliser des parenthèses pour rendre l'ordre d'évaluation explicite. Par exemple :

```
A = X + Y - 2/2 + Z;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

a une signification très différente de le même instruction avec un groupe particulier de parenthèses

```
A = X + (Y - 2)/(2 + Z);
```

(Essayez d'évaluer le résultat avec X = 1, Y = 2, and Z = 3.)

3.3.2 - Auto incrémentation et décrémentation

Le C, et donc le C++, sont pleins des raccourcis. Les raccourcis peuvent rendre le code beaucoup plus facile à écrire et parfois plus difficile à lire. Peut-être les concepteurs du langage C ont-ils pensé qu'il serait plus facile de comprendre un morceau de code astucieux si vos yeux ne devaient pas balayer une large zone d'affichage.

L'un des raccourcis les plus intéressants sont les opérateurs d'auto-incrémentation et d'auto-décrementation. On emploie souvent ces derniers pour modifier les variables de boucle, qui commandent le nombre d'exécution d'une boucle.

L'opérateur d'auto-décrementation est ' **--**' et veut dire " diminuer d'une unité. " l'opérateur d'auto-incrémentation est le ' **++**' et veut dire "augmentation d'une unité." Si **A**est un **int**, par exemple, l'expression **++A**est équivalente à (**A = A + 1**). Les opérateurs Auto-incrémentation et auto-décrementation produisent comme résultat la valeur de la variable. Si l'opérateur apparaît avant la variable, (c.-à-d., **++A**), l'opération est effectuée d'abord puis la valeur résultante est produite. Si l'opérateur apparaît après la variable (c.-à-d. **A++**), la valeur courante est produite, puis l'opération est effectuée. Par exemple :

```
//: C03:AutoIncrement.cpp
// montre l'utilisation des operateurs d'auto-incrémentation
// et auto-décrementation .
#include <iostream>
using namespace std;

int main() {
  int i = 0;
  int j = 0;
  cout << ++i << endl; // Pre-incrementation
  cout << j++ << endl; // Post-incrementation
  cout << --i << endl; // Pre-décrementation
  cout << j-- << endl; // Post décrementation
} ///:~
```

Si vous vous êtes déjà interrogés sur le mot "C++," maintenant vous comprenez. Il signifie "une étape au delà de C. "

3.4 - Introduction aux types de données

Les types de données définissent la façon dont vous utilisez le stockage (mémoire) dans les programmes que vous écrivez. En spécifiant un type de données, vous donnez au compilateur la manière de créer un espace de stockage particulier ainsi que la façon de manipuler cet espace.

Les types de données peuvent être intégrés ou abstraits. Un type de données intégré est compris intrinsèquement par le compilateur, et codé directement dans le compilateur. Les types de données intégrés sont quasiment identiques en C et en C++. À l'opposé, un type défini par l'utilisateur correspond à une classe créée par vous ou par un autre programmeur. On les appelle en général des types de données abstraits. Le compilateur sait comment gérer les types intégrés lorsqu'il démarre ; il « apprend » à gérer les types de données abstraits en lisant les fichiers d'en-tête contenant les déclarations des classes (vous étudierez ce sujet dans les chapitres suivants).

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

3.4.1 - Types intégrés de base

La spécification du C standard (dont hérite le C++) pour les types intégrés ne mentionne pas le nombre de bits que chaque type intégré doit pouvoir contenir. À la place, elle stipule les valeurs minimales et maximales que le type intégré peut prendre. Lorsqu'une machine fonctionne en binaire, cette valeur maximale peut être directement traduite en un nombre minimal de bits requis pour stocker cette valeur. Cependant, si une machine utilise, par exemple, le système décimal codé en binaire (BCD) pour représenter les nombres, la quantité d'espace nécessaire pour stocker les nombres maximum de chaque type sera différente. Les valeurs minimales et maximales pouvant être stockées dans les différents types de données sont définis dans les fichiers d'en-tête du système **limits.h**et **float.h**(en C++ vous incluerez généralement **climits**et **cfloat**à la place).

Le C et le C++ ont quatre types intégrés de base, décrits ici pour les machines fonctionnant en binaire. Un **char**est fait pour le stockage des caractères et utilise au minimum 8 bits (un octet) de stockage, mais peut être plus grand. Un **int**stocke un nombre entier et utilise au minumum deux octets de stockage. Les types **float**et **double**stockent des nombres à virgule flottante, habituellement dans le format IEEE. **float**est prévu pour les flottants simple précision et **double**est prévu pour les flottants double précision.

Comme mentionné précédemment, vous pouvez définir des variables partout dans une portée, et vous pouvez les définir et les initialiser en même temps. Voici comment définir des variables utilisant les quatre types de données de base :

```
                          //: C03:Basic.cpp
// Utilisation des quatre
// types de données de base en C en C++

int main() {
  // Définition sans initialisation :
  char proteine;
  int carbohydrates;
  float fibre;
  double graisse;
  // Définition & initialisation simultannées :
  char pizza = 'A', soda = 'Z';
  int machin = 100, truc = 150,
    chose = 200;
  float chocolat = 3.14159;
  // Notation exponentielle :
  double ration_de_creme = 6e-4;
} ///:~
```

La première partie du programme définit des variables des quatre types de données de base sans les initialiser. Si vous n'initialisez pas une variable, le standard indique que son contenu n'est pas défini (ce qui signifie en général qu'elle contient n'importe quoi). La seconde partie du programme définit et initialise en même temps des variables (c'est toujours mieux, si possible, de donner une valeur initiale au moment de la définition). Notez l'utilisation de la notation exponentielle dans la constant 6e-4, signifiant « 6 fois 10 puissance -4 »

3.4.2 - bool, true, & false

Avant que **bool**fasse partie du C++ standard, tout le monde avait tendance à utiliser des techniques différentes pour obtenir un comportement booléen. Ces techniques causaient des problèmes de portabilité et pouvait introduire des erreurs subtiles.

Le type **bool**du C++ standard possède deux états, exprimés par les constantes intégrées **true**(qui est convertie en l'entier 1) et **false**(qui est convertie en l'entier 0). De plus, certains éléments du langage ont été adaptés :

Comme il existe une grande quantité de code qui utilise un **int**pour représenter un marqueur, le compilateur

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

convertira implicitement un **int**en **bool**(les valeurs non nulles produisent **true**tandis que les valeurs nulles produisent **false**). Idéalement, le compilateur vous avertira pour vous suggérer de corriger cette situation.

Un idiome, considéré comme un « mauvais style de programmation », est d'utiliser **++**pour mettre la valeur d'un marqueur à vrai. Cet idiome est encore autorisé, mais *déprécié*, ce qui signifie qu'il deviendra illégal dans le futur. Le problème vient du fait que vous réalisez une conversion implicite de **bool**vers **int**en incrémentant la valeur (potentiellement au-delà de l'intervalle normal des valeurs de **bool**, 0 et 1), puis la convertissez implicitement dans l'autre sens.

Les pointeurs (qui seront introduits plus tard dans ce chapitre) sont également convertis en **bool**lorsque c'est nécessaire.

3.4.3 - Spécificateurs

Les spécificateurs modifient la signification des types intégrés de base et les étendent pour former un ensemble plus grand. Il existe quatre spécificateurs : **long**, **short**, **signed**et **unsigned**.

longet **short**modifient les valeurs maximales et minimales qu'un type de données peut stocker. Un **int**simple doit être au moins de la taille d'un **short**. La hiérarchie des tailles des types entier est la suivante : **short int**, **int**, **long int**. Toutes les tailles peuvent être les mêmes, tant qu'elles respectent les conditions sur les valeurs minimales et maximales. Sur une machine avec des mots de 64 bits, par exemple, tous les types de données peuvent être longs de 64 bits.

La hiérarchie des tailles pour les nombres à virgule flottante est : **float**, **double**et **long double**. « long float » n'est pas un type légal. Il n'y a pas de flottants **short**.

Les spécificateurs **signed**et **unsigned**donnent au compilateur la manière de traiter le bit de signe des types entiers et des caractères (les nombres à virgule flottante ont toujours un signe). Un nombre **unsigned**n'a pas de signe et a donc un bit en plus de disponible ; il peut ainsi stocker des nombres positifs deux fois plus grands que les nombres positifs qui peuvent être stockés dans un nombre **signed**. **signed**est implicite, sauf pour **char**; **char**peut être implicitement signé ou non. En spécifiant **signed char**, vous forcez l'utilisation du bit de signe.

L'exemple suivant montre les tailles en octet des types de données en utilisant l'opérateur **sizeof**, introduit plus tard dans ce chapitre :

```
                        //: C03:Specify.cpp
// Montre l'utilisation des spécificateurs
#include <iostream>
using namespace std;

int main() {
  char c;
  unsigned char cu;
  int i;
  unsigned int iu;
  short int is;
  short iis; // Même chose que short int
  unsigned short int isu;
  unsigned short iisu;
  long int il;
  long iil;  // Même chose que long int
  unsigned long int ilu;
  unsigned long iilu;
  float f;
  double d;
  long double ld;
  cout
    << "\n char= " << sizeof(c)
    << "\n unsigned char = " << sizeof(cu)
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
      << "\n int = " << sizeof(i)
      << "\n unsigned int = " << sizeof(iu)
      << "\n short = " << sizeof(is)
      << "\n unsigned short = " << sizeof(isu)
      << "\n long = " << sizeof(il)
      << "\n unsigned long = " << sizeof(ilu)
      << "\n float = " << sizeof(f)
      << "\n double = " << sizeof(d)
      << "\n long double = " << sizeof(ld)
      << endl;
} ///:~
```

Notez que les résultats donnés par ce programme seront probablement différents d'une machine à l'autre, car (comme mentionné précédemment), la seule condition qui doit être respectée est que chaque type puisse stocker les valeurs minimales et maximales spécifiées dans le standard.

Lorsque vous modifiez un **int**par **short**ou par **long**, le mot-clé **int**est facultatif, comme montré ci-dessus.

3.4.4 - Introduction aux pointeurs

À chaque fois que vous lancez un programme, il est chargé dans la mémoire de l'ordinateur (en général depuis le disque). Ainsi, tous les éléments du programme sont situés quelque part dans la mémoire. La mémoire est généralement arrangée comme une suite séquentielle d'emplacements mémoire ; nous faisons d'habitude référence à ces emplacements par des *octets*de huit bits, mais la taille de chaque espace dépend en fait de l'architecture particulière d'une machine et est en général appelée la *taille du mot*de cette machine. Chaque espace peut être distingué de façon unique de tous les autres espaces par son *adresse*. Au cours de cette discussion, nous considérerons que toutes les machines utilisent des octets qui ont des adresses séquentielles commençant à zéro et s'étendant jusqu'à la fin de la mémoire disponible dans l'ordinateur.

Puisque votre programme réside en mémoire au cours de son exécution, chaque élément du programme a une adresse. Supposons que l'on démarre avec un programme simple :

```
                        //: C03:YourPets1.cpp
#include <iostream>
using namespace std;

int chien, chat, oiseau, poisson;

void f(int animal) {
  cout << "identifiant de l'animal : " << animal << endl;
}

int main() {
  int i, j, k;
} ///:~
```

Chaque élément de ce programme se voit attribuer un emplacement mémoire à l'exécution du programme. Même la fonction occupe de la place mémoire. Comme vous le verrez, il s'avère que la nature d'un élément et la façon dont vous le définissez détermine en général la zone de mémoire dans laquelle cet élément est placé.

Il existe un opérateur en C et et C++ qui vous donne l'adresse d'un élément. Il s'agit de l'opérateur ‘ **&**’. Tout ce que vous avez à faire est de faire précéder le nom de l'identifiant par ‘ **&**’ et cela produira l'adresse de cet identifiant. **YourPets.cpp**peut être modifié pour afficher l'adresse de tous ses éléments, de cette façon :

```
                        //: C03:YourPets2.cpp
#include <iostream>
using namespace std;

int chien, chat, oiseau, poisson;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
void f(int pet) {
  cout << "identifiant de l'animal : " << pet << endl;
}

int main() {
  int i, j, k;
  cout << "f() : " << (long)&f << endl;
  cout << "chien : " << (long)&chien << endl;
  cout << "chat : " << (long)&chat << endl;
  cout << "oiseau : " << (long)&oiseau << endl;
  cout << "poisson : " << (long)&poisson << endl;
  cout << "i : " << (long)&i << endl;
  cout << "j : " << (long)&j << endl;
  cout << "k : " << (long)&k << endl;
} ///:~
```

L'expression **(long)**est une conversion. Cela signifie « ne considère pas ceci comme son type d'origine, considère le comme un **long**». La conversion n'est pas obligatoire, mais si elle n'était pas présente, les adresses auraient été affichées en hexadécimal, et la conversion en **long**rend les choses un peu plus lisibles.

Les résultats de ce programme varient en fonction de votre ordinateur, de votre système et d'autres facteurs, mais ils vous donneront toujours des informations intéressantes. Pour une exécution donnée sur mon ordinateur, les résultats étaient les suivants :

```
                        f(): 4198736
chien  : 4323632
chat : 4323636
oiseau : 4323640
poisson : 4323644
i: 6684160
j: 6684156
k: 6684152
```

Vous pouvez remarquer que les variables définies dans **main( )**sont dans une zone différente des variables définies en dehors de **main( )**; vous comprendrez la raison en apprenant plus sur ce langage. De plus, **f( )**semble être dans sa propre zone ; en mémoire, le code est généralement séparé des données.

Notez également que les variables définies l'une après l'autre semblent être placées séquentiellement en mémoire. Elles sont séparées par le nombre d'octets dicté par leur type de donnée. Ici, le seul type utilisé est **int**, et **chat**est à quatre octets de **chien**, **oiseau**est à quatre octets de **chat**, etc. Il semble donc que, sur cette machine, un **int**est long de quatre octets.

En plus de cette expérience intéressante montrant l'agencement de la mémoire, que pouvez-vous faire avec une adresse ? La chose la plus importante que vous pouvez faire est de la stocker dans une autre variable pour vous en servir plus tard. Le C et le C++ ont un type spécial de variable pour contenir une adresse. Cette variable est appelée un *pointeur*.

L'opérateur qui définit un pointeur est le même que celui utilisé pour la multiplication, ‘ *****’. Le compilateur sait que ce n'est pas une multiplication grace au contexte dans lequel il est utilisé, comme vous allez le voir.

Lorsque vous définissez un pointeur, vous devez spécifier le type de variable sur lequel il pointe. Vous donnez d'abord le nom du type, puis, au lieu de donner immédiatement un identifiant pour la variable, vous dites « Attention, c'est un pointeur » en insérant une étoile entre le type et l'identifiant. Un pointeur sur un **int**ressemble donc à ceci :

```
int* ip; // ip pointe sur une variable de type int
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

L'association de l'opérateur ' ***' a l'air raisonnable et se lit facilement mais peut induire en erreur. Vous pourriez être enclins à penser à « pointeurSurInt » comme un type de données distinct. Cependant, avec un **int**ou un autre type de données de base, il est possible d'écrire :

```
int a, b, c;
```

tandis qu'avec un pointeur, vous *aimeriez*écrire :

```
int* ipa, ipb, ipc;
```

La syntaxe du C (et par héritage celle du C++) ne permet pas ce genre d'expressions intuitives. Dans les définitions ci-dessus, seul **ipa**est un pointeur, tandis que **ipb**et **ipc**sont des **int**ordinaires (on peut dire que « * est lié plus fortement à l'identifiant »). Par conséquent, les meilleurs résultats sont obtenus en ne mettant qu'une définition par ligne ; vous obtiendrez ainsi la syntaxe intuitive sans la confusion :

```
int* ipa;
int* ipb;
int* ipc;
```

Comme une recommendation générale pour la programmation en C++ est de toujours initialiser une variable au moment de sa définition, cette forme fonctionne mieux. Par exemple, les variables ci-dessus ne sont pas initialisées à une valeur particulière ; elles contiennent n'importe quoi. Il est plus correct d'écrire quelque chose du genre :

```
int a = 47;
int* ipa = &a;
```

De cette façon, **a**et **ipa**ont été initialisés, et **ipa**contient l'adresse de **a**.

Une fois que vous avez un pointeur initialisé, son utilisation la plus élémentaire est de modifier la valeur sur laquelle il pointe. Pour accéder à une variable par un pointeur, on *déréférence*le pointeur en utilisant le même opérateur que pour le définir, de la façon suivante :

```
*ipa = 100;
```

Maintenant, **a**contient la valeur 100 à la place de 47.

Vous venez de découvrir les bases des pointeurs : vous pouvez stocker une adresse et utiliser cette adresse pour modifier la variable d'origine. Une question reste en suspens : pourquoi vouloir modifier une variable en utilisant une autre variable comme intermédiaire ?

Dans le cadre de cette introduction aux pointeurs, on peut classer la réponse dans deux grandes catégories :

1. Pour changer des « objets extérieurs » depuis une fonction. Ceci est probablement l'usage le plus courant des pointeurs et va être présenté maintenant.
2. Pour d'autres techniques de programmation avancées, qui seront présentées en partie dans le reste de ce livre.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

3.4.5 - Modification d'objets extérieurs

Habituellement, lorsque vous passez un argument à une fonction, une copie de cet argument est faite à l'intérieur de la fonction. Ceci est appelé le *passage par valeur*. Vous pouvez en voir les effets dans le programme suivant :

```
                        //: C03:PassByValue.cpp
#include <iostream>
using namespace std;

void f(int a) {
  cout << "a = " << a << endl;
  a = 5;
  cout << "a = " << a << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  f(x);
  cout << "x = " << x << endl;
} ///:~
```

Dans **f( )**, **a**est une *variable locale*, elle n'existe donc que durant l'appel à la fonction **f( )**. Comme c'est un argument de fonction, la valeur de **a**est initialisée par les arguments qui sont passés lorsque la fonction est appelée ; dans **main( )**l'argument est **x**, qui a une valeur de 47, et cette valeur est copiée dans **a**lorsque **f( )**est appelée.

En exécutant ce programme, vous verrez :

```
                        x = 47
a = 47
a = 5
x = 47
```

La valeur initiale de **x**est bien sûr 47. Lorsque **f()**est appelée, un espace temporaire est créé pour stocker la variable **a**pour la durée de l'appel de fonction, et **a**est initialisée en copiant la valeur de **x**, ce qui est vérifié par l'affichage. Bien sûr, vous pouvez changer la valeur de **a**et montrer que cette valeur a changé. Mais lorsque **f( )**se termine, l'espace temporaire qui a été créé pour **a**disparait, et on s'aperçoit que la seule connexion qui existait entre **a**et **x**avait lieu lorsque la valeur de **x**était copiée dans **a**.

À l'intérieur de **f( )**, **x**est l'objet extérieur (dans ma terminologie) et, naturellement, une modification de la variable locale n'affecte pas l'objet extérieur, puisqu'ils sont à deux emplacements différents du stockage. Que faire si vous **voulez**modifier un objet extérieur ? C'est là que les pointeurs se révèlent utiles. D'une certaine manière, un pointeur est un synonyme pour une autre variable. En passant un *pointeur*à une fonction à la place d'une valeur ordinaire, nous lui passons un synonyme de l'objet extérieur, permettant à la fonction de modifier cet objet, de la façon suivante :

```
                        //: C03:PassAddress.cpp
#include <iostream>
using namespace std;

void f(int* p) {
  cout << "p = " << p << endl;
  cout << "*p = " << *p << endl;
  *p = 5;
  cout << "p = " << p << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  cout << "&x = " << &x << endl;
  f(&x);
  cout << "x = " << x << endl;
} ///:~
```

De cette façon, **f( )**prend un pointeur en argument, et déréférence ce pointeur pendant l'affectation, ce qui cause la modification de l'objet extérieur **x**. Le résultat est :

```
                        x = 47
&ax = 0065FE00
p = 0065FE00
*p = 47
p = 0065FE00
x = 5
```

Notez que la valeur contenue dans **p**est la même que l'adresse de **x**– le pointeur **p**pointe en effet sur **x**. Si cela n'est pas suffisamment convaincant, lorsque **p**est déréférencé pour lui affecter la valeur 5, nous voyons que la valeur de **x**est également changée en 5.

Par conséquent, passer un pointeur à une fonction permet à cette fonction de modifier l'objet extérieur. Vous découvrirez beaucoup d'autres utilisations pour les pointeurs par la suite, mais ceci est sans doute la plus simple et la plus utilisée.

3.4.6 - Introduction aux références en C++

Les pointeurs fonctionnent globalement de la même façon en C et en C++, mais le C++ ajoute une autre manière de passer une adresse à une fonction. Il s'agit du *passage par référence*, qui existe dans plusieurs autres langages, et n'est donc pas une invention du C++.

Votre première impression sur les références peut être qu'elles sont inutiles, et que vous pourriez écrire tous vos programmes sans références. En général ceci est vrai, à l'exception de quelques cas importants présentés dans la suite de ce livre. Vous en apprendrez également plus sur les références plus tard, mais l'idée de base est la même que pour la démonstration sur l'utilisation des pointeurs ci-dessus : vous pouvez passer l'adresse d'un argument en utilisant une référence. La différence entre les références et les pointeurs est que l' *appel*d'une fonction qui prend des références est plus propre au niveau de la syntaxe que celui d'une fonction qui prend des pointeurs (et c'est cette même différence syntaxique qui rend les références indispensables dans certaines situations). Si **PassAddress.cpp**est modifié pour utiliser des références, vous pouvez voir la différence d'appel de fonction dans **main( )**:

```
                        //: C03:PassReference.cpp
#include <iostream>
using namespace std;

void f(int& r) {
  cout << "r = " << r << endl;
  cout << "&r = " << &r << endl;
  r = 5;
  cout << "r = " << r << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(x); // Ressemble à un passage par valeur
        // c'est en fait un passage par référence
  cout << "x = " << x << endl;
} ///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Dans la liste d'arguments de **f( )**, à la place d'écrire **int***pour passer un pointeur, on écrit **int&**pour passer une référence. À l'intérieur de **f( )**, en écrivant simplement ' **r**' (ce qui donnerait l'adresse si **r**était un pointeur), vous *récupérez la valeur de la variable que* ***r****référence* . En affectant quelque chose à **r**, vous affectez cette chose à la variable que **r**référence. La seule manière de récupérer l'adresse contenue dans **r**est d'utiliser l'opérateur ' **&**'.

Dans **main( )**, vous pouvez voir l'effet principal de l'utilisation des références dans la syntaxe de l'appel à **f( )**, qui se ramène à **f(x)**. Bien que cela ressemble à un passage par valeur ordinaire, la référence fait que l'appel prend l'adresse et la transmet, plutôt que de simplement copier la valeur. La sortie est :

```
                        x = 47
&x = 0065FE00
r = 47
&r = 0065FE00
r = 5
x = 5
```

Vous pouvez ainsi voir que le passage par référence permet à une fonction de modifier l'objet extérieur à l'instar d'un pointeur (vous pouvez aussi voir que la référence cache le passage de l'adresse, ceci sera examiné plus tard dans ce livre). Pour les besoins de cette introduction simple, vous pouvez considérer que les références ne sont qu'une autre syntaxe (ceci est parfois appelé « sucre syntactique ») pour réaliser ce que font les pointeurs : permettre aux fonctions de changer des objets extérieurs.

3.4.7 - Pointeurs et références comme modificateurs

Jusqu'à maintenant, vous avez découvert les types de données de base **char**, **int**, **float**et **double**, ainsi que les spécificateurs **signed**, **unsigned**, **short**et **long**qui peuvent être utilisés avec les types de données de base dans de nombreuses combinaisons. Nous venons d'ajouter les pointeurs et les références, qui sont orthogonaux aux types de données de base et aux spécificateurs et qui donnent donc un nombre de combinaisons triplé :

```
                        //: C03:AllDefinitions.cpp
// Toutes les définitions possibles des types de données
// de base, des spécificateurs, pointeurs et références
#include <iostream>
using namespace std;

void f1(char c, int i, float f, double d);
void f2(short int si, long int li, long double ld);
void f3(unsigned char uc, unsigned int ui,
  unsigned short int usi, unsigned long int uli);
void f4(char* cp, int* ip, float* fp, double* dp);
void f5(short int* sip, long int* lip,
  long double* ldp);
void f6(unsigned char* ucp, unsigned int* uip,
  unsigned short int* usip,
  unsigned long int* ulip);
void f7(char& cr, int& ir, float& fr, double& dr);
void f8(short int& sir, long int& lir,
  long double& ldr);
void f9(unsigned char& ucr, unsigned int& uir,
  unsigned short int& usir,
  unsigned long int& ulir);

int main() {} ///:~
```

Les pointeurs et les références peuvent aussi être utilisés pour passer des objets dans une fonction et retourner des objets depuis une fonction ; ceci sera abordé dans un chapitre suivant.

Il existe un autre type fonctionnant avec les pointeurs : **void**. En écrivant qu'un pointeur est un **void***, cela signifie que n'importe quel type d'adresse peut être affecté à ce pointeur (tandis que si avez un **int***, vous ne pouvez affecter que l'adresse d'une variable **int**à ce pointeur). Par exemple :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
//: C03:VoidPointer.cpp
int main() {
  void* vp;
  char c;
  int i;
  float f;
  double d;
  // L'adresse de n'importe quel type
  // peut être affectée à un pointeur void
  vp = &c;
  vp = &i;
  vp = &f;
  vp = &d;
} ///:~
```

Une fois que vous affectez une adresse à un **void***, vous perdez l'information du type de l'adresse. Par conséquent, avant d'utiliser le pointeur, vous devez le convertir dans le type correct :

```
//: C03:CastFromVoidPointer.cpp
int main() {
  int i = 99;
  void* vp = &i;
  // On ne peut pas déréférencer un pointeur void
  // *vp = 3; // Erreur de compilation
  // Il faut le convertir en int avant de le déréférencer
  *((int*)vp) = 3;
} ///:~
```

La conversion **(int*)vp**dit au compilateur de traiter le **void***comme un **int***, de façon à ce qu'il puisse être déréférencé. Vous pouvez considérer que cette syntaxe est laide, et elle l'est, mais il y a pire – le **void***crée un trou dans le système de types du langage. En effet, il permet, et même promeut, le traitement d'un type comme un autre type. Dans l'exemple ci-dessus, je traite un **int**comme un **int**en convertissant **vp**en **int***, mais rien ne m'empêche de le convertir en **char***ou en **double***, ce qui modifierait une zone de stockage d'une taille différente que celle qui a été allouée pour l' **int**, faisant potientiellement planter le programme. En général, les pointeurs sur **void**devraient être évités et n'être utilisés que dans des cas bien précis que vous ne rencontrerez que bien plus tard dans ce livre.

Vous ne pouvez pas créer de références sur **void**, pour des raisons qui seront expliquées au chapitre 11.

3.5 - Portée des variables

Les règles de portée d'une variable nous expliquent la durée de validité d'une variable, quand elle est créée, et quand elle est détruite (i.e.: lorsqu'elle sort de la portée). La portée d'une variable s'étend du point où elle est définie jusqu'à la première accolade "fermante" qui correspond à la plus proche accolade "ouvrante" précédant la définition de la variable. En d'autres termes, la portée est définie par le plus proche couple d'accolades entourant la variable. L'exemple qui suit, illustre ce sujet:

```
//: C03:Scope.cpp
// Portée des variables
int main() {
  int scp1;
  // scp1 est utilisable ici
  {
    // scp1 est encore utilisable ici
    //.....
    int scp2;
    // scp2 est utilisable ici
    //.....
    {
      // scp1 & scp2 sont toujours utilisables ici
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
        //..
        int scp3;
        // scp1, scp2 & scp3 sont utilisables ici
        // ...
      } // <-- scp3 est détruite ici
      // scp3 n'est plus utilisable ici
      // scp1 & scp2 sont toujours utilisables ici
      // ...
    } // <-- scp2 est détruite ici
    // scp3 & scp2 ne sont plus utilisables ici
    // scp1 est toujours utilisable ici
    //..
  } // <-- scp1 est détruite ici
  ///:~
```

L'exemple ci-dessus montre quand les variables sont utilisables (on parle aussi de visibilité) et quand elles ne sont plus utilisables (quand elles sortent de la portée). Une variable ne peut être utilisée qu'à l'intérieur de sa portée. Les portées peuvent être imbriquées, indiquées par une paire d'accolades à l'intérieur d'autres paires d'accolades. Imbriqué veut dire que vous pouvez accéder à une variable se trouvant dans la portée qui englobe la portée dans laquelle vous vous trouvez. Dans l'exemple ci-dessus, la variable **scp1**est utilisable dans toutes les portées alors que la variable **scp3**n'est utilisable que dans la portée la plus imbriquée.

3.6 - Définir des variables "à la volée"

Comme expliqué plus tôt dans ce chapitre, il y a une différence significative entre C et C++ dans la définition des variables. Les deux langages requièrent que les variables soient définies avant qu'elles ne soient utilisées, mais C (et beaucoup d'autres langages procéduraux) vous oblige à définir toutes les variables en début de portée, ainsi lorsque le compilateur crée un bloc, il peut allouer la mémoire pour ces variables.

Quand on lit du code C, un bloc de définition de variables est habituellement la première chose que vous voyez quand vous entrez dans une portée. Déclarer toutes les variables au début du bloc demande, de la part du programmeur, d'écrire d'une façon particulière, à cause des détails d'implémentation du langage. La plupart des programmeurs ne savent pas quelles variables vont être utilisées avant d'écrire le code, ainsi ils doivent remonter au début du bloc pour insérer de nouvelles variables ce qui est maladroit et source d'erreurs. Ces définitions de variables en amont ne sont pas très utiles pour le lecteur, et elles créent la confusion parce qu'elles apparaissent loin du contexte où elles sont utilisées.

C++ (mais pas C) vous autorise à définir une variable n'importe où dans la portée, ainsi vous pouvez définir une variable juste avant de l'utiliser. De plus, vous pouvez initialiser la variable lors de sa définition, ce qui évite un certain type d'erreur. Définir les variables de cette façon, rend le code plus facile à écrire et réduit les erreurs que vous obtenez quand vous effectuez des allers-retours dans la portée. Le code est plus facile à comprendre car vous voyez la définition d'une variable dans son contexte d'utilisation. Ceci est particulièrement important quand vous définissez et initialisez une variable en même temps - vous pouvez comprendre la raison de cette initialisation grâce à la façon dont cette variable est utilisée.

Vous pouvez aussi définir les variables à l'intérieur des expressions de contrôle de boucle **for**ou de boucle **while**, à l'intérieur d'un segment conditionnel **if**et à l'intérieur d'une sélection **switch**.Voici un exemple de définition de variables "à la volée":

```
//: C03:OnTheFly.cpp
// Définitions de variables à la volée
#include <iostream>
using namespace std;

int main() {
  //..
  { // Commence une nouvelle portée
    int q = 0; // C demande les définitions de variables ici
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
      //..
      // Définition à l'endroit de l'utilisation
      for(int i = 0; i < 100; i++) {
        q++; // q provient d'une portée plus grande
        // Définition à la fin d'une portée
        int p = 12;
      }
      int p = 1;  // Un p différent
    } // Fin de la portée contenant q et le p extérieur
    cout << "Tapez un caractère:" << endl;
    while(char c = cin.get() != 'q') {
      cout << c << " n'est ce pas ?" << endl;
      if(char x = c == 'a' || c == 'b')
        cout << "Vous avez tapé a ou b" << endl;
      else
        cout << "Vous avez tapé" << x << endl;
    }
    cout << "Tapez A, B, ou C" << endl;
    switch(int i = cin.get()) {
      case 'A': cout << "Snap" << endl; break;
      case 'B': cout << "Crackle" << endl; break;
      case 'C': cout << "Pop" << endl; break;
      default: cout << "Ni A, B ou C!" << endl;
    }
  } ///:~
```

Dans la portée la plus intérieure, **p**est défini juste avant la fin de la portée, c'est réellement sans intérêt (mais cela montre que vous pouvez définir une variable n'importe où). Le **p**de la portée extérieure est dans la même situation.

La définition de **i**dans l'expression de contrôle de la boucle **for**est un exemple de la possibilité de définir une variable *exactement*à l'endroit où vous en avez besoin (ceci n'est possible qu'en C++). La portée de **i**est la portée de l'expression contrôlée par la boucle **for**, ainsi vous pouvez réutiliser **i**dans une prochaine boucle **for**. Ceci est pratique et communément utilisé en C++ : **i**est un nom de variable classique pour les compteurs de boucle et vous n'avez pas besoin d'inventer de nouveaux noms.

Bien que l'exemple montre également la définition de variables dans les expressions **while**, **if**et **switch**, ce type de définition est moins courant, probablement parce que la syntaxe est contraignante. Par exemple, vous ne pouvez pas mettre de parenthèses. Autrement dit, vous ne pouvez pas écrire :

```
while((char c = cin.get()) != 'q')
```

L'addition de parenthèses supplémentaires peut sembler innocent et efficace, mais vous ne pouvez pas les utiliser car les résultats ne sont pas ceux escomptés. Le problème vient du fait que ' **!=**' a une priorité supérieure à ' **=**', ainsi le **char c**renvoie un **bool**convertit en **char**. A l'écran, sur de nombreux terminaux, vous obtiendrez un caractère de type "smiley".

En général, vous pouvez considérer que cette possibilité de définir des variables dans les expressions **while**, **if**et **switch**n'est là que pour la beauté du geste mais vous utiliserez ce type de définition dans une boucle **for**(où vous l'utiliserez très souvent).

3.6 - Définir l'allocation mémoire

Quand vous créez une variable, vous disposez de plusieurs options pour préciser sa durée de vie, comment la mémoire est allouée pour cette variable, et comment la variable est traitée par le compilateur.

3.6.1 - Variables globales

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Les variables globales sont définies hors de tout corps de fonction et sont disponibles pour tous les éléments du programme (même le code d'autres fichiers). Les variables globales ne sont pas affectées par les portées et sont toujours disponibles (autrement dit, une variable globale dure jusqu'à la fin du programme). Si une variable globale est déclarée dans un fichier au moyen du mot-clé **extern**et définie dans un autre fichier, la donnée peut être utilisée par le second fichier. Voici un exemple d'utilisation de variables globales :

```
                        //: C03:Global.cpp
//{L} Global2
// Exemple de variables globales
#include <iostream>
using namespace std;

int globe;
void func();
int main() {
  globe = 12;
  cout << globe << endl;
  func(); // Modifies globe
  cout << globe << endl;
} ///:~
```

Ici un fichier qui accède à **globe**comme un **extern**:

```
                        //: C03:Global2.cpp {O}
// Accès aux variables globales externes
extern int globe;
// (The linker resolves the reference)
void func() {
  globe = 47;
} ///:~
```

Le stockage de la variable **globe**est créé par la définition dans **Global.cpp**, et le code dans **Global2.cpp**accède à cette même variable. Comme le code de **Global2.cpp**est compilé séparément du code de **Global.cpp**, le compilateur doit être informé que la variable existe ailleurs par la déclaration

```
                        extern int globe;
```

A l'exécution du programme, vous verrez que, de fait, l'appel à **func( )**affecte l'unique instance globale de **globe**.

Dans **Global.cpp**, vous pouvez voir la balise de commentaire spéciale (qui est de ma propre conception):

```
                        //{L} Global2
```

Cela dit que pour créer le programme final, le fichier objet **Global2**doit être lié (il n'y a pas d'extension parce que l'extension des fichiers objets diffère d'un système à l'autre). Dans **Global2.cpp**, la première ligne contient aussi une autre balise de commentaire spéciale **{O},**qui dit "n'essayez pas de créer un exécutable à partir de ce fichier, il est en train d'être compilé afin de pouvoir être lié dans un autre exécutable." Le programme **ExtractCode.cpp**dans le deuxième volume de ce livre (téléchargeable à *www.BruceEckel.com*) lit ces balises et crée le **makefile**approprié afin que tout se compile proprement (vous étudierez les **makefile**s à la fin de ce chapitre).

3.6.2 - Variables locales

Les variables locales existent dans un champ limité ; elles sont "locales" à une fonction. Elle sont souvent appelées variables *automatiques*parce qu'elles sont créés automatiquement quand on entre dans le champ et disparaissent automatiquement quand le champ est fermé. Le mot clef **auto**rend la chose explicite, mais les variables locales

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

sont par défaut **auto**afin qu'il ne soit jamais nécessaire de déclarer quelque chose **auto**.

Variables de registre

Une variable de registre est un type de variable locale. Le mot clef **register**dit au compilateur "rend l'accès à cette donnée aussi rapide que possible". L'accroissement de la vitesse d'accès aux données dépend de l'implémentation, mais, comme le suggère le nom, c'est souvent fait en plaçant la variable dans un registre. Il n'y a aucune garantie que la variable sera placée dans un registre ou même que la vitesse d'accès sera augmentée. C'est une suggestion au compilateur.

Il y a des restrictions à l'usage des variables de **registre**. Vous ne pouvez pas prendre ou calculer leur adresse. Elles ne peuvent être déclarées que dans un bloc (vous ne pouvez pas avoir de variables de **registre**globales ou **static**). Toutefois, vous pouvez utiliser une variable de **registre**comme un argument formel dans une fonction (i.e., dans la liste des arguments).

En général, vous ne devriez pas essayer de contrôler l'optimiseur du compilateur, étant donné qu'il fera probablement un meilleur travail que vous. Ainsi, il vaut mieux éviter le mot-clef **register**.

3.6.3 - static

Le mot-clef **static**a différentes significations. Normalement, les variables définies dans une fonction disparaissent à la fin de la fonction. Quand vous appelez une fonction à nouveau, l'espace de stockage pour la variable est recréé et les valeurs ré-initialisées. Si vous voulez qu'une valeur soit étendue à toute la durée de vie d'un programme, vous pouvez définir la variable locale d'une fonction **static**et lui donner une valeur initiale. L'initialisation est effectuée uniquement la première fois que la fonction est appelée, et la donnée conserve sa valeur entre les appels à la fonction. Ainsi, une fonction peut "se souvenir" de morceaux d'information entre les appels.

Vous pouvez vous demander pourquoi une variable globale n'est pas utilisée à la place ? La beauté d'une variable **static**est qu'elle est indisponible en dehors du champ de la fonction et ne peut donc être modifiée par inadvertance. Ceci localise les erreurs.

Voici un exemple d'utilisation des variables **static**:

```
                        //: C03:Static.cpp
// Utiliser une variable static dans une fonction
#include <iostream>
using namespace std;

void func() {
  static int i = 0;
  cout << "i = " << ++i << endl;
}

int main() {
  for(int x = 0; x < 10; x++)
    func();
} ///:~
```

A chaque fois que **func**( ) est appelée dans la boucle for, elle imprime une valeur différente. Si le mot-clef **static**n'est pas utilisé, la valeur utilisée sera toujours ‘1’.

Le deuxième sens de **static**est relié au premier dans le sens “indisponible en dehors d'un certain champ”. Quand **static**est appliqué au nom d'une fonction ou à une variable en dehors de toute fonction, cela signifie “Ce nom est indisponible en dehors de ce fichier.” Le nom de la focntion ou de la variable est local au fichier ; nous disons qu'il a la *portée d'un fichier*. Par exemple, compiler et lier les deux fichiers suivants causera une erreur d'édition de liens

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

:

```
                        //: C03:FileStatic.cpp
// Démonstration de la portée à un fichier. Compiler et
// lier ce fichier avec FileStatic2.cpp
// causera une erreur d'éditeur de liens

// Portée d'un fichier signifie disponible seulement dans ce fichier :
static int fs;

int main() {
  fs = 1;
} ///:~
```

Même si la variable **fs**est déclaré exister comme une **extern**dan le fichier suivant, l'éditeur de liens ne la trouvera pas parce qu'elle a été déclarée **static**dans **FileStatic.cpp**.

```
                        //: C03:FileStatic2.cpp {O}
// Tentative de référencer fs
extern int fs;
void func() {
  fs = 100;
} ///:~
```

Le mot-clef **static**peut aussi être utilisé dans une classe. Ceci sera expliqué plus loin, quand vous aurez appris à créer des classes.

3.6.4 - extern

Le mot-clef **extern**a déjà été brièvement décrit et illustré. Il dit au compilateur qu'une variable ou une fonction existe, même si le compilateur ne l'a pas encore vu dans le fichier en train d'être compilé. Cette variable ou cette fonction peut être définie dans un autre fichier ou plus loin dans le même fichier. Comme exemple du dernier cas :

```
                        //: C03:Forward.cpp
// Fonction forward & déclaration de données
#include <iostream>
using namespace std;

// Ce n'est pas vraiment le cas externe, mais il
// faut dire au compilateur qu'elle existe quelque part :
extern int i;
extern void func();
int main() {
  i = 0;
  func();
}
int i; // Création de la donnée
void func() {
  i++;
  cout << i;
} ///:~
```

Quand le compilateur rencontre la déclaration ‘ **extern int i**’, il sait que la définition de **i**doit exister quelque part comme variable globale. Quand le compilateur atteint la définition de **i**, il n'y a pas d'autre déclaration visible, alors il sait qu'il a trouvé le même **i**déclaré plus tôt dans le fichier. Si vous définissiez **i static**, vous diriez au compilateur que **i**est défini globalement (via **extern**), mais qu'il a aussi une portée de fichier (via **static**), et le compilateur génèrera une erreur.

Edition de lien

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Pour comprendre le comportement des programmes en C et C++, vous devez connaître l'édition de liens (*linkage*). Dans un programme exécutable un identifiant est représenté par un espace mémoire qui contient une variable ou le corps d'une fonction compilée. L'édition de liens décrit cet espace comme il est vu par l'éditeur de liens (*linker*). Il y a deux types d'édition de liens : l'édition de liens *interne*et *externe*.

L'édition de liens interne signifie que l'espace mémoire est créé pour représenter l'identifiant seulement pour le fichier en cours de compilation. D'autres fichiers peuvent utiliser le même nom d'identifiant avec l'édition interne de liens, ou pour une variable globale, et aucun conflit ne sera détecté par l'éditeur de liens – un espace différent est créé pour chaque identifiant. L'édition de liens interne est spécifiée par le mot-clef **static**en C et C++.

L'édition de liens externe signifie qu'un seul espace de stockage est créé pour représenter l'identifiant pour tous les fichiers compilés. L'espace est créé une fois, et l'éditeur de liens doit assigner toutes les autres références à cet espace. Les variables globales et les noms de fonctions ont une édition de liens externe. Ceux-ci sont atteints à partir des autres fichiers en les déclarant avec le mot-clef **extern**. Les variables définies en dehors de toute fonction (à l'exception de **cont**en C++) et les définitions de fonctions relèvent par défaut de l'édition de liens externe. Vous pouvez les forcer spécifiquement à avoir une édition interne de liens en utilisant le mot-clef **static**. Vous pouvez déclarer explicitement qu'un identifiant a une édition de liens externe en le définissant avec le mot-clef **extern**. Définir une variable ou une fonction avec **extern**n'est pas nécessaire en C, mais c'est parfois nécessaire pour **const**en C++.

Les variables (locales) automatiques existent seulement temporairement, sur la pile, quand une fonction est appelée. L'éditeur de liens ne connaît pas les variables automatiques, et celles-ci n'ont donc *pas d'édition de liens*.

3.6.5 - Constantes

Dans l'ancien C (pré-standard), si vous vouliez créer une constante, vous deviez utiliser le préprocesseur :

```
#define PI 3.14159
```

Partout où vous utilisiez **PI**, la valeur 3.14159 était substitué par le préprocesseur (vous pouvez toujours utiliser cette méthode en C et C++).

Quand vous utilisez le préprocesseur pour créer des constantes, vous placez le contrôle de ces constantes hors de la portée du compilateur. Aucune vérification de type n'est effectuée sur le nom **PI**et vous ne pouvez prendre l'adresse de **PI**(donc vous ne pouvez pas passer un pointeur ou une référence à **PI**). **PI**ne peut pas être une variable d'un type défini par l'utlisateur. Le sens de **PI**dure depuis son point de définition jusqu'à la fin du fichier ; le préprocesseur ne sait pas gérer la portée.

C++ introduit le concept de constante nommée comme une variable, sauf que sa valeur ne peut pas être changée. Le modificateur **const**dit au compilateur qu'un nom représente une constante. N'importe quel type de données, prédéfini ou défini par l'utilisateur, peut être défini **const**. Si vous définissez quelque chose **const**et essayez ensuite de le modifier, le compilateur génère une erreur.

Vous devez définir le type de **const**, ainsi :

```
const int x = 10;
```

En C et C++ standard, vous pouvez utiliser une constante nommée dans une liste d'arguments, même si l'argument auquel il correspond est un pointeur ou un référence (i.e., vous pouvez prendre l'adresse d'une **const**). Une **const**a une portée, exactement comme une variable normale, vous pouvez donc "cacher" une **const**dans une

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

fonction et être sûr que le nom n'affectera pas le reste du programme.

consta été emprunté au C++ et incorporé en C standard, mais de façon relativement différente. En C, le compilateur traite une **const**comme une variable qui a une étiquette attachée disant "Ne me changez pas". Quand vous définissez une **const**en C, le compilateur crée un espace pour celle-ci, donc si vous définissez plus d'une **const**avec le même nom dans deux fichiers différents (ou mettez la définition dans un fichier d'en-tête (*header*)), l'éditeur de liens générera des messages d'erreur de conflits. L'usage voulu de **const**en C est assez différent de celui voulu en C++ (pour faire court, c'est plus agréable en C++).

Valeurs constantes

En C++, une **const**doit toujours avoir une valeur d'initialisation (ce n'est pas vrai en C). Les valeurs constantes pour les types prédéfinis sont les types décimal, octal, hexadécimal, nombres à virgule flottante (*floating-point numbers*) (malheureusement, les nombres binaires n'ont pas été considérés importants), ou caractère.

En l'absence d'autre indication, le compilateur suppose qu'une valeur constante est un nombre décimal. Les nombres 47, 0 et 1101 sont tous traités comme des nombres décimaux.

Une valeur constante avec 0 comme premier chiffre est traitée comme un nombre octal (base 8). Les nombres en base 8 peuvent contenir uniquement les chiffres 0-7 ; le compilateur signale les autres chiffres comme des erreurs. Un nombre octal valide est 017 (15 en base 10).

Une valeur constante commençant par 0x est traitée comme un nombre hexadécimal (base 16). Les nombres en base 16 contiennent les chiffres 0 à 9 et les lettres A à F. Un nombre hexadécimal valide peut être 0x1fe (510 en base 10).

Les nombres à virgule flottante peuvent contenir un point décimal et une puissance exponentielle (représentée par e, ce qui veut dire "10 à la puissance"). Le point décimal et le **e**sont tous deux optionnels. Si vous assignez une constante à une variable en virgule flottante, le compilateur prendra la valeur constante et la convertira en un nombre à virgule flottante (ce procédé est une forme de ce que l'on appelle *la conversion de type implicite*). Toutefois, c'est une bonne idée d'utiliser soit un point décimal ou un **e**pour rappeler au lecteur que vous utilisez un nombre à virgule flottante ; des compilateurs plus anciens ont également besoin de cette indication.

Les valeurs constantes à virgule flottante valides sont : 1e4, 1.0001, 47.0, 0.0, et -1.159e-77. Vous pouvez ajouter des suffixes pour forcer le type de nombre à virgule flottante : **f**ou **F**force le type **float**, **L**ou **l**force le type **long double**; autrement le nombre sera un **double**.

Les caractères constants sont des caractères entourés par des apostrophes, comme : ‘ **A**’, ‘ **0**’, ‘ ‘. Remarquer qu'il y a une grande différence entre le caractère ‘ **0**’ (ASCII 96) et la valeur **0**. Des caractères spéciaux sont représentés avec un échappement avec “backslash”: ‘ **\n**’ (nouvelle ligne), ‘ **\t**’ (tabulation), ‘ ****’ (backslash), ‘ **\r**’ (retour chariot), ‘ **"**’ (guillemets), ‘ **'** ’ (apostrophe), etc. Vous pouvez aussi exprimer les caractères constants en octal : ‘ **\17**’ ou hexadecimal : ‘ **\xff**’.

3.6.6 - volatile

Alors que la déclaration **const**dit au compilateur “Cela ne change jamais” (ce qui permet au compilateur d'effectuer des optimisations supplémentaires), la déclaration **volatile**dit au compilateur “On ne peut pas savoir quand cela va changer” et empêche le compilateur d'effectuer des optimisations basée sur la stabilité de cette variable. Utilisez ce mot-clef quand vous lisez une valeur en dehors du contrôle de votre code, comme un registre dans une partie de communication avec le hardware. Une variable **volatile**est toujours lue quand sa valeur est requise, même si elle a été lue à la ligne précédente.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Un cas spécial d'espace mémoire étant “en dehors du contrôle de votre code” est dans un programme multithreadé. Si vous utilisez un flag modifié par une autre thread ou process, ce flag devrait être **volatile**afin que le compilateur ne suppose pas qu'il peut optimiser en négligeant plusieurs lectures des flags.

Notez que **volatile**peut ne pas avoir d'effet quand un compilateur n'optimise pas, mais peut éviter des bugs critiques quand vous commencez à optimiser le code (c'est alors que le compilateur commencera à chercher des lectures redondantes).

Les mots-clefs **const**et **volatile**seront examinés davantage dans un prochain chapitre.

3.7 - Operateurs et leurs usages

Cette section traite de tous les opérateurs en C et en C++.

Tous les opérateurs produisent une valeur à partir de leurs opérandes. Cette valeur est produite sans modifier les opérandes, excepté avec les opérateurs d'affectation, d'incrémentation et de décrémentation. Modifier un opérande est appelé *effet secondaire*. L'usage le plus commun pour les opérateurs est de modifier ses opérandes pour générer l'effet secondaire, mais vous devez garder à l'esprit que la valeur produite est disponible seulement pour votre usage comme dans les opérateurs sans effets secondaires.

3.7.1 - L'affectation

L'affectation est éxécutée par l'opérateur **=**. Il signifie “Prendre le côté droit (souvent appelé la *valeur droite ou rvalue*) et la copier dans le côté gauche (souvent appelé la *valeur gauche ou lvalue*).” Une valeur droite est une constante, une variable, ou une expression produisant une valeur, mais une valeur gauche doit être distinguée par un nom de variable (c'est-à-dire, qu'il doit y avoir un espace physique dans lequel on stocke les données). Par exemple, vous pouvez donner une valeur constante à une variable (**A = 4;**), mais vous ne pouvez rien affecter à une valeur constante – Elle ne peut pas être une valeur l (vous ne pouvez pas écrire **4 = A;**).

3.7.2 - Opérateurs mathématiques

Les opérateurs mathématiques de base sont les mêmes que dans la plupart des langages de programmation: addition (**+**), soustraction (**-**), division (**/**), multiplication (*****), et modulo (**%**; qui retourne le reste de la division entière). La division de entière tronque le résultat (ne provoque pas un arrondi). L'opérateur modulo ne peut pas être utilisé avec des nombres à virgule.

C et C++ utilisent également une notation condensée pour exécuter une opération et une affectation en même temps. On le note au moyen d'un opérateur suivi par le signe égal, et est applicable avec tous les opérateurs du langage (lorsque ceci à du sens). Par exemple, pour ajouter 4 à une variable **x**et affecter **x**au résultat, vous écrivez: **x += 4;**.

Cet exemple montre l'utilisation des opérateurs mathématiques:

```
                        //: C03:Mathops.cpp
// Opérateurs mathématiques
#include <iostream>
using namespace std;

// Une macro pour montrer une chaîne de caractères et une valeur.
#define PRINT(STR, VAR) \
  cout << STR " = " << VAR << endl

int main() {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    int i, j, k;
    float u, v, w;  // Appliquable aussi aux doubles
    cout << "saisir un entier: ";
    cin >> j;
    cout << "saisissez un autre entier: ";
    cin >> k;
    PRINT("j",j);  PRINT("k",k);
    i = j + k; PRINT("j + k",i);
    i = j - k; PRINT("j - k",i);
    i = k / j; PRINT("k / j",i);
    i = k * j; PRINT("k * j",i);
    i = k % j; PRINT("k % j",i);
    // La suite ne fonctionne qu'avec des entiers:
    j %= k; PRINT("j %= k", j);
    cout << "saisissez un nombre à virgule: ";
    cin >> v;
    cout << "saisissez un autre nombre à virgule:";
    cin >> w;
    PRINT("v",v); PRINT("w",w);
    u = v + w; PRINT("v + w", u);
    u = v - w; PRINT("v - w", u);
    u = v * w; PRINT("v * w", u);
    u = v / w; PRINT("v / w", u);
    // La suite fonctionne pour les entiers, les caractères,
    // et les types double aussi:
    PRINT("u", u); PRINT("v", v);
    u += v; PRINT("u += v", u);
    u -= v; PRINT("u -= v", u);
    u *= v; PRINT("u *= v", u);
    u /= v; PRINT("u /= v", u);
} ///:~
```

La valeur droite de toutes les affectations peut, bien sûr, être beaucoup plus complexe.

Introduction aux macros du préprocesseur

Remarquez l'usage de la macro **PRINT( )**pour économiser de la frappe(et les erreurs de frappe!). Les macros pour le préprocesseur sont traditionnellement nommées avec toutes les lettres en majuscules pour faire la différence – vous apprendrez plus tard que les macros peuvent vite devenir dangereuses (et elles peuvent aussi être très utiles).

Les arguments dans les parenthèses suivant le nom de la macro sont substitués dans tout le code suivant la parenthèse fermante. Le préprocesseur supprime le nom **PRINT**et substitue le code partout où la macro est appelée, donc le compilateur ne peut générer aucun message d'erreur utilisant le nom de la macro, et il ne peut vérifier les arguments (ce dernier peut être bénéfique, comme dans la macro de débogage à la fin du chapitre).

3.7.3 - Opérateurs relationnels

Les opérateurs relationnels établissent des relations entre les valeurs des opérandes. Ils produisent un booléen (spécifié avec le mot-clé **bool**en C++) **true**si la relation est vraie, et **false**si la relation est fausse. Les opérateurs relationnels sont: inférieur à (**<**), supérieur à (**>**), inférieur ou égal à (**<=**), supérieur ou égal à (**>=**), équivalent (**==**), et non équivalent (**!=**). Ils peuvent être utilisés avec tous les types de données de base en C et en C++. Ils peuvent avoir des définitions spéciales pour les types de données utilisateurs en C++ (vous l'apprendrez dans le chapitre 12, dans la section surcharge des opérateurs).

3.7.4 - Opérateurs logiques

Les opérateurs logiques *et*(**&&**) et *ou*(**||**) produisent **vrai**ou **faux**selon les relations logiques de ses arguments. Souvenez vous qu'en C et en C++, une expression est **true**si elle a une valeur non-égale à zéro, et **false**si elle a une valeur à zéro. Si vous imprimez un **bool**, vous verrez le plus souvent un ' **1**' pour **true**et ' **0**' pour **false**.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Cet exemple utilise les opérateurs relationnels et les opérateurs logiques:

```
                        //: C03:Boolean.cpp
// Opérateurs relationnels et logiques.
#include <iostream>
using namespace std;

int main() {
  int i,j;
  cout << "Tapez un entier: ";
  cin >> i;
  cout << "Tapez un autre entier: ";
  cin >> j;
  cout << "i > j is " << (i > j) << endl;
  cout << "i < j is " << (i < j) << endl;
  cout << "i >= j is " << (i >= j) << endl;
  cout << "i <= j is " << (i <= j) << endl;
  cout << "i == j is " << (i == j) << endl;
  cout << "i != j is " << (i != j) << endl;
  cout << "i && j is " << (i && j) << endl;
  cout << "i || j is " << (i || j) << endl;
  cout << " (i < 10) && (j < 10) is "
       << ((i < 10) && (j < 10))  << endl;
} ///:~
```

Vous pouvez remplacer la définition de **int**avec **float**ou **double**dans le programme précédent. Soyez vigilant cependant, sur le fait que la comparaison d'un nombre à virgule avec zéro est stricte; Un nombre, aussi près soit-il d'un autre, est toujours ”non égal.” Un nombre à virgule qui est le plus petit possible est toujours vrai.

3.7.5 - Opérateurs bit à bit

Les opérateurs bit à bit vous permettent de manipuler individuellement les bits dans un nombre (comme les valeurs à virgule flottante utilisent un format interne spécifique, les opérateurs de bits travaillent seulement avec des types entiers: **char**, **int**et **long**). Les opérateurs bit à bit exécutent l'algèbre booléene sur les bits correspondant dans les arguments pour produire le résultat.

L'opérateur bit à bit *et*(**&**) donne 1 pour bit se sortie si les deux bits d'entrée valent 1; autrement il produit un zéro. L'opérateur bit à bit ou (**|**) produit un Un sur la sortie si l'un ou l'autre bit est un Un et produit un zéro seulement si les deux bits d'entrés sont à zéro. L'opérateur bit à bit *ou exclusif*, ou *xor*(**^**) produit un Un dans le bit de sortie si l'un ou l'autre bit d'entré est à Un, mais pas les deux. L'opérateur bit à bit *non*(**~**, aussi appelé le *complément de Un*) est un opérateur unaire – Il prend seulement un argument (tous les autres opérateurs bit à bit sont des opérateurs binaires). L'opérateur bit à bit *non*produit l'opposé du bit d'entrée – un Un si le bit d'entré est zéro, un zéro si le bit d'entré est Un.

Les opérateurs bit à bit peuvent être combinés avec le signe **=**pour regrouper l'opération et l'affectation: **&=**, **|=,**et **^=**sont tous des opérations légitimes (comme **~**est un opérateur unitaire, il ne peut pas être combiné avec le signe **=**).

3.7.6 - Opérateurs de décalage

Les opérateurs de décalages manipulent aussi les bits. L'opérateur de décalage à gauche (**<<**) retourne l'opérande situé à gauche de l'opérateur décalé vers la gauche du nombre de bits spécifié après l'opérateur. L'opérateur de décalage à droite (**>>**) retourne l'opérande situé à gauche de l'opérateur décalé vers la droite du nombre de bits spécifié après l'opérateur. Si la valeur après l'opérateur de décalage est supérieure au nombre de bits de l'opérande de gauche, le résultat est indéfini. Si l'opérande de gauche est non signée, le décalage à droite est un décalage logique donc les bits supérieurs seront remplis avec des zéros. Si l'opérande de gauche est signé, le décalage à droite peut être ou non un décalage logique (c'est-à-dire, le comportement est non défini).

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Les décalages peuvent être combinés avec le signe (**<<=**et **>>=**). La valeur gauche est remplacée par la valeur gauche décalé par la valeur droite.

Ce qui suit est un exemple qui démontre l'utilisation de tous les opérateurs impliquant les bits. D'abord, voici une fonction d'usage universel qui imprime un octet dans le format binaire, créée séparément de sorte qu'elle puisse être facilement réutilisée. Le fichier d'en-tête déclare la fonction :

```
                        //: C03:printBinary.h
// imprime un bit au format binaire
void printBinary(const unsigned char val);
///:~
```

Ici; ce trouve l'implémentation de la fonction:

```
                        //: C03:printBinary.cpp {O}
#include <iostream>
void printBinary(const unsigned char val) {
  for(int i = 7; i >= 0; i--)
    if(val & (1 << i))
      std::cout << "1";
    else
      std::cout << "0";
} ///:~
```

La fonction **printBinary( )**prend un simple octet et l'affiche bit par bit. L'expression

```
                        (1 &lt;&lt; i)
```

produit un Un successivement dans chaque position; en binaire: 00000001, 00000010, etc. Si on fait un *et*bit à bit avec **val**et que le résultat est non nul, cela signifie qu'il y avait un Un dans cette position en **val**.

Finalement, la fonction est utilisée dans l'exemple qui montre la manipulation des opérateurs de bits:

```
                        //: C03:Bitwise.cpp
//{L} printBinary
// Démonstration de la manipulation de bit
#include "printBinary.h"
#include <iostream>
using namespace std;

// Une macro pour éviter de la frappe
#define PR(STR, EXPR) \
  cout << STR; printBinary(EXPR); cout << endl;

int main() {
  unsigned int getval;
  unsigned char a, b;
  cout << "Entrer un nombre compris entre 0 et 255: ";
  cin >> getval; a = getval;
  PR("a in binary: ", a);
  cout << "Entrer un nombre compris entre 0 et 255: ";
  cin >> getval; b = getval;
  PR("b en binaire: ", b);
  PR("a | b = ", a | b);
  PR("a & b = ", a & b);
  PR("a ^ b = ", a ^ b);
  PR("~a = ", ~a);
  PR("~b = ", ~b);
  // Une configuration binaire intéressante:
  unsigned char c = 0x5A;
  PR("c en binaire: ", c);
  a |= c;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    PR("a |= c; a = ", a);
    b &= c;
    PR("b &= c; b = ", b);
    b ^= a;
    PR("b ^= a; b = ", b);
} ///:~
```

Une fois encore, une macro préprocesseur est utilisée pour économiser de la frappe. Elle imprime la chaîne de caractère de votre choix, puis la représentation binaire d'une expression, puis une nouvelle ligne.

Dans **main( )**, les variables sont **unsigned**. Parce que, en général, vous ne voulez pas de signe quand vous travaillez avec des octets. Un **int**doit être utilisé au lieu d'un **char**pour **getval**parce que l'instruction " **cin >>**" va sinon traiter le premier chiffre comme un caractère. En affectant **getval**à **a**et **b**, la valeur est convertie en un simple octet(en le tronquant).

Les **<<**et **>>**permettent d'effectuer des décalages de bits, mais quand ils décalent les bits en dehors de la fin du nombre, ces bits sont perdus. Il est commun de dire qu'ils sont tombés dans le *seau des bits perdus*, un endroit où les bits abandonnés finissent, vraisemblablement ainsi ils peuvent être réutilisés...). Quand vous manipulez des bits vous pouvez également exécuter une *rotation*, ce qui signifie que les bits qui sont éjectés d'une extrémité sont réinsérés à l'autre extrémité, comme s'ils faisait une rotation autour d'une boucle. Quoique la plupart des processeurs d'ordinateur produise une commande de rotation au niveau machine (donc vous pouvez voir cela dans un langage d'assembleur pour ce processeur), Il n'y a pas de support direct pour les "rotations" en C et C++. Vraisemblablement les concepteurs du C percevaient comme justifié de laisser les "rotations" en dehors (visant, d'après eux, un langage minimal) parce que vous pouvez construire votre propre commande de rotation. Par exemple, voici les fonctions pour effectuer des rotations gauches et droites:

```
                        //: C03:Rotation.cpp {O}
// effectuer des rotations gauches et droites

unsigned char rol(unsigned char val) {
  int highbit;
  if(val & 0x80) // 0x80 est le bit de poids fort seulement
    highbit = 1;
  else
    highbit = 0;
  // décalage à gauche (le bit de poids faible deviens 0):
  val <<= 1;
  // Rotation du bit de poids fort sur le bit de poids faible:
  val |= highbit;
  return val;
}

unsigned char ror(unsigned char val) {
  int lowbit;
  if(val & 1) // vérifie le bit de poids faible
    lowbit = 1;
  else
    lowbit = 0;
  val >>= 1; // décalage à droite par une position
  // Rotation du bit de poids faible sur le bit de poids fort:
  val |= (lowbit << 7);
  return val;
} ///:~
```

Essayez d'utiliser ces fonctions dans **Bitwise.cpp**. Noter que les définitions (ou au moins les déclarations) de **rol( )**et **ror( )**doivent être vues par le compilateur dans **Bitwise.cpp**avant que les fonctions ne soit utilisées.

Les fonctions bit à bit sont généralement extrêmement efficaces à utiliser parce qu'elles sont directement traduites en langage d'assembleur. Parfois un simple traitement en C ou C++ peut être généré par une simple ligne de code d'assembleur.

3.7.7 - Opérateurs unaires

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

L'opérateur bit à bit *not*n'est pas le seul opérateur qui prend un argument unique. Son compagnon, le *non logique*(**!**), va prendre une valeur **true**et produire une valeur **false**. L'unaire moins (**-**) et l'unaire plus (**+**) sont les même opérateurs que les binaires moins et plus; le compilateur trouve quelle utilisation est demandée en fonction de la façon dout vous écrivez l'expression.Par exemple, le traitement

```
x = -a;
```

a une signification évidente. Le compilateur peut comprendre:

```
x = a * -b;
```

mais le lecteur peut être troublé, donc il est préférable d'écrire:

```
x = a * (-b);
```

L'unaire moins produit l'opposé de la valeur. L'unaire plus produit la symétrie avec l'unaire moins, bien qu'il ne fasse actuellement rien.

Les opérateurs d'incrémentation et de décrémentation (**++**et **--**) ont été introduits plus tôt dans ce chapitre. Ils sont les seuls autres opérateurs hormis ceux impliquant des affectations qui ont des effets de bord. Ces opérateurs incrémentent et décrémentent la variable d'une unité, bien qu'une “unité” puisse avoir différentes significations selon le type de la donnée, c'est particulièrement vrai avec les pointeurs.

Les derniers opérateurs unaires sont adresse-de (**&**),déréférence (***et** **->**), et les opérateurs de transtypage en C et C++, et **new**et **delete**en C++. L'adresse-de et la déréférence sont utilisés avec les pointeurs, décrit dans ce chapitre. Le transtypage est décrit plus tard dans ce chapitre, et **new**et **delete**sont introduits dans ce chapitre 4.

3.7.8 - L'opérateur ternaire

Le ternaire **if-else**est inhabituel parce qu'il a trois opérandes. C'est un vrai opérateur parce qu'il produit une valeur, à la difference de l'instruction ordinaire **if-else**. Il est composé de trois expressions: si la première expression (suivie par **?**) est évaluée à **vrai**, l'expression suivant le **?**est évaluée et son résultat devient la valeur produite par l'opérateur. Si la première expression est **fausse**, la troisième expression (suivant le **:**) est évaluée et le résultat devient la valeur produite par l'opérateur.

L'opérateur conditionnel peut être utilise pour son effet de bord ou pour la valeur qu'il produit. Voici un fragment de code qui démontre cela :

```
a = --b ? b : (b = -99);
```

Ici, la condition produit la valeur droite. **a**est affectée à la valeur de **b**si le résultat de la décrémentation de **b**n'est pas zéro. Si **b**devient zéro, **a**et **b**sont tous les deux assignés à -99. **b**est toujours décrémenté, mais il est assigné à -99 seulement si la décrémentation fait que **b**deviens 0. Un traitement similaire peut être utilisé sans le “ **a =**” juste pour l'effet de bord:

```
--b ? b : (b = -99);
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Ici le second B est superflu, car la valeur produite par l'opérateur n'est pas utilisée. Une expression est requise entre le **?**et le **:**. Dans ce cas, l'expression peut simplement être une constante qui va produire un code un peu plus rapide.

3.7.9 - L'opérateur virgule

La virgule n'est pas restreinte à séparer les noms de variables dans les définitions multiples, comme dans

```
int i, j, k;
```

Bien sûr, c'est aussi utilisé dans les listes d'arguments de fonctions. Pourtant, il peut aussi être utilisé comme un opérateur pour séparer les expressions – dans ce cas cela produit seulement la valeur de la dernière expression. Toutes les autres expressions dans une liste séparée par des virgules sont évaluées seulement pour leur effet seondaire. Cet exemple incrémente une liste de variables et utilise la dernière comme la valeur droite:

```
//: C03:CommaOperator.cpp
#include <iostream>
using namespace std;
int main() {
  int a = 0, b = 1, c = 2, d = 3, e = 4;
  a = (b++, c++, d++, e++);
  cout << "a = " << a << endl;
  // Les parentheses sont obligatoires ici.
  // Sans celle ci, le traitement sera évalué par:
  (a = b++), c++, d++, e++;
  cout << "a = " << a << endl;
} ///:~
```

En général, il est préférable d'éviter d'utiliser la virgule comme autre chose qu'un séparateur, car personne n'a l'habitude de le voir comme un opérateur.

3.7.10 - Piège classique quand on utilise les opérateurs

Comme illustré précédemment, un des pièges quand on utilise les opérateurs est d'essayer de se passer de parenthèses alors que vous n'êtes pas sûr de comment une expression va être évaluée (consulter votre manuel C pour l'ordre d'évaluation des expressions).

Une autre erreur extrêmement commune ressemble à ceci:

```
//: C03:Pitfall.cpp
// Erreur d'opérateur

int main() {
  int a = 1, b = 1;
  while(a = b) {
    // ....
  }
} ///:~
```

Le traitement **a = b**sera toujours évalué à vrai quand **b**n'est pas nul. La variable **a**est assignée à la valeur de **b**, et la valeur de **b**est aussi produite par l'opérateur **=**. En général, vous voulez utiliser l'opérateur d'équivalence **==**à l'intérieur du traitement conditionnel, et non l'affectation. Cette erreur est produite par un grand nombre de programmeurs (pourtant, certains compilateurs peuvent vous montrer le problème, ce qui est utile).

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Un problème similaire est l'utilisation des opérateurs bit à bit *and*et *or*à la place des opérateurs logique associés. Les opérateurs bit à bit *and*et *or*utilise un des caractère (**&**ou **|**), alors *and*et *or*logique utilisent deux (**&&**et **||**). Tout comme **=**et **==**, il est facile de taper un caractère à la place de deux. Un moyen mnémotechnique est d'observer que les “ bits sont plus petits, donc ils n'ont pas besoin de beaucoup de caractères dans leurs opérateurs.”

3.7.11 - Opérateurs de transtypage

Le mot *transtypage*(*cast*en anglais, ndt) est utilisé dans le sens de “fondre dans un moule.” Le compilateur pourra automatiquement changer un type de donnée en un autre si cela a un sens. Par exemple, si vous affectez une valeur entière à une valeur à virgule, le compilateur fera secrètement appel a une fonction (ou plus probablement, insérera du code) pour convertir le **int**en un **float**. Transtyper vous permet de faire ce type de conversion explicitement, ou de le forcer quand cela ne se ferait pas normalement.

Pour accomplir un transtypage, mettez le type de donnée désiré (incluant tout les modifieurs) à l'intérieur de parenthèses à la gauche de la valeur. Cette valeur peut être une variable, une constante, la valeur produite par une expression, ou la valeur de retour d'une fonction. Voici un exemple :

```
                        //: C03:SimpleCast.cpp
int main() {
  int b = 200;
  unsigned long a = (unsigned long int)b;
} ///:~
```

Le transtypage est puissant, mais il peut causer des maux de tête parce que dans certaine situations il peut forcer le compilateur à traiter les données comme si elles étaient (par exemple) plus larges qu'elles ne le sont en réalité, donc cela peut occuper plus d'espace en mémoire ; et peut écraser d'autres données. Cela arrive habituellement quand un pointeur est transtypé, et non quand un simple transtypage est fait comme celui montré plus tôt.

C++ a une syntaxe de transtypage supplémentaire, qui suit la syntaxe d'appel de fonction. Cette syntaxe met des parenthèses autour de l'argument, comme un appel de fonction, plutôt qu'autour du type de la donnée :

```
                        //: C03:FunctionCallCast.cpp
int main() {
  float a = float(200);
  // Ceci est équivalent à:
  float b = (float)200;
} ///:~
```

Bien sûr dans le cas précédent vous ne pouvez pas réellement avoir besoin de transtypage; vous pouvez juste dire **200 .f** ou **200.0f** (en effet, c'est ce que le compilateur fera normalement pour l'expression précédente). Le transtypage est habituellementutilisé avec des variables, plutôt qu' avecles constantes.

3.7.12 - Transtypage C++ explicite

Le transtypage doit être utilisé avec précaution, parce que ce que vous faites est de dire au compilateur “oublie le contrôle des types – traite le comme cet autre type à la place.” C'est à dire, vous introduisez une faille dans le système de types du C++ et empechez le compilateur de vous dire que vous êtes en train de faire quelque chose de mal avec ce type. Ce qui est pire, le compilateur vous croit implicitement et ne peut exécuter aucun autre contrôle pour détecter les erreurs. Une fois que vous commencez à transtyper, vous vous ouvrez à toutes sortes de problèmes. En fait, tout programme qui utilise beaucoup de transtypages doit être abordé avec suspicion, peut importe le nombre de fois où on vous dit que ça “doit” être fait ainsi.En général, les transtypages devraient être peu nombreux et réduits au traitement de problèmes spécifiques.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Une fois que vous avez compris cela et que vous êtes en leur présence dans un programme bogué, votre premier réflexe peut être de regarder les transtypages comme pouvant être les coupables. Mais comment localiser les transtypages du style C ? Ils ont simplement un nom de type entre parenthèses, et si vous commencez à chercher ces choses vous découvrirez que c'est souvent difficile de les distinguer du reste du code.

Le standard C++ inclut une syntaxe de transtypage explicite qui peut être utilisée pour remplacer complètement l'ancien style C de transtypage (bien sûr, les transtypages de style C ne peuvent pas être declarés hors la loi sans briser la compatibilité avec du code existant, mais les compilateurs peuvent facilement vous signaler un transtypage de l'ancien style). La syntaxe de transtypage explicite est ainsi faite que vous pouvez facilement la trouver, comme vous pouvez la voir par son nom :

Le trois premiers transtypages explicites seront décrits dans la prochaine section, alors que le dernier sera expliqué seulement après que vous en ayez appris plus, dans le chapitre 15.

static_cast

Un **static_cast**est utilisé pour toutes les conversions qui sont bien définies. Ceci inclut les conversions “sûres” que le compilateur peut vous autoriser de faire sans un transtypage et les conversions moins sûres qui sont néanmoins bien définies. Les types de conversions couverts par **static_cast**incluent typiquement les conversions de type sans danger (implicites), les conversions limitantes (pertes d'information), le forçage d'une conversion d'un **void***, conversions implicite du type, et la navigation statique dans la hiérarchie des classes (comme vous n'avez pas vu les classes et l'héritage actuellement, ce dernier est repoussé au chapitre 15):

```
                        //: C03:static_cast.cpp
void func(int) {}

int main() {
  int i = 0x7fff; // Max pos value = 32767
  long l;
  float f;
  // (1) Conversion typique sans transtypage:
  l = i;
  f = i;
  // fonctionne aussi:
  l = static_cast<long>(i);
  f = static_cast<float>(i);

  // (2) conversion limitante:
  i = l; // Peut perdre des chiffres
  i = f; // Peut perdre des informations
  // Dis #Je sais,# elimine les avertissements:
  i = static_cast<int>(l);
  i = static_cast<int>(f);
  char c = static_cast<char>(i);

  // (3) Forcer une conversion depuis un void* :
  void* vp = &i;
  // Ancienne forme: produit une conversion dangereuse:
  float* fp = (float*)vp;
  // La nouvelle façon est également dangereuse:
  fp = static_cast<float*>(vp);

  // (4) Conversion de type implicite, normalement
  // exécutée par le compilateur:
  double d = 0.0;
  int x = d; // Conversion de type automatique
  x = static_cast<int>(d); // Plus explicite
  func(d); // Conversion de type automatique
  func(static_cast<int>(d)); // Plus explicite
} ///:~
```

Dans la section (1), vous pouvez voir le genre de conversion que vous utilisiez en C, avec ou sans transtypage. Promouvoir un **int**en un **long**ou **float**n'est pas un problème parce que ces derniers peuvent toujours contenir que qu'un **int**peut contenir. Bien que ce ne soit pas nécessaire, vous pouvez utiliser **static_cast**pour mettre en valeur

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

cette promotion.

La conversion inverse est montrée dans (2). Ici, vous pouvez perdre des données parce que un **int**n'est pas “large” comme un **long**ou un **float**; ce ne sont pas des nombres de même taille. Ainsi ces convertions sont appelées *conversions limitantes*. Le compilateur peut toujours l'effectuer, mais peut aussi vous retourner un avertissement. Vous pouvez éliminer le warning et indiquer que vous voulez vraiment utiliser un transtypage.

L'affectation à partir d'un **void***n'est pas permise sans un transtypage en C++ (à la différence du C), comme vu dans (3). C'est dangereux et ça requiert que le programmeur sache ce qu'il fait. Le **static_cast**, est plus facile à localiser que l'ancien standard de transtypage quand vous chassez les bugs.

La section (4) du programme montre le genre de conversions implicites qui sont normalement effectuées automatiquement par le compilateur. Celles-ci sont automatiques et ne requièrent aucun transtypage, mais à nuoveau un **static_cast**met en évidence l'action dans le cas où vous voudriez le faire apparaitre clairement ou le reperer plus tard.

const_cast

Si vous voulez convertir d'un **const**en un non **const**ou d'un **volatile**en un non **volatile**, vous utilisez **const_cast**. C'est la *seule*conversion autorisée avec **const_cast**; si une autre conversion est impliquée, il faut utiliser une expression séparée ou vous aurez une erreur de compilation.

```
                        //: C03:const_cast.cpp
int main() {
  const int i = 0;
  int* j = (int*)&i; // Obsolete
  j  = const_cast<int*>(&i); // A privilegier
  // Ne peut faire simultanément de transtypage additionnel:
//! long* l = const_cast<long*>(&i); // Erreur
  volatile int k = 0;
  int* u = const_cast<int*>(&k);
} ///:~
```

Si vous prenez l'adresse d'un objet **const**, vous produisez un pointeur sur un **const**, et il ne peut être assigné à un pointeur non **const**sans un transtypage. L'ancien style de transtypage peut l'accomplir, mais le **const_cast**est approprié pour cela. Ceci est vrai aussi pour un **volatile**.

reinterpret_cast

Ceci est le moins sûr des mécanismes de transtypage, et le plus apprecié pour faire des bugs. Un **reinterpret_cast**prétend qu'un objet est juste un ensemble de bit qui peut être traité (pour quelques obscures raisons) comme si c'était un objet d'un type entièrement différent. C'est le genre de bricolage de bas niveau qui a fait mauvaise réputation au C. Vous pouvez toujours virtuellement avoir besoin d'un **reinterpret_cast**pour retourner dans le type original de la donnée(ou autrement traiter la variable comme son type original) avant de faire quoi que ce soit avec elle.

```
                        //: C03:reinterpret_cast.cpp
#include <iostream>
using namespace std;
const int sz = 100;

struct X { int a[sz]; };

void print(X* x) {
  for(int i = 0; i < sz; i++)
    cout << x->a[i] << ' ';
  cout << endl << "--------------------" << endl;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
} 

int main() {
  X x;
  print(&x);
  int* xp = reinterpret_cast<int*>(&x);
  for(int* i = xp; i < xp + sz; i++)
    *i = 0;
  // Ne pas utiliser xp comme un X* à ce point
  // à moins de le retranstyper dans son état d'origine:
  print(reinterpret_cast<X*>(xp));
  // Dans cette exemple, vous pouvez aussi juste utiliser
  // l'identifiant original:
  print(&x);
} ///:~
```

Dans cet exemple simple, **struct X**contiens seulement un tableau de **int**, mais quand vous en créez un sur la pile comme dans **X x**, la valeur de chacun des **int**s est n'importe quoi (ceci est montré en utilisant la fonction **print( )**pour afficher le contenu de la **struct**). Pour les initialiser , l'adresse de **X**est prise et transtypée en un pointeur de type **int**, le tableau est alors parcouru pour mettre chaque **int**à zéro. Notez comment la limite haute pour **i**est calculée par “l'addition” de **sz**avec **xp**; le compilateur sait que vous voulez actuellement **sz**positions au départ de **xp**et utilise l'arithmétique de pointeur pour vous.

L'idée du **reinterpret_cast**est que quand vous l'utilisez, ce que vous obtenez est à ce point différent du type original que vous ne pouvez l'utiliser comme tel à moins de le transtyper à nouveau. Ici, nous voyons le transtypage précédent pour un **X***dans l'appel de print, mais bien sûr dès le début vous avez l'identifiant original que vous pouvez toujours utiliser comme tel. Mais le **xp**est seulement utile comme un **int***, qui est vraiment une “réinterpretation” du **X**original.

Un **reinterpret_cast**peut aussi indiquer une imprudence et/ou un programme non portable, mais est disponible quand vous décidez que vous devez l'utiliser.

3.7.13 - sizeof – Un opérateur par lui même

L'opérateur **sizeof**reste seul parce qu'il satisfait un besoin non usuel. **sizeof**vous donne des informations à propos de la quantité de mémoire allouée pour une donnée. Comme décrit plus tôt dans ce chapitre, **sizeof**vous dit le nombre d'octets utilisés par n'importe quelle variable. Il peut aussi donner la taille du type de la donnée (sans nom de variable):

```
                        //: C03:sizeof.cpp
#include <iostream>
using namespace std;
int main() {
  cout << "sizeof(double) = " << sizeof(double);
  cout << ", sizeof(char) = " << sizeof(char);
} ///:~
```

Avec la définition de **sizeof**tout type de **char**(**signed**, **unsigned**ou simple) est toujours un, sans se soucier du fait que le stockage sous-jacent pour un **char**est actuellement un octet. Pour tous les autres types, le résultat est la taille en octets.

Notez que **sizeof**est un opérateur, et non une fonction. Si vous l'appliquez à un type, il doit être utilisé avec les parenthèses comme vu précédemment, mais si vous l'appliquez à une variable vous pouvez l'utiliser sans les parenthèses:

```
                        //: C03:sizeofOperator.cpp
int main() {
  int x;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  int i = sizeof x;
} ///:~
```

sizeofpeut aussi vous donnez la taille des données d'un type défini par l'utilisateur. C'est utilisé plus tard dans le livre.

3.7.14 - Le mot clef asm

Ceci est un mécanisme d'échappement qui vous permet d'écrire du code assembleur pour votre matériel dans un programme C++. Souvent vous êtes capable de faire référence à des variables C++ dans le code assembleur, ceci signifie que vous pouvez facilement communiquer avec votre code C++ et limiter les instructions en code assembleur pour optimiser des performances ou pour faire appel à des instructions microprocesseur précises. La syntaxe exacte que vous devez utiliser quand vous écrivez en langage assembleur est dépendante du compilateur et peut être découverte dans la documentation de votre compilateur.

3.7.15 - Opérateurs explicites

Ces mots clefs sont pour les opérateurs de bit et les opérateurs logiques. Les programmeurs non américains sans les caractères du clavier comme **&**, **|**, **^**, et ainsi de suite, sont forcés d'utiliser les horribles *trigraphes*C, ce qui n'est pas seulement pénible, mais obscur à lire. Cela a été arrangé en C++ avec l'ajout des mots clefs :

Si votre compilateur se conforme au standard C++, il supportera ces mots clefs.

3.8 - Création de type composite

Les types de données fondamentaux et leurs variantes sont essentiels, bien que primitifs. C et C++ fournissent des outils qui vous autorisent à composer des types de données plus sophistiqués à partir des types fondamentaux. Comme vous le verrez, le plus important de ces types est **struct**, qui est le fondement des **class**es du C++. Cependant, la façon la plus simple de créer des types plus sophistiqués est de simplement créer un alias d'un nom vers un autre nom grâce à **typedef**.

3.8.1 - Alias de noms avec typedef

Ce mot clé promet plus qu'il n'agit : **typedef**suggère “une définition de type” alors qu'“alias” serait probablement une description plus juste, puisque c'est ce qu'il fait réellement. Sa syntaxe est :

typedef description-type-existant nom-alias;

Le **typedef**est fréquemment utilisé quand les noms des types de données deviennent quelque peu compliqués, simplement pour économiser quelques frappes. Voici un exemple d'utilisation commune du **typedef**:

```
typedef unsigned long ulong;
```

Maintenant, si vous dites **ulong**le compilateur sait que vous voulez dire **unsigned long**. Vous pensez peut-être que cela pourrait être si facilement résolu avec une substitution pré processeur, mais il existe des situations pour lesquelles le compilateur doit savoir que vous traitez un nom comme s'il était un type, donc **typedef**est essentiel.

Un endroit pour lequel **typedef**est pratique est pour les types pointeurs. Comme mentionné précédemment, si vous dites :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
int* x, y;
```

Le code va en fait créer un **int***qui est **x**et un **int**(pas un **int***) qui est **y**. Cela vient du fait que le '*' s'associe par la droite, et non par la gauche. Cependant si vous utilisez un ' **typedef**' :

```
typedef int* IntPtr;
IntPtr x, y;
```

Alors **x**et **y**sont tous les deux du type **int***.

Vous pourriez argumenter qu'il est plus explicite et donc plus lisible d'éviter les **typedef**sur les types primitifs, et que les programmes deviendraient rapidement difficiles à lire quand beaucoup de **typedef**sont utilisés. Cependant, les **typedef**deviennent particulièrement importants en C quand ils sont utilisés avec des **struct**ures.

3.8.2 - Combiner des variables avec des struct

Une **struct**est une manière de rassembler un groupe de variables dans une structure. Une fois que vous créez une **struct**, vous pouvez alors créer plusieurs instances de ce "nouveau" type de variable que vous venez d'inventer. Par exemple :

```
//: C03:SimpleStruct.cpp
struct Structure1 {
  char c;
  int i;
  float f;
  double d;
};

int main() {
  struct Structure1 s1, s2;
  s1.c = 'a'; // Sélectionnez un élément en utilisant un '.'
  s1.i = 1;
  s1.f = 3.14;
  s1.d = 0.00093;
  s2.c = 'a';
  s2.i = 1;
  s2.f = 3.14;
  s2.d = 0.00093;
} ///:~
```

La déclaration d'une **struct**doit être terminée par un point-virgule. Dans notre **main( )**, deux instances de **Structure1**sont créées : **s1**et **s2**. Chacune de ces instances dispose de ses propres versions distinctes de **c**, **i**, **f**et **d**. ainsi **s1**et **s2**représentent des blocs de variables totalement indépendants. Pour sélectionner un des éléments encapsulé dans **s1**ou **s2**, vous utilisez un ' **.**', syntaxe que vous avez rencontré dans le précédent chapitre en utilisant des objets de **class**es C++ – comme les **class**es sont des **struct**ures évoluées, voici donc d'où vient la syntaxe.

Une chose que vous noterez est la maladresse d'utilisation de **Structure1**(comme cela ressort, c'est requis en C uniquement, pas en C++). En C, vous ne pouvez pas juste dire **Structure1**quand vous définissez des variables, vous devez dire **struct Structure1**. C'est ici que le **typedef**devient particulièrement pratique en C :

```
//: C03:SimpleStruct2.cpp
// Utilisation de typedef avec des struct
typedef struct {
  char c;
  int i;
  float f;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  double d;
} Structure2;

int main() {
  Structure2 s1, s2;
  s1.c = 'a';
  s1.i = 1;
  s1.f = 3.14;
  s1.d = 0.00093;
  s2.c = 'a';
  s2.i = 1;
  s2.f = 3.14;
  s2.d = 0.00093;
} ///:~
```

En utilisant **typedef**de cette façon, vous pouvez prétendre (en C tout du moins ; essayez de retirer le **typedef**pour C++) que **Structure2**est un type natif, au même titre que **int**ou **float**, quand vous définissez **s1**et **s2**(mais notez qu'il a uniquement des –caractéristiques– de données mais sans inclure de comportement particulier, comportements que l'on peut définir avec de vrais objets en C++). Vous noterez que l'identifiant **struct**a été abandonné au début, parce que le but était de créer un type via le **typedef**. Cependant, il y a des fois où vous pourriez avoir besoin de vous référer au **struct**pendant sa définition. Dans ces cas, vous pouvez en fait répéter le **struct**dans le nom de la **struct**ure à travers le **typedef**:

```
                        //: C03:SelfReferential.cpp
// Autoriser une struct à faire référence à elle-même

typedef struct SelfReferential {
  int i;
  SelfReferential* sr; // Déjà mal à la tête ?
} SelfReferential;

int main() {
  SelfReferential sr1, sr2;
  sr1.sr = &sr2;
  sr2.sr = &sr1;
  sr1.i = 47;
  sr2.i = 1024;
} ///:~
```

Si vous regardez ceci de plus près, vous remarquerez que **sr1**et **sr2**pointent tous les deux l'un vers l'autre, comme s'ils contenaient une donnée quelconque.

En fait, le nom de la **struct**n'est pas obligatoirement le même que celui du **typedef**, mais on procède généralement de cette façon pour préserver la simplicité du procédé.

Pointeurs et structs

Dans les exemples ci-dessus, toutes les **struct**ures sont manipulées comme des objets. Cependant, comme tout élément de stockage, vous pouvez prendre l'adresse d'une **struct**(comme montré dans l'exemple **SelfReferential.cpp**ci-dessous). Pour sélectionner les éléments d'un objet **struct**particulier, On utilise le ‘ **.**’, comme déjà vu plus haut. Cela dit, si vous disposez d'un pointeur sur une **struct**, vous devez sélectionner ses éléments en utilisant un opérateur différent : le ‘ **->**’. Voici un exemple :

```
                        //: C03:SimpleStruct3.cpp
// Utilisation de pointeurs de struct
typedef struct Structure3 {
  char c;
  int i;
  float f;
  double d;
} Structure3;

int main() {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  Structure3 s1, s2;
  Structure3* sp = &s1;
  sp->c = 'a';
  sp->i = 1;
  sp->f = 3.14;
  sp->d = 0.00093;
  sp = &s2; // Pointe vers une struct différent
  sp->c = 'a';
  sp->i = 1;
  sp->f = 3.14;
  sp->d = 0.00093;
} ///:~
```

Dans **main( )**, le pointeur de **struct sp**pointe initialement sur **s1**, et les membres de **s1**sont initialisés en les sélectionnant grâce au ‘ **->**’ (et vous utilisez ce même opérateur pour lire ces membres). Mais par la suite **sp**pointe vers **s2**, et ses variables sont initialisées de la même façon. Aussi vous pouvez voir qu'un autre avantage des pointeurs est qu'ils peuvent être redirigés dynamiquement pour pointer vers différents objets ; ceci apporte d'avantage de flexibilité à votre programmation, comme vous allez le découvrir.

Pour l'instant, c'est tout ce que vous avez besoin de savoir à propos des **struct**, mais vous allez devenir très familiers (et particulièrement avec leurs plus puissants successeurs, les **class**es) au fur et à mesure de la lecture de ce livre.

3.8.3 - Eclaircir les programmes avec des enum

Un type de données énuméré est une manière d'attacher un nom a des nombres, et ainsi d'y donner plus de sens pour quiconque qui lit le code. Le mot-clé **enum**(du C) énumère automatiquement une liste d'identifiants que vous lui donnez en affectant des valeurs 0, 1, 2, etc. On peut déclarer des variables **enum**(qui sont toujours représentées comme des valeurs intégrales). La déclaration d'une **enum**ération est très proche à celle d'une **struct**ure.

Un type de données énuméré est utile quand on veut garder une trace de certaines fonctionnalités :

```
                          //: C03:Enum.cpp
// Suivi des formes

enum ShapeType {
  circle,
  square,
  rectangle
};  // Se termine par un point-virgule

int main() {
  ShapeType shape = circle;
  // Activités ici....
  // Faire quelque chose en fonction de la forme :
  switch(shape) {
    case circle:  /* c'est un cercle */ break;
    case square:  /* C'est un carré */ break;
    case rectangle:  /* Et voici le rectangle */ break;
  }
} ///:~
```

shapeest une variable du type de données énuméré **ShapeType**, et sa valeur est comparée à la valeur de l'énumération. Puisque **shape**est réellement juste un **int**, il est possible de lui affecter n'importe quelle valeur qu'un **int**peut prendre (y compris les valeurs négatives). Vous pouvez aussi comparer une variable de type **int**à une valeur de l'énumération.

Vous devez savoir que l'exemple de renommage ci-dessus se révèle une manière de programmer problématique. Le C++ propose une façon bien meilleure pour faire ce genre de choses, l'explication de ceci sera donnée plus loin dans le livre.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Si vous n'aimez pas la façon dont le compilateur affecte les valeurs, vous pouvez le faire vous-même, comme ceci :

```
enum ShapeType {
  circle = 10, square = 20, rectangle = 50
};
```

Si vous donnez des valeurs à certains noms mais pas à tous, le compilateur utilisera la valeur entière suivante. Par exemple,

```
enum snap { crackle = 25, pop };
```

Le compilateur donne la valeur 26 à **pop**.

Vous pouvez voir alors combien vous gagnez en lisibilité du code en utilisant des types de données énumérés. Cependant, d'une certaine façon, cela reste une tentative (en C) d'accomplir des choses que l'on peut faire avec une **class**en C++, c'est ainsi que vous verrez que les **enum**sont moins utilisées en C++.

Vérification de type pour les énumérations

Les énumérations du C sont très primitives, en associant simplement des valeurs intégrales à des noms, mais elles ne fournissent aucune vérification de type. En C++, comme vous pouvez vous y attendre désormais, le concept de type est fondamental, et c'est aussi vrai avec les énumérations. Quand vous créez une énumération nommée, vous créez effectivement un nouveau type tout comme vous le faites avec une classe : le nom de votre énumération devient un mot réservé pour la durée de l'unité de traduction.

De plus, la vérification de type est plus stricte pour les énumérations en C++ qu'en C. Vous noterez cela, en particulier, dans le cas d'une énumération **color**appelée **a**. En C, vous pouvez écrire **a++**, chose que vous ne pouvez pas faire en C++. Ceci parce que l?incrémentation d?une énumération effectue en réalité deux conversions de type, l'une d'elle légale en C++, mais l'autre illégale. D'abord, la valeur de l'énumération est implicitement convertie de **color**vers un **int**, puis la valeur est incrémentée, et reconvertie en **color**. En C++, ce n'est pas autorisé, parce que **color**est un type distinct et n'est pas équivalent à un **int**. Cela a du sens, parce que comment saurait-on si le résultat de l'incrémentation de **blue**sera dans la liste de couleurs? Si vous souhaitez incrémenter un **color**, alors vous devez utiliser une classe (avec une opération d'incrémentation) et non pas une **enum**, parce que la classe peut être rendue plus sûre. Chaque fois que vous écrirez du code qui nécessite une conversion implicite vers un type **enum**, Le compilateur vous avertira du danger inhérent à cette opération.

Les unions (décrites dans la prochaine section) possèdent la même vérification additionnelle de type en C++.

3.8.4 - Economiser de la mémoire avec union

Parfois, un programme va manipuler différents types de donnée en utilisant la même variable. Dans de tels cas, deux possibilités: soit vous créez une **struct**qui contient tous les types possibles que vous auriez besoin d'enregistrer, soit vous utilisez une **union**. Une **union**empile toutes les données dans un même espace; cela signifie que la quantité de mémoire nécessaire sera celle de l'élément le plus grand que vous avez placé dans l' **union**. Utilisez une **union**pour économiser de la mémoire.

Chaque fois que vous écrivez une valeur dans une **union**, cette valeur commence toujours à l'adresse de début de l' **union**, mais utilise seulement la mémoire nécessaire. Ainsi, vous créez une “super-variable” capable d'utiliser chacune des variables de l' **union**. Toutes les adresses des variables de l' **union**sont les mêmes (alors que dans une classe ou une **struct**, les adresses diffèrent).

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Voici un simple usage d'une **union**. Essayez de supprimer quelques éléments pour voir quel effet cela a sur la taille de l' **union**. Notez que cela n'a pas de sens de déclarer plus d'une instance d'un simple type de données dans une **union**(à moins que vous ne fassiez que pour utiliser des noms différents).

```
                          //: C03:Union.cpp
// Simple usage d'une union
#include <iostream>
using namespace std;

union Packed { // Déclaration similaire à une classe
  char i;
  short j;
  int k;
  long l;
  float f;
  double d;
  // L'union sera de la taille d'un
  // double, puisque c'est l?élément le plus grand
};  // Un point-virgule termine une union, comme une struct

int main() {
  cout << "sizeof(Packed) = "
       << sizeof(Packed) << endl;
  Packed x;
  x.i = 'c';
  cout << x.i << endl;
  x.d = 3.14159;
  cout << x.d << endl;
} ///:~
```

Le compilateur effectuera l'assignation correctement selon le type du membre de l'union que vous sélectionnez.

Une fois que vous avez effectué une affectation, le compilateur se moque de ce que vous ferez par la suite de l'union. Dans l'exemple précédent, on aurait pu assigner une valeur flottante à **x**:

```
                          x.f = 2.222;
```

Et l'envoyer sur la sortie comme si c'était un **int**:

```
                          cout << x.i;
```

Ceci aurait produit une valeur sans aucun sens.

3.8.5 - Tableaux

Les tableaux sont une espèce de type composite car ils vous autorisent à agréger plusieurs variables ensemble, les unes à la suite des autres, sous le même nom. Si vous dites :

```
                          int a[10];
```

Vous créez un emplacement mémoire pour 10 variables **int**empilées les unes sur les autres, mais sans un nom unique pour chacune de ces variables. A la place, elles sont toutes réunies sous le nom **a**.

Pour accéder à l'un des *éléments*du tableau, on utilise la même syntaxe utilisant les crochets que celle utilisée pour définir un tableau :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
a[5] = 47;
```

Cependant, vous devez retenir que bien que la *taille*de **a**soit 10, on sélectionne les éléments d'un tableau en commençant à zero (ceci est parfois appelé *indexation basée sur zero*), donc vous ne pouvez sélectionner que les éléments 0-9 du tableau, comme ceci :

```
//: C03:Arrays.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  for(int i = 0; i < 10; i++) {
    a[i] = i * 10;
    cout << "a[" << i << "] = " << a[i] << endl;
  }
} ///:~
```

L'accès aux tableaux est extrêmement rapide. Cependant, si votre index dépasse la taille du tableau, il n'y a aucun filet de sécurité – vous pointerez sur d'autres variables. L'autre inconvénient est que vous devez spécifier la taille du tableau au moment de la compilation ; si vous désirez changer la taille lors de l'exécution, vous ne pouvez pas le faire avec la syntaxe précédente (le C propose une façon de créer des tableaux dynamiquement, mais c'est assurément plus sale). Le type **vector**fournit par C++, présenté au chapitre précédent, nous apporte un type semblable à un tableau qui redéfinit sa taille automatiquement, donc c'est généralement une meilleure solution si la taille de votre tableau ne peut pas être connue lors de la compilation.

Vous pouvez créer un tableau de n'importe quel type, y compris de **struct**ures :

```
//: C03:StructArray.cpp
// Un tableau de struct

typedef struct {
  int i, j, k;
} ThreeDpoint;

int main() {
  ThreeDpoint p[10];
  for(int i = 0; i < 10; i++) {
    p[i].i = i + 1;
    p[i].j = i + 2;
    p[i].k = i + 3;
  }
} ///:~
```

Remarquez comment l'identifiant **i**de la **struct**ure est indépendant de celui de la boucle **for**.

Pour vérifier que tous les éléments d'un tableau se suivent, on peut afficher les adresses comme ceci :

```
//: C03:ArrayAddresses.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  cout << "sizeof(int) = "<< sizeof(int) << endl;
  for(int i = 0; i < 10; i++)
    cout << "&a[" << i << "] = "
         << (long)&a[i] << endl;
} ///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Quand vous exécutez ce programme, vous verrez que chaque élément est éloigné de son précédent de la taille d'un **int**. Ils sont donc bien empilés les uns sur les autres.

Pointeurs et tableaux

L'identifiant d'un tableau n'est pas comme celui d'une variable ordinaire. Le nom d'un tableau n'est pas une lvalue ; vous ne pouvez pas lui affecter de valeur. C'est seulement un point d'ancrage pour la syntaxe utilisant les crochets '[]', et quand vous utilisez le nom d'un tableau, sans les crochets, vous obtenez l'adresse du début du tableau:

```
                        //: C03:ArrayIdentifier.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  cout << "a = " << a << endl;
  cout << "&a[0] =" << &a[0] << endl;
} ///:~
```

En exécutant ce programme, vous constaterez que les deux adresses (affichées en hexadécimal, puisque aucun cast en **long**n'est fait) sont identiques.

Nous pouvons considérer que le nom d?un tableau est un pointeur en lecture seule sur le début du tableau. Et bien que nous ne puissions pas changer le nom du tableau pour qu'il pointe ailleurs, nous pouvons, en revanche, créer un autre pointeur et l'utiliser pour se déplacer dans le tableau. En fait, la syntaxe avec les crochets marche aussi avec les pointeurs normaux également :

```
                        //: C03:PointersAndBrackets.cpp
int main() {
  int a[10];
  int* ip = a;
  for(int i = 0; i < 10; i++)
    ip[i] = i * 10;
} ///:~
```

Le fait que nommer un tableau produise en fait l'adresse de départ du tableau est un point assez important quand on s'intéresse au passage des tableaux en paramètres de fonctions. Si vous déclarez un tableau comme un argument de fonction, vous déclarez en fait un pointeur. Dans l'exemple suivant, **func1( )**et **func2( )**ont au final la même liste de paramètres :

```
                        //: C03:ArrayArguments.cpp
#include <iostream>
#include <string>
using namespace std;

void func1(int a[], int size) {
  for(int i = 0; i < size; i++)
    a[i] = i * i - i;
}

void func2(int* a, int size) {
  for(int i = 0; i < size; i++)
    a[i] = i * i + i;
}

void print(int a[], string name, int size) {
  for(int i = 0; i < size; i++)
    cout << name << "[" << i << "] = "
         << a[i] << endl;
}

int main() {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    int a[5], b[5];
    // Probablement des valeurs sans signification:
    print(a, "a", 5);
    print(b, "b", 5);
    // Initialisation des tableaux:
    func1(a, 5);
    func1(b, 5);
    print(a, "a", 5);
    print(b, "b", 5);
    // Les tableaux sont toujours modifiés :
    func2(a, 5);
    func2(b, 5);
    print(a, "a", 5);
    print(b, "b", 5);
} ///:~
```

Même si **func1( )**et **func2( )**déclarent leurs paramètres différemment, leur utilisation à l'intérieur de la fonction sera la même. Il existe quelques autres problèmes que l'exemple suivant nous révèle : les tableaux ne peuvent pas être passés par valeur A moins que vous ne considériez l'approche stricte selon laquelle “ tous les paramètres en C/C++ sont passés par valeur, et que la ‘valeur’ d'un tableau est ce qui est effectivement dans l'identifiant du tableau : son adresse.” Ceci peut être considéré comme vrai d'un point de vue du langage assembleur, mais je ne pense pas que cela aide vraiment quand on travaille avec des concepts de plus haut niveau. L'ajout des références en C++ ne fait qu'accentuer d'avantage la confusion du paradigme “tous les passages sont par valeur”, au point que je ressente plus le besoin de penser en terme de “passage par valeur” opposé à “passage par adresse”, car vous ne récupérez jamais de copie locale du tableau que vous passez à une fonction. Ainsi, quand vous modifiez un tableau, vous modifiez toujours l'objet extérieur. Cela peut dérouter au début, si vous espériez un comportement de passage par valeur tel que fourni avec les arguments ordinaires.

Remarquez que **print( )**utilise la syntaxe avec les crochets pour les paramètres tableaux. Même si les syntaxes de pointeurs et avec les crochets sont effectivement identiques quand il s'agit de passer des tableaux en paramètres, les crochets facilitent la lisibilité pour le lecteur en lui explicitant que le paramètre utilisé est bien un tableau.

Notez également que la **taille**du tableau est passée en paramètre dans tous les cas. Passer simplement l'adresse d'un tableau n'est pas une information suffisante; vous devez toujours savoir connaître la taille du tableau à l'intérieur de votre fonction, pour ne pas dépasser ses limites.

Les tableaux peuvent être de n'importe quel type, y compris des tableaux de pointeurs. En fait, lorsque vous désirez passer à votre programme des paramètres en ligne de commande, le C et le C++ ont une liste d'arguments spéciale pour **main( )**, qui ressemble à ceci :

```
int main(int argc, char* argv[]) { // ...
```

Le premier paramètre est le nombre d'éléments du tableau, lequel tableau est le deuxième paramètre. Le second paramètre est toujours un tableau de **char***, car les arguments sont passés depuis la ligne de commande comme des tableaux de caractères (et souvenez vous, un tableau ne peut être passé qu'en tant que pointeur). Chaque portion de caractères délimitée par des espaces est placée dans une chaîne de caractères séparée dans le tableau. Le programme suivant affiche tous ses paramètres reçus en ligne de commande en parcourant le tableau :

```
//: C03:CommandLineArgs.cpp
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
  cout << "argc = " << argc << endl;
  for(int i = 0; i < argc; i++)
    cout << "argv[" << i << "] = "
         << argv[i] << endl;
} ///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Notez que **argv[0]**est en fait le chemin et le nom de l'application elle-même. Cela permet au programme de récupérer des informations sur lui. Puisque que cela rajoute un élément de plus au tableau des paramètres du programme, une erreur souvent rencontrée lors du parcours du tableau est d'utiliser **argv[0]**alors qu'on veut en fait **argv[1]**.

Vous n'êtes pas obligés d'utiliser **argc**et **argv**comme identifiants dans **main( )**; ils sont utilisés par pure convention (mais ils risqueraient de perturber un autre lecteur si vous ne les utilisiez pas). Aussi, il existe une manière alternative de déclarer **argv**:

```
int main(int argc, char** argv) { // ...
```

Les deux formes sont équivalentes, mais je trouve la version utilisée dans ce livre la plus intuitive pour relire le code, puisqu'elle dit directement “Ceci est un tableau de pointeurs de caractères”.

Tout ce que vous récupérez de la ligne de commande n'est que tableaux de caractères; si vous voulez traiter un argument comment étant d'un autre type, vous avez la responsabilité de le convertir depuis votre programme. Pour faciliter la conversion en nombres, il existe des fonctions utilitaires de la librairie C standard, déclarées dans **<cstdlib>**. Les plus simples à utiliser sont **atoi( )**, **atol( ),**et **atof( )**pour convertir un tableau de caractères ASCII en valeurs **int**, **long,**et **double**, respectivement. Voici un exemple d'utilisation de **atoi( )**(les deux autres fonctions sont appelées de la même manière) :

```
//: C03:ArgsToInts.cpp
// Convertir les paramètres de la ligne de commande en int
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char* argv[]) {
  for(int i = 1; i < argc; i++)
    cout << atoi(argv[i]) << endl;
} ///:~
```

Dans ce programme, vous pouvez saisir n'importe quel nombre de paramètres en ligne de commande. Vous noterez que la boucle **for**commence à la valeur **1**pour ignorer le nom du programme en **argv[0]**. Mais, si vous saisissez un nombre flottant contenant le point des décimales sur la ligne de commande, **atoi( )**ne prendra que les chiffres jusqu'au point. Si vous saisissez des caractères non numériques, **atoi( )**les retournera comme des zéros.

Exploration du format flottant

La fonction **printBinary( )**présentée précédemment dans ce chapitre est pratique pour scruter la structure interne de types de données divers. Le plus intéressant de ceux-ci est le format flottant qui permet au C et au C++ d'enregistrer des nombres très grands et très petits dans un espace mémoire limité. Bien que tous les détails ne puissent être exposés ici, les bits à l'intérieur d'un **float**et d'un **double**sont divisées en trois régions : l'exposant, la mantisse et le bit de signe; le nombre est stocké en utilisant la notation scientifique. Le programme suivant permet de jouer avec les modèles binaires de plusieurs flottants et de les imprimer à l'écran pour que vous vous rendiez compte par vous même du schéma utilisé par votre compilateur (généralement c'est le standard IEEE, mais votre compilateur peut ne pas respecter cela) :

```
//: C03:FloatingAsBinary.cpp
//{L} printBinary
//{T} 3.14159
#include "printBinary.h"
#include <cstdlib>
#include <iostream>
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
using namespace std;

int main(int argc, char* argv[]) {
  if(argc != 2) {
    cout << "Vous devez fournir un nombre" << endl;
    exit(1);
  }
  double d = atof(argv[1]);
  unsigned char* cp =
    reinterpret_cast<unsigned char*>(&d);
  for(int i = sizeof(double)-1; i >= 0 ; i -= 2) {
    printBinary(cp[i-1]);
    printBinary(cp[i]);
  }
} ///:~
```

Tout d'abord, le programme garantit que le bon nombre de paramètres est fourni en vérifiant **argc**, qui vaut deux si un seul argument est fourni (il vaut un si aucun argument n'est fourni, puisque le nom du programme est toujours le premier élément de **argv**). Si ce test échoue, un message est affiché et la fonction de la bibliothèque standard du C **exit( )**est appelée pour terminer le programme.

Puis le programme récupère le paramètre de la ligne de commande et convertit les caractères en **double**grâce à **atof( )**. Ensuite le double est utilisé comme un tableau d'octets en prenant l'adresse et en la convertissant en **unsigned char***. Chacun de ces octets est passé à **printBinary( )**pour affichage.

Cet exemple a été réalisé de façon à ce que que le bit de signe apparaisse d'abord sur ma machine. La vôtre peut être différente, donc vous pourriez avoir envie de réorganiser la manière dont sont affichées les données. Vous devriez savoir également que le format des nombres flottants n'est pas simple à comprendre ; par exemple, l'exposant et la mantisse ne sont généralement pas arrangés sur l'alignement des octets, mais au contraire un nombre de bits est réservé pour chacun d'eux, et ils sont empaquetés en mémoire de la façon la plus serrée possible. Pour vraiment voir ce qui se passe, vous devrez trouver la taille de chacune des parties (le bit de signe est toujours un seul bit, mais l'exposant et la mantisse ont des tailles différentes) et afficher les bits de chaque partie séparément.

Arithmétique des pointeurs

Si tout ce que l'on pouvait faire avec un pointeur qui pointe sur un tableau était de l'utiliser comme un alias pour le nom du tableau, les pointeurs ne seraient pas très intéressants. Cependant, ce sont des outils plus flexibles que cela, puisqu'ils peuvent être modifiés pour pointer n'importe où ailleurs (mais rappelez vous que l'identifiant d'un tableau ne peut jamais être modifié pour pointer ailleurs).

*Arithmétique des pointeurs*fait référence à l'application de quelques opérateurs arithmétiques aux pointeurs. La raison pour laquelle l'arithmétique des pointeurs est un sujet séparé de l'arithmétique ordinaire est que les pointeurs doivent se conformer à des contraintes spéciales pour qu'ils se comportent correctement. Par exemple, un opérateur communément utilisé avec des pointeurs est le **++**, qui “ajoute un au pointeur”. Cela veut dire en fait que le pointeur est changé pour se déplacer à “la valeur suivante”, quoi que cela signifie. Voici un exemple :

```
                          //: C03:PointerIncrement.cpp
#include <iostream>
using namespace std;

int main() {
  int i[10];
  double d[10];
  int* ip = i;
  double* dp = d;
  cout << "ip = " << (long)ip << endl;
  ip++;
  cout << "ip = " << (long)ip << endl;
  cout << "dp = " << (long)dp << endl;
  dp++;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  cout << "dp = " << (long)dp << endl;
} ///:~
```

Pour une exécution sur ma machine, voici le résultat obtenu :

```
                        ip = 6684124
ip = 6684128
dp = 6684044
dp = 6684052
```

Ce qui est intéressant ici est que bien que l'opérateur **++**paraisse être la même opération à la fois pour un **int***et un **double***, vous remarquerez que le pointeur a avancé de seulement 4 octets pour l' **int***mais de 8 octets pour le **double***. Ce n'est pas par coïncidence si ce sont les tailles de ces types sur ma machine. Et tout est là dans l'arithmétique des pointeurs : le compilateur détermine le bon déplacement à appliquer au pointeur pour qu'il pointe sur l'élément suivant dans le tableau (l'arithmétique des pointeurs n'a de sens qu'avec des tableaux). Cela fonctionne même avec des tableaux de **struct**s:

```
                        //: C03:PointerIncrement2.cpp
#include <iostream>
using namespace std;

typedef struct {
  char c;
  short s;
  int i;
  long l;
  float f;
  double d;
  long double ld;
} Primitives;

int main() {
  Primitives p[10];
  Primitives* pp = p;
  cout << "sizeof(Primitives) = "
       << sizeof(Primitives) << endl;
  cout << "pp = " << (long)pp << endl;
  pp++;
  cout << "pp = " << (long)pp << endl;
} ///:~
```

L'affichage sur ma machine a donné :

```
                        sizeof(Primitives) = 40
pp = 6683764
pp = 6683804
```

Vous voyez ainsi que le compilateur fait aussi les choses correctement en ce qui concerne les pointeurs de **struct**ures (et de **class**es et d' **union**s).

L'arithmétique des pointeurs marche également avec les opérateurs **--**, **+,**et **-**, mais les deux derniers opérateurs sont limités : Vous ne pouvez pas additionner deux pointeurs, et si vous retranchez des pointeurs, le résultat est le nombre d'élément entre les deux pointeurs. Cependant, vous pouvez ajouter ou soustraire une valeur entière et un pointeur. Voici un exemple qui démontre l'utilisation d'une telle arithmétique :

```
                        //: C03:PointerArithmetic.cpp
#include <iostream>
using namespace std;

#define P(EX) cout << #EX << ": " << EX << endl;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
int main() {
  int a[10];
  for(int i = 0; i < 10; i++)
    a[i] = i; // Attribue les valeurs de l?index
  int* ip = a;
  P(*ip);
  P(*++ip);
  P(*(ip + 5));
  int* ip2 = ip + 5;
  P(*ip2);
  P(*(ip2 - 4));
  P(*--ip2);
  P(ip2 - ip); // Renvoie le nombre d?éléments
} ///:~
```

Il commence avec une nouvelle macro, mais celle-ci utilise une fonctionnalité du pré processeur appelée *stringizing*(transformation en chaîne de caractères - implémentée grâce au symbole ‘ #’ devant une expression) qui prend n'importe quelle expression et la transforme en tableau de caractères. C'est très pratique puisqu'elle permet d'imprimer une expression, suivie de deux-points, suivie par la valeur de l'expression. Dans **main( )**vous pouvez voir l'avantage que cela produit.

De même, les version pré et post fixées des opérateurs **++**et **--**sont valides avec les pointeurs, même si seule la version préfixée est utilisée dans cet exemple parce qu'elle est appliquée avant que le pointeur ne soit déréférencé dans les expressions ci-dessus, on peut donc voir les effets des opérations. Notez que seules les valeurs entières sont ajoutées et retranchées ; si deux pointeurs étaient combinés de cette façon, le compilateur ne l'aurait pas accepté.

Voici la sortie du programme précédent :

```
                        *ip: 0
*++ip: 1
*(ip + 5): 6
*ip2: 6
*(ip2 - 4): 2
*--ip2: 5
```

Dans tous les cas, l'arithmétique des pointeurs résulte en un pointeur ajusté pour pointer au “bon endroit”, basé sur la taille des éléments pointés.

Si l'arithmétique des pointeurs peut paraître accablante de prime abord, pas de panique. La plupart du temps, vous aurez simplement besoin de créer des tableaux et des index avec **[ ]**, et l'arithmétique la plus sophistiquée dont vous aurez généralement besoin est **++**et **--**. L'arithmétique des pointeurs est en général réservée aux programmes plus complexes, et la plupart des conteneurs standards du C++ cachent tous ces détails intelligents pour que vous n'ayez pas à vous en préoccuper.

3.9 - Conseils de déboguage

Dans un environnement idéal, vous disposez d'un excellent débogueur qui rend aisément le comportement de votre programme transparent et qui vous permet de découvrir les erreurs rapidement. Cependant, la plupart des débogueurs ont des "angles morts" qui vont vous forcer à insérer des fragments de code dans votre programme afin de vous aider à comprendre ce qu'il s'y passe. De plus, vous pouvez être en train de développer dans un environnement (par exemple un système embarqué, comme moi durant mes années de formation) qui ne dispose pas d'un débogueur, et fournit même peut-être des indications très limitées (comme une ligne d'affichage à DEL). Dans ces cas, vous devenez créatif dans votre manière de découvrir et d'afficher les informations à propos de l'exécution de votre code. Cette section suggère quelques techniques de cet ordre.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

3.9.1 - Drapeaux de déboguage

Si vous branchez du code de déboguage en dur dans un programme, vous risquez de rencontrer certains problèmes. Vous commencez à être inondé d'informations, ce qui rend le bogue difficile à isoler. Lorsque vous pensez avoir trouvé le bogue, vous commencez à retirer le code, juste pour vous apercevoir que vous devez le remettre. Vous pouvez éviter ces problèmes avec deux types de drapeaux de déboguage : les drapeaux de précompilation et ceux d'exécution.

Drapeaux de déboguage de précompilation

En utilisant le préprocesseur pour définir (instruction **#define**) un ou plusieurs drapeaux (de préférence dans un fichier d'en-tête), vous pouvez tester le drapeau (à l'aide de l'instruction **#ifdef**) et inclure conditionnellement du code de déboguage. Lorsque vous pensez avoir terminé, vous pouvez simplement désactiver (instruction **#undef**) le(s) drapeau(x) et le code sera automatiquement exclu de la compilation (et vous réduirez la taille et le coût d'exécution de votre fichier).

Il est préférable de choisir les noms de vos drapeaux de déboguage avant de commencer la construction de votre projet afin qu'ils présentent une certaine cohérence. Les drapeaux de préprocesseur sont distingués traditionnellement des variables en les écrivant entièrement en majuscules. Un nom de drapeau classique est simplement **DEBUG**(mais évitez d'utiliser **NDEBUG**, qui, lui, est reservé en C). La séquence des instructions pourrait être :

```
                        #define DEBUG // Probablement dans un fichier d'en-tête
//...
#ifdef DEBUG // Teste l'état du drapeau
/* code de déboguage */
#endif // DEBUG
```

La plupart des implémentations C et C++ vous laissent aussi utiliser **#define**et **#undef**pour contrôler des drapeaux à partir de la ligne de commande du compilateur, de sorte que vous pouvez re-compiler du code et insérer des informations de deboguage en une seule commande (de préférence via le makefile, un outil que nous allons décrire sous peu). Consultez votre documentation préferée pour plus de détails.

Drapeau de déboguage d'exécution

Dans certaines situations, il est plus adapté de lever et baisser des drapeaux de déboguage durant l'exécution du programme, particulièrement en les contrôlant au lancement du programme depuis la ligne de commande. Les gros programmes sont pénibles à recompiler juste pour insérer du code de déboguage.

Pour activer et désactiver du code de déboguage dynamiquement, créez des drapeaux booléens (**bool**) ::

```
//: C03:DynamicDebugFlags.cpp
#include <iostream>
#include <string>
using namespace std;
// Les drapeaux de deboguage ne sont pas nécéssairement globaux :
bool debug = false;

int main(int argc, char* argv[]) {
  for(int i = 0; i < argc; i++)
    if(string(argv[i]) == "--debug=on")
      debug = true;
  bool go = true;
  while(go) {
    if(debug) {
      // Code de déboguage ici
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
        cout << "Le débogueur est activé!" << endl;
      } else {
        cout << " Le débogueur est désactivé." << endl;
      }
      cout << "Activer le débogueur [oui/non/fin]: ";
      string reply;
      cin >> reply;
      if(reply == "oui") debug = true; // Activé
      if(reply == "non") debug = false; // Désactivé
      if(reply == "fin") break; // Sortie du 'while'
    }
  } ///:~
```

Ce programme vous permet d'activer et de désactiver la drapeau de deboguage jusqu'à ce que vous tapiez “fin” pour lui indiquer que vous voulez sortir. Notez qu'il vous faut taper les mots en entier, pas juste une lettre (vous pouvez bien sûr changer ça, si vous le souhaitez). Vous pouvez aussi fournir un argument de commande optionnel qui active le déboguage au démarrage; cet argument peut être placé à n'importe quel endroit de la ligne de commande, puisque le code de démarrage dans la fonction **main**( ) examine tous les arguments. Le test est vraiment simple grâce à l'expression:

```
                        string(argv[i])
```

Elle transforme le tableau de caractères **argv[i]**en une chaîne de caractères (**string**), qui peut en suite être aisément comparée à la partie droite du **==**. Le programme ci-dessus recherche la chaîne **--debug=on**en entier. Vous pourriez aussi chercher **--debug=**et regarder ce qui se trouve après pour offrir plus d'options. Le Volume 2 (disponible depuis *www.BruceEckel.com*) dédie un chapitre à la classe **string**du Standard C++.

Bien qu'un drapeau de déboguage soit un des rares exemples pour lesquels il est acceptable d'utiliser une variable globale, rien n'y oblige. Notez que la variable est en minuscules pour rappeler au lecteur que ce n'est pas un drapeau du préprocesseur.

3.9.2 - Transformer des variables et des expressions en chaînes de caractère

Lorsqu'on ecrit du code de déboguage, il devient vite lassant d'écrire des expressions d'affichage formées d'un tableau de caractères contenant le nom d'une variable suivi de la variable elle même. Heureusement, le Standard C inclut l'opérateur de transformation en chaîne de caractères ' #', qui a déjà été utilisé dans ce chapitre. Lorsque vous placez un #devant un argument dans une macro du préprocesseur, il transforme cet argument en un tableau de caractères. Cela, combiné avec le fait que des tableaux de caractères mis bout à bout sans ponctuation sont concaténés en un tableau unique, vous permet de créer une macro très pratique pour afficher la valeur de variables durant le déboguage :

```
                        #define PR(x) cout << #x " = " << x << "\n";
```

Si vous affichez la variable **a**en utilisant la macro **PR(a)**, cela aura le même effet que le code suivant:

```
                        cout << "a = " << a << "\n";
```

Le même processus peut s'appliquer a des expressions entières. Le programme suivant utilise une macro pour créer un raccourcis qui affiche le texte d'une expression puis évalue l'expression et affiche le résultat:

```
                        //: C03:StringizingExpressions.cpp
  #include <iostream>
  using namespace std;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
#define P(A) cout << #A << ": " << (A) << endl;

int main() {
  int a = 1, b = 2, c = 3;
  P(a); P(b); P(c);
  P(a + b);
  P((c - a)/b);
} ///:~
```

Vous pouvez voir comment une telle technique peut rapidement devenir indispensable, particulièrement si vous êtes sans debogueur (ou devez utiliser des environnements de développement multiples). Vous pouvez aussi insérer un **#ifdef**pour redéfinir **P(A)**à “rien” lorsque vous voulez retirer le déboguage.

3.9.3 - la macro C assert( )

Dans le fichier d'en-tête standard **<cassert>**vous trouverez **assert( )**, qui est une macro de déboguage très utile. Pour utiliser **assert( )**, vous lui donnez un argument qui est une expression que vous “considérez comme vraie.” Le préprocesseur génère du code qui va tester l'assertion. Si l'assertion est fausse, le programme va s'interrompre après avoir émis un message d'erreur indiquant le contenu de l'assertion et le fait qu'elle a échoué. Voici un exemple trivial:

```
                        //: C03:Assert.cpp
// Utilisation de la macro assert()
#include <cassert>  // Contient la macro
using namespace std;

int main() {
  int i = 100;
  assert(i != 100); // Échec
} ///:~
```

La macro vient du C standard, elle est donc disponible également dans le fichier **assert.h**.

Lorsque vous en avez fini avec le déboguage, vous pouvez retirer le code géneré par la macro en ajoutant la ligne :

```
                        #define NDEBUG
```

dans le programme avant d'inclure **<cassert>**, ou bien en définissant NDEBUG dans la ligne de commande du compilateur. NDEBUG est un drapeau utilisé dans **<cassert>**pour changer la façon dont le code est géneré par les macros.

Plus loin dans ce livre, vous trouverez des alternatives plus sophistiquées à **assert( )**.

3.10 - Adresses de fonctions

Une fois qu'une fonction est compilée et chargée dans l'ordinateur pour être exécutée, elle occupe un morceau de mémoire. Cette mémoire, et donc la fonction, a une adresse.

Le C n'a jamais été un langage qui bloquait le passage là où d'autres craignent de passer. Vous pouvez utiliser les adresses de fonctions avec des pointeurs simplement comme vous utiliseriez des adresses de variable. La déclaration et l'utilisation de pointeurs de fonctions semblent un peu plus opaque de prime abord, mais suit la logique du reste du langage.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

3.10.1 - Définir un pointeur de fonction

Pour définir un pointeur sur une fonction qui ne comporte pas d'argument et ne retourne pas de valeur, vous écrivez:

```
void (*funcPtr)();
```

Quand vous regardez une définition complexe comme celle-ci, la meilleure manière de l'attaquer est de commencer par le centre et d'aller vers les bords. “Commencer par le centre” signifie commencer par le nom de la variable qui est **funcPtr**. “Aller vers les bords” signifie regarder à droite l'élément le plus proche (rien dans notre cas; la parenthèse droite nous arrête), puis à gauche (un pointeur révélé par l'astérisque), puis à droite (une liste d'argument vide indiquant une fonction ne prenant aucun argument), puis regarder à gauche (**void,**qui indique que la fonction ne retourne pas de valeur). Ce mouvement droite-gauche-droite fonctionne avec la plupart des déclarations.

Comme modèle, “commencer par le centre” (“ **funcPtr**est un ...”), aller à droite (rien ici – vous êtes arrêté par la parenthèse de droite), aller à gauche et trouver le ‘ *****’ (“... pointeur sur ...”), aller à droite et trouver la liste d'argument vide (“... fonction qui ne prend pas d'argument ... ”), aller à gauche et trouver le **void**(“ **funcPtr**est un pointeur sur une fonction qui ne prend aucun argument et renvoie **void**”).

Vous pouvez vous demander pourquoi ***funcPtr**requiert des parenthèses. Si vous ne les utilisez pas, le compilateur verra:

```
void *funcPtr();
```

Vous déclareriez une fonction (qui retourne **void***) comme on définit une variable. Vous pouvez imaginer que le compilateur passe par le même processus que vous quand il se figure ce qu'une déclaration ou une définition est censée être. Les parenthèses sont nécessaires pour que le compilateur aille vers la gauche et trouve le ‘ *****’, au lieu de continuer vers la droite et de trouver la liste d'argument vide.

3.10.2 - Déclarations complexes & définitions

Cela mis à part, une fois que vous savez comment la syntaxe déclarative du C et du C++ fonctionne, vous pouvez créer des déclarations beaucoup plus compliquées. Par exemple:

```
//: C03:ComplicatedDefinitions.cpp
/* 1. */    void * (*(*fp1)(int))[10];
/* 2. */    float (*(*fp2)(int,int,float))(int);
/* 3. */    typedef double (*(*(*fp3)())[10])();
            fp3 a;
/* 4. */    int (*(*f4())[10])();
int main() {} ///:~
```

Les prendre une par une et employer la méthode de droite à gauche pour les résoudre. Le premier dit que “ **fp1**est un pointeur sur une fonction qui prend un entier en argument et retourne un pointeur sur un tableau de 10 pointeurs **void**.”

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Le second dit que “ **fp2**est un pointeur sur une fonction qui prend trois arguments (**int**, **int,**et **float**) et retourne un **float**.”

Si vous créez beaucoup de définitions complexes, vous voudrez sans doute utiliser un **typedef**. Le troisième exemple montre comment un **typedef**enregistre à chaque fois les descriptions complexes. Cet exemple nous dit que “ **fp3**est un pointeur sur une fonction ne prenant aucun argument et retourne un pointeur sur un tableau de 10 pointeurs de fonctions qui ne prennent aucun argument et retournent des double.” Il nous dit aussi que “ **a**est du type **fp3**.” **typedef**est en général très utile pour établir des descriptions simples à partir de descriptions complexes.

Le numéro 4 est une déclaration de fonction plutôt qu'une définition de variable. “ **f4**est une fonction qui retourne un pointeur sur un tableau de 10 pointeurs de fonctions qui retournent des entiers.”

Vous aurez rarement besoin de déclarations et définitions aussi compliquées que ces dernières. Cependant, si vous vous entraînez à ce genre d'exercice vous ne serez pas dérangé avec les déclarations légèrement compliquées que vous pourrez rencontrer dans la réalité.

3.10.3 - Utiliser un pointeur de fonction

Une fois que vous avez défini un pointeur de fonction, vous devez l'assigner à une adresse de fonction avant de l'utiliser. Tout comme l'adresse du tableau **arr[10]**est produite par le nom du tableau sans les crochets, l'adresse de la fonction **func()**est produite par le nom de la fonction sans la liste d'argument (**func**). Vous pouvez également utiliser la syntaxe suivante, plus explicite, **&func()**. Pour appeler une fonction, vous déférencez le pointeur de la même façon que vous l'avez défini (rappelez-vous que le C et le C++ essaient de produire des définitions qui restent semblables lors de leur utilisation). L'exemple suivant montre comment un pointeur sur une fonction est défini et utilisé:

```
                        //: C03:PointerToFunction.cpp
// Définir et utiliser un pointeur de fonction
#include <iostream>
using namespace std;

void func() {
  cout << "func() called..." << endl;
}

int main() {
  void (*fp)();  // Définir un pointeur de fonction
  fp = func;  // L'initialiser
  (*fp)();    // Le déférencement appelle la fonction
  void (*fp2)() = func;  // Définir et initialiser
  (*fp2)();
} ///:~
```

Après que le pointeur de fonction **fp**soit défini, il est assigné à l'adresse de la fonction **func()**avec **fp = func**(notez que la liste d'arguments manque au nom de la fonction). Le second cas montre une déclaration et une initialisation simultanées.

3.10.4 - Tableau de pointeurs de fonction

L'une des constructions les plus intéressantes que vous puissiez créer est le tableau de pointeurs de fonctions. Pour sélectionner une fonction, il vous suffit d'indexer dans le tableau et de référencer le pointeur. Cela amène le concept de *code piloté par table*; plutôt que d'utiliser un cas ou une condition, vous sélectionnez les fonctions à exécuter en vous basant sur une variable d'état (ou une combinaison de variables d'état). Ce type de design peut être très utile si vous ajoutez ou supprimez souvent des fonctions à la table (ou si vous voulez créer ou changer de table dynamiquement).

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

L'exemple suivant crée des fonctions factices en utilisant un macro du préprocesseur, et crée un tableau de pointeurs sur ces fonctions en utilisant une initialisation globale automatique. Comme vous pouvez le constater, il est facile d'ajouter ou supprimer des fonctions de la table (et de cette façon, les fonctionnalités du programme) en changeant une petite portion du code:

```
                        //: C03:FunctionTable.cpp
// Utilisation d'un tableau de pointeurs de fonctions
#include <iostream>
using namespace std;

// Une macro qui définit des fonctions factices:
#define DF(N) void N() { \
   cout << "la fonction " #N " est appelee..." << endl; }

DF(a); DF(b); DF(c); DF(d); DF(e); DF(f); DF(g);

void (*func_table[])() = { a, b, c, d, e, f, g };

int main() {
  while(1) {
    cout << "pressez une touche de 'a' a 'g' "
      "or q to quit" << endl;
    char c, cr;
    cin.get(c); cin.get(cr); // le second pour CR
    if ( c == 'q' )
      break; // ... sortie du while(1)
    if ( c < 'a' || c > 'g' )
      continue;
    (*func_table[c - 'a'])();
  }
} ///:~
```

A ce stade, vous êtes à même d'imaginer combien cette technique peut être utile lorsqu'on crée une espèce d'interpréteur ou un traitement de liste.

3.11 - Make: gestion de la compilation séparée

Lorsque vous utilisez la *compilation séparée*(découpage du code en plusieurs unités de compilation), il vous faut une manière de compiler automatiquement chaque fichier et de dire à l'éditeur de liens de joindre tous les morceaux - ainsi que les bibliothèques nécessaires et le code de lancement - pour en faire un fichier exécutable. La plupart des compilateurs vous permettent de faire ça avec une seule ligne de commande. Par exemple, pour le compilateur GNU C++, vous pourriez dire

```
                    g++ SourceFile1.cpp SourceFile2.cpp
```

Le problème de cette approche est que le compilateur va commencer par compiler chaque fichier, indépendamment du fait que ce fichier *ait besoin*d'être recompilé ou pas. Pour un projet avec de nombreux fichiers, il peut devenir prohibitif de recompiler tout si vous avez juste changé un seul fichier.

La solution de ce problème, développée sous Unix mais disponible partout sous une forme ou une autre, est un programme nommé **make**. L'utilitaire **make**gère tous les fichiers d'un projet en suivant les instructions contenues dans un fichier texte nommé un **makefile**. Lorsque vous éditez des fichiers dans un projet puis tapez **make**, le programme **make**suis les indications du **makefile**pour comparer les dates de fichiers source aux dates de fichiers cible correspondants, et si un fichier source est plus récent que son fichier cible, **make**déclanche le compilateur sur le fichier source. **make**recompile uniquement les fichiers source qui ont été changés, ainsi que tout autre fichier source affecté par un fichier modifié. En utilisant **make**vous évitez de recompiler tous les fichiers de votre projet à chaque changement, et de vérifier que tout à été construit correctement. Le **makefile**contient toutes les commandes pour construire votre projet. Apprendre **make**vous fera gagner beaucoup de temps et éviter beaucoup de frustration. Vous allez aussi découvrir que **make**est le moyen typique d'installer un nouveau logiciel sous

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Linux/Unix (bien que, le **makefile**pour cela ait tendance à être largement plus complexe que ceux présentés dans ce livre, et que vous générerez souvent le **makefile**pour votre machine particulière pendant le processus d'installation).

Étant donné que **make**est disponible sous une forme ou une autre pour virtuellement tous les compilateurs C++ (et même si ce n'est pas le cas, vous pouvez utiliser un **make**disponible gratuitement avec n'importe quel compilateur), c'est l'outil que nous utiliserons à travers tout ce livre. Cependant, les fournisseurs de compilateur ont aussi créé leur propre outil de construction de projet. Ces outils vous demandent quels fichiers font partie de votre projet et déterminent toutes les relations entre eux par eux-mêmes. Ces outils utilisent quelque chose de similaire à un fichier **makefile**, généralement nommé un *fichier de projet*, mais l'environnement de développement maintient ce fichier de sorte que vous n'avez pas à vous en soucier. La configuration et l'utilisation de fichiers de projets varie d'un environnement de développement à un autre, c'est pourquoi vous devez trouver la documentation appropriée pour les utiliser (bien que les outils de fichier de projet fournis par les vendeurs de compilateur sont en général si simples à utiliser que vous pouvez apprendre juste en jouant avec - ma façon préférée d'apprendre).

Les fichiers **makefile**utilisés à travers ce livre devraient fonctionner même si vous utilisez aussi un outil de construction spécifique.

3.11.1 - Les actions du Make

Lorsque vous tapez **make**(ou le nom porté par votre incarnation de "make"), le programme **make**cherche un fichier nommé **makefile**dans le répertoire en cours, que vous aurez créé si c'est votre projet. Ce fichier liste les dépendances de vos fichiers source. **make**examine la date des fichiers. Si un dépendant est plus ancien qu'un fichier dont il dépend, **make**exécute la *règle*donnée juste après la définition de dépendance.

Tous les commentaires d'un **makefile**commencent par un #et continuent jusqu'à la fin de la ligne.

Un exemple simple de **makefile**pour un programme nommé "bonjour" pourrait être :

```
                        # Un commentaire
bonjour.exe: bonjour.cpp
        moncompilateur bonjour.cpp
```

Cela signifie que **bonjour.exe**(la cible) dépend de **bonjour.cpp**. Quand **bonjour.cpp**a une date plus récente que **bonjour.exe**, **make**applique la "règle" **moncompilateur hello.cpp**. Il peut y avoir de multiples dépendances et de multiples règles. De nombreux programmes de **make**exigent que les règles débutent par un caractère de tabulation. Ceci mis à part, les espaces sont ignorés ce qui vous permet de formater librement pour une meilleure lisibilité.

Les règles ne sont pas limitées à des appels au compilateur; vous pouvez appeler n'importe quel programme depuis **make**. En créant des groupes interdépendants de jeux de règles de dépendance, vous pouvez modifier votre code source, taper **make**et être certain que tous les fichiers concernés seront reconstruits correctement.

Macros

Un **makefile**peut contenir des *macros*(notez bien qu'elle n'ont rien à voir avec celles du préprocesseur C/C++). Les macros permettent le remplacement de chaînes de caractères. Les **makefile**de ce livre utilisent une macro pour invoquer le compilateur C++. Par exemple,

```
                        CPP = moncompilateur
hello.exe: hello.cpp
        $(CPP) hello.cpp
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Le signe **=**est utilisé pour identifier **CPP**comme une macro, et le **$**avec les parenthèses permettent d'utiliser la macro. Dans ce cas, l'utilisation signifie que l'appel de macro **$(CPP)**sera remplacé par la chaîne de caractères **moncompilateur**. Avec la macro ci-dessus, si vous voulez passer à un autre compilateur nommé **cpp**, vous avez simplement à modifier la macro comme cela:

```
CPP = cpp
```

Vous pouvez aussi ajouter des drapeaux de compilation, etc., à la macro, ou bien encore utiliser d'autres macros pour ajouter ces drapeaux.

Règles de suffixes

Il devient vite lassant d'invoquer **make**pour chaque fichier **cpp**de votre projet, alors que vous savez que c'est le même processus basique à chaque fois. Puisque **make**a été inventé pour gagner du temps, il possède aussi un moyen d'abréger les actions, tant qu'elles dépendent du suffixe des noms de fichiers. Ces abréviations se nomment des *règles de suffixes*. Une telle règle permet d'apprendre à **make**comment convertir un fichier d'un type d'extension (**.cpp**, par exemple) en un fichier d'un autre type d'extension (**.obj**ou **.exe**). Une fois que **make**connaît les règles pour produire un type de fichier à partir d'un autre, tout ce qu'il vous reste à lui dire est qui dépend de qui. Lorsque **make**trouve un fichier avec une date plus ancienne que le fichier dont il dépend, il utilise la règle pour créer un nouveau fichier.

La règle de suffixes dit à **make**qu'il n'a pas besoin de règle explicite pour construire tout, mais qu'il peut trouver comment le faire simplement avec les extensions de fichier. Dans ce cas, elle dit "pour construire un fichier qui se termine par **exe**à partir d'un qui finit en **cpp**, activer la commande suivante". Voilà à quoi cela ressemble pour cette règle:

```
CPP = moncompilateur
.SUFFIXES: .exe .cpp
.cpp.exe:
        $(CPP) $&lt;
```

La directive **.SUFFIXES**dit à **make**qu'il devra faire attention aux extensions de fichier suivantes parce qu'elles ont une signification spéciale pour ce makefile particulier. Ensuite, vous voyez la règle de suffixe **.cpp.exe**, qui dit "voilà comment convertir un fichier avec l'extension **cpp**en un avec l'extension **exe**" (si le fichier **cpp**est plus récent que le fichier **exe**). Comme auparavant, la macro **$(CPP)**est utilisée, mais ensuite vous apercevez quelque chose de nouveau: **$<**. Comme ça commence avec un " **$**", c'est une macro, mais c'est une des macros spéciales intégrées à **make**. Le **$<**ne peut être utilisé que dans les règles de suffixe, et il signifie "le dépendant qui a déclenché la règle", ce qui, dans ce cas, se traduit par "le fichier **cpp**qui a besoin d'être compilé".

Une fois les règles des suffixes en place, vous pouvez simplement dire, par exemple, " **make Union.exe**", et la règle de suffixes s'activera bien qu'il n'y ait pas la moindre mention de "Union" dans le **makefile**.

Cibles par défaut

Après les macros et les règles de suffixes, **make**examine la première "cible" dans un fichier, et la construit, si vous n'avez pas spécifié autrement. Ainsi pour le **makefile**suivant:

```
CPP = moncompilateur
.SUFFIXES: .exe .cpp
.cpp.exe:
        $(CPP) $<
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
cible1.exe:
cible2.exe:
```

si vous tapez juste " **make**", ce sera **cible1.exe**qui sera construit (à l'aide de la règle de suffixe par défaut) parce que c'est la première cible que **make**rencontre. Pour construire **cible2.exe**vous devrez explicitement dire " **make cible2.exe**". Cela devient vite lassant, et pour y remédier, on créé normalement une cible "fictive" qui dépend de toutes les autres cibles de la manière suivante :

```
                        CPP = moncompilateur
.SUFFIXES: .exe .cpp
.cpp.exe:
        $(CPP) $<
all: cible1.exe cible2.exe
```

Ici, " **all**" n'existe pas et il n'y a pas de fichier nommé " **all**", du coup, à chaque fois que vous tapez **make**, le programme voit " **all**" comme première cible de la liste (et donc comme cible par défaut), ensuite il voit que " **all**" n'existe pas et qu'il doit donc le construire en vérifiant toutes les dépendances. Alors, il examine **cible1.exe**et (à l'aide de la règle de suffixe) regarde (1) si **cible1.exe**existe et (2) si **cible1.cpp**est plus récent que **cible1.exe**, et si c'est le cas, exécute la règle de suffixe (si vous fournissez une règle explicite pour une cible particulière, c'est cette règle qui sera utilisée à la place). Ensuite, il passe au fichier suivant dans la liste de la cible par défaut. Ainsi, en créant une liste de cible par défaut (typiquement nommée **all**par convention, mais vous pouvez choisir n'importe quel nom) vous pouvez déclancher la construction de tous les exécutables de votre projet en tapant simplement " **make**". De plus, vous pouvez avoir d'autres listes de cibles non-défaut qui font d'autres choses - par exemple vous pouvez arranger les choses de sorte qu'en tapant " **make debug**" vous reconstruisiez tous vos fichiers avec le déboguage branché.

3.11.2 - Les makefiles de ce livre

En utilisant le programme **ExtractCode.cpp**du Volume 2 de ce livre, tous les listings sont automatiquement extraits de la version en texte ASCII de ce livre et placé dans des sous-réertoires selon leur chapitre. De plus, **ExtractCode.cpp**crée plusieurs **makefiles**dans chaque sous-répertoire (avec des noms distincts) pour que vous puissiez simplement vous placer dans ce sous-répertoire et taper **make -f moncompilateur.makefile**(en substituant le nom de votre compilateur à "moncompilateur", le drapeau " **-f**" signifie "utilise ce qui suit comme **makefile**"). Finalement, **ExtractCode.cpp**crée un **makefile**maître dans le répertoire racine où les fichiers du livre ont été décompressés, et ce **makefile**descend dans chaque sous-répertoire et appelle **make**avec le **makefile**approprié. De cette manière, vous pouvez compiler tout le code du livre en invoquant une seule commande **make**, et le processus s'arrêtera dès que votre compilateur rencontrera un problème avec un fichier particulier (notez qu'un compilateur compatible avec le Standard C++ devrait pouvoir compiler tous les fichiers de ce livre). Du fait que les implémentations de **make**varient d'un système à l'autre, seules les fonctionnalités communes de base sont utilisées dans les **makefile**s générés.

3.11.3 - Un exemple de makefile

Comme indiqué, l'outil d'extraction de code **ExtractCode.cpp**génère automatiquement des **makefiles**pour chaque chapitre. De ce fait, ils ne seront pas inclus dans le livre (il sont tous joints au code source que vous pouvez télécharger depuis *www.BruceEckel.com*). Cependant il est utile de voir un exemple. Ce qui suit est une version raccourcie d'un **makefile**qui a été automatiquement généré pour ce chapitre par l'outil d'extraction du livre. Vous trouverez plus d'un **makefile**dans chaque sous-répertoire (ils ont des noms différents ; vous invoquez chacun d'entre eux avec " **make -f**"). Celui-ci est pour GNU C++:

```
                        CPP = g++
OFLAG = -o
.SUFFIXES : .o .cpp .c
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
.cpp.o :
  $(CPP) $(CPPFLAGS) -c $<
.c.o :
  $(CPP) $(CPPFLAGS) -c $<

all: \
  Return \
  Declare \
  Ifthen \
  Guess \
  Guess2
# Le reste des fichiers de ce chapitre est omis

Return: Return.o
  $(CPP) $(OFLAG)Return Return.o

Declare: Declare.o
  $(CPP) $(OFLAG)Declare Declare.o

Ifthen: Ifthen.o
  $(CPP) $(OFLAG)Ifthen Ifthen.o

Guess: Guess.o
  $(CPP) $(OFLAG)Guess Guess.o

Guess2: Guess2.o
  $(CPP) $(OFLAG)Guess2 Guess2.o

Return.o: Return.cpp
Declare.o: Declare.cpp
Ifthen.o: Ifthen.cpp
Guess.o: Guess.cpp
Guess2.o: Guess2.cpp
```

La macro CPP affectée avec le nom du compilateur. Pour utiliser un autre compilateur, vous pouvez soit éditer le **makefile**, soit changer la valeur de la macro sur la ligne de commande de la manière suivante:

```
                        make CPP=cpp
```

Notez cependant, que **ExtractCode.cpp**utilise un système automatique pour construire les fichiers **makefile**des autres compilateurs.

La seconde macro **OFLAG**est le drapeau qui est uilisé pour indiquer le nom de fichier en sortie. Bien que de nombreux compilateurs supposent automatiquement que le fichier de sortie aura le même nom de base que le fichier d'entrée, d'autre ne le font pas (comme les compilateurs Linux/Unix, qui créent un fichier nommé **a.out**par défaut).

Vous pouvez voir deux règles de suffixes, une pour les fichiers **cpp**et une pour les fichiers **c**(au cas où il y aurait du code source C à compiler). La cible par défaut est **all**, et chaque ligne de cette cible est continuée en utilisant le caractère ****, jusqu'à **Guess2**qui est le dernier de la ligne et qui n'en a pas besoin. Il y a beaucoup plus de fichiers dans ce chapitre, mais seulement ceux là sont présents ici par soucis de brièveté.

Les règles de suffixes s'occupent de créer les fichiers objets (avec une extension **.o**) à partir des fichiers **cpp**, mais en général, vous devez spécifier des règles pour créer les executables, parce que, normalement, un exécutable est créé en liant de nombreux fichiers objets différents et **make**ne peut pas deviner lesquels. De plus, dans ce cas (Linux/Unix) il n'y a pas d'extension standard pour les exécutables, ce qui fait qu'une règle de suffixe ne pourrait pas s'appliquer dans ces situations simples. C'est pourquoi vous voyez toutes les règles pour construire les exécutables finaux énoncées explicitement.

Ce **makefile**choisit la voie la plus absolument sure possible; il n'utilise que les concepts basiques de cible et dépendance, ainsi que des macros. De cette manière il est virtuellement garanti de fonctionner avec autant de programmes **make**que possible. Cela a tendance à produire un **makefile**plus gros, mais ce n'est pas si grave

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

puisqu'il est géneré automatiquement par **ExtractCode.cpp**.

Il existe de nombreuses autres fonctions de **make**que ce livre n'utilisera pas, ainsi que de nouvelles et plus intelligentes versions et variations de **make**avec des raccourcis avancés qui peuvent faire gagner beaucoup de temps. Votre documentation favorite décrit probablement les fonctions avancées de votre **make**, et vous pouvez en apprendre plus sur **make**grâce à *Managing Projects with Make*de Oram et Talbott (O'Reilly, 1993). D'autre part, si votre votre vendeur de compilateur ne fournit pas de **make**ou utilise un **make**non-standard, vous pouvez trouver le make de GNU pour virtuellement n'importe quel système existant en recherchant les archives de GNU (qui sont nombreuses) sur internet.

3.12 - Résumé

Ce chapitre était une excursion assez intense parmi les notions fondamentales de la syntaxe du C++, dont la plupart sont héritées du C et en commun avec ce dernier (et résulte de la volonté du C++ d'avoir une compatibilité arrière avec le C). Bien que certaines notions de C++ soient introduites ici, cette excursion est principalement prévue pour les personnes qui sont familières avec la programmation, et doit simplement donner une introduction aux bases de la syntaxe du C et du C++. Si vous êtes déjà un programmeur C, vous pouvez avoir déjà vu un ou deux choses ici au sujet du C qui vous était peu familières, hormis les dispositifs de C++ qui étaient très probablement nouveaux pour vous. Cependant, si ce chapitre vous a semblé un peu accablant, vous devriez passer par le cours du cédérom *Thinking in C: Foundations for C++ and Java*(qui contient des cours, des exercices et des solutions guidées), qui est livré avec ce livre, et également disponible sur *www.BruceEckel.com*.

3.13 - Exercices

Les solutions de exercices suivants peuvent être trouvés dans le document électronique *The Thinking in C++ Annotated Solution Guide*, disponible à petit prix sur www.BruceEckel.com.

1 Créer un fichier d'en-tête (avec une extension ‘ **.h**’). Dans ce fichier, déclarez un groupe de fonctions qui varient par leur liste d'arguments et qui retournent des valeurs parmi les types suivants : **void**, **char**, **int**, and **float**. A présent créez un fichier **.cpp**qui inclut votre fichier d'en-tête et crée les définitions pour toutes ces fonctions. Chaque fonction doit simplement imprimer à l'écran son nom, la liste des paramètres, et son type de retour de telle façon que l'on sâche qu'elle a été appelée. Créez un second fichier **.cpp**qui inclut votre en-tête et définit **int main( )**, qui contient des appels à toutes vos fonctions. Compilez et lancez votre programme.

2 Ecrivez un programme qui utilise deux boucles **for**imbriquées et l'opérateur modulo (**%**) pour détecter et afficher des nombres premiers (nombres entiers qui ne sont divisibles que pas eux-même ou 1).

3 Ecrivez un programme qui utilise une boucle **while**pour lire des mots sur l'entrée standard (**cin**) dans une **string**. C'est une boucle **while**“infinie”, de laquelle vous sortirez (et quitterez le programme) grâce une instruction **break**. Pour chaque mot lu, évaluez le dans un premier temps grâce à une série de **if**pour “associer” une valeur intégrale à ce mot, puis en utilisant une instruction **switch**sur cet entier comme sélecteur (cette séquence d'événements n'est pas présentée comme étant un bon style de programmation ; elle est juste supposée vous fournir une source d'entraînement pour vous exercer au contrôle de l'exécution). A l'intérieur de chaque **case**, imprimez quelque chose qui a du sens. Vous devez choisir quels sont les mots “intéressants” et quelle est leur signification. Vous devez également décider quel mot signalera la fin du programme. Testez le programme en redirigeant un fichier vers l'entrée standard de votre programme (Pour économiser de la saisie, ce fichier peut être le fichier source de votre programme).

4 Modifiez **Menu.cpp**pour utiliser des instructions **switch**au lieu d'instructions **if**.

5 Ecrivez un programme qui évalue les deux expressions dans la section “précédence.”

6 Modifiez **YourPets2.cpp**pour qu'il utilise différents types de données (**char**, **int**, **float**, **double,**et leurs variantes). Lancez le programme et créez une carte de l'arrangement de mémoire résultant. Si vous avez accès à plus d'un type de machine, système d'exploitation, ou compilateur, essayez cette expérience avec autant de variations que you pouvez.

7 Créez deux fonctions, l'une qui accepte un **string***et une autre qui prend un **string&**. Chacune de ces

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

fonctions devraient modifier l'objet **string**externe de sa propre façon. Dans **main( )**, créez et iniatilisez un objet **string**, affichez le, puis passez le à chacune des deux fonctions en affichant les résultats.

8 Ecrivez un programme qui utilise tous les trigraphes pour vérifier que votre compilateur les supporte.

9 Compilez et lancez **Static.cpp**. Supprimez le mot clé **static**du code, recompilez et relancez le, en expliquant ce qui s'est passé.

10 Essayez de compiler et de lier **FileStatic.cpp**avec **FileStatic2.cpp**. Qu'est-il indiqué par le message d'erreur ? Que signifie-t-il ?

11 Modifiez **Boolean.cpp**pour qu'il travaille sur des valeurs **double**plutot que des **int**s.

12 Modifiez **Boolean.cpp**et **Bitwise.cpp**pour qu'ils utilisent des opérateurs explicites (si votre compilateur est conforme au standard C++ il les supportera).

13 Modifiez **Bitwise.cpp**pour utiliser les fonctions définies dans **Rotation.cpp**. Assurez vous d'afficher les résultats de façon suffisamment claire pour être représentative de ce qui se passe pendant les rotations.

14 Modifiez **Ifthen.cpp**pour utiliser l'opérateur ternaire **if-else**(**?:**).

15 Créez une **struct**ure qui manipule deux objets **string**et un **int**. Utilisez un **typedef**pour le nom de la **struct**ure. Créez une instance de cette **struct**, initialisez ses trois membres de votre instance, et affichez les. Récupérez l'adresse de votre instance, et affichez la. Affectez ensuite cette adresse dans un pointeur sur le type de votre structure. Changez les trois valeurs dans votre instance et affichez les, tout en utilisant le pointeur.

16 Créez un programme qui utilise une énumération de couleurs. Créez une variable du type de cette **enum**et affichez les numéros qui correspondent aux noms des couleurs, en utilisant une boucle **for**.

17 Amusez vous à supprimer quelques **union**de **Union.cpp**et regardez comment évolue la taille des objets résultants. Essayez d'affecter un des éléments d'une **union**et de l'afficher via un autre type pour voir ce qui se passe.

18 Créez un programme qui définit deux tableaux d' **int**, l'un juste derrière l'autre. Indexez la fin du premier tableau dans le second, et faites une affectation. Affichez le second tableau pour voir les changements que ceci a causé. Maintenant essayez de définir une variable **char**entre la définition des deux tableaux, et refaites un test. Vous pourrez créer une fonction d'impression pour vous simplifier la tâche d'affichage.

19 Modifiez **ArrayAddresses.cpp**pour travailler sur des types de données **char**, **long int**, **float,**et **double**.

20 Appliquez la technique présentée dans **ArrayAddresses.cpp**pour afficher la taille de la **struct**et les adresses des éléments du tableau dans **StructArray.cpp**.

21 Créez un tableau de **string**et affectez une string à chaque élément. Affichez le tableau grâce à une boucle **for**.

22 Créez deux nouveaux programmes basés sur **ArgsToInts.cpp**pour qu'ils utilisent respectivement **atol( )**et **atof( )**.

23 Modifiez **PointerIncrement2.cpp**pour qu'il utilise une **union**au lieu d'une **struct**.

24 Modifiez **PointerArithmetic.cpp**pour travailler avec des **long**et des **long double**.

25 Definissez une variable du type **float**. Récupérez son adresse, transtypez la en **unsigned char**, et affectez la à un pointeur d' **unsigned char**. A l'aide de ce pointeur et de **[ ]**, indexez la variable **float**et utilisez la fonction **printBinary( )**définie dans ce chapitre pour afficher un plan de la mémoire du **float**(allez de 0 à **sizeof(float)**). Changez la valeur du **float**et voyez si vous pouvez expliquer ce qui se passe (le **float**contient des données encodées).

26 Définissez un tableau d' **int**s Prenez l'adresse du premier élément du tableau et utilisez l'opérateur **static_cast**pour la convertir en **void***. Ecrivez une fonction qui accepte un **void***, un nombre (qui indiquera un nombre d'octets), et une valeur (qui indiquera la valeur avec laquelle chaque octet sera affecté) en paramètres. La fonction devra affecter chaque octet dans le domaine spécifié à la valeur reçue. Essayez votre fonction sur votre tableau d' **int**s.

27 Créez un tableau constant (**const**) de **double**s et un tableau **volatile**de **double**s. Indexez chaque tableau et utilisez **const_cast**pour convertir chaque élément en non- **const**et non- **volatile**, respectivement, et affectez une valeur à chaque element.

28 Créez une fonction qui prend un pointeur sur un tableau de **double**et une valeur indiquant la taille du tableau. La fonction devrait afficher chaque élément du tableau. Maintenant créez un tableau de **double**et initialisez chaque élément à zero, puis utilisez votre fonction pour afficher le tableau. Ensuite, utilisez **reinterpret_cast**pour convertir l'adresse de début du tableau en **unsigned char***, et valuer chaque octet du tableau à 1 (astuce : vous aurez besoin de **sizeof**pour calculer le nombre d'octets d'un **double**). A présent utilisez votre fonction pour afficher les nouveaux résultats. Pourquoi, d'après vous, chaque élément n'est pas

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

égal à la valeur 1.0 ?

29 (Challenge) Modifez **FloatingAsBinary.cpp**pour afficher chaque partie du **double**comme un groupe de bits séparé. Il vous faudra remplacer les appels à **printBinary( )**avec votre propre code spécialisé (que vous pouvez dériver de **printBinary( )**), et vous aurez besoin de comprendre le format des nombres flottants en parallèle avec l'ordre de rangement des octets par votre compilateur (c'est la partie challenge).

30 Créez un makefile qui compile non seulement **YourPets1.cpp**et **YourPets2.cpp**(pour votre compilateur en particulier) mais également qui exécute les deux programmes comme cible par défaut. Assurez vous d'utiliser la règle suffixe.

31 Modifiez **StringizingExpressions.cpp**pour que **P(A)**soit conditionné par **#ifdef**pour autoriser le code en déboggage à être automatiquement démarré grâce à un flag sur la ligne de commande. Vous aurez besoin de consulter la documentation de votre compilateur pour savoir comment définir des valeurs du préprocesseur en ligne de commande.

32 Définissez une fonction qui prend en paramètre un **double**et retourne un **int**. Créez et initialisez un pointeur sur cette fonction, et appelez là à travers ce pointeur.

33 Déclarez un pointeur de fonction recevant un paramètre **int**et retournant un pointeur de fonction qui reçoit un **char**et retourne un **float**.

34 Modifiez **FunctionTable.cpp**pour que chaque fonction retourne une **string**(au lieu d'afficher un message) de telle façon que cette valeur soit affichée directement depuis **main( )**.

35 Créez un **makefile**pour l'un des exercices précédents (de votre choix) qui vous permettra de saisir **make**pour un build de production du programme, et **make debug**pour un build de l'application comprenant les informations de déboggage.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

4 - Abstraction des données

C++ est un outil destiné à augmenter la productivité. Sinon, pourquoi feriez-vous l'effort (et c'est un effort, indépendamment de la facilité que nous essayons de donner à cette transition)

de passer d'un langage que vous connaissez déjà et avec lequel vous êtes productif à un nouveau langage avec lequel vous serez *moins*productif l'espace de quelques temps, jusqu'à ce que vous le maîtrisiez? C'est parce que vous avez été convaincus des avantages importants que vous allez obtenir avec ce nouvel outil.

La productivité, en termes de programmation informatique, signifie qu'un nombre réduit de personnes pourront écrire des programmes plus complexes et plus impressionnants en moins de temps. Il y a certainement d'autres enjeux qui interviennent lors du choix d'un langage, comme l'efficacité (la nature même du langage est-elle source de ralentissement et de gonflement du code source?), la sûreté (le langage vous aide-t'il à assurer que votre programme fera toujours ce que vous avez prévu, et qu'il traîte les erreurs avec élégance?), et la maintenance (le langage vous aide-t'il à créer du code facile à comprendre, à modifier, et à étendre?). Ce sont, à n'en pas douter, des facteurs importants qui seront examinés dans cet ouvrage.

La productivité en tant que telle signifie qu'un programme dont l'écriture prenait une semaine à trois d'entre vous, ne mobilisera maintenant qu'un seul d'entre vous durant un jour ou deux. Cela touche l'économie à plusieurs niveaux. Vous êtes content, car vous récoltez l'impression de puissance qui découle de l'acte de construire quelque chose, votre client (ou patron) est content, car les produits sont développés plus rapidement et avec moins de personnel, et les consommateurs sont contents, car ils obtiennent le produit à meilleur prix. La seule manière d'obtenir une augmentation massive de productivité est de s'appuyer sur le code d'autres personnes, c'est-à-dire d'utiliser des bibliothèques.

Une bibliothèque est simplement une collection de codes qu'une tierce personne a écrits et assemblés dans un paquetage. Souvent, un paquetage minimal se présente sous la forme d'un fichier avec une extension telle que **lib**et un ou plusieurs fichiers d'en-tête destinés à informer votre compilateur du contenu de la bibliothèque. L'éditeur de liens sait comment rechercher au sein du fichier de la bibliothèque et extraire le code compilé approprié. Mais il s'agit là seulement d'une manière de distribuer une bibliothèque. Sur des plateformes qui couvrent plusieurs architectures, telles que Linux/Unix, la seule façon de distribuer une bibliothèque est de la distribuer avec son code source, de manière qu'il puisse être reconfiguré et recompilé sur la nouvelle cible.

Ainsi, l'usage de bibliothèques est le moyen le plus important destiné à accroître la productivité, et un des objectifs de conception principaux de C++ est de rendre l'utilisation de bibliothèques plus aisée. Ceci implique qu'il y a quelque chose de compliqué concernant l'utilisation de bibliothèques en C. La compréhension de ce facteur vous donnera un premier aperçu de la conception de C++, et par conséquent un aperçu de comment l'utiliser.

4.1 - Une petite bibliothèque dans le style C

Une bibliothèque commence habituellement comme une collection de fonctions, mais si vous avez utilisé des bibliothèques C écrites par autrui, vous savez qu'il s'agit généralement de plus que cela, parce que la vie ne se limite pas à des comportements, des actions et des fonctions. On y trouve également des caractéristiques (bleu, livres, texture, luminance), qui sont représentées par des données. Et lorsque vous commencez à travailler avec un ensemble de caractéristiques en C, il est très pratique de les rassembler dans une structure, particulièrement si vous désirez représenter plus d'un objet similaire dans l'espace de votre problème. De cette manière, vous pouvez définir une variable du type de cette structure pour chaque objet.

Ainsi, la plupart des bibliothèques C se composent d'un ensemble de structures et d'un ensemble de fonctions qui agissent sur ces structures. Comme exemple de ce à quoi un tel système peut ressembler, considérez un objet qui se comporte comme un tableau, mais dont la taille peut être établie à l'exécution, lors de sa création. Je l'appellerai **CSTash**. Bien qu'il soit écrit en C++, il utilise un style qui correspond à ce que vous écririez en C:

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
//: C04:CLib.h
// Fichier d'en-tête pour une bibliothèque
// écrite dans le style C
// Un entité semblable à un tableau créée à l'exécution

typedef struct CStashTag {
  int size;      // Taille de chaque espace
  int quantity;  // Nombre d'espaces de stockage
  int next;      // Prochain espace libre
  // Tableau d'octets alloué dynamiquement:
  unsigned char* storage;
} CStash;

void initialize(CStash* s, int size);
void cleanup(CStash* s);
int add(CStash* s, const void* element);
void* fetch(CStash* s, int index);
int count(CStash* s);
void inflate(CStash* s, int increase);
///:~
```

Un nom tel que **CStashTag**est généralement employé pour une structure au cas où vous auriez besoin de référencer cette structure à l'intérieur d'elle-même. Par exemple, lors de la création d'une *liste chaînée*(chaque élément dans votre liste contient un pointeur vers l'élément suivant), vous avez besoin d'un pointeur sur la prochaine variable **struct**, vous avez donc besoin d'un moyen d'identifier le type de ce pointeur au sein du corps même de la structure. Aussi, vous verrez de manière presque universelle le mot clé **typedef**utilisé comme ci-dessus pour chaque **struct**présente dans une bibliothèque C. Les choses sont faites de cette manière afin que vous puissiez traîter une structure comme s'il s'agissait d'un nouveau type et définir des variables du type de cette structure de la manière suivante:

```
CStash A, B, C;
```

Le pointeur **storage**est de type **unsigned char***. Un **unsigned char**est la plus petite unité de stockage que supporte un compilateur C, bien que, sur certaines machines, il puisse être de la même taille que la plus grande. Cette taille dépend de l'implémentation, mais est souvent de un octet. Vous pourriez penser que puisque **CStash**est conçu pour contenir n'importe quel type de variable, **void***serait plus approprié. Toutefois, l'idée n'est pas ici de traîter cet espace de stockage comme un bloc d'un type quelconque inconnu, mais comme un bloc contigu d'octets.

Le code source du fichier d'implémentation (que vous n'obtiendrez pas si vous achetez une bibliothèque commerciale - vous recevrez seulement un **obj**, ou un **lib**, ou un **dll**, etc. compilé) ressemble à cela:

```
                    //: C04:CLib.cpp {O}
// Implantation de l'exemple de bibliothèque écrite
// dans le style C
// Declaration de la structure et des fonctions:
#include "CLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Quantité d'éléments à ajouter
// lorsqu'on augmente l'espace de stockage:
const int increment = 100;

void initialize(CStash* s, int sz) {
  s->size = sz;
  s->quantity = 0;
  s->storage = 0;
  s->next = 0;
}

int add(CStash* s, const void* element) {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  if(s->next >= s->quantity) //Il reste suffisamment d'espace?
    inflate(s, increment);
  // Copie l'élément dans l'espace de stockage,
  // en commençant au prochain espace vide:
  int startBytes = s->next * s->size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < s->size; i++)
    s->storage[startBytes + i] = e[i];
  s->next++;
  return(s->next - 1); // Numéro de l'indice
}

void* fetch(CStash* s, int index) {
  // Vérifie les valeurs limites de l'indice:
  assert(0 <= index);
  if(index >= s->next)
    return 0; // Pour indiquer la fin
  // Produit un pointer sur l'élément désiré:
  return &(s->storage[index * s->size]);
}

int count(CStash* s) {
  return s->next;  // Eléments dans CStash
}

void inflate(CStash* s, int increase) {
  assert(increase > 0);
  int newQuantity = s->quantity + increase;
  int newBytes = newQuantity * s->size;
  int oldBytes = s->quantity * s->size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = s->storage[i]; // Copie l'ancien espace vers le nouveau
  delete [](s->storage); // Ancien espace
  s->storage = b; // Pointe sur le nouvel espace mémoire
  s->quantity = newQuantity;
}

void cleanup(CStash* s) {
  if(s->storage != 0) {
   cout << "freeing storage" << endl;
   delete []s->storage;
  }
} ///:~
```

initialize()effectue le travail d'initialisation pour la structure **CStash**en fixant les variables internes à une valeur appropriée. Initialement, le pointeur **storage**est mis à zéro - aucun espace de stockage n'est alloué.

La fonction **add()**insère un élément dans le **CStash**à la prochaine position disponible. D'abord, elle contrôle si il reste de l'espace à disposition. Si ce n'est pas le cas, elle étend l'espace de stockage en utilisant la fonction **inflate()**, décrite plus loin.

Parce que le compilateur ne connaît pas le type spécifique de la variable stockée (tout ce que la fonction reçoit est un **void***), vous ne pouvez pas simplement faire une affectation, ce qui serait certainement chose pratique. A la place, vous devez copier la variable octet par octet. La manière la plus évidente de réaliser cette copie est par itération sur les indices d'un tableau. Typiquement, **storage**contient déjà des octets de données, ce qui est indiqué par la valeur de **next**. Afin de démarrer avec le décalage d'octets approprié, **next**est multiplié par la taille de chaque élément (en octets) de manière à produire **startBytes**. Puis, l'argument **element**est converti en un **unsigned char***de façon telle qu'il peut être adressé octet par octet et copié dans l'espace de stockage disponible. **next**est incrémenté de manière à indiquer le prochain emplacement disponible, et l'"indice" où la valeur a été placée afin de pouvoir récupérer cette dernière en utilisant cet indice avec **fetch()**.

fetch()vérifie que l'indice n'est pas en dehors des limites et retourne l'adresse de la variable désirée, calculée à l'aide de l'argument **index**. Puisque **index**représente le nombre d'éléments à décaler dans **CStash**, il doit être multiplié par le nombre d'octets occupés par chaque entité pour produire le décalage numérique en octets. Lorsque ce décalage est utilisé pour accéder à un élément de **storage**en utilisant l'indiçage d'un tableau, vous n'obtenez pas l'adresse, mais au lieu de cela l'octet stocké à cette adresse. Pour produire l'adresse, vous devez utiliser

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

l'opérateur *adresse-de* **&**.

count()peut au premier abord apparaître un peu étrange au programmeur C expérimenté. Cela ressemble à une complication inutile pour faire quelque chose qu'il serait probablement bien plus facile de faire à la main. Si vous avez une structure **CStash**appelée intStash, par exemple, il semble bien plus évident de retrouver le nombre de ses éléments en appelant **inStash.next**plutôt qu'en faisant un appel de fonction (qui entraîne un surcoût), tel que **count(&intStash)**. Toutefois, si vous désirez changer la représentation interne de CStash, et ainsi la manière dont le compte est calculé, l'appel de fonction apporte la flexibilité nécessaire. Mais hélas, la plupart des programmeurs ne s'ennuieront pas à se documenter sur la conception "améliorée" de votre bibliothèque. Ils regarderont la structure et prendront directement la valeur de **next**, et peut-être même qu'ils modifieront **next**sans votre permission. Si seulement il y avait un moyen pour le concepteur de bibliothèque d'obtenir un meilleur contrôle sur de telles opérations! (Oui, c'est un présage.)

4.1.1 - Allocation dynamique de mémoire

Vous ne savez jamais la quantité maximale de mémoire dont vous pouvez avoir besoin pour un **CStash**, ainsi l'espace mémoire pointé par **storage**est alloué sur le tas. Le tas est un grand bloc de mémoire utilisé pour allouer de plus petits morceaux à l'exécution. Vous utilisez le tas lorsque vous ne connaissez pas la taille de l'espace mémoire dont vous aurez besoin au moment où vous écrivez un programme. C'est-à-dire que seulement à l'exécution, vous découvrez que vous avez besoin de la mémoire nécessaire pour stocker 200 variables **Airplane**au lieu de 20. En C standard, les fonctions d'allocation dynamique de mémoire incluent **malloc()**, **calloc**, **realloc**, et **free()**. En lieu et place d'appels à la bibliothèque standard, le C++ utilise une approche plus sophistiquée (bien que plus simple d'utilisation) à l'allocation dynamique de mémoire qui est intégrée au langage via les mots clés **new**et **delete**.

La fonction **inflate()**utilise **new**pour obtenir un plus grand espace de stockage pour **CStash**. Dans ce cas, nous nous contenterons d'étendre l'espace mémoire et ne le rétrécirons pas, et l'appel à **assert()**nous garantira qu'aucun nombre négatif ne sera passé à **inflate()**en tant que valeur de **increase**. Le nouveau nombre d'éléments pouvant être stocké (après complétion de **inflate()**) est calculé en tant que **newQuantity**, et il est multiplié par le nombre d'octets par élément afin de produire **newBytes**, qui correspond au nombre d'octets alloués. De manière à savoir combien d'octets copier depuis l'ancien emplacement mémoire, **oldBytes**est calculée en utilisant l'ancienne valeur **quantity**.

L'allocation d'espace mémoire elle-même a lieu au sein de l'expression *new*, qui est l'expression mettant en jeu le mot clé **new**:

```
new unsigned char[newBytes];
```

La forme générale de l'expression *new*est:

new Type;

dans laquelle **Type**décrit le type de la variable que vous voulez allouer sur le tas. Dans ce cas, on désire un tableau de **unsigned char**de longueur **newBytes**, c'est donc ce qui apparaît à la place de **Type**. Vous pouvez également allouer l'espace pour quelque chose d'aussi simple qu'un **int**en écrivant:

```
new int;
```

et bien qu'on le fasse rarement, vous pouvez constater que la forme de l'expression reste cohérente.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Une expression *new*retourne un *pointeur*sur un objet du type exact de celui que vous avez demandé. Ainsi, si vous écrivez **new Type**, vous obtenez en retour un pointeur sur un **Type**. Si vous écrivez **new int**, vous obtenez un pointeur sur un **int**. Si vous désirez un **new** *tableau*de **unsigned char**, vous obtenez une pointeur sur le premier élément de ce tableau. Le compilateur s'assurera que vous affectiez la valeur de retour de l'expression *new*à un pointeur du type correct.

Bien entendu, à chaque fois que vous demandez de la mémoire, il est possible que la requête échoue, s'il n'y a plus de mémoire. Comme vous l'apprendrez, C++ possède des mécanismes qui entrent en jeu lorsque l'opération d'allocation de mémoire est sans succès.

Une fois le nouvel espace de stockage alloué, les données contenues dans l'ancien emplacement doivent être copiées dans le nouveau; Ceci est une nouvelle fois accompli par itération sur l'indice d'un tableau, en copiant un octet après l'autre dans une boucle. Après que les données aient été copiées, l'ancien espace mémoire doit être libéré de manière à pouvoir être utilisé par d'autres parties du programme au cas où elles nécessiteraient de l'espace mémoire supplémentaire. Le mot clé **delete**est le complément de **new**, et doit être utilisé pour libérer tout espace alloué par **new**(si vous oubliez d'utiliser **delete**, cet espace mémoire demeure indisponible, et si ce type de *fuite de mémoire*, comme on l'appelle communément, apparaît suffisamment souvent, il est possible que vous arriviez à court de mémoire). Par ailleurs, il existe une syntaxe spéciale destinée à effacer un tableau. C'est comme si vous deviez rappeler au compilateur que ce pointeur ne pointe pas simplement sur un objet isolé, mais sur un tableau d'objets: vous placez une paire de crochets vides à gauche du pointeur à effacer:

```
delete []myArray;
```

Une fois que l'ancien espace a été libéré, le pointeur sur le nouvel emplacement mémoire peut être affecté au pointeur **storage**, la variable **quantity**est ajustée, et **inflate()**a terminé son travail.

Notez que le gestionnaire du tas est relativement primitif. Il vous fournit des morceaux d'espace mémoire et les récupère lorsque vous les libérez. Il n'y a pas d'outil inhérent de compression du tas, qui compresse le tas de manière à obtenir de plus grands espaces libres. Si un programme alloue et libère de la mémoire sur le tas depuis un moment, vous pouvez finir avec un tas *fragmenté*qui possède beaucoup de mémoire libre, mais sans aucun morceau de taille suffisante pour allouer l'espace dont vous avez besoin maintenant. Un défragmenteur de tas complique le programme parce qu'il déplace des morceaux de mémoire, de manière telle que vos pointeurs ne conserveront pas leur valeur. Certains environnements d'exploitation possèdent un défragmenteur de tas intégré, mais ils exigent que vous utilisiez des *manipulateurs*spéciaux de mémoire (qui peuvent être convertis en pointeurs de manière temporaire, après avoir verrouillé l'espace mémoire en question pour le défragmenteur de tas ne puisse le déplacer) à la place des pointeurs. Vous pouvez également mettre en oeuvre votre propre schéma de compression du tas, mais ce n'est pas là une tâche à entreprendre à la légère.

Lorsque vous créez une variable sur la pile à la compilation, l'espace pour cette variable est automatiquement créé et libéré par le compilateur. Le compilateur sait exactement quelle quantité de mémoire est nécessaire, et il connaît la durée de vie des variables grâce à leur portée. Avec l'allocation dynamique de mémoire, toutefois, le compilateur ne sait pas de combien d'espace mémoire vous aurez besoin, et il ne connaît pas la durée de vie de cet espace. C'est-à-dire, la mémoire n'est pas libérée automatiquement. Pour cette raison, vous êtes responsable de la restitution de la mémoire à l'aide de **delete**, qui indique au gestionnaire du tas que l'espace mémoire en question peut être utilisé par le prochain appel à **new**. L'emplacement logique où ce mécanisme de libération doit être mis en oeuvre dans la bibliothèque, c'est dans la fonction **cleanup()**, parce que c'est à cet endroit que tout le nettoyage de fermeture est effectué.

Dans le but de tester la bibliothèque, deux **CStash**es sont créés. Le premier contient des **int**s et le deuxième contient un tableau de 80 **char**s:

```
//: C04:CLibTest.cpp
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
//{L} CLib
// Teste la bibliothèque écrite dans le style C
#include "CLib.h"
#include <fstream>
#include <iostream>
#include <string>
#include <cassert>
using namespace std;

int main() {
  // Définit les variables au début
  // d'un bloc, comme en C:
  CStash intStash, stringStash;
  int i;
  char* cp;
  ifstream in;
  string line;
  const int bufsize = 80;
  // Maintenant, rappelez-vous d'initialiser les variables:
  initialize(&intStash, sizeof(int));
  for(i = 0; i < 100; i++)
    add(&intStash, &i);
  for(i = 0; i < count(&intStash); i++)
    cout << "fetch(&intStash, " << i << ") = "
         << *(int*)fetch(&intStash, i)
         << endl;
  // Contient des chaînes de 80 caractères:
  initialize(&stringStash, sizeof(char)*bufsize);
  in.open("CLibTest.cpp");
  assert(in);
  while(getline(in, line))
    add(&stringStash, line.c_str());
  i = 0;
  while((cp = (char*)fetch(&stringStash,i++))!=0)
    cout << "fetch(&stringStash, " << i << ") = "
         << cp << endl;
  cleanup(&intStash);
  cleanup(&stringStash);
} ///:~
```

En suivant la forme requise par le C, toutes les variables sont créées au début de la portée de **main()**. Bien entendu, vous devez vous souvenir d'initialiser les variables **CStash**plus tard dans le bloc en appelant **initialize()**. Un des problèmes avec les bibliothèques C est que vous devez consciencieusement inculquer à l'utilisateur l'importance des fonctions d'initialisation et de nettoyage. Si ces fonctions ne sont pas appelées, il y aura de nombreux problèmes. Malheureusement, l'utilisateur ne se demande pas toujours si l'initialisation et le nettoyage sont obligatoires. Ils savent ce qu'ils veulent accomplir, et ils ne se sentent pas si concernés que cela en vous voyant faire de grands signes en clamant, "Hé, attendez, vous devez d'abord faire *ça*!" Certains utilisateurs initialisent même les éléments d'une structure par eux-même. Il n'existe aucun mécanisme en C pour l'empêcher (encore un présage).

La variable de type **intStash**est remplie avec des entiers, tandis que la variable de type **stringStash**est remplie avec des tableaux de caractères. Ces tableaux de caractères sont produits en ouvrant le fichier source, **CLibTest.cpp**, et en lisant les lignes de ce dernier dans une variable **string**appelée **line**, et puis en produisant un pointeur sur la représentation tableau de caractères de **line**à l'aide de la fonction membre **c_str()**.

Après que chaque **Stash**ait été chargé en mémoire, il est affiché. La variable de type **intStash**est imprimée en utilisant une boucle **for**, qui utilise **count()**pour établir la condition limite. La variable de type **stringStash**est imprimée à l'aide d'une boucle **while**qui s'interrompt lorsque **fetch()**retourne zéro pour indiquer qu'on est en dehors des limites.

Vous aurez également noté une convertion supplémentaire dans

```
cp = (char*)fetch(&stringStash,i++)
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

C'est dû à la vérification plus stricte des types en C++, qui ne permet pas d'affecter simplement un **void*** à n'importe quel autre type (C permet de faire cela).

4.1.2 - Mauvaises conjectures

Il y a un enjeu plus important que vous devriez comprendre avant que l'on regarde les problèmes généraux liés à la création de bibliothèques en C. Notez que le fichier d'en-tête **CLib.h** *doit*être inclus dans chaque fichier qui fait référence à **CStash**parce que le compilateur est incapable de deviner à quoi cette structure peut ressembler. Toutefois, il peut deviner à quoi ressemble une fonction; ceci peut apparaître comme une fonctionnalité mais il s'agit là d'un piège majeur du C.

Bien que vous devriez toujours déclarer des fonctions en incluant un fichier d'en-tête, les déclarations de fonctions ne sont pas essentielles en C. Il est possible en C (mais pas en C++) d'appeler une fonction que vous n'avez pas déclarée. Un bon compilateur vous avertira que vous avez probablement intérêt à d'abord déclarer une fonction, mais ce n'est pas imposé pas la norme du langage C. Il s'agit là d'une pratique dangereuse parce que le compilateur C peut supposer qu'une fonction que vous appelez avec un **int**en argument possède une liste d'argument contenant **int**, même si elle contient en réalité un **float**. Ce fait peut entraîner des bugs qui sont très difficiles à démasquer, comme vous allez le voir.

Chaque fichier d'implantation séparé (avec l'extension **.c**) représente une *unité de traduction*. Cela signifie que le compilateur travaille séparément sur chaque unité de traduction, et lorsque qu'il est en train de travailler, il ne connaît que cette unité. Par conséquent, toute information que vous lui fournissez en incluant des fichiers d'en-tête est importante, car cela conditionne la compréhension qu'aura le compilateur du reste de votre programme. Les déclarations dans les fichiers d'en-tête sont particulièrement importantes, parce que partout où un fichier d'en-tête est inclus, le compilateur saura exactement quoi faire. Si, par exemple, vous avez une déclaration dans un fichier d'en-tête qui dit **void func(float)**, le compilateur sait que si vous appelez cette fonction avec un argument entier, il doit convertir cet **int**en un **float**lors du passage de l'argument (ce processus est appelé *promotion*). En l'absence de déclaration, le compilateur C fera simplement l'hypothèse qu'une fonction **func(int)**existe, il n'effectuera pas la promotion, et la donnée erronée se verra ainsi passée silencieusement à **func()**.

Pour chaque unité de traduction, le compilateur crée un fichier objet, avec l'extension **.o**ou **.obj**ou quelque chose de similaire. Ces fichiers objets, en plus du code de démarrage nécessaire, doivent être collectés par l'éditeur de liens au sein d'un programme exécutable. Durant l'édition des liens, toutes les références externes doivent être résolues. Par exemple, dans **CLibTest.cpp**, des fonctions telles que **initialize()**et **fetch()**sont déclarées (cela signifie que le compilateur est informé de ce à quoi elles ressemblent) et utilisées, mais pas définies. Elles sont définies ailleurs, dans CLib.cpp. Par conséquent, les appels dans **CLibTest.cpp**sont des références externes. L'éditeur de liens doit, lorsqu'il réunit tous les fichiers objets, traîter les références externes non résolues et trouver les adresses auxquelles elles font référence. Ces adresses sont placées dans le programme exécutable en remplacement des références externes.

Il est important de réaliser qu'en C, les références externes que l'éditeur de liens recherche sont simplement des noms de fonctions, généralement précédés d'un caractère de soulignement. Ainsi, tout ce que l'éditeur de liens a à faire, c'est de réaliser la correspondance entre le nom d'une fonction, lorsque celle-ci est appelée, et le corps de cette fonction dans un fichier objet, et c'est tout. Si vous faites accidentellement un appel que le compilateur interprète comme **func(int)**et qu'il y a un corps de fonction pour **func(float)**dans un autre fichier objet, l'éditeur de liens verra _**func**d'un côté et _**func**de l'autre, et il pensera que tout est en ordre. L'appel à **func()**placera un **int**sur la pile, alors que le corps de la fonction **func()**attend la présence d'un **float**au sommet de la pile. Si la fonction se contente de lire la valeur et n'écrit pas dans cet emplacement, cela ne fera pas sauter la pile. En fait, la valeur **float**qu'elle lit depuis la pile peut même avoir un sens. C'est pire, car il est alors plus difficile de découvrir le bug.

4.2 - Qu'est-ce qui ne va pas?

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Nous avons des capacités d'adaptation remarquables, y compris dans les situations dans lesquelles nous ne *devrions*peut être pas nous adapter. Le modèle de la bibliotheque **CStash**était une entrée en matière destinée aux programmeurs en langage C, mais si vous la regardiez pendant un moment, vous pourriez noter qu'elle est plutôt... maladroite. Quand vous l'employez, vous devez passer l'adresse de la structure à chaque fonction de la bibliothèque. En lisant le code, le mécanisme de fonctionnement de la bibliothèque se mélange avec la signification des appels de fonction, ce qui crée la confusion quand vous essayez de comprendre ce qui se passe.

Un des obstacles majeurs à l'utilisation de bibliothèques C est cependant le problème de *collision des noms*. C possède un espace de nom unique pour les fonctions; c'est-à-dire que, quand l'éditeur de liens recherche le nom d'une fonction, il regarde dans une liste principale unique. De plus, quand le compilateur travaille sur une unité de traduction, il ne peut travailler qu'avec une seule fonction ayant un nom donné.

Supposez maintenant que vous décidiez d'acheter deux bibliothèques différentes à deux fournisseurs différents, et que chaque bibliothèque dispose d'une structure qui doit etre initialisée et nettoyée. Chaque fournisseur décide que **initialize( )**et **cleanup( )**sont des noms adaptés. Si vous incluez leur deux fichiers d'en-tête dans une même unité de traduction, que fait le compilateur C? Heureusement, C vous donne une erreur, et vous indique qu'il y a une disparité dans les deux listes d'arguments des fonctions déclarées. Mais même si vous ne les incluez pas dans la même unité de traduction, l'éditeur de lien aura toujours des problèmes. Un bon éditeur de liens détectera qu'il y a une collision de noms, mais certains autres prendront le premier nom qu'ils trouvent, en cherchant dans la listes de fichiers objets selon l'ordre dans lequelle vous les lui avez donnés. (Ce peut même être considéré comme une fonctinonnalité du fait qu'il vous permet de remplacer une fonction de bibliothèque par votre propre version.)

D'une manière ou d'une autre, vous ne pourrez pas utiliser deux bibliothèques C contenant une fonction ayant un nom identique. Pour résoudre ce problème, les fournisseurs de bibliothèques ajouteront souvent une séquence unique de caractères au début de tous les noms de leurs fonctions. Ainsi **initialize( )**et **cleanup( )**deviendront **CStash_initialize( )**et **CStash_cleanup( )**. C'est une chose logique à faire car cela “décore” le nom de la fonction avec le nom de la **struct**sur laquelle elle travaille.

Maintenant, il est temps de suivre la première étape vers la création de classes en C++. Les noms de variables dans une **struct**n'entrent pas en collision avec ceux des variables globales. Alors pourquoi ne pas tirer profit de ceci pour des noms de fonctions, quand ces fonctions opèrent sur une **struct**particulière ? Autrement dit, pourquoi ne pas faire des fonctions membres de **struct**s ?

4.3 - L'objet de base

La première étape est exactement celle-là. Les fonctions C++ peuvent être placées à l'intérieur de structures sous la forme de "fonctions membres". Voilà à quoi peut ressembler le code après conversion de la version C de **CStash**en une version C++ appelée **Stash**:

```
                    //: C04:CppLib.h
// Bibliothèque dans le style C convertie en C++

struct Stash {
  int size;      // Taille de chaque espace
  int quantity;  // Nombre d'espaces de stockage
  int next;      // Prochain emplacement libre
   // Allocation dynamique d'un tableau d'octets:
  unsigned char* storage;
  // Fonctions!
  void initialize(int size);
  void cleanup();
  int add(const void* element);
  void* fetch(int index);
  int count();
  void inflate(int increase);
}; ///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Tout d'abord, notez qu'il n'y a pas de **typedef**. A la place de vous demander de créer un **typedef**, le compilateur C++ transforme le nom de la structure en un nouveau nom de type disponible pour le programme (de la même manière que **int**, **char**, **float**et **double**sont des noms de type).

Tous les membres données sont exactement les mêmes que précédemment, mais maintenant, les fonctions se trouvent à l'intérieur du corps de la structure. Par ailleurs, notez que le premier argument présent dans la version C de la bibliothèque a été éliminé. En C++, au lieu de vous forcer à passer l'adresse de la structure en premier argument de toutes les fonctions qui agissent sur cette structure, le compilateur le fait secrètement à votre place. Maintenant, les seuls arguments passés aux fonctions concernent ce que la fonction *fait*, et non le mécanisme sous-jacent lié au fonctionnement de cette fonction.

C'est important de réaliser que le code des fonctions est effectivement le même que dans la version C de la bibliothèque. Le nombre d'arguments est le même (même si vous ne voyez pas l'adresse de la structure, elle est toujours là), et il y a exactement un corps de fonction pour chaque fonction. C'est à dire que le simple fait d'écrire

```
Stash A, B, C;
```

ne signifie pas une fonction **add()**différente pour chaque variable.

Ainsi, le code généré est presque identique à celui que vous auriez écrit pour la version C de la bibliothèque. De manière assez intéressante, cela inclus la "décoration des noms" que vous auriez probablement mise en oeuvre afin de produire **Stash_initialize()**, **Stash_cleanup()**, etc. Lorsque le nom de fonction se trouve à l'intérieur d'une structure, le compilateur réalise effectivement les mêmes opérations. C'est pourquoi **initialize()**situé à l'intérieur de la structure **Stash**n'entre pas en collision avec une fonction appelée **initialize()**située dans une autre structure, ou même avec une fonction globale nommée **initialize()**. La plupart du temps vous n'avez pas besoin de vous préoccuper de la décoration des noms de fonctions - vous utilisez les noms non décorés. Mais parfois, vous avez besoin de pouvoir spécifier que cet **initialize()**appartient à la structure **Stash**, et pas à n'importe quelle autre structure. En particulier, lorsque vous définissez la fonction, vous avez besoin de spécifier de manière complète de quelle fonction il s'agit. Pour réaliser cette spécification complète, C++ fournit un opérateur (**::**) appelé *opérateur de résolution de portée*(appelé ainsi, car les noms peuvent maintenant exister sous différentes portées: à un niveau global ou au sein d'une structure). Par exemple, si vous désirez spécifier **initialize()**, qui appartient à **Stash**, vous écrivez **Stash::initialize(int size)**. Vous pouvez voir ci-dessous comment l'opérateur de résolution de portée est utilisé pour la définition de fonctions:

```
//: C04:CppLib.cpp {O}
// Bibliothèque C convertie en C++
// Déclare une structure et des fonctions:
#include "CppLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Quantité d'éléments à ajouter
// lorsqu'on augmente l'espace de stockage:
const int increment = 100;

void Stash::initialize(int sz) {
  size = sz;
  quantity = 0;
  storage = 0;
  next = 0;
}

int Stash::add(const void* element) {
  if(next >= quantity) // Reste-il suffisamment de place?
    inflate(increment);
  // Copie element dans storage,
  // en commençant au prochain espace libre:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Numéro d'indice
}

void* Stash::fetch(int index) {
  // Check index boundaries:
  assert(0 <= index);
  if(index >= next)
    return 0; // Pour indiquer la fin
  // Retourne un pointeur sur l'élément désiré:
  return &(storage[index * size]);
}

int Stash::count() {
  return next; // Nombre d'éléments dans CStash
}

void Stash::inflate(int increase) {
  assert(increase > 0);
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copie l'ancien dans le nouveau
  delete []storage; // Ancienne espace
  storage = b; // Pointe vers le nouvel espace mémoire
  quantity = newQuantity;
}

void Stash::cleanup() {
  if(storage != 0) {
    cout << "freeing storage" << endl;
    delete []storage;
  }
} ///:~
```

Il y a plusieurs autres points qui diffèrent entre C et C++. Tout d'abord, les déclarations dans les fichiers d'en-tête sont *requises*par le compilateur. En C++, vous ne pouvez pas appeler une fonction sans la déclarer d'abord. Faute de quoi, le compilateur vous renverra un message d'erreur. C'est une manière importante de s'assurer que les appels de fonctions sont cohérents entre l'endroit où elles sont appelées et l'endroit où elles sont définies. En vous forçant à déclarer une fonction avant de l'appeler, le compilateur C++ s'assure virtuellement que vous réaliserez cette déclaration en incluant le fichier d'en-tête. Si vous incluez également le même fichier d'en-tête à l'endroit où les fonctions sont définies, alors le compilateur vérifie que la déclaration dans l'en-tête et dans la définition de la fonction correspondent. Cela signifie que le fichier d'en-tête devient un dépôt validé de déclarations de fonctions et assure que ces fonctions seront utilisées d'une manière cohérente dans toutes les unités de traduction du projet.

Bien entendu, les fonctions globales peuvent toujours être déclarées à la main à chaque endroit où elles sont définies et utilisées. (C'est si ennuyeux à réaliser que cette manière de faire est très improbable.) Toutefois, les structures doivent toujours être déclarées avant qu'elles soient définies ou utilisées, et l'endroit le plus approprié pour y placer la définition d'une structure, c'est dans un fichier d'en-tête, à l'exception de celles que vous masquer intentionnellement dans un fichier.

Vous pouvez observer que toutes les fonctions membres ressemblent à des fonctions C, à l'exception de la résolution de portée et du fait que le premier argument issu de la version C de la bibliothèque n'apparaît plus de façon explicite. Il est toujours là, bien sûr, parce que la fonction doit être en mesure de travailler sur une variable **struct**particulière. Mais notez qu'à l'intérieur d'une fonction membre, la sélection du membre a également disparu! Ainsi, à la place d'écrire **s->size = sz;**vous écrivez **size = sz;**et éliminez le **s->**ennuyeux, qui n'ajoutait de toute manière véritablement rien au sens de ce que vous vouliez faire. Le compilateur C++ fait cela à votre place. En fait, il utilise le premier argument "secret" (l'adresse de la structure que nous passions auparavant à la main) et lui applique le sélecteur de membre à chaque fois que vous faites référence à une donnée membre de cette structure. Cela signifie qu'à chaque fois que vous vous trouvez à l'intérieur d'une fonction membre d'une autre structure, vous pouvez faire référence à n'importe quel membre (inclus une autre fonction membre) en utilisant simplement son nom. Le compilateur recherchera parmi les noms locaux à la structure avant de rechercher une version globale du

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

même nom. Vous vous rendrez compte que cette fonctionnalité signifie que votre code sera non seulement plus facile à écrire, il sera aussi beaucoup plus facile à lire.

Mais que ce passe-t'il si, pour une raison quelconque, vous désirez manipuler l'adresse de la structure? Dans la version C de la bibliothèque, c'était facile, car le premier argument de chaque fonction était un pointeur de type **CStash***appelé **s**. En C++, les choses sont encore plus cohérentes. Il y a un mot clé spécial, appelé **this**, qui produit l'adresse de la structure. C'est l'équivalent de **s**dans la version C de la bibliothèque. Ainsi, on peut retrouver le style utilisé en C en écrivant:

```
                    this->size = Size;
```

Le code généré par le compilateur est exactement le même, si bien que vous n'avez pas besoin d'utiliser **this**de cette manière; occasionnellement, vous verrez du code où les gens utilisent **this->**partout de façon explicite, mais cela n'ajoute rien à la signification du code et c'est souvent révélateur d'un programmeur inexpérimenté. Habituellement, vous n'utilisez pas souvent **this**, mais lorsque vous en avez besoin, il est là (certains des exemples que vous rencontrerez plus loin dans le livre utilisent **this**).

Il reste un dernier point à mentionner. En C, vous pouvez affecter un pointeur de type **void***à n'importe quel autre pointeur de la façon suivante:

```
                    int i = 10;
void* vp = &i; // OK aussi bien en C qu'en C++
int* ip = vp; // Acceptable uniquement en C
```

et il n'y avait aucune plainte de la part du compilateur. Mais en C++, cette instruction n'est pas autorisée. Pourquoi? Parce que C n'est pas aussi précis au sujet de l'information de type, ainsi il vous autorise à affecter un pointeur avec un type non spécifié à un pointeur avec un type spécifié. Rien de cela avec C++. Le typage est une chose critique en C++, et le compilateur sort ses griffes lorsqu'il aperçoit des violations au sujet de l'information de type. Ca a toujours été important, mais ça l'est spécialement en C++, parce que vous avez des fonctions membres à l'intérieur des structures. Si vous pouviez passer des pointeurs de **struct**n'importe comment en toute impunité en C++, il est possible que vous finissiez par appeler une fonction membre pour une structure qui n'existe même pas pour la structure effectivement traitée! Une voie directe vers le désastre. Par conséquent, tandis que C++ autorise l'affectation de n'importe quel type de pointeur à un **void***(c'était la raison d'être originelle de **void***, qui a la contrainte d'être suffisamment grand pour contenir un pointeur de n'importe quel type), il ne vous permettra pas d'affecter un pointeur **void**à n'importe quel autre type de pointeur. Une conversion est toujours nécessaire pour avertir le lecteur et le compilateur que vous voulez véritablement le traîter comme le type de destination.

Ce point soulève un aspect intéressant. Un des objectifs importants de C++ est de compiler autant de code C existant que possible afin de permettre une transition aisée vers ce nouveau langage. Malgré tout, cela ne signifie pas que n'importe quel code autorisé en C sera automatiquement accepté en C++. Il y a de nombreuses choses qu'un compilateur C laisse passer qui sont dangereuses et susceptibles d'entraîner des erreurs. (Nous les étudierons au fur et à mesure que le livre progresse.) Le compilateur C++ génère des avertissements et des erreurs dans ces situations. Il s'agit là souvent plus d'un avantage que d'un obstacle. En fait, il existe de nombreuses situations où vous essayez de traquer une erreur en C et ne parvenez pas à la trouver, mais aussitôt que vous recompilez le programme en C++, le compilateur montre le problème du doigt! En C, vous vous rendrez souvent compte que vous pouvez amener le programme à compiler, mais que la prochaine étape est de le faire fonctionner correctement. En C++, lorsque le programme compile comme il le doit, souvent en plus, il fonctionne! C'est parce que le langage est beaucoup plus strict avec les types.

Vous pouvez voir un certain nombre de nouvelles choses dans la façon dont la version C++ de **Stash**est utilisée dans le programme de test suivant:

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                    //: C04:CppLibTest.cpp
//{L} CppLib
// Test de la bibliothèque C++
#include "CppLib.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash;
  intStash.initialize(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  // Contient des chaînes de 80 caractères
  Stash stringStash;
  const int bufsize = 80;
  stringStash.initialize(sizeof(char) * bufsize);
  ifstream in("CppLibTest.cpp");
  assure(in, "CppLibTest.cpp");
  string line;
  while(getline(in, line))
    stringStash.add(line.c_str());
  int k = 0;
  char* cp;
  while((cp =(char*)stringStash.fetch(k++)) != 0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
  intStash.cleanup();
  stringStash.cleanup();
} ///:~
```

Une chose que vous noterez, c'est que les variables sont toutes définies "à la volée" (comme introduit au chapitre précédent). C'est à dire qu'elles sont définies à n'importe quel endroit au sein d'un bloc, au lieu d'être contraintes - comme en C - de l'être au début du bloc.

Le code est relativement similaire à **CLibTest.cpp**, mais lorsqu'une fonction membre est appelée, l'appel a lieu en utilisant l'opérateur de sélection de membre ' **.**' précédé par le nom de la variable. C'est un syntaxe pratique parce qu'elle imite la sélection d'un membre donnée de la structure. La différence est qu'il s'agit là d'une fonction membre, et qu'elle possède une liste d'arguments.

Bien entendu, l'appel que le compilateur génère effectivement ressemble beaucoup plus à la fonction originale de la bibliothèque C. Ainsi, en considérant la décoration du nom et le passage de **this**, l'appel de fonction C++ **inStash.initialize(sizeof(int), 100)**devient quelque chose comme **Stash_initialize(&intStash, sizeof(int), 100)**.Si vous vous demandez un jour ce qui se passe sous le capot, souvenez-vous que **cfront**, le compilateur C++ original de AT&T, produisait en sortie du code C qui était alors compilé par un compilateur C sous-jacent. Cette approche signifiait que **cfront**pouvait être rapidement porté sur n'importe quelle machine possédant un compilateur C, et cela a contribué à favoriser la dissémination rapide de la technologie du compilateur C++. Mais, parce que le compilateur C++ devait générer du C, vous savez qu'il doit être possible, d'une manière ou d'une autre, de représenter la syntaxe C++ en C (certains compilateurs vous permettent encore de produire du code C).

Il ya un autre changement par rapport à **CLibTest.cpp**qui consiste en l'introduction du fichier d'en-tête **require.h**. Il s'agit d'un fichier d'en-tête que j'ai créé pour ce livre afin d'effectuer une vérification d'erreur plus sophistiquée que celle fournie par **assert()**. Il contient plusieurs fonctions dont celle utilisée ici et appelée **assure()**, qui est utilisée pour les fichiers. Cette fonction contrôle que le fichier a été ouvert avec succès, et dans le cas contraire, affiche sur le flux d'erreur standard que le fichier n'a pu être ouvert (elle a donc besoin du nom du fichier en second argument) et quitte le programme. Les fonctions de **require.h**seront utilisées tout au long de ce livre, en particulier pour nous assurer que la ligne de commande comporte le bon nombre d'arguments et que les fichiers sont ouverts proprement. Les fonctions de **require.h**remplacent le code de vérification d'erreur répétitif et qui consitute une

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

distraction, et fournissent essentiellement des messages d'erreur utiles. Ces fonctions seront expliquées de façon de complète plus loin dans le livre.

4.4 - Qu'est-ce qu'un objet?

Maintenant que vous avez vu un premier exemple, il est temps de faire marche arrière et de jeter un oeil à la terminologie. Le fait d'apporter des fonctions dans une structure est la base de ce que le C++ apporte au C, et cela introduit une nouvelle façon de penser les structures: comme des concepts. En C, une **struct**est une agglomération de données, un moyen d'empaqueter les données de manière à ce que vous puissiez les traiter dans un bloc. Mais il est difficile d'y penser autrement que comme une commodité de programmation. Les fonctions qui agissent sur ces structures sont ailleurs. Cependant, avec les fonctions dans le paquetage, la structure devient une nouvelle créature, capable de décrire à la fois des caractéristiques (comme le fait une **struct**C) *et*des comportements. Le concept de l'objet, une entité libre et bornée qui peut se souvenir *et*agir, se suggère de lui-même.

En C++, un objet est simplement une variable, et la plus pure définition est “une zone de stockage” (c'est un moyen plus spécifique de dire, “un objet doit avoir un identifiant unique”, qui dans le cas du C++ est une adresse mémoire unique). C'est un emplacement dans lequel vous pouvez stocker des données, et qui implique qu'il y a également des opérations qui peuvent être effectuées sur ces données.

Malheureusement, il n'y a pas complète uniformité entre les langages sur ces termes, bien qu'ils soient assez bien acceptés. Vous rencontrerez parfois des désaccords sur ce qu'est un langage orienté objet, bien que cela semble raisonnablement bien défini maintenant. Il y a des langages qui sont *à base d'objets*, ce qui signifie qu'il y a des objets comme les structures avec fonctions du C++ que vous avez vu jusqu'à présent. Ce n'est cependant qu'une partie de la condition nécessaire à un langage orienté objet, et les langages qui s'arrêtent à l'empaquetage des fonctions dans les structures de données sont à base d'objets, et non orientés objet.

4.5 - Typage de données abstraites

La capacité d'empaqueter des données avec des fonctions vous permet de créer de nouveaux types de données. Cela est généralement appelé *encapsulation* Ce terme peut prêter à polémique. Certains l'utilisent défini ainsi; d'autres l'utilisent pour décrire le contrôle d'accès, dont il est question dans le chapitre suivant.. Un type de donnée existant peut avoir plusieurs morceaux de données empaquetées ensemble. Par exemple, un **float**a un exposant, une mantisse, et un bit de signe. Vous pouvez lui dire de faire des choses: l'ajouter à un autre **float**ou à un **int**, et ainsi de suite. Il a des caractéristiques et un comportement.

La définition de **Stash**crée un nouveau type de données. Vous pouvez ajouter (**add( )**), chercher (**fetch( )**), et gonfler (**inflate( )**). Vous en créez un en disant **Stash s**, tout comme vous créez un **float**en disant **float f**. Un **Stash**a aussi des caractéristiques et un comportement. Bien qu'il se comporte comme un type réel, intégré, nous nous y référons comme à un *type de données abstrait*, peut-être parce qu'il nous permet de abstraire un concept de l'espace des problèmes vers l'espace des solutions. En plus, le compilateur C++ le traite comme un nouveau type de données, et si vous dites qu'une fonction attend un **Stash**, le compilateur s'assurera que vous passiez un **Stash**à cette fonction. Ainsi le même niveau de vérification de type se produit avec les types de données abstraits (parfois appelés *types définis par l'utilisateur*) qu'avec les types intégrés.

Vous pouvez immédiatement voir la différence, cependant, à la façon dont vous effectuez des opérations sur les objets. Vous dites **object.memberFunction(arglist)**. C'est “l'appel d'une fonction membre pour un objet.” Mais dans le jargon orienté objet, c'est également mentionné comme “l'envoi d'un message à un objet.” Pour un **Stash s**, l'instruction **s.add(&i)**envoie à **s**un message disant, “ **ajoute**toi ceci.” En réalité, la programmation orientée objet peut se résumer en une seule phrase: *envoyer des messages à des objets*. C'est vraiment tout ce que vous faites – créer un groupe d'objets et leur envoyer des messages. L'astuce, bien sûr, est de vous représenter ce que *sont*vos objets et vos messages , mais une fois que vous l'avez fait, l'implémentation en C++ est étonnamment

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

simple.

4.6 - Détails sur les objest

Une question qui revient souvent dans les séminaires est “Quelle est la taille d'un objet, et à quoi ressemble-t-il ?” La réponse dépend “ce que vous voulez faire d'un **struct**C.” En fait, le code que le compilateur C produit pour un **struct**C (avec aucun ornement C++) est souvent *exactement*le même que celui produit par un compilateur C++. C'est rassurant pour ces programmeurs C qui se reposent sur les détails de taille et de disposition de leur code, et qui,pour certaines raisons, accédent directement aux octets de la structure au lieu d'employer des identifiants (compter sur une taille et une disposition particulières pour une structure est une activité non portable).

La taille d'une structure est la taille combinée de tout ses membres. Quelquefois quand le compilateur génère un **struct**, il ajoute des octets supplémentaires pour faire ressortir nettement les frontières – ceci peut augmenter l'efficacité de l'exécution. Dans le Chapitre 15, vous verrez comment dans certains cas des pointeurs “secrets” sont ajoutés à la structure, mais vous n'avez pas besoin de vous en soucier pour l'instant.

Vous pouvez déterminer la taille d'un **struct**en utilisant l'opérateur **sizeof**. Voici un petit exemple:

```
                    //: C04:Sizeof.cpp
// Taille des structures
#include "CLib.h"
#include "CppLib.h"
#include <iostream>
using namespace std;

struct A {
  int   i[100];
};

struct B {
  void f();
};

void B::f() {}

int main() {
  cout << "sizeof struct A = " << sizeof(A) << " bytes" << endl;
  cout << "sizeof struct B = " << sizeof(B) << " bytes" << endl;
  cout << "sizeof CStash in C = " << sizeof(CStash) << " bytes" << endl;
  cout << "sizeof Stash in C++ = " << sizeof(Stash) << " bytes" << endl;
} ///:~
```

Sur ma machine (vos résultats peuvent varier) le premier rapport d'impression donne 200 parce que chaque **int**occupe deux octets. **struct B**est une espèce d'anomalie parce que c'est un **struct**sans données membres. En C c'est illégal, mais en C++ nous avons besoin de pouvoir créer une structure dont la tache est d'étendre les noms de fonctions, et c'est donc autorisé. Dans tous les cas, le résultat produit par le deuxième rapport d'impression est une valeur non nulle un peu étonnante. Dans les premières versions du langage, la taille était zéro, mais une situation maladroite surgit quand vous créez de tels objets: Ils ont la même adresse que l'objet créé directement après eux, ils sont donc indistincts. Une des règles fondamentales des objets est que chaque objet a une adresse unique, ainsi les structures sans données membres ont toujours une taille minimale non nulle.

Les deux derniers **sizeof**vous montrent que la taille de la structure en C++ est la même que la taille de la version équivalente en C. Le C++ essaie de ne pas ajouter de suppléments inutiles.

4.7 - L'étiquette d'un fichier d'en-tête

Lorsque vous créez une structure contenant des fonctions membres, vous êtes en train de créer un nouveau type de donnée. En général, vous voulez que ce type soit facilement accessible à vous-même et aux autres. Par

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

ailleurs, vous désirez séparer l'interface (la déclaration) de l'implémentation (la définition des fonctions membres) de manière à ce que l'implémentation puisse être modifiée sans forcer une re-compilation du système entier. Vous y parvenez en plaçant les déclarations concernant votre nouveau type dans un fichier d'en-tête.

Lorsque j'ai d'abord appris à programmer en C, le fichier d'en-tête était un mystère pour moi. Beaucoup d'ouvrages sur le C ne semblent pas mettre l'accent dessus, et le compilateur n'imposait pas les déclarations de fonction, de telle manière que j'avais la plupart du temps l'impression que c'était optionnel, sauf quand des structures étaient déclarées. En C++, l'usage de fichiers d'en-tête devient clair comme de l'eau de roche. Ils sont obligatoires pour un développement de programme facile, et on y place des informations très spécifiques: les déclarations. Le fichier d'en-tête informe le compilateur de ce qui est disponible dans votre bibliothèque. Vous êtes en mesure d'utiliser la bibliothèque même si vous ne possédez que le fichier d'en-tête, ainsi que le fichier objet ou le fichier de bibliothèque. Vous n'avez pas besoin du code source du fichier **cpp**. Le fichier d'en-tête est l'endroit où est sauvegardé la spécification de l'interface.

Bien que ce ne soit pas imposé par le compilateur, la meilleure approche pour construire de grands projets en C est d'utiliser des bibliothèques; collecter des fonctions associées dans un même module objet ou bibliothèque, et utiliser un fichier d'en-tête pour contenir toutes les déclarations de fonctions. Cette pratique est de rigueur en C++. Vous pouviez placer n'importe quelle fonction dans une bibliothèque C, mais le type abstrait de donnée du C++ détermine les fonctions associées par leur accès commun aux données d'une même structure. N'importe quelle fonction membre doit être déclarée dans une déclaration de structure. Vous ne pouvez pas le faire ailleurs. L'usage de bibliothèques de fonctions était encouragé en C, mais institutionnalisé en C++.

4.7.1 - L'importance des fichiers d'en-tête

Lorsque vous utilisez une fonction d'une bibliothèque, le langage C vous autorise à ignorer le fichier d'en-tête et à déclarer simplement les fonctions à la main. Dans le passé, certaines personnes procédaient ainsi afin d'accélérer un peu le travail du compilateur en lui épargnant la tâche d'ouvrir et d'inclure le fichier (ce n'est généralement pas un sujet de préoccupation avec les compilateurs modernes). Par exemple, voici une déclaration extrêmement nonchalante de la fonction C **printf( )**(de **<stdio.h>**):

```
printf(...);
```

Les ellipses spécifient une liste variable d'arguments Pour écrire la définition d'une fonction qui reçoit une liste variable d'arguments, vous devez utiliser varargs, bien que cette pratique doive être évitée en C++. Vous pouvez trouver des détails au sujet de varargs dans votre manuel C, ce qui signifie: **printf( )**reçoit certains arguments, chacun d'eux a un type, mais ignore cela. Prend tous les arguments que tu rencontres et accepte-les. En utilisant ce type de déclaration, vous mettez en veilleuse tout le système de vérification d'erreur sur les arguments.

Cette pratique peut entraîner des problèmes subtils. Si vous déclarer des fonctions à la main, dans un fichier, il est possible que vous fassiez une erreur. Puisque le compilateur ne voit dans ce fichier que la déclaration que vous avez faite à la main, il est capable de s'adapter à votre erreur. Ainsi, le programme se comportera correctement à l'édition des liens, mais l'usage de cette fonction dans le fichier en question sera erroné. C'est une erreur difficile à démasquer, et il est facile de l'éviter en utilisant un fichier d'en-tête.

Si vous placez toutes vos déclarations de fonctions dans un fichier d'en-tête, et que vous incluez ce fichier partout où vous utilisez la fonction, ainsi qu'à l'endroit où vous définissez la fonction, vous vous assurez d'une déclaration cohérente sur l'ensemble du système. Vous vous assurez également que la déclaration et la définition correspondent en incluant l'en-tête dans le fichier de définition.

Si une structure est déclarée dans un fichier d'en-tête en C++, vous *devez*inclure ce fichier d'en-tête partout où la structure en question est utilisée, et à l'endroit où sont définies les fonctions membres de cette structure. Le compilateur C++ retournera une erreur si vous essayez d'appeler une fonction régulière, ou d'appeler ou de définir

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

une fonction membre, sans la déclarer auparavant. En forçant l'usage correct des fichiers d'en-tête, le langage assure la cohérence au sein des bibliothèques, et réduit le nombre de bugs en imposant l'utilisation de la même interface partout.

L'en-tête est un contrat entre vous et l'utilisateur de votre bibliothèque. Ce contrat décrit vos structures de données, les états des arguments et valeurs de retour pour l'appel des fonctions. Il dit: "Voici ce que ma bibliothèque fait." L'utilisateur a besoin de certaines de ces informations pour développer l'application et le compilateur a besoin de toutes les informations pour générer du code propre. L'utilisateur de la structure inclut simplement le fichier d'en-tête, crée des objets (instances) de cette structure, et lie avec le module objet ou la bibliothèque (c-à-d: le code compilé).

Le compilateur impose ce contrat en exigeant que vous déclariez toutes les structures et fonctions avant qu'elles ne soient utilisées et, dans le cas des fonctions membres, avant qu'elles ne soient définies. Ainsi, vous êtes forcés de placer les déclarations dans un fichier d'en-tête et d'inclure cet en-tête dans le fichier où les fonctions membres sont définies, et dans le(s) fichier(s) où elles sont utilisées. Parce qu'un fichier d'en-tête unique décrivant votre bibliothèque est inclus dans tout le système, le compilateur peut garantir la cohérence et éviter les erreurs.

Il y a certains enjeux que vous devez avoir à l'esprit pour organiser votre code proprement et écrire des fichiers d'en-tête efficaces. Le premier de ces enjeux concerne ce que vous pouvez mettre dans des fichiers d'en-tête. La règle de base est "uniquement des déclarations", c'est-à-dire seulement des informations destinées au compilateur, mais rien qui alloue de la mémoire en générant du code ou en créant des variables. La raison de cette limitation vient du fait qu'un fichier d'en-tête sera typiquement inclus dans plusieurs unités de compilation au sein d'un projet, et si de la mémoire est allouée pour un identifiant à plus d'un endroit, l'éditeur de liens retournera une erreur de définition multiple (il s'agit de la règle de la définition unique du C++: vous pouvez déclarer les choses autant de fois que vous voulez, mais il ne peut y avoir qu'une seule définition pour chaque chose).

Cette règle n'est pas complètement rigide. Si vous définissez une variable "statique" (dont la visibilité est limitée au fichier) dans un fichier d'en-tête, il y aurait de multiples instances de cette donnée à travers le projet, mais l'éditeur de liens ne subira aucune collision. En C++ standard, le mot clé static destiné à limiter la portée au fichier est une fonctionnalité dépréciée.. De manière générale, vous ne ferez rien dans un fichier d'en-tête qui entraînera une ambiguïté à l'édition des liens.

4.7.2 - Le problème des déclarations multiples

Le deuxième enjeu relatif aux fichiers d'en-tête est le suivant: lorsque vous placez une déclaration de structure dans un fichier d'en-tête, il est possible que ce fichier soit inclus plus d'une fois dans un programme compliqué. Les flux d'entrées/sorties sont de bons exemples. A chaque fois qu'une structure fait des entrées/sorties, elle inclut un des fichiers d'en-tête iostream. Si le fichier **cpp**, sur lequel vous êtes en train de travailler, utilise plus qu'une sorte de structure (typiquement en incluant un fichier d'en-tête pour chacune d'elles), vous courez le risque d'inclure l'en-tête **<iostream>**plus d'une fois et de re-déclarer des flux d'entrées/sorties.

Le compilateur considère la redéclaration d'une structure (déclarée à l'aide du mot clé **struct**ou **class**) comme une erreur, puisque, dans le cas contraire, cela reviendrait à autoriser l'utilisation d'un même nom pour différents types. Afin d'éviter cette erreur lorsque de multiples fichiers d'en-tête sont inclus, vous avez besoin de doter vos fichiers d'en-tête d'une certaine intelligence en utilisant le préprocesseur (les fichiers d'en-tête standards du C++, comme **<iostream>**possèdent déjà cette "intelligence").

Aussi bien C que C++ vous autorisent à redéclarer une fonction, du moment que les deux déclarations correspondent, mais aucun des deux n'autorise la redéclaration d'une structure. En C++, cette règle est particulièrement importante, parce que si le compilateur vous autorisait à redéclarer une structure et que les deux déclarations différaient, laquelle des deux utiliserait-il?

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Le problème de la redéclaration est d'autant plus important en C++, parce que chaque type de donnée (structure avec des fonctions) possède en général son propre fichier d'en-tête, et vous devez inclure un en-tête dans l'autre si vous voulez créer un autre type de donnée qui utilise le premier. Dans chaque fichier **cpp**de votre projet, il est probable que vous allez inclure plusieurs fichiers qui eux-mêmes incluent le même fichier d'en-tête. Au cours d'un processus de compilation donné, le compilateur est en mesure de rencontrer le même fichier d'en-tête à plusieurs reprises. A moins que vous fassiez quelque chose contre cela, le compilateur va voir la redéclaration de votre structure et reporter une erreur à la compilation. Afin de résoudre le problème, vous avez besoin d'en savoir un peu plus au sujet du préprocesseur.

4.7.3 - Les directives #define, #ifdef et #endif du préprocesseur

La directive du préprocesseur #define peut être utilisée afin de créer des symboles à la compilation. Vous avez deux possibilités: vous pouvez simplement dire au préprocesseur que le symbole est défini, sans spécifier de valeur:

```
#define FLAG
```

ou alors vous pouvez lui donner une valeur (ce qui est la manière typique en C de définir une constante):

```
#define PI 3.14159
```

Dans chacun de cas, l'étiquette peut maintenant être testée par le préprocesseur afin de voir si elle est définie:

```
#ifdef FLAG
```

La valeur vrai sera retournée, et le code qui suit le **#ifdef**sera inclus dans le paquetage envoyé au compilateur. Cette inclusion s'arrête lorsque le préprocesseur rencontre l'instruction

```
#endif
```

ou

```
#endif // FLAG
```

Toute autre chose qu'un commentaire sur la même ligne, à la suite du **#endif**est illégal, même si certains compilateurs l'acceptent. La paire **#ifdef**/ **#endif**peut être imbriquée.

Le complément de **#define**est **#undef**(abréviation pour "un-define"), qui fera qu'une instruction **#ifdef**utilisant la même variable retournera le résultat faux. **#undef**entraînera également l'arrêt de l'usage d'une macro par le préprocesseur. Le complément de **#ifdef**est **#ifndef**, qui retourne vrai si l'étiquette n'a pas été définie (C'est l'instruction que nous allons utiliser pour les fichiers d'en-tête).

Il y a d'autres fonctionnalités utiles dans le préprocesseur du C. Vous devriez consulter votre documentation locale pour un tour d'horizon complet.

4.7.4 - Un standard pour les fichiers d'en-tête

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Dans chaque fichier d'en-tête qui contient une structure, vous devriez d'abord vérifier si l'en-tête a déjà été inclus dans le fichier **cpp**en question. Vous accomplissez cela en testant la définition d'un symbole du préprocesseur. Si le symbole n'est pas défini, le fichier n'a pas été inclus, vous devriez alors définir ce symbole (de telle manière que la structure ne puisse être redéclarée) puis déclarer la structure. Si le symbole a été défini, alors ce type a déjà été déclaré, et vous deviez simplement ignorer le code qui le déclare à nouveau. Voici à quoi devrait ressembler le fichier d'en-tête:

```
#ifndef HEADER_FLAG
#define HEADER_FLAG
// Ici vient la déclaration du type...
#endif // HEADER_FLAG
```

Comme vous pouvez le voir, la première fois que le fichier d'en-tête est inclus, le contenu de ce fichier (y compris votre déclaration de type) sera inclus par le préprocesseur. Toute inclusion subséquente dans une unité de compilation donnée verra la déclaration du type ignorée. Le nom HEADER_FLAG peut être n'importe quel nom unique, mais un standard fiable est de mettre le nom du fichier d'en-tête en lettres majuscules et de remplacer les points par des caractères de soulignement (les caractères de soulignement en tête du nom sont toutefois réservé aux noms du système). Voici un exemple:

```
                        //: C04:Simple.h
// Simple header that prevents re-definition
#ifndef SIMPLE_H
#define SIMPLE_H

struct Simple {
  int i,j,k;
  initialize() { i = j = k = 0; }
};
#endif // SIMPLE_H ///:~
```

Bien que le **SIMPLE_H**, après le **#endif**, soit commenté et ainsi ignoré du préprocesseur, il est utile à des fins de documentation.

Ces instructions du préprocesseur qui permettent de prévenir l'inclusion multiple sont souvent appelées des *gardes d'inclusion*.

4.7.5 - Les espaces de nommage dans les fichiers d'en-tête

Vous noterez que des *directives using*sont présentes dans presque tous les fichier **cpp**de cet ouvrage, habituellement sous la forme:

```
                        using namespace std;
```

Puisque **std**est l'espace de nommage qui entoure l'ensemble de la bibliothèque standard du C++, cette instruction using autorise les noms de la bibliothèques standard du C++ à être utilisés sans qualification. Toutefois, vous ne verrez pratiquement jamais une directive using dans un fichier d'en-tête (du moins en dehors de toute portée). La raison de ce fait est que la directive using élimine la protection de cet espace de nommage, et ce jusqu'à la fin de l'unité de compilation courante. Si vous placez une directive using (en dehors de toute portée) dans un fichier d'en-tête, cela signifie que cette perte de "protection de l'espace de nommage" sera effective dans tout fichier incluant l'en-tête en question, souvent d'autres fichiers d'en-tête. Par conséquent, si vous commencez à placer des directives using dans les fichiers d'en-tête, il est très facile de finir par éliminer les espaces de nommage partout, et ainsi de neutraliser les effets bénéfiques apportés par ces espaces de nommage.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

En résumé, ne placez pas de directives using dans des fichiers d'en-tête.

4.7.6 - Utiliser des fichiers d'en-tête dans des projets

Lors de la construction d'un projet en C++, vous le créerez habituellement par rassemblement d'un grand nombre de types différents (structures de données avec fonctions associées). Vous placerez habituellement la déclaration pour chaque type ou pour un groupe de types associés dans des fichiers d'en-tête séparés, puis vous définirez les fonctions relatives à ce type dans une unité de traduction. Lorsque vous utilisez ce type, vous devrez inclure le fichier d'en-tête afin d'effectuer les déclarations proprement.

Parfois, cette façon de faire sera respectée dans ce livre, mais la plupart du temps les exemples seront très simples, de telle manière que tout – les déclaration de structures, les définitions de fonctions et la fonction **main( )**– peut se trouver dans un fichier unique. Toutefois, gardez à l'esprit qu'en pratique, vous utiliserez de préférence des fichiers séparés, ainsi que des fichiers d'en-tête.

4.8 - Structures imbriquées

La commodité de sortir les noms de données et de fonctions de l'espace de nom global s'étend aux structures. Vous pouvez imbriquer une structure dans une autre, ce qui conserve les éléments associés ensemble. La syntaxe de la déclaration est celle à laquelle on peut s'attendre, comme vous pouvez le voir dans la structure suivante, qui implémente une pile classique au moyen d'une liste simplement chaînée de manière à ce qu'elle ne manque "jamais" de mémoire:

```
                    //: C04:Stack.h
// struct imbriquée dans une liste chaînée
#ifndef STACK_H
#define STACK_H

struct Stack {
  struct Link {
    void* data;
    Link* next;
    void initialize(void* dat, Link* nxt);
  }* head;
  void initialize();
  void push(void* dat);
  void* peek();
  void* pop();
  void cleanup();
};
#endif // STACK_H ///:~
```

Le **struct**imbriqué s'appelle **Link**, et il contient un pointeur sur le prochain **Link**dans la liste ainsi qu'un pointeur sur la donnée stockée dans le **Link**. Si le pointeur **next**vaut zéro, cela signifie que vous êtes à la fin de la liste.

Notez que le pointeur **head**est défini juste après la déclaration du **struct Link**, au lieu d'une définition séparée **Link* head**. C'est une syntaxe issue du C, mais cela souligne l'importance du point-virgule après la déclaration de structure; le point-virgule indique la fin de la liste des définitions pour ce type de structure. Les divers éléments de cette liste de définitions sont séparés par une virgule. (Généralement la liste est vide.)

La structure imbriquée possède sa propre fonction **initialize( )**, comme toutes les structures vues précédemment, pour assurer son initialisation correcte. **Stack**possède les deux fonctions **initialize( )**et **cleanup( )**, ainsi que **push( )**, qui prend en paramètre un pointeur sur le **data**que vous voulez stocker (elle considère qu'il a été alloué sur le tas), et **pop( )**, qui retourne le pointeur de **data**se trouvant en haut de la pile avant de le supprimer du haut de la pile. (quand vous dépilez - **pop( )**- un élément, vous êtes responsable de la destruction de l'objet pointé par **data**.) La fonction **peek( )**retourne également le pointeur de **data**se trouvant en haut de la pile, mais elle conserve

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

cet élément en haut de la pile.

Voici les définitions des fonctions membres:

```
                    //: C04:Stack.cpp {O}
// Liste chaînée avec imbrication
#include "Stack.h"
#include "../require.h"
using namespace std;

void Stack::Link::initialize(void* dat, Link* nxt) {
  data = dat;
  next = nxt;
}

void Stack::initialize() { head = 0; }

void Stack::push(void* dat) {
  Link* newLink = new Link;
  newLink->initialize(dat, head);
  head = newLink;
}

void* Stack::peek() {
  require(head != 0, "Pile vide");
  return head->data;
}

void* Stack::pop() {  if(head == 0) return 0;
  void* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

void Stack::cleanup() {
  require(head == 0, "Pile non vide");
} ///:~
```

La première définition est particulièrement intéressante parce qu'elle vous montre comment définir un membre d'une structure imbriquée. Vous utilisez simplement un niveau supplémentaire de résolution de portée pour spécifier le nom du **struct**englobant. **Stack::Link::initialize( )**prend les arguments et les assigne à ses membres.

Stack::initialize( )met le pointeur **head**à zéro, ainsi l'objet sait que sa liste est vide.

Stack::push( )prend l'argument, qui est un pointeur sur la variable dont vous voulez conserver la trace, et l'ajoute sur le haut de la pile. Pour cela, la fonction commence par utiliser **new**pour allouer la mémoire pour le **Link**que l'on va insérer au dessus. Puis elle appelle la fonction **initialize( )**de **Link**pour assigner les valeurs appropriées aux membres de **Link**. Notez que le pointeur **next**est affecté au pointeur **head**courant; puis le pointeur **head**est affecté avec la valeur du nouveau pointeur **Link**. Cela pousse effectivement le **Link**en haut de la liste.

Stack::pop( )capture le pointeur **data**situé sur le haut de la pile; puis fait descendre le pointeur **head**et supprime l'ancien sommet de la pile, et retourne enfin le pointeur capturé. Quand **pop( )**supprime le dernier élément, alors le pointeur **head**vaut à nouveau zéro, ce qui signifie que la pile est vide.

Stack::cleanup( )ne fait en fait aucun nettoyage. Au lieu de cela, il établit une politique ferme qui est que “vous (le programmeur client qui utilise cet objet **Stack**) êtes responsable du dépilement de tous les éléments du **Stack**et de leur suppression.” **require( )**est utilisé pour indiquer qu'une erreur de programmation s'est produite si la pile n'est pas vide.

Pourquoi le destructeur de **Stack**ne pourrait-il pas être responsable de tous les objets que le programmeur client n'a pas dépilé ? Le problème est que **Stack**est en possession de pointeurs **void**, et vous apprendrez au Chapitre

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

13 qu'appeler **delete**sur un **void***ne nettoie pas les choses proprement. Le fait de savoir “qui est responsable de la mémoire” n'est pas *si*simple, comme nous le verrons dans les prochains chapitres.

Voici un exemple pour tester la pile **Stack**:

```
                    //: C04:StackTest.cpp
//{L} Stack
//{T} StackTest.cpp
// Test d'une liste chaînée imbriquée
#include "Stack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // Le nom du fichier est passé en argument
  ifstream in(argv[1]);
  assure(in,argv[1]);
  Stack textlines;
  textlines.initialize();
  string line;
  // Lit le fichier et stocke les lignes dans la pile:
    while(getline(in, line))
  textlines.push(new string(line));
  // Dépile les lignes de la pile et les affiche:
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << *s << endl;
    delete s;
  }
  textlines.cleanup();
} ///:~
```

Ceci est similaire à l'exemple précédent, mais on empile des lignes d'un fichier (sous forme de pointeur de **string**) sur le **Stack**puis on les dépile, ce qui provoque un affichage inversé du fichier à l'écran. Notez que la fonction membre **pop( )**retourne un **void***et que celui-ci doit être casté en **string***avant de pouvoir être utilisé. Pour afficher le **string**à l'écran, le pointeur est déférencé.

Comme **textlines**est rempli, le contenu de **line**est “cloné” pour chaque **push( )**en faisant un **new string(line)**. La valeur retournée par l'expression new est un pointeur sur un nouveau **string**qui a été créé et qui a copié l'information dans **line**. Si vous aviez simplement passé l'adresse de **line**à **push()**, vous auriez obtenu un **Stack**rempli d'adresses identiques, pointant toutes vers **line**. Vous en apprendrez plus à propos de ce processus de “clonage” plus tard dans ce livre.

Le nom de fichier est récupéré depuis la ligne de commande. Pour s'assurer qu'il y ait assez d'arguments dans la ligne de commande, vous pouvez voir l'utilisation d'une deuxième fonction issue du fichier d'en-tête **require.h**: **requireArgs( )**, qui compare **argc**au nombre désiré d'arguments et affiche à l'écran un message d'erreur approprié avant de terminer le programme s'il n'y a pas assez d'arguments.

4.8.1 - Résolution de portée globale

L'opérateur de résolution de portée vous sort des situations dans lesquelles le nom que le compilateur choisit par défaut (le nom “le plus proche”) n'est pas celui que vous voulez. Par exemple, supposez que vous ayiez une structure avec un identificateur local **a**, et que vous vouliez sélectionner un identificateur global **a**depuis l'intérieur d'une fonction membre. Le compilateur va par défaut choisir celui qui est local, et donc vous êtes obligé de lui dire de faire autrement. Quand vous voulez spécifier un nom global en utilisant la résolution de portée, vous utilisez l'opérateur avec rien devant. Voici un exemple qui montre une résolution de portée globale à la fois pour une variable et une fonction :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                          //: C04:Scoperes.cpp
// Résolution de portée globale
int a;
void f() {}

struct S {
  int a;
  void f();
};

void S::f() {
  ::f();  // autrement il y aurait récurrence!
  ::a++;  // Sélectionne le a global
  a--;    // Le a dans la portée du struct
}

int main() { S s; f(); } ///:~
```

Sans la résolution de portée dans **S::f( )**, le compilateur aurait par défaut sélectionné les versions membres de **f( )**et **a**.

4.9 - Résumé

Dans ce chapitre, vous avez appris le “tournant” fondamental du C++: vous pouvez mettre des fonctions dans les structures. Ce nouveau type de structure est appelé un *type de données abstraites*, et les variables que vous créez en utilisant ces structures sont appelés *objets*, ou *instances*, de ce type. Appeler une fonction membre d'un objet est appelé *envoyer un message*à cet objet. L'action première en programmation orientée objet est d'envoyer des messages aux objets.

Bien qu'empaqueter les données et les fonctions ensembles apporte un bénéfice considérable à l'organisation du code et simplifie l'utilisation des bibliothèques parce que cela empêche les conflits de noms en les cachant, vous pouvez faire beaucoup plus pour programmer de façon plus sûre en C++. Dans le prochain chapitre, vous apprendrez comment protéger certains membres d'un **struct**pour que vous soyez le seul à pouvoir les manipuler. Cela établit une frontière claire entre ce que l'utilisateur de la structure peut changer et ce que seul le programmeur peut changer.

4.10 - Exercices

Les solutions de exercices suivants peuvent être trouvés dans le document électronique *The Thinking in C++ Annotated Solution Guide*, disponible à petit prix sur www.BruceEckel.com.

1 Au sein de la bibliothèque standard du langage C, la fonction **puts()**imprime un tableau de caractères sur la console (ainsi vous pouvez écrire **puts("hello")**). Ecrivez un programme C qui utilise **puts()**mais n'inclut pas < **stdio.h**> ou autrement dit déclarez la fonction. Compilez ce programme à l'aide de votre compilateur C. (Certains compilateurs C++ ne sont pas distincts de leur compilateur C; dans ce cas, vous devez rechercher une option à passer à la ligne de commande qui force une compilation C.) Maintenant, compilez-le avec un compilateur C++ et observez la différence.

2 Créez une déclaration de structure avec une fonction membre unique, puis créez une définition pour cette fonction membre. Créez une instance de votre nouveau type de donnée, et appelez la fonction membre.

3 Modifiez votre solution de l'exercice 2 de telle manière à déclarer la structure dans un fichier d'en-tête "protégé" de façon adéquate contre les inclusions multiples, avec la définition de la fonction dans un fichier **cpp**et votre **main()**dans un autre.

4 Créez une structure contenant un membre donnée unique de type **int**, et deux fonctions globales prenant chacune en argument un pointeur sur cette structure. La première fonction prend un second argument de type **int**et affecte la valeur de cet argument au membre **int**de la structure, la seconde affiche la valeur du membre **int**de cette structure. Testez ces fonctions.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

5 Répétez l'exercice 4 mais déplacez les fonctions de manière à ce qu'elles soient des fonctions membres de la structure, et testez à nouveau ces fonctions.
6 Créer une classe qui (de façon redondante) effectue la sélection d'un membre donnée ainsi que l'appel d'une fonction membre en utilisant le mot clé **this**(qui fait référence à l'adresse de l'objet courant).
7 Créer un **Stash**qui contient des **double**s. Remplissez-le avec 25 valeurs de type **double**, puis affichez-les sur la console.
8 Répétez l'exercice 7 avec **Stack**.
9 Créer un fichier contenant une fonction **f()**qui prend un argument de type **int**et l'affiche sur la console en utilisant la fonction **printf()**déclarée dans < **stdio.h**> en écrivant: **printf("%d\n", i)**où **i**est l'entier que vous désirez afficher. Créez un fichier séparé contenant **main()**, et dans ce fichier, déclarez **f()**comme prenant un argument de type **float**. Appelez **f()**depuis **main()**. Essayez de compiler et de lier votre programme à l'aide d'un compilateur C++ et observez ce qui se passe. Maintenant compilez et liez ce programme en utilisant un compilateur C, et regardez ce qui se passe lorsqu'il s'exécute. Expliquez les comportements observés.
10 Trouvez comment produire du code assembleur à l'aide de vos compilateurs C et C++. Ecrivez une fonction en C et une structure avec une fonction membre unique en C++. Générez les codes assembleur correspondants et recherchez les noms de fonctions qui sont produits par votre fonction C et votre fonction membre C++, de telle manière que vous puissiez voir quelle décoration de nom est mise en oeuvre par le compilateur.
11 Ecrivez un programme avec du code de compilation conditionnelle au sein de **main()**, de telle manière que lorsqu'une constante pré-processeur est définie, un message est affiché, alors qu'un autre message est affiché lorsqu'elle n'est pas définie. Compilez ce code en expérimentant avec un **#define**dans le programme, puis recherchez comment vous pouvez passer des définitions pré-processeur via la ligne de commande et expérimentez.
12 Ecrivez un programme qui utilise **assert()**avec un argument qui est toujours faux (zéro) pour voir ce qui se passe lorsque vous l'exécutez. Maintenant, compilez-le avec **#define NDEBUG**et exécutez-le à nouveau pour voir la différence.
13 Créez un type abstrait de donnée qui représente une cassette vidéo dans un étalage de location de vidéos. Essayez de considérer toutes les données et opérations qui peuvent être nécessaire au type **Video**pour se comporter de manière adéquate au sein du système de gestion de location de vidéos. Incluez une fonction membre **print()**qui affiche les informations concernant la **Video**.
14 Créer un objet **Stack**pour contenir les objets **Video**de l'exercice 13. Créez plusieurs objets **Video**, stockez-les dans l'objet **Stack**, et affichez-les en utilisant **Video::print()**.
15 Ecrivez un programme qui affiche toutes les tailles des types de données fondamentaux sur votre ordinateur en utilisant **sizeof**.
16 Modifiez **Stash**de manière à utiliser un **vector<char>**comme structure de donnée sous-jacente.
17 Créez dynamiquement des emplacements mémoires pour les types suivants, en utilisant **new**: **int**, **long**, un tableau de 100 **char**s, un tableau de 100 **float**s. Affichez leurs adresses et puis libérez les espaces alloués à l'aide de **delete**.
18 Ecrivez une fonction qui prend un **char***en argument. En utilisant **new**, allouez dynamiquement un tableau de **char**s qui est de la taille du tableau de **char**s passé à la fonction. En utilisant l'itération sur les indices d'un tableau, copiez les caractères du tableau passé en argument vers celui alloué dynamiquement (n'oubliez pas le marqueur de fin de chaîne null) et retournez le pointeur sur la copie. Dans votre fonction **main()**, testez la fonction en y passant une constante chaîne de caractères statique entre guillemets, puis récupérez le résultat et passez-le à son tour à la fonction. Affichez les deux chaînes de caractères et les deux pointeurs de manière à vous rendre compte qu'ils correspondent à des emplacements mémoire différents. A l'aide de **delete**, nettoyez tout l'espace alloué dynamiquement.
19 Montrez un exemple d'une structure déclarée à l'intérieur d'une autre structure (une structure imbriquée). Déclarez des membres données dans chacunes des structures, et déclarez et définissez des fonctions membres dans chacunes des structures. Ecrivez une fonction **main()**qui teste vos nouveaux types.
20 Quelle est la taille d'une structure? Ecrivez un morceau de code qui affiche la taille de différentes structures. Créez des structures qui comportent seulement des données membres et d'autres qui ont des données membres ainsi que des fonctions membres. Puis, créez une structure qui n'a aucun membre du tout. Affichez toutes les tailles correspondantes. Expliquez le pourquoi du résultat obtenu pour la structure ne contenant aucun membre donnée du tout.
21 C++ crée automatiquement l'équivalent d'un **typedef**pour les structures, comme vous l'avez appris dans ce

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

chapitre. Il fait la même chose pour les énumérations et les unions. Ecrivez un petit programme qui démontre cela.

22 Créez un **Stack**qui contient des **Stash**es. Chaque **Stash**contiendra 5 lignes d'un fichier passé en entrée. Créez les **Stash**es en utilisant **new**. Chargez le contenu d'un fichier dans votre **Stack**, puis réaffichez-le dans sa forme origninale en extrayant les données de la structure **Stack**.

23 Modifiez l'exercice 22 de manière à créer une structure qui encapsule le **Stack**de **Stashes**es. L'utilisateur doit non seulement être en mesure d'ajouter et d'obtenir des lignes par l'intermédiaire de fonctions membres, mais sous le capot, la structure doit utiliser un **Stack**de **Stash**es.

24 Créez une structure qui contient un **int**et un pointeur sur une autre instance de la même structure. Ecrivez une fonction qui prend l'adresse d'une telle structure et un **int**indiquant la longueur de la liste que vous désirez créer. Cette fonction créera une chaîne entière de ces structures (*une liste chaînée*), en démarrant à la position indiquée par l'argument (la *tête*de la liste), chaque instance pointant sur la suivante. Créez les nouvelles structures en utilisant **new**, et placez le compte (de quel numéro d'objet il s'agit) dans le **int**. Dans la dernière structure de la liste, mettez une valeur de zéro dans le pointeur afin d'indiquer que c'est la fin. Ecrivez une seconde fonction qui prend en argument la tête de votre liste, et qui se déplace jusqu'à la fin en affichant la valeur du pointeur et la valeur du **int**pour chaque noeud de la chaîne.

25 Répétez l'exercice 24, mais placez les fonctions à l'intérieur d'une structure au lieu d'avoir des structures "brutes" et des fonctions séparées.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

5 - Cacher l'implémentation

Une bibliothèque C typique contient un **struct**et quelques fonctions associées pour agir sur cette structure. Jusqu'ici vous avez vu comment le C++ prend les fonctions qui sont *conceptuellement*associées et les associe *littéralement*en

mettant les déclarations de fonctions à l'intérieur de la portée de la structure, en changeant la façon dont les fonctions sont appelées par la structure, en éliminant le passage de l'adresse de la structure en premier argument, et en ajoutant un nouveau nom de type au programme (donc vous n'avez pas à créer un **typedef**pour le label de la structure).

Tout ceci est très pratique – cela vous aide à organiser votre code et à le rendre plus facile à écrire et à lire. Cependant, il y a d'autres questions importantes quand on fait des bibliothèques simplifiées en C++, en particulier sur les problèmes de la sûreté et du contrôle. Ce chapitre s'intéresse au sujet des limites des structures.

5.1 - Fixer des limites

Dans toute relation il est important d'avoir des limites respectées par toutes les parties concernées. Quand vous créez une bibliothèque, vous établissez une relation avec le *programmeur client*qui utilise la bibliothèque pour construire une application ou une autre bibliothèque.

Dans un **struct**C, comme avec la plupart des choses en C, il n'y a pas de règles. Les programmeurs clients peuvent faire ce qu'ils veulent avec la structure, et il n'y a aucune façon de forcer un comportement particulier. Par exemple, bien que vous ayez vu dans le dernier chapitre l'importance des fonctions appelées **initialize( )**et **cleanup( )**, le programmeur client à la possibilité de ne pas appeler ces fonctions. (nous verrons une meilleure approche dans le prochain chapitre.) Et bien que vous préfèreriez vraiment que le programmeur client ne manipule pas directement certains membres de votre structure, en C il n'y aucun moyen de s'en prémunir. Tout est nu en ce monde.

Il y a deux raisons pour contrôler l'accès aux membres. La première est d'empêcher le programmeur client d'accéder à des outils auxquels il ne devrait pas toucher, des outils qui sont nécessaires pour les processus internes du type de données, mais pas de la partie de l'interface dont le programmeur client a besoin pour résoudre son problème particulier. C'est réellement un service rendu aux programmeurs clients parce qu'ils peuvent facilement voir ce qui est important pour eux et ce qu'ils peuvent ignorer.

La deuxième raison du contrôle d'accès est de permettre au concepteur de bibliothèque de changer les fonctionnements internes de la structure sans s'inquiéter de la façon dont cela affectera le programmeur client. Dans l'exemple de la **Stack**du dernier chapitre, vous pourriez vouloir assigner le stockage dans de grandes sections, pour la vitesse, plutôt que de créer un nouveau stockage chaque fois qu'un élément est ajouté. Si l'interface et l'exécution sont clairement séparées et protégées, vous pouvez accomplir ceci et exiger seulement un relink par le programmeur client.

5.2 - Le contrôle d'accès en C++

Le C++ introduit trois nouveaux mots-clefs pour fixer les limites d'une structure : **public**, **private**et **protected**. Leur sens et leur usage sont remarquablement clairs. Ces *spécificateurs d'accès*sont utilisés seulement dans la déclaration d'une structure, et ils changent les limites pour toutes les déclarations qui viennent après eux. Quand vous utilisez un tel spécificateur, il doit être suivi par deux points.

publicsignifie que tous les membres qui suivent cette déclaration sont disponibles à tout le monde. Les membres **public**sont comme les membres d'un **struct**. Par exemple, les déclarations de structures suivantes sont

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

équivalentes :

```
                    //: C05:Public.cpp
// Public est exactement comme une structure (struct) en C

struct A {
  int i;
  char j;
  float f;
  void func();
};

void A::func() {}

struct B {
public:
  int i;
  char j;
  float f;
  void func();
};

void B::func() {}

int main() {
  A a; B b;
  a.i = b.i = 1;
  a.j = b.j = 'c';
  a.f = b.f = 3.14159;
  a.func();
  b.func();
} ///:~
```

Le mot-clef **private**, à l'inverse, signifie que personne ne peut accéder à ce membre sauf vous, le créateur de ce type, dans les fonctions membres de ce type. **private**est une brique dans le mur entre vous et le programmeur client ; si quelqu'un essaye d'accéder à un membre **private**, ils obtiennent une erreur de compilation (compile-time error). Dans **struct B**dans l'exemple ci-dessus, vous pourriez vouloir rendre des morceaux de la représentation (c'est-à-dire, des données membres) cachés, accessibles uniquement par vous :

```
                    //: C05:Private.cpp
// Fixer les limites

struct B {
private:
  char j;
  float f;
public:
  int i;
  void func();
};

void B::func() {
  i = 0;
  j = '0';
  f = 0.0;
};

int main() {
  B b;
  b.i = 1;    // OK, public
//!  b.j = '1';  // Illégal, private
//!  b.f = 1.0;  // Illégal, private
} ///:~
```

Bien que **func( )**puisse accéder à n'importe quel membre de **B**(car **func( )**est un membre de **B**, ce qui lui garantit automatiquement la permission), une fonction globale ordinaire comme **main( )**ne le peut pas. Bien sûr, un membre d'une autre structure ne le peut pas non plus. Seules, les fonctions qui sont clairement écrites dans la déclaration de la structure (le "contrat") peuvent accéder aux membres **private**.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Il n'y a pas d'ordre requis pour les spécificateurs d'accès, et ils peuvent apparaître plus d'une fois. Ils affectent tous les membres déclarés après eux et avant le spécificateur d'accès suivant.

5.2.1 - protected

Le dernier spécificateur est **protected**. **protected**agit exactement comme **private**, avec une exception dont nous ne pouvons pas vraiment parler maintenant : les structures "héritées" (qui ne peuvent accéder aux membres **protected**) peuvent accéder aux membres **protected**. Ceci deviendra plus clair au Chapitre 14 quand l'héritage sera introduit. Pour le moment, considérez que **protected**a le même effet que **private**.

5.3 - L'amitié

Que faire si vous voulez donner accès à une fonction qui n'est pas membre de la structure courante ? Ceci est accompli en déclarant cette fonction **friend**(amie) *dans*la déclaration de la structure. Il est important que la déclaration **friend**ait lieu à l'intérieur de la déclaration de la structure parce que vous (ainsi que le compilateur) devez être capables de lire la déclaration de la structure et d'y voir toutes les règles concernant la taille et le comportement de ce type de données. Et une règle très importante dans toute relation est "qui peut accéder à mon implémentation privée ?"

La classe contrôle le code qui a accès à ses membres. Il n'y a pas de moyen magique de "forcer le passage" depuis l'extérieur si vous n'êtes pas **friend**; vous ne pouvez pas déclarer une nouvelle classe et dire "Salut, je suis *friend*(un ami, ndt) de **Bob**.

Vous pouvez déclarer une fonction globale **friend**, et vous pouvez également déclarer une fonction membre d'une autre structure, ou même une structure entière, en tant que **friend**. Voici un exemple :

```
                    //: C05:Friend.cpp
// Friend permet des accès spéciaux

// Déclaration (spécification du type incomplète ) :
struct X;

struct Y {
  void f(X*);
};

struct X { // Définition
private:
  int i;
public:
  void initialize();
  friend void g(X*, int); // friend global
  friend void Y::f(X*);  // friend membre d'une structure
  friend struct Z; // Structure entière comme friend
  friend void h();
};

void X::initialize() {
  i = 0;
}

void g(X* x, int i) {
  x->i = i;
}

void Y::f(X* x) {
  x->i = 47;
}

struct Z {
private:
  int j;
public:
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  void initialize();
  void g(X* x);
};

void Z::initialize() {
  j = 99;
}

void Z::g(X* x) {
  x->i += j;
}

void h() {
  X x;
  x.i = 100; // Manipulation directe des données
}

int main() {
  X x;
  Z z;
  z.g(&x);
} ///:~
```

struct Ya une fonction membre **f( )**qui modifiera un objet de type **X**. Cela ressemble à un casse-tête car le compilateur C++ exige que vous déclariez tout avant de pouvoir y faire référence, donc **struct Y**doit être déclaré avant que son membre **Y::f(X*)**puisse être déclaré comme **friend**dans **struct X**. Mais pour déclarer **Y::f(X*)**, **struct X**doit d'abord être déclaré !

Voici la solution. Remarquez que **Y::f(X*)**prend l' *adresse*d'un objet **X**. C'est critique parce que le compilateur sait toujours comment passer une adresse, qui est d'une taille fixe quelque soit le type d'objet passé, même s'il n'a pas toutes les informations à propos de la taille du type concerné. Toutefois, si vous essayez de passer l'objet complet le compilateur doit voir la déclaration de la structure **X**en intégralité, pour connaître sa taille et savoir comment le passer, avant qu'il ne vous permette de déclarer une fonction comme **Y::g(X)**.

En passant l'adresse d'un **X**, le compilateur vous permet de faire une *spécification de type incomplète*de **X**avant de déclarer **Y::f(X*)**. Ceci est accompli par la déclaration :

```
struct X;
```

Cette déclaration dit simplement au compilateur qu'il existe une structure portant ce nom, et donc que c'est OK pour y faire référence tant que vous n'avez pas besoin de plus de détails que le nom.

A présent, dans **struct X**, la fonction **Y::f(X*)**peut être déclarée comme **friend**sans problème. Si vous aviez essayé de la déclarer avant que le compilateur eût vu la définition complète de **Y**, cela aurait généré une erreur. C'est une sécurité pour assurer la cohérence et éliminer les bugs.

Notez les deux autres fonctions **friend**. La première déclaration concerne une fonction globale ordinaire **g( )**. Mais **g( )**n'a pas été déclarée précédemment dans la portée générale ! Il s'avère que **friend**peut être utilisé de cette manière pour simultanément déclarer la fonction *et*lui donner le statut **friend**. Ce comportement s'applique aux structures en intégralité :

```
friend struct Z;
```

est une spécification de type incomplète pour **Z**, et donne à toute la structure le statut **friend**.

5.3.1 - Amis emboîtés

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Faire une structure emboîtée ne lui donne pas automatiquement accès aux membres **private**. Pour obtenir cela, vous devez suivre une procédure particulière : d'abord, déclarer (sans la définir) la structure emboîtée, puis la déclarer en tant que **friend**, et finalement définir la structure. La définition de la structure doit être séparée de la déclaration **friend**, autrement elle serait vu par le compilateur comme étant non membre. Voici un exemple :

```
                        //: C05:NestFriend.cpp
// friends emboîtés
#include <iostream>
#include <cstring> // memset()
using namespace std;
const int sz = 20;

struct Holder {
private:
  int a[sz];
public:
  void initialize();
  struct Pointer;
  friend struct Pointer;
  struct Pointer {
  private:
    Holder* h;
    int* p;
  public:
    void initialize(Holder* h);
    // Se déplace dans le tableau:
    void next();
    void previous();
    void top();
    void end();
    // Accession à des valeurs:
    int read();
    void set(int i);
  };
};

void Holder::initialize() {
  memset(a, 0, sz * sizeof(int));
}

void Holder::Pointer::initialize(Holder* rv) {
  h = rv;
  p = rv->a;
}

void Holder::Pointer::next() {
  if(p < &(h->a[sz - 1])) p++;
}

void Holder::Pointer::previous() {
  if(p > &(h->a[0])) p--;
}

void Holder::Pointer::top() {
  p = &(h->a[0]);
}

void Holder::Pointer::end() {
  p = &(h->a[sz - 1]);
}

int Holder::Pointer::read() {
  return *p;
}

void Holder::Pointer::set(int i) {
  *p = i;
}

int main() {
  Holder h;
  Holder::Pointer hp, hp2;
  int i;

  h.initialize();
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    hp.initialize(&h);
    hp2.initialize(&h);
    for(i = 0; i < sz; i++) {
      hp.set(i);
      hp.next();
    }
    hp.top();
    hp2.end();
    for(i = 0; i < sz; i++) {
      cout << "hp = " << hp.read()
           << ", hp2 = " << hp2.read() << endl;
      hp.next();
      hp2.previous();
    }
} ///:~
```

Quand **Pointer**est déclaré, l'accès aux membres privés de **Holder**lui est accordé en disant :

```
                        friend struct Pointer;
```

struct Holdercontient un tableau de **int**s et **Pointer**vous permet d'y accéder. Parce que **Pointer**est fortement lié avec **Holder**, il est judicieux d'en faire une structure membre de **Holder**. Mais comme **Pointer**est une classe différente de **Holder**, vous pouvez en créer plusieurs instances dans **main( )**et les utiliser pour sélectionner différentes parties du tableau. **Pointer**est une structure au lieu d'un simple pointeur C, donc vous pouvez garantir qu'il pointera toujours sans risque dans **Holder**.

La fonction **memset( )**de la bibliothèque C standard (dans **<cstring>**) est utilisée par commodité dans le programme ci-dessus. Elle initialise toute la mémoire démarrant à une certaine addresse (le premier argument) à une valeur particulière (le deuxième argument) sur **n**octets à partir de l'adresse de départ (**n**est le troisième argument). Bien sûr, vous auriez pu simplement utiliser une boucle pour itérer sur toute la mémoire, mais **memset( )**est disponible, abondamment testée (donc il est moins probable que vous introduisiez une erreur), et probablement plus efficace que si vous le codiez à la main.

5.3.2 - Est-ce pur ?

La définition de classe vous donne une piste de vérification, afin que vous puissiez voir en regardant la classe quelles fonctions ont la permission de modifier les parties privées de la classe. Si une fonction est **friend**, cela signifie que ce n'est pas un membre, mais que vous voulez quand-même lui donner la permission de modifier des données privées, et elle doit être listée dans la définition de la classe afin que tout le monde puisse voir que c'est une des fonctions privilégiées.

Le C++ est un langage objet hybride, pas objet pur, et le mot-clé **friend**a été ajouté pour régler certains problèmes pratiques qui ont surgi. Il n'est pas choquant de souligner que cela rend le langage moins "pur" car C++ *est*conçu pour être pragmatique, et non pas par pour aspirer à un idéal abstrait.

5.4 - Organisation physique d'un objet

Le chapitre 4 affirmait qu'un **struct**écrit pour un compilateur C puis compilé avec C++ resterait inchangé. Cette affirmation faisait principalement référence à l'organisation physique d'un **struct**, c'est-à-dire à l'emplacement mémoire individuel des variables au sein de la mémoire allouée pour l'objet. Si le compilateur C++ modifiait l'organisation des **struct**s conçus en C, alors tout code C que vous auriez écrit et qui serait basé sur la connaissance de l'emplacement précis des variables dans un **struct**cesserait de fonctionner.

Cependant, quand vous commencez à utiliser des spécificateurs d'accès, vous entrez de plein pied dans le royaume du C++, et les choses changent un peu. Dans un "bloc d'accès" particulier (un groupe de déclarations

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

délimité par des spécificateurs d'accès), on a la garantie que les variables seront positionnées de manière contigües en mémoire, comme en C. Toutefois, les blocs d'accès peuvent ne pas apparaître au sein de l'objet dans l'ordre dans lequel vous les avez déclarés. Bien que le compilateur dispose *en général*les blocs exactement comme vous les voyez, il n'y a pas de règles à ce sujet, car l'architecture d'une machine particulière et/ou d'un système pourrait avoir un support explicite des mots-clefs **private**et **protected**qui imposerait à ces blocs de se trouver dans des emplacements mémoires particuliers. Les spécifications du langage ne veulent pas priver une implémentation de ce type d'avantage.

Les spécificateurs d'accès font partie de la structure et n'affectent pas les objets créés à partir de la structure. Toutes les informations relatives aux spécifications d'accès disparaissent avant que le programme ne soit exécuté ; en général, ceci se produit au moment de la compilation. Lors de l'exécution, les objets deviennent des "espaces de stockage" et rien de plus. Si vous le voulez vraiment, vous pouvez enfreindre toutes les règles et accéder directement à la mémoire, comme en C. C++ n'est pas conçu pour vous éviter de faire des choses imprudentes. Il vous fournit simplement une alternative bien plus simple, et autrement plus souhaitable.

En général, ce n'est pas une bonne idée de dépendre de quelque chose de spécifique à l'implémentation quand vous écrivez un programme. Quand vous devez avoir de telles dépendances, encapsulez-les dans une structure de façon à ce que les changements nécessaires au portage soient concentrés en un même endroit.

5.5 - La classe

Le contrôle d'accès est souvent appelé le *masquage de l'implémentation*. Inclure les fonctions dans les structures (souvent désigné par le terme encapsulation Comme nous l'avons dit précédemment, on appelle parfois le contrôle d'accès, l'encapsulation.)produisent un type de données avec des caractéristiques et des comportements, mais le contrôle d'accès impose des limites à ce type de données, pour deux motifs importants. La première est d'établir ce que le programmeur client peut et ne peut pas utiliser. Vous pouver construire vos mécanismes internes dans la structure sans vous soucier que des programmeurs clients pensent à ces mécanismes qui font partie de l'interface qu'ils devront employer.

Ceci amène directement la deuxième raison, qui est de séparer l'interface de l'implémentation. Si la structure est employée dans un ensemble de programmes, mais que les programmeurs clients ne peuvent faire rien d'autre qu'envoyer des messages à l'interface publique, alors vous pouvez changer tout ce qui est privé sans exiger des modifications à leur code.

L'encapsulation et le contrôle d'accès, pris ensemble, créent quelque chose de plus que la **struct**C. Nous sommes maintenant dans le monde de la programmation orientée-objet, où une structure décrit une classe d'objets comme vous pouvez décrire une classe de poissons ou une classe d'oiseaux : Tout objet appartenant à cette classe partagera ces caractéristiques et comportements. C'est ce qu'est devenue la déclaration de structure, une description de la façon dont tous les objets de ce type agiront et à quoi ils ressembleront.

Dans le langage POO d'origine, Simula-67, le mot-clé **class**était utilisé pour décrire un nouveau type de données. Ceci a apparamment inspiré à Stroustrup de choisir le même mot-clé pour le C++, pour souligner que c'était le point focal de tout le langage : la création de nouveaux types de données qui sont plus que des **struct**s C avec des fonctions. Cela semble être une justification adéquate pour un nouveau mot-clé.

Cependant, **class**est proche d'être un mot-clé inutile en C++. Il est identique au mot-clé **struct**à tous les points de vue sauf un : les membres d'une **class**sont **private**par défaut, tandis que ceux d'une **struct**sont **public**. Nous avons ici deux structures qui produisent le même résultat :

```
                    //: C05:Class.cpp
// Similarité entre une structure et une classe

struct A {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
private:
  int i, j, k;
public:
  int f();
  void g();
};

int A::f() {
  return i + j + k;
}

void A::g() {
  i = j = k = 0;
}

// On obtient des résultats identiques avec :

class B {
  int i, j, k;
public:
  int f();
  void g();
};

int B::f() {
  return i + j + k;
}

void B::g() {
  i = j = k = 0;
}

int main() {
  A a;
  B b;
  a.f(); a.g();
  b.f(); b.g();
} ///:~
```

La classe est le concept fondamental de la POO en C++. C'est un des mots-clés qui *ne sera pas*mis en gras dans ce livre – ça deviendrait fatigant avec un mot répété aussi souvent que “class.” Le changement avec les classes est si important que je suspecte que Stroustrup aurait préféré mettre définitivement aux clous la structure, mais le besoin de compatibilité ascendante avec le C ne permettait pas cela.

Beaucoup de personnes préfèrent un style de création de classe plutôt façon **struct**que façon classe parce que vous surchargez le comportement privé par défaut de la classe en commençant par les éléments **public**s :

```
                    class X {
public:
  void interface_function();
private:
  void private_function();
  int internal_representation;
};
```

La logique sous-jacente est qu'il est plus sensé pour le lecteur de voir les membres qui ont le plus d'intérêt pour lui, ainsi il peut ignorer tout ce qui est privé. En effet, la seule raison pour laquelle tous les autres membres doivent être déclarés dans la classe est qu'ainsi le compilateur sait de quelle taille est l'objet et peut les allouer correctement, et ainsi peut garantir l'uniformité.

Les exemples de ce livre, cependant, mettront les membres privés en premier, comme ceci :

```
                    class X {
  void private_function();
  int internal_representation;
public:
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  void interface_function();
};
```

Certains se donnent même la peine de décorer leurs propres noms privés :

```
                    class Y {
public:
  void f();
private:
  int mX;  // nom "décoré"
};
```

Comme **mX**est déjà caché dans la portée de **Y**, le **m**(pour “membre”) n'est pas nécessaire. Cependant dans les projets avec beaucoup de variables globales (quelque chose que vous devriez essayer d'éviter, mais qui est parfois inévitable dans des projets existants), il est utile de pouvoir distinguer dans une définition de fonction membre une donnée globale d'une donnée membre.

5.5.1 - Modifier Stash pour employer le contrôle d'accès

Il est intéressant de prendre les exemples du chapitre 4 et de les modifier pour employer les classes et le contrôle d'accès. Observez la façon dont la partie de l'interface accessible au client se distingue maintenant clairement, de telle sorte qu'il n'y a aucune possibilité pour les programmeurs clients de manipuler accidentellement une partie de la classe qu'ils ne devraient pas.

```
                        //: C05:Stash.h
// Converti pour utiliser le contrôle d'accès
#ifndef STASH_H
#define STASH_H

class Stash {
  int size;      // Taille de chaque espace
  int quantity;  // Nombre d'espaces de stockage
  int next;      // Prochain espace vide
  // Tableaux d'octets alloués dynamiquement :
  unsigned char* storage;
  void inflate(int increase);
public:
  void initialize(int size);
  void cleanup();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH_H ///:~
```

La fonction **inflate( )**a été déclaré privée parce qu'elle est utilisée seulement par la fonction **add( )**et fait donc partie de l'implémentation interne, pas de l'interface. Cela signifie que, plus tard, vous pourrez changer l'implémentation interne pour utiliser un système différent pour la gestion de la mémoire.

En dehors du nom du fichier d'include, l'en-tête ci-dessus est la seule chose qui ait été changée pour cet exemple. Le fichier d'implémentation et le fichier de test sont identiques.

5.5.2 - Modifier Stack pour employer le contrôle d'accès

Comme deuxième exemple, voici **Stack**changé en classe. Maintenant la structure imbriquée des données est privée, ce qui est est une bonne chose car cela assure que le programmeur client ne verra pas la représentation interne de la **Stack**et ne dépendra pas de cette dernière :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                          //: C05:Stack2.h
// Structure imbriquée via une liste chaînée
#ifndef STACK2_H
#define STACK2_H

class Stack {
  struct Link {
    void* data;
    Link* next;
    void initialize(void* dat, Link* nxt);
  }* head;
public:
  void initialize();
  void push(void* dat);
  void* peek();
  void* pop();
  void cleanup();
};
#endif // STACK2_H ///:~
```

Comme précédemment, l'implémentation ne change pas et donc on ne la répète pas ici. Le test, lui aussi, est identique. La seule chose qui ait changé c'est la robustesse de l'interface de la classe. L'intérêt réel du contrôle d'accès est de vous empêcher de franchir les limites pendant le développement. En fait, le compilateur est la seule chose qui connaît le niveau de protection des membres de la classe. Il n'y a aucune information de contrôle d'accès dans le nom du membre au moment de l'édition de liens. Toute la vérification de protection est faite par le compilateur ; elle a disparu au moment de l'exécution.

Notez que l'interface présentée au programmeur client est maintenant vraiment celle d'une pile push-down. Elle est implémentée comme une liste chaînée, mais vous pouvez changer cela sans affecter ce avec quoi le programmeur client interagit, ou (ce qui est plus important) une seule ligne du code client.

5.6 - Manipuler les classes

Les contrôles d'accès du C++ vous permettent de séparer l'interface de l'implémentation, mais la dissimulation de l'implémentation n'est que partielle. Le compilateur doit toujours voir les déclarations de toutes les parties d'un objet afin de le créer et de le manipuler correctement. Vous pourriez imaginer un langage de programmation qui requiert seulement l'interface publique d'un objet et autorise l'implémentation privée à être cachée, mais C++ effectue autant que possible la vérification des types de façon statique (au moment de la compilation). Ceci signifie que vous apprendrez aussi tôt que possible s'il y a une erreur, et que votre programme est plus efficace. Toutefois, inclure l'implémentation privée a deux effets : l'implémentation est visible même si vous ne pouvez réellement y accéder, et elle peut causer des recompilations inutiles.

5.6.1 - Dissimuler l'implémentation

Certains projets ne peuvent pas se permettre de rendre leur implémentation visible au programmeur client. Cela peut révéler des informations stratégiques dans les fichiers d'en-tête d'une librairie que la compagnie ne veut pas rendre disponible aux concurrents. Vous pouvez travailler sur un système où la sécurité est un problème (un algorithme d'encryptage, par exemple) et vous ne voulez pas laisser le moindre indice dans un fichier d'en-tête qui puisse aider les gens à craquer votre code. Ou vous pouvez mettre votre librairie dans un environnement "hostile", où les programmeurs auront de toute façon directement accès aux composants privés, en utilisant des pointeurs et du forçage de type. Dans toutes ces situations, il est intéressant d'avoir la structure réelle compilée dans un fichier d'implémentation plutôt que dans un fichier d'en-tête exposé.

5.6.2 - Réduire la recompilation

Le gestionnaire de projet dans votre environnement de programmation causera la recompilation d'un fichier si ce

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

fichier est touché (c'est-à-dire modifié) *ou*si un autre fichier dont il dépend (un fichier d'en-tête inclu) est touché. Cela veut dire qu'à chaque fois que vous modifiez une classe, que ce soit l'interface publique ou les déclarations de membres privés, vous forcerez une recompilation de tout ce qui inclut ce fichier d'en-tête. On appelle souvent cela le *problème de la classe de base fragile*. Pour un grand projet dans ses étapes initiales cela peut être peu pratique car l'implémentation sous-jacente peut changer souvent ; si le projet est très gros, le temps de compilation peut prévenir tout changement de direction.

La technique pour résoudre ce problème est souvent appelée *manipulation de classes*ou "chat du Cheshire" Ce nom est attribué à John Carolan, un des premiers pionniers du C++, et, bien sûr, Lewis Carrol. Cette technique peut aussi être vue comme une forme du “bridge” design pattern, décrit dans le Volume 2.– tout ce qui concerne l'implémentation disparaît sauf un unique pointeur, le "sourire". Le pointeur fait référence à une structure dont la définition est dans le fichier d'implémentation avec toutes les définitions des fonctions membres. Ainsi, tant que l'interface n'est pas modifiée, le fichier d'en-tête reste intouché. L'implémentation peut changer à volonté, et seuls les fichiers d'implémentation ont besoin d'être recompilés et relinkés avec le projet.

Voici un exemple simple démontrant la technique. Le fichier d'en-tête contient uniquement l'interface publique et un unique pointeur de classe incomplètement spécifiée :

```
                          //: C05:Handle.h
// Handle classes
#ifndef HANDLE_H
#define HANDLE_H

class Handle {
  struct Cheshire; // Déclaration de classe uniquement
  Cheshire* smile;
public:
  void initialize();
  void cleanup();
  int read();
  void change(int);
};
#endif // HANDLE_H ///:~
```

Voici tout ce que le programmeur client est capable de voir. La ligne

```
                          struct Cheshire;
```

est une *spécification incomplète*ou une *déclaration de classe*(une *définition de classe*inclut le corps de la classe). Ce code dit au compilateur que **Cheshire**est un nom de structure, mais ne donne aucun détail à propos du **struct**. Cela représente assez d'information pour créer un pointeur pour le **struct**; vous ne pouvez créer d'objet tant que le corps du **struct**n'a pas été fourni. Avec cette technique, le corps de cette structure est dissimulé dans le fichier d'implémentation :

```
                          //: C05:Handle.cpp {O}
// Handle implementation
#include "Handle.h"
#include "../require.h"

// Define Handle's implementation:
struct Handle::Cheshire {
  int i;
};

void Handle::initialize() {
  smile = new Cheshire;
  smile->i = 0;
}

void Handle::cleanup() {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  delete smile;
}

int Handle::read() {
  return smile->i;
}

void Handle::change(int x) {
  smile->i = x;
} ///:~
```

Cheshireest une structure imbriquée, donc elle doit être définie avec la définition de portée :

```
                        struct Handle::Cheshire {
```

Dans **Handle::initialize( )**, le stockage est alloué pour une structure **Cheshire**, et dans **Handle::cleanup( )**ce stockage est libéré. Ce stockage est utilisé en lieu et place de tous les éléments que vous mettriez normalement dans la section **private**de la classe. Quand vous compilez **Handle.cpp**, cette définition de structure est dissimulée dans le fichier objet où personne ne peut la voir. Si vous modifiez les éléments de **Cheshire**, le seul fichier qui doit être recompilé est **Handle.cpp**car le fichier d'en-tête est intouché.

L'usage de **Handle**est similaire à celui de toutes les classes: inclure l'en-tête, créer les objets, et envoyer des messages.

```
                        //: C05:UseHandle.cpp
//{L} Handle
// Use the Handle class
#include "Handle.h"

int main() {
  Handle u;
  u.initialize();
  u.read();
  u.change(1);
  u.cleanup();
} ///:~
```

La seule chose à laquelle le programmeur client peut accéder est l'interface publique, donc tant que l'implémentation est la seule chose qui change, le fichier ci-dessus ne nécessite jamais une recompilation. Ainsi, bien que ce ne soit pas une dissimulation parfaite de l'implémentation, c'est une grande amélioration.

5.7 - Résumé

Le contrôle d'accès en C++ donne un bon contrôle au créateur de la classe. Les utilisateurs de la classe peuvent [clairement] voir exactement ce qu'ils peuvent utiliser et ce qui est à ignorer. Plus important, ceci-dit, c'est la possibilité de s'assurer qu'aucun programmeur client ne devienne dépendant d'une partie quelconque [partie] de l'implémentation interne d'une classe. Si vous faites cela en tant que créateur de la classe, vous pouvez changer l'implémentation interne tout en sachant qu'aucun programmeur client ne sera affecté par les changements parce qu'ils ne peuvent accéder à cette partie de la classe.

Quand vous avez la capacité de changer l'implémentation interne, vous pouvez non seulement améliorer votre conception ultérieurement, mais vous avez également la liberté de faire des erreurs. Peu importe le soin avec lequel vous planifiez et concevez, vous ferez des erreurs. Savoir que vous pouvez faire des erreurs avec une sécurité relative signifie que vous expérimenterez plus, vous apprendrez plus vite, et vous finirez votre projet plus tôt.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

L'interface publique dans une classe est ce que le programmeur client *peut*voir, donc c'est la partie la plus importante à rendre “correcte” pendant l'analyse et la conception. Mais même cela vous permet une certaine marge de sécurité pour le changement. Si vous n'obtenez pas la bonne d'interface la première fois, vous pouvez *ajouter*plus de fonctions, tant que vous n'en enlevez pas que les programmeurs clients ont déjà employés dans leur code.

5.8 - Exercices

Les solutions des exercices suivants peuvent être trouvées dans le document électronique *The Thinking in C++ Annotated Solution Guide*, disponible à petit prix sur www.BruceEckel.com.

1 Créez une classe avec des données et des fonctions membres **public**, **private**, et **protected**. Créez un objet de cette classe et regardez quel genre de message le compilateur vous donne quand vous essayez d'accéder à chacun des membres.
2 Ecrivez un **struct**appelé **Lib**qui contienne trois objets **string a**, **b,**et **c**. Dans **main( )**créez un objet **Lib**appelé **x**et assignez une valeur à **x.a**, **x.b**, et **x.c**. Affichez les valeurs à l'écran. A présent remplacez **a**, **b,**et **c**par un tableau de **string s[3]**. Vérifiez que le code dans **main( )**ne compile plus suite à ce changement. Créez maintenant une classe appelée **Libc**, avec des objets **string private a**, **b,**et **c**, et les fonctions membres **seta( )**, **geta( )**, **setb( )**, **getb( )**, **setc( )**, et **getc( )**pour assigner (*set*en angais, ndt) et lire (*get*signifie obtenir en anglais, ndt) les valeurs. Ecrivez **main( )**comme précédemment. A présent, changez les objets **string private a**, **b,**et **c**en un tableau de **string s[3] private**. Vérifiez que le code dans **main( )**n'est *pas*affecté par ce changement.
3 Créez une classe et une fonction **friend**globale qui manipule les données **private**dans la classe.
4 Ecrivez deux classes, chacune ayant une fonction membre qui reçoit un pointeur vers un objet de l'autre classe. Créez une instance des deux objets dans **main( )**et appelez la fonction membre mentionnée ci-dessus dans chaque classe.
5 Créez trois classes. La première classe contient des données **private**, et accorde le status de **friend**à l'ensemble de la deuxième classe ainsi qu'à une fonction membre de la troisième. Dans **main( )**, vérifiez que tout marche correctement.
6 Créez une classe **Hen**. Dans celle-ci, imbriquez une classe **Nest**. Dans **Nest**, placez une classe **Egg**. Chaque classe devrait avoir une fonction membre **display( )**. Dans **main( )**, créez une instance de chaque classe et appelez la fonction **display( )**de chacune d'entre elles.
7 Modifiez l'exercice 6 afin que **Nest**et **Egg**contiennent chacune des données **private**. Donnez accès à ces données privées aux classes englobantes au moyen du mot-clé **friend**.
8 Créez une classe avec des données membres réparties au sein de nombreuses sections **public**, **private,**et **protected**. Ajoutez une fonction membre **showMap( )**qui affiche à l'écran les noms de chacune des données membre ainsi que leurs addresses. Si possible, compilez et exécutez ce programme sur plusieurs compilateurs et/ou ordinateur et/ou système d'exploitation pour voir s'il y a des différences d'organisation dans l'objet.
9 Copiez l'implémentation et les fichiers de test pour **Stash**du Chapitre 4 afin que vous puissiez compiler et tester **Stash.h**dans ce chapitre.
10 Placez des objets de la class **Hen**de l'Exercice 6 dans un **Stash**. Développez les ensembles et affichez-les à l'écran (si vous ne l'avez pas déjà fait, vous aurez besoin d'ajouter **Hen::print( )**).
11 Copiez l'implémentation et les fichiers de test pour **Stack**du Chapitre 4 afin que vous puissiez compiler et tester **Stack2.h**dans ce chapitre.
12 Placez des objets de la classe **Hen**de l'Exercice 6 dans un **Stack**. Développez les ensemble et affichez-les à l'écran (si vous ne l'avez pas déjà fait, vous aurez besoin d'ajouter **Hen::print( )**).
13 Modifiez **Cheshire**dans **Handle.cpp**, et vérifiez que votre gestionnaire de projets recompile et refasse l'édition de liens uniquement pour ce fichier, mais ne recompile pas **UseHandle.cpp**.
14 Créez une classe **StackOfInt**(une pile qui contient des **int**s) en utilisant la technique du "chat du Cheshire" qui disimule les structures de données de bas niveau utilisées pour stocker les éléments dans une classe appelée **StackImp**. Implémentez deux versions de **StackImp**: une qui utilise un tableau de **int**de taille fixe, et une qui utilise un **vector<int>**. Assignez une taille maximum dans le premier cas pour la pile afin que vous n'ayiez pas à vous soucier de l'agrandissement du tableau dans la première version. Remarquez que la

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

classe **StackOfInt.h**n'a pas besoin d'être modifiée quand **StackImp**change.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

6 - Initialisation & Nettoyage

Le chapitre 4 a apporté une amélioration significative dans l'utilisation d'une bibliothèque en prenant tous les composants dispersés d'une bibliothèque typique du C et en les encapsulant dans une structure (un type de données abstrait, appelé dorénavant une classe).

Ceci fournit non seulement un point d'entrée unique dans un composant de bibliothèque, mais cela cache également les noms des fonctions dans le nom de classe. Dans le chapitre 5, le contrôle d'accès (le masquage de l'implémentation) a été présenté. Ceci donne au concepteur de classe une manière d'établir des frontières claires pour déterminer ce que le programmeur client a la permission de manoeuvrer et ce qui hors des limites. Cela signifie que les mécanismes internes d'une opération d'un type de données sont sous le contrôle et la discrétion du concepteur de la classe, et il est clair pour les programmeurs de client à quels membres ils peuvent et devraient prêter attention.

Ensemble, l'encapsulation et le contrôle d'accès permettent de franchir une étape significative en améliorant la facilité de l'utilisation de la bibliothèque. Le concept du “nouveau type de données” qu'ils fournissent est meilleur par certains côtés que les types de données intégrés existants du C. Le compilateur C++ peut maintenant fournir des garanties de vérification de type pour ce type de données et assurer ainsi un niveau de sûreté quand ce type de données est employé.

Cependant quand il est question de sécurité, le compilateur peut en faire beaucoup plus pour nous que ce qui est proposé par le langage C. Dans ce chapitre et de futurs, vous verrez les dispositifs additionnels qui ont été mis en #uvre en C++ qui font que les bogues dans votre programme vous sautent presque aux yeux et vous interpellent, parfois avant même que vous ne compiliez le programme, mais habituellement sous forme d'avertissements et d'erreurs de compilation. Pour cette raison, vous vous habituerez bientôt au scénario inhabituel d'un programme C++ qui compile fonctionne du premier coup.

Deux de ces questions de sûreté sont l'initialisation et le nettoyage. Un grand partie des bogues C se produisent quand le programmeur oublie d'initialiser ou de vider une variable. C'est particulièrement vrai avec des bibliothèques C, quand les programmeurs de client ne savent pas initialiser une structure, ou même ce qu'ils doivent initialiser. (Les bibliothèques n'incluent souvent pas de fonction d'initialisation, et le programmeur client est forcé d'initialiser la struture à la main.) Le nettoyage est un problème spécial parce que les programmeurs en langage C oublient facilement les variables une fois qu'elles ne servent plus, raison pour laquelle le nettoyage qui peut être nécessaire pour une structure d'une bibliothèque est souvent oublié.

En C++, le concept d'initialisation et de nettoyage est essentiel pour une utilisation facile d'une bibliothèque et pour éliminer les nombreux bogues subtiles qui se produisent quand le programmeur de client oublie d'exécuter ces actions. Ce chapitre examine les dispositifs C++ qui aident à garantir l'initialisation appropriée et le nettoyage.

6.1 - Initialisation garantie avec le constructeur

Les deux classes **Stash/ cachette**et **Stack/pile**définies plus tôt ont une fonction appelée **initialisation( )**, qui indique par son nom qu'elle doit être appelée avant d'utiliser l'objet de quelque manière que ce soit. Malheureusement, ceci signifie que le client programmeur doit assurer l'initialisation appropriée. Les clients programmeurs sont enclins à manquer des détails comme l'initialisation dans la précipitation pour utiliser votre incroyable librairie pour résoudre leurs problèmes. En C++, l'initialisation est trop importante pour la laisser au client programmeur. Le concepteur de la classe peut garantir l'initialisation de chaque objet en fournissant une fonction spéciale appelé *constructeur*. Si une classe a un constructeur, le compilateur appellera automatiquement ce constructeur au moment où l'objet est créé, avant que le programmeur client puisse toucher à l'objet. Le constructeur appelé n'est pas une option pour le client programmeur ; il est exécuté par le compilateur au moment où l'objet est définie.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Le prochain défi est de donner un nom à cette fonction. Il y a deux problèmes. La première est que le nom que vous utilisez peut potentiellement être en conflit avec un nom que vous pouvez utiliser pour un membre de la classe. Le second est que comme le compilateur est responsable de l'appel au constructeur, il doit toujours savoir quelle fonction appeler. La solution que Stroustrup a choisi semble la plus simple et la plus logique : Le nom du constructeur est le même que le nom de la classe. Cela semble raisonnable qu'une telle fonction puisse être appelé automatiquement à l'initialisation.

Voici une classe simple avec un constructeur:

```
                    class X {
  int i;
public:
  X();  // Constructeur
};
```

Maintenant, quand un objet est défini,

```
                    void f() {
  X a;
  // ...
}
```

la même chose se produit que si **a**était un **int**: le stockage est alloué pour l'objet. Mais quand le programme atteint le *point de séquence*(point d'exécution) où **a**est définie, le constructeur est appelé automatiquement. C'est le compilateur qui insère discrètement l'appel à **X::X( )**pour l'objet **a**au point de définition. Comme n'importe quelle fonction membre, le premier argument (secret) pour le constructeur est le pointeur **this**- l'adresse de l'objet pour lequel il est appellé. Dans le cas d'un constructeur, cependant, **this**pointe sur un block non initialisé de mémoire, et c'est le travail du constructeur d'initialiser proprement cette mémoire.

Comme n'importe quelle fonction, le constructeur peut avoir des arguments pour vous permettre d'indiquer comment un objet est créé, lui donner des valeurs d'initialisation, et ainsi de suite. Les arguments du constructeur vous donnent une manière de garantir que toutes les parties de votre objet sont initialisées avec des valeurs appropriées. Par exemple, si une classe **Arbre**a un constructeur qui prend un seul entier en argument donnant la hauteur de l'arbre, vous devez créer un objet arbre comme cela:

```
                    Arbre a(12);  // arbre de 12 pieds (3,6 m)
```

Si **Arbre(int)**est votre seul constructeur, le compilateur ne vous laissera pas créer un objet d'une autre manière. (Nous allons voir les constructeurs multiples et les différentes possibilités pour appeler les constructeurs dans le prochain chapitre.)

Voici tout ce que fait un constructeur ; c'est une fonction avec un nom spéciale qui est appelé automatiquement par le compilateur pour chaque objet au moment de sa création. En dépit de sa simplicité, c'est très précieux parce qu'il élimine une grande classe de problèmes et facilite l'écriture et la lecture du code. Dans le fragment de code ci-dessus, par exemple vous ne voyez pas un appel explicite à une quelconque fonction **initialisation( )**qui serait conceptuellement différente de la définition. En C++, définition et initialisation sont des concepts unifiés, vous ne pouvez pas avoir l'un sans l'autre.

Le constructeur et le destructeur sont des types de fonctions très peu communes : elles n'ont pas de valeur de retour. C'est clairement différent d'une valeur de retour **void**, où la fonction ne retourne rien mais où vous avez toujours l'option de faire quelque chose d'autre. Les constructeurs et destructeurs ne retournent rien et vous ne pouvez rien y changer. L'acte de créer ou détruire un objet dans le programme est spécial, comme la naissance et la mort, et le compilateur fait toujours les appels aux fonctions par lui même, pour être sûr qu'ils ont lieu. Si il y avait

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

une valeur de retour, et si vous pouviez sélectionner la votre, le compilateur devrait d'une façon ou d'une autre savoir que faire avec la valeur de retour, ou bien le client programmeur devrait appeler explicitement le constructeur et le destructeur, ce qui détruirait leurs sécurité.

6.2 - Garantir le nettoyage avec le destructeur

En tant que programmeur C++, vous pensez souvent à l'importance de l'initialisation, mais il est plus rare que vous pensiez au nettoyage. Après tout, que devez-vous faire pour nettoyer un **int**? Seulement l'oublier. Cependant, avec des bibliothèques, délaisser tout bonnement un objet lorsque vous en avez fini avec lui n'est pas aussi sûr. Que se passe-t-il s'il modifie quelquechose dans le hardware, ou affiche quelque chose à l'écran, ou alloue de la mémoire sur le tas/pile ? Si vous vous contentez de l'oublier, votre objet n'accomplit jamais sa fermeture avant de quitter ce monde. En C++, le nettoyage est aussi important que l'initialisation et est donc garanti par le destructeur.

La syntaxe du destructeur est semblable à celle du constructeur : le nom de la classe est employé pour le nom de la fonction. Cependant, le destructeur est distingué du constructeur par un tilde (**~**) en préfixe. En outre, le destructeur n'a jamais aucun argument parce que la destruction n'a jamais besoin d'aucune option. Voici la déclaration pour un destructeur :

```
                    class Y {
public:
  ~Y();
};
```

Le destructeur est appelé automatiquement par le compilateur quand l'objet sort de la portée. Vous pouvez voir où le constructeur est appelé lors de la définition de l'objet, mais la seule preuve d'un appel au destructeur est l'accolade de fermeture de la portée qui entoure l'objet. Pourtant le destructeur est toujours appelé, même lorsque vous employez **goto**pour sauter d'une portée. (**goto**existe en C++ pour la compatibilité ascendante avec le C et pour les fois où il est pratique.) Notez qu'un *goto non local*, implémenté par les fonctions de la bibliothèque standard du C **setjmp( )**et **longjmp( )**, ne provoque pas l'appel des destructeurs. (Ce sont les spécifications, même si votre compilateur ne les met pas en application de cette façon. Compter sur un dispositif qui n'est pas dans les spécifications signifie que votre code est non portable.)

Voici un exemple illustrant les dispositifs des constructeurs et destructeurs que vous avez vu jusqu'à maintenant :

```
                    //: C06:Constructor1.cpp
// Construteurs & destructeurs
#include <iostream>
using namespace std;

class Tree {
  int height;
public:
  Tree(int initialHeight);  // Constructeur
  ~Tree();  // Destructeur
  void grow(int years);
  void printsize();
};

Tree::Tree(int initialHeight) {
  height = initialHeight;
}

Tree::~Tree() {
  cout << "au c#ur du destructeur de Tree" << endl;
  printsize();
}

void Tree::grow(int years) {
  height += years;
}
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
void Tree::printsize() {
  cout << "La taille du Tree est " << height << endl;
}

int main() {
  cout << "avant l'ouvertude de l'accolade" << endl;
  {
    Tree t(12);
    cout << "apres la création du Tree" << endl;
    t.printsize();
    t.grow(4);
    cout << "avant la fermeture de l'accolade" << endl;
  }
  cout << "apres la fermeture de l'accolade" << endl;
} ///:~
```

Voici la sortie du programme précédent :

```
                    avant l'ouverture de l'accolade
apres la création du Tree
La taille du Tree est 12
avant la fermeture de l'accolade
au c#ur du destructeur de Tree
La taille du Tree est 16
apres la fermeture de l'accolade
```

Vous pouvez voir que le destructeur est automatiquement appelé à la fermeture de la portée qui entoure l'objet.

6.3 - Elimination de la définition de bloc

En C, vous devez toujours définir toutes les variables au début d'un bloc, après l'ouverture des accolades. Ce n'est pas une exigence inhabituelle dans les langages de programmation, et la raison donnée a souvent été que c'est un “bon style de programmation ”. Sur ce point, j'ai des doutes. Il m'a toujours semblé malcommode, comme programmeur, de retourner au début d'un bloc à chaque fois que j'ai besoin d'une nouvelle variable. Je trouve aussi le code plus lisible quand la déclaration d'une variable est proche de son point d'utilisation.

Peut-être ces arguments sont-ils stylistiques ? En C++, cependant, il y a un vrai problème à être forcé de définir tous les objets au début de la portée. Si un constructeur existe, il doit être appelé quand l'objet est créé. Cependant, si le constructeur prend un ou plusieurs arguments d'initialisation, comment savez vous que vous connaîtrez cette information d'initialisation au début de la portée? En générale, vous ne la connaîtrez pas. Parce que le C n'a pas de concept de **private**, cette séparation de définition et d'initialisation n'est pas problématique. Cependant, le C++ garantit que quand un objet est créé, il est simultanément initialisé. Ceci garantit que vous n'aurez pas d'objet non initialisés se promenant dans votre système. Le C n'en a rien à faire ; en fait, le C *encourage*cette pratique en requérant que vous définissiez les variables au début du bloc avant que vous ayez nécessairement l'information d'initialisation C99, La version à jour du standard C, permet aux variables d'être définies à n'importe quel point d'une portée, comme C++..

En général, le C++ ne vous permettra pas de créer un objet avant que vous ayez les informations d'initialisation pour le constructeur. A cause de cela, le langage ne fonctionnerait pas si vous aviez à définir les variables au début de la portée. En fait, le style du langage semble encourager la définition d'un objet aussi près de son utilisation que possible. En C++, toute règle qui s'applique à un “objet” s'applique également automatiquement à un objet de type intégré. Ceci signifie que toute classe d'objet ou variable d'un type intégré peut aussi être définie à n'importe quel point de la portée. Ceci signifie que vous pouvez attendre jusqu'à ce que vous ayez l'information pour une variable avant de la définir, donc vous pouvez toujours définir et initialiser au même moment:

```
                    //: C06:DefineInitialize.cpp
// Définir les variables n'importe où
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
#include "../require.h"
#include <iostream>
#include <string>
using namespace std;

class G {
  int i;
public:
  G(int ii);
};

G::G(int ii) { i = ii; }

int main() {
  cout << "valeur d'initialisation? ";
  int retval = 0;
  cin >> retval;
  require(retval != 0);
  int y = retval + 3;
  G g(y);
} ///:~
```

Vous constatez que du code est exécuté, puis **retval**est définie, initialisé, et utilisé pour récupérer l'entrée de l'utilisateur, et puis **y**et **g**sont définies. C, par contre, ne permet pas à une variable d'être définie n'importe où excepté au début de la portée.

En général, vous devriez définir les variables aussi près que possible de leur point d'utilisation, et toujours les initialiser quand elles sont définies. (Ceci est une suggestion de style pour les types intégrés, pour lesquels l'initialisation est optionnelle). C'est une question de sécurité. En réduisant la durée de disponibilité d'une variable dans la portée, vous réduisez les chances d'abus dans une autre partie de la portée. En outre, la lisibilité est augmentée parce que le lecteur n'a pas besoin de faire des allées et venues au début de la portée pour connaître le type d'une variable.

6.3.1 - les boucles

En C++, vous verrez souvent un compteur de boucle **for**défini directement dans l'expression **for**:

```
                        for(int j = 0; j < 100; j++) {
    cout << "j = " << j << endl;
}
for(int i = 0; i < 100; i++)
 cout << "i = " << i << endl;
```

Les déclarations ci-dessus sont des cas spéciaux importants, qui embarassent les nouveaux programmeurs en C++.

Les variables **i**et **j**sont définies directement dans l'expression **for**(ce que vous ne pouvez pas faire en C). Elles peuvent être utilisées dans la boucle **for**. C'est une syntaxe vraiment commode parce que le contexte répond à toute question concernant le but de **i**et **j**, donc vous n'avez pas besoin d'utiliser des noms malcommodes comme **i_boucle_compteur**pour plus de clarté.

Cependant, vous pouvez vous tromper si vous supposez que la durée de vie des variables **i**et **j**se prolonge au delà de la portée de la boucle for – ce n'est pas le cas Un ancien brouillon du standard du C++ dit que la durée de vie de la variable se prolonge jusqu'à la fin de la portée qui contient la boucle **for**. Certains compilateurs implémentent toujours cela, mais ce n'est pas correct. Votre code ne sera portable que si vous limitez la portée à la boucle **for**. .

Le chapitre 3 souligne que les déclarations **while**et **switch**permettent également la définition des objets dans leurs expressions de contrôle, bien que cet emploi semble beaucoup moins important que pour les boucles **for**.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Méfiez-vous des variables locales qui cachent des variables de la portée englobant la boucle. En général, utiliser le même nom pour une variable imbriquée et une variable globale est confus et enclin à l'erreur Le langage Java considère ceci comme une si mauvaise idée qu'il signale un tel code comme une erreur..

Je considère les petites portées comme des indicateurs de bonne conception. Si une simple fontion fait plusieurs pages, peut être que vous essayez de faire trop de choses avec cette fonction. Des fonctions plus granulaires sont non seulement plus utiles, mais permettent aussi de trouver plus facilement les bugs.

6.3.2 - Allocation de mémoire

Une variable peut maintenant être définie à n'importe quel point de la portée, donc il pourrait sembler que le stockage pour cette variable ne peut pas être défini jusqu'à son point de déclaration. Il est en fait plus probable que le compilateur suivra la pratique du C d'allouer tous les stockages d'une portée à l'ouverture de celle ci. Ce n'est pas important, parce que, comme programmeur, vous ne pouvez pas accéder aux stockages (ou l'objet) jusqu'à ce qu'il ait été défini D'accord, vous pourriez probablement en jouant avec des pointers, mais vous seriez très, très méchants.. Bien que le stockage soit alloué au commencement d'un bloc, l'appel au constructeur n'a pas lieu avant le point de séquence où l'objet est défini parce que l'identifiant n'est pas disponible avant cela. Le compilateur vérifie même que vous ne mettez pas la définition de l'objet (et ainsi l'appel du constructeur) là où le point de séquence passe seulement sous certaines conditions, comme dans un **switch**ou à un endroit qu'un **goto**peut sauter. Ne pas commenter les déclarations dans le code suivant produira des warnings ou des erreurs :

```
                          //: C06:Nojump.cpp
// ne peut pas sauter le constructeur

class X {
public:
  X();
};

X::X() {}

void f(int i) {
  if(i < 10) {
    //! goto jump1; // Erreur: goto outrepasse l'initialisation
  }
  X x1;  // Constructeur appelé ici
 jump1:
  switch(i) {
    case 1 :
      X x2;  // Constructeur appelé ici
      break;
  //! case 2 : // Erreur: goto outrepasse l'initialisation
      X x3;  // Constructeur appelé ici
      break;
  }
}

int main() {
  f(9);
  f(11);
}///:~
```

Dans le code ci-dessus, le **goto**et le **switch**peuvent tout deux sauter le point de séquence où un constructeur est appelé. Cet objet sera alors dans la portée même sans que le constructeur ait été appelé, donc le compilateur génère un message d'erreur. Ceci garantit encore une fois qu'un objet ne soit pas créé sans être également initialisé.

Toutes les allocations mémoires discutées ici, se produisent, bien sûr, sur la pile. Le stockage est alloué par le compilateur en déplaçant le pointeur de pile vers le “bas” (un terme relatif, ce qui peut indiquer une augmentation ou une diminution de la valeur réelle du pointeur de pile, selon votre machine). Les objets peuvent aussi être alloués sur la pile en utilisant **new**, ce que nous explorerons dans le chapitre 13.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

6.4 - Stash avec constructueur et destructeur

Les exemples des chapitres précédents ont des fonctions évidentes qui établissent les constructeurs et destructeurs : **initialize( )**et **cleanup( )**. Voici l'en-tête **Stash**utilisant les constructeurs et destructeurs :

```
                    //: C06:Stash2.h
// Avec constructeurs & destructeurs
#ifndef STASH2_H
#define STASH2_H

class Stash {
  int size;      // Taille de chaque espace
  int quantity;  // Nombre d'espaces de stockage
  int next;      // Espace vide suivant
  // Tableau d'octet alloué dynamiquement
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int size);
  ~Stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH2_H ///:~
```

Les seules définitions de fonctions qui changent sont **initialize( )**et **cleanup( )**, qui ont été remplacées par un constructeur et un destructeur :

```
                    //: C06:Stash2.cpp {O}
// Constructeurs & destructeurs
#include "Stash2.h"
#include "../require.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  storage = 0;
  next = 0;
}

int Stash::add(void* element) {
  if(next >= quantity) // Reste-t-il suffisamment de place ?
    inflate(increment);
  // Copier l'élément dans l'espce de stockage,
  // à partir de l'espace vide suivant :
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Nombre indice
}

void* Stash::fetch(int index) {
  require(0 <= index, "Stash::fetch (-)index");
  if(index >= next)
    return 0; // Pour indiquer la fin
  // Produire un pointeur vers l'élément voulu :
  return &(storage[index * size]);
}

int Stash::count() {
  return next; // Nombre d'éléments dans CStash
}
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
void Stash::inflate(int increase) {
  require(increase > 0,
    "Stash::inflate zero or negative increase");
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copie vieux dans nouveau
  delete [](storage); // Vieux stockage
  storage = b; // Pointe vers la nouvelle mémoire
  quantity = newQuantity;
}

Stash::~Stash() {
  if(storage != 0) {
   cout << "freeing storage" << endl;
   delete []storage;
  }
} ///:~
```

Vous pouvez constatez que les fonctions **require.h**sont utilisées pour surveiller les erreurs du programmeur, à la place de **assert( )**. La sortie d'un échec de **assert( )**n'est pas aussi utile que celle des fonctions **require.h**(qui seront présentées plus loin dans ce livre).

Comme **inflate( )**est privé, la seule façon pour qu'un **require( )**puisse échouer est si une des autres fonctions membres passe accidentellement une valeur erronée à **inflate( )**. Si vous êtes sûr que cela ne peut arriver, vous pouvez envisager d'enlever le **require( )**, mais vous devriez garder cela en mémoire jusqu'à ce que la classe soit stable ; il y a toujours la possibilité que du nouveau code soit ajouté à la classe, qui puisse causer des erreurs. Le coût du **require( )**est faible (et pourrait être automatiquement supprimé en utilisant le préprocesseur) et la valeur en terme de robustesse du code est élevée.

Notez dans le programme test suivant comment les définitions pour les objets **Stash**apparaissent juste avant qu'elles soient nécessaires, et comment l'initialisation apparaît comme une part de la définition, dans la liste des arguments du constructeur :

```
                    //: C06:Stash2Test.cpp
//{L} Stash2
// Constructeurs & destructeurs
#include "Stash2.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize);
  ifstream in("Stash2Test.cpp");
  assure(in, " Stash2Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Notez également comment les appels à **cleanup( )**ont été éliminés, mais les destructeurs sont toujours appelés automatiquement quand **intStash**et **stringStash**sortent du champ.

Une chose dont il faut être conscient dans les exemples de **Stash**: je fais très attention d'utiliser uniquement des types intégrés ; c'est-à-dire ceux sans destructeurs. Si vous essayiez de copier des classes d'objets dans le **Stash**, vous rencontreriez beaucoup de problèmes et cela ne fontionnerait pas correctement. La librairie standard du C++ peut réellement faire des copies d'objets correctes dans ses conteneurs, mais c'est un processus relativement sale et complexe. Dans l'exemple **Stack**suivant, vous verrez que les pointeurs sont utilisés pour éviter ce problème, et dans un prochain chapitre le **Stash**sera modifié afin qu'il utilise des pointeurs.

6.5 - Stack avec des constructeurs & des destructeurs

Réimplémenter la liste chaînée (dans **Stack**) avec des constructeurs et des destructeurs montrent comme les constructeurs et les destructeurs marchent proprement avec **new**et **delete**. Voici le fichier d'en-tête modifié :

```
                    //: C06:Stack3.h
// Avec constructeurs/destructeurs
#ifndef STACK3_H
#define STACK3_H

class Stack {
  struct Link {
    void* data;
    Link* next;
    Link(void* dat, Link* nxt);
    ~Link();
  }* head;
public:
  Stack();
  ~Stack();
  void push(void* dat);
  void* peek();
  void* pop();
};
#endif // STACK3_H ///:~
```

Il n'y a pas que **Stack**qui ait un constructeur et un destructeur, mais la struct **Link**imbriquée également :

```
                    //: C06:Stack3.cpp {O}
// Constructeurs/destructeurs
#include "Stack3.h"
#include "../require.h"
using namespace std;

Stack::Link::Link(void* dat, Link* nxt) {
  data = dat;
  next = nxt;
}

Stack::Link::~Link() { }

Stack::Stack() { head = 0; }

void Stack::push(void* dat) {
  head = new Link(dat,head);
}

void* Stack::peek() {
  require(head != 0, "Stack vide");
  return head->data;
}

void* Stack::pop() {
  if(head == 0) return 0;
  void* result = head->data;
  Link* oldHead = head;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  head = head->next;
  delete oldHead;
  return result;
}

Stack::~Stack() {
  require(head == 0, "Stack non vide");
} ///:~
```

Le constructeur **Link::Link( )**initialise simplement les pointeurs **data**et **next**, ainsi dans **Stack::push( )**la ligne

```
                    head = new Link(dat,head);
```

ne fait pas qu'allouer un nouveau **Link**(en utilisant la création dynamique d'objet avec le mot-clé **new**, introduit au Chapitre 4), mais initialise également proprement les pointeurs pour ce **Link**.

Vous pouvez vous demander pourquoi le destructeur de **Link**ne fait rien – en particulier, pourquoi ne supprime-t-il pas le pointeur **data**? Il y a deux problèmes. Au chapitre 4, où la **Stack**a été présentée, on a précisé que vous ne pouvez pas correctement supprimer un pointeur **void**s'il pointe un objet (une affirmation qui sera prouvée au Chapitre 13). En outre, si le destructeur de **Link**supprimait le pointeur **data**, **pop( )**finirait par retourner un pointeur sur un objet supprimé, ce qui serait certainement un bogue. Ce problème est désigné parfois comme la question de la *propriété*: **Link**, et par là **Stack**, utilise seulement les pointeurs, mais n'est pas responsable de leur libération. Ceci signifie que vous devez faire très attention de savoir qui est responsable. Par exemple, si vous ne dépilez et ne supprimez pas tous les pointeurs de la **Stack**, ils ne seront pas libérés automatiquement par le destructeur de **Stack**. Ceci peut être un problème récurrent et mène aux fuites de mémoire, ainsi savoir qui est responsable la destruction d'un objet peut faire la différence entre un programme réussi et un bogué – C'est pourquoi **Stack::~Stack( )**affiche un message d'erreur si l'objet **Stack**n'est pas vide lors de la destruction.

Puisque l'allocation et la désallocation des objets **Link**sont cachés dans la **Stack**– cela fait partie de l'implémentation interne – vous ne la voyez pas se produire dans le programme de test, bien que *vous*soyez responsable de supprimer les pointeurs qui arrivent de **pop( )**:

```
                    //: C06:Stack3Test.cpp
//{L} Stack3
//{T} Stack3Test.cpp
// Constructeurs/destructeurs
#include "Stack3.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // Le nom de fichier est un argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  string line;
  // Lectrure du fichier et stockage des lignes dans la pile :
  while(getline(in, line))
    textlines.push(new string(line));
  // Dépiler les lignes de la pile et les afficher :
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << *s << endl;
    delete s;
  }
} ///:~
```

Dans ce cas-là, toutes les lignes dans les **textlines**sont dépilées et supprimées, mais si elles ne l'étaient pas, vous recevriez un message **require( )**qui signifie qu'il y a une fuite de mémoire.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

6.6 - Initialisation d'aggrégat

Un *aggrégat*est exactement ce qu'il a l'air d'être : un paquet de choses rassemblées ensembles. Cette définition inclut des aggrégats de type mixte, comme les **struct**ures et les **class**es. Un tableau est un aggrégat d'un type unique.

Initialiser les aggrégats peut être enclin à l'erreur et fastidieux. L' *initialisation d'aggrégat*du C++ rend l'opération beaucoup plus sure. Quand vous créez un objet qui est un aggrégat, tout ce que vous avez à faire est de faire une déclaration, et l'initialisation sera prise en charge par le compilateur. Cette déclaration peut avoir plusieurs nuances, selon le type d'aggrégat auquel vous avez à faire, mais dans tous les cas les éléments de la déclaration doivent être entourés par des accolades. Pour un tableau de types intégrés, c'est relativement simple :

```
int a[5] = { 1, 2, 3, 4, 5 };
```

Si vous essayez de passer plus de valeurs qu'il n'y a d'élements dans le tableau, le compilateur génère un message d'erreur. Mais que se passe-t-il si vous passez *moins*de valeurs ? Par exemple :

```
int b[6] = {0};
```

Ici, le compilateur utilisera la première valeur pour le premier élément du tableau, et ensuite utilisera zéro pour tous les éléments sans valeur fournie. Remarquez que ce comportement ne se produit pas si vous définissez un tableau sans liste de valeurs. Donc, l'expression ci-dessus est un moyen succint d'initialiser un tableau de zéros, sans utiliser une boucle **for**, et sans possibilité d'erreur de bord (selon les compilateurs, ce procédé peut même être plus efficace que la boucle **for**).

Un deuxième raccourci pour les tableaux est le *comptage automatique*, dans lequel vous laissez le compilateur déterminer la taille d'un tableau à partir du nombre de valeurs passées à l'initialisation :

```
int c[] = { 1, 2, 3, 4 };
```

A présent, si vous décidez d'ajouter un nouvel élément au tableau, vous ajoutez simplement une autre valeur initiale. Si vous pouvez concevoir vote code de façon à ce qu'il n'ait besoin d'être modifié qu'en un seul endroit, vous réduisez les risques d'erreur liés à la modification. Mais comment déterminez-vous la taille du tableau ? L'expression **sizeof c / sizeof *c**(taille totale du tableau divisée par la taille du premier élément) fait l'affaire sans avoir besoin d'être modifié si la taille du tableau varie. Dans le Volume 2 de ce livre (disponible gratuitement à www.BruceEckel.com), vous verrez une manière plus succinte de calculer la taille d'un tableau en utilisant les templates.:

```
for(int i = 0; i < sizeof c / sizeof *c; i++)
  c[i]++;
```

Comme les structures sont également des aggrégats, elles peuvent être initialisées de façon similaire. Comme les membres d'un **struct**type C sont tous **public**, ils peuvent être initialisés directement :

```
struct X {
  int i;
  float f;
  char c;
};
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
X x1 = { 1, 2.2, 'c' };
```

Si vous avez un tableau d'objets de ce genre, vous pouvez les initialiser en utilisant un ensemble d'accolades imbriquées pour chaque objet :

```
X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} };
```

Ici, le troisième objet est initialisé à zéro.

Si n'importe laquelle des données membres est **private**(ce qui est typiquement le cas pour une classe C++ correctement conçue), ou même si tout est **public**mais qu'il y a un contructeur, les choses sont différentes. Dans les exemples ci-dessus, les valeurs initiales étaient assignées directement aux éléments de l'aggrégat, mais les contructeurs ont une façon de forcer l'initialisation à se produire à travers une interface formelle. Donc, si vous avez un **struct**qui ressemble à cela :

```
struct Y {
  float f;
  int i;
  Y(int a);
};
```

vous devez indiquez les appels au constructeur. La meilleure approche est explicite, comme celle-ci :

```
Y y1[] = { Y(1), Y(2), Y(3) };
```

Vous avez trois objets et trois appels au constructeur. Chaque fois que vous avez un constructeur, que ce soit pour un **struct**dont tous les membres sont **public**ou bien une classe avec des données membres **private**, toutes les initialisations doivent passer par le constructeur, même si vous utilisez l'intialisation d'aggrégats.

Voici un deuxième exemple montrant un constructeur à paramètres multiples :

```
//: C06:Multiarg.cpp
// constructeur à paramètres multiples
// avec initialisation d'aggrégat
#include <iostream>
using namespace std;

class Z {
  int i, j;
public:
  Z(int ii, int jj);
  void print();
};

Z::Z(int ii, int jj) {
  i = ii;
  j = jj;
}

void Z::print() {
  cout << "i = " << i << ", j = " << j << endl;
}

int main() {
  Z zz[] = { Z(1,2), Z(3,4), Z(5,6), Z(7,8) };
  for(int i = 0; i < sizeof zz / sizeof *zz; i++)
    zz[i].print();
} ///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Remarquez qu'il semble qu'un constructeur est appelé pour chaque objet du tableau.

6.7 - Les constructeurs par défaut

Un *constructeur par défaut*est un constructeur qui peut être appelé sans arguments. Un constructeur par défaut est utilisé pour créer un “objet basique”, mais il est également important quand on fait appel au compilateur pour créer un objet mais sans donner aucun détail. Par exemple, si vous prenez la **struct Y**définit précédemment et l'utilisez dans une définition comme celle-là

```
Y y2[2] = { Y(1) };
```

Le compilateur se plaindra qu'il ne peut pas trouver un constructeur par défaut. Le deuxième objet dans le tableau veut être créé sans arguments, et c'est là que le compilateur recherche un constructeur par défaut. En fait, si vous définissez simplement un tableau d'objets **Y**,

```
Y y3[7];
```

le compilateur se plaindra parce qu'il doit avoir un constructeur par défaut pour initialiser tous les objets du tableau.

Le même problème apparaît si vous créez un objet individuel de cette façon :

```
Y y4;
```

Souvenez-vous, si vous avez un constructeur, le compilateur s'assure que la construction se produise *toujours*, quelque soit la situation.

Le constructeur par défaut est si important que *si*(et seulement si) il n'y a aucun constructeur pour une structure (**struct**ou **class**), le compilateur en créera automatiquement un pour vous. Ainsi ceci fonctionne :

```
//: C06:AutoDefaultConstructor.cpp
// Génération automatique d'un constructeur par défaut

class V {
  int i;  // privé
}; // Pas de constructeur

int main() {
  V v, v2[10];
} ///:~
```

Si un ou plusieurs constructeurs quelconques sont définis, cependant, et s'il n'y a pas de constructeur par défaut, les instances de **V**ci-dessus produiront des erreurs au moment de la compilation.

Vous pourriez penser que le constructeur généré par le compilateur doit faire une initialisation intelligente, comme fixer toute la mémoire de l'objet à zéro. Mais ce n'est pas le cas – cela ajouterait une transparence supplémentaire mais serait hors du contrôle du programmeur. Si vous voulez que la mémoire soit initialisée à zéro, vous devez le faire vous-même en écrivant le constructeur par défaut explicitement.

Bien que le compilateur crée un constructeur par défaut pour vous, le comportement du constructeur généré par le compilateur est rarement celui que vous voulez. Vous devriez traiter ce dispositif comme un filet de sécurité, mais l'employer à petite dose. Généralement vous devriez définir vos constructeurs explicitement et ne pas permettre au

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

compilateur de le faire pour vous.

6.8 - Résumé

Les mécanismes apparemment raffinés fournis par le C++ devraient vous donner un indice fort concernant l'importance critique de l'initialisation et du nettoyage dans ce langage. Lorsque Stroustrup concevait le C++, une des premières observations qu'il a faites au sujet de la productivité en C était qu'une partie significative des problèmes de programmation sont provoqués par une initialisation incorrectes des variables. Il est difficile de trouver ce genre de bogues, et des problèmes similaires concernent le nettoyage incorrect. Puisque les constructeurs et les destructeurs vous permettent de *garantir*l'initialisation et le nettoyage appropriés (le compilateur ne permettra pas à un objet d'être créé et détruit sans les appels appropriés au constructeur et au destructeur), vous obtenez un contrôle et une sûreté complets.

L'initialisation agrégée est incluse de manière semblable – elle vous empêche de faire les erreurs typiques d'initialisation avec des agrégats de types intégrés et rend votre code plus succinct.

La sûreté pendant le codage est une grande question en C++. L'initialisation et le nettoyage sont une partie importante de celui-ci, mais vous verrez également d'autres questions de sûreté au cours de votre lecture.

6.9 - Exercices

Les solutions de exercices suivants peuvent être trouvés dans le document électronique *The Thinking in C++ Annotated Solution Guide*, disponible à petit prix sur www.BruceEckel.com.

1 Ecrire une classe simple appelée **Simple**avec un constructeur qui affiche quelque chose pour vous dire qu'il a été appelé. Dans le **main( )**, créez un objet de votre classe.
2 Ajoutez un destructeur à l'Exercice 1 qui affiche un message qui vous dit qu'il a été appelé.
3 Modifiez l'Exercice 2 pour que la classe contienne un membre **int**. Modifiez le constructeur pour qu'il prenne un **int**en argument qui sera stocké dans le membre de la classe. Le constructeur et le destructeur doivent afficher la valeur de l' **int**dans leur message, afin que vous puissiez voir les objets lorsqu'ils sont créés et détruits.
4 Montrez que les destructeurs sont appelés même quand on utilise un **goto**pour sortir d'une boucle.
5 Ecrivez deux boucles **for**qui affichent les valeurs de 0 à 10. Dans la première, définissez le compteur de boucle avant la boucle **for**, et dans la seconde, définissez le compteur de boucle dans l'expression de contrôle de la boucle **for**. Dans la deuxième partie de cet exercice, donnez au compteur de la deuxième boucle le même nom que le compteur de la première et regardez la réaction du compilateur.
6 Modifiez les fichiers **Handle.h**, **Handle.cpp**, et **UseHandle.cpp**de la fin du chapitre 5 pour utiliser des constructeurs et des destructeurs.
7 Utilisez l'initialisation agrégée pour créer un tableau de **double**pour lequel vous spécifiez la taille mais sans remplir aucun élément. Affichez ce tableau en utilisant **sizeof**pour déterminer la taille du tableau. Maintenant créez un tableau de **double**en utilisante l'initialisation agrégée *and*le compteur automatique. Affichez le tableau.
8 Utilisez l'initialisation agrégée pour créer un tableau d'objets **string**. Créez une **Stack**pour contenir ces **string**s et déplacez-vous dans votre tableau, empilez chaque **string**dans votre **Stack**. Pour terminer, dépilez les **string**s de votre **Stack**et affichez chacune d'elles.
9 Illustrez le comptage automatique et l'initialisation agrégée avec un tableau d'objets de la classe que vous avez créé à l'Exercice 3. Ajoutez une fonction membre à cette classe qui affiche un message. Calculez la taille du tableau et déplacez-vous dedans en appelant votre nouvelle fonction membre.
10 Créez une classe sans aucun constructeur, et montrez que vous pouvez créer des objets avec le constructeur par défaut. Maintenant créez un constructeur particulier (avec des arguments) pour cette classe, et essayer de compiler à nouveau. Expliquez ce qui se passe.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

7 - Fonctions surchargée et arguments par défaut

Un des dispositifs importants dans n'importe quel langage de programmation est l'utilisation commode des noms.

Quand vous créez un objet (une variable), vous donnez un nom à une région de stockage. Une fonction est un nom pour une action. En composant des noms pour décrire le système actuel, vous créez un programme qu'il est plus facile pour d'autres personnes de comprendre et de changer. Cela se rapproche beaucoup de la prose d'écriture - le but est de communiquer avec vos lecteurs.

Un problème surgit en traçant le concept de la nuance dans la langue humaine sur un langage de programmation. Souvent, le même mot exprime un certain nombre de différentes significations selon le contexte. C'est-à-dire qu'un mot simple a des significations multiples - il est surchargé. C'est très utile, particulièrement quand en on vient aux différences insignifiantes.Vous dites : « lave la chemise, lave la voiture. » Il serait idiot d'être forcé de dire : « je fais un lavage de chemise à la chemise, je fais un lavage de voiture à la voiture ». Ainsi l'auditeur ne doit faire aucune distinction au sujet de l'action a exécuté. Les langues humaines ont la redondance intégrée, ainsi même si vous manquez quelques mots, vous pouvez déterminer la signification. Nous n'avons pas besoin de marques uniques - nous pouvons déduire la signification à partir du contexte.

La plupart des langages de programmation, cependant, exigent que vous ayez une marque unique pour chaque fonction. Si vous avez trois types différents de données que vous voulez imprimer : **int**, **char**, et **float**, vous devez généralement créer trois noms de fonction différents, par exemple, **print_int ()**, **print_char ()**, et **print_float ()**. Ceci ajoute du travail supplémentaire pour l'écriture du programme, et pour les lecteurs pendant qui essayent de comprendre votre code.

En C++, un autre facteur force la surcharge des noms de fonction : le constructeur. Puisque le nom du constructeur est prédéterminé par le nom de la classe, il semblerait qu'il ne puisse y avoir qu'un constructeur. Mais si vous voulez en créer un en plus du constructeur classique ? Par exemple, supposer que vous voulez établir une classe qui peut s'initialiser d'une manière standard et également par une information de lecture à partir d'un dossier. Vous avez besoin de deux constructeurs, un qui ne prennent aucun argument (le constructeur par défaut)et un qui prennent une chaine de caractère comme argument, qui est le nom du dossier pour initialiser l'objet. Tous les deux sont des constructeurs, ainsi ils doivent avoir le même nom : le nom de la classe. Ainsi, la surcharge de fonction est essentielle pour permettre au même nom de fonction - le constructeur dans ce cas-ci - d'être employée avec différents types d'argument.

Bien que la surcharge de fonction soit une nécessité pour des constructeurs, c'est une convenance générale et elle peut être employé avec n'importe quelle fonction, pas uniquement avec une fonction membre d'une classe. En plus, les fonctions surchargent les significations, si vous avez deux bibliothèques qui contiennent des fonctions du même nom, elles n'entrerons pas en conflit tant que les listes d'arguments sont différentes. Nous regarderons tous ces facteurs en détail dans ce chapitre.

Le thème de ce chapitre est l'utilisation correcte des noms de fonction. La surcharge de fonction vous permet d'employer le même nom pour différentes fonctions, mais il y a une deuxième manière de faire appel à une fonction de manière plus correcte. Que se passerait-il si vous voudriez appeler la même fonction de différentes manières ? Quand les fonctions ont de longues listes d'arguments, il peut devenir pénible d'écrire (et difficile à lire) des appels de fonction quand la plupart des arguments sont les mêmes pour tous les appels. Un dispositif utilisé généralement en C++ s'appelle les arguments par défaut. Un argument par défaut est inséré par le compilateur si on ne l'indique pas dans l'appel de fonction. Ainsi, les appels f ("bonjour"), f ("salut", 1), et f ("allo", 2, `c') peuvent tout être des appels à la même fonction. Cela pourraient également être des appels à trois fonctions surchargées, mais quand les listes d'argument sont semblables, vous voudrez habituellement un comportement semblable, qui réclame une fonction simple.

Les fonctions surchargées et les arguments par défaut ne sont pas vraiment très compliqués. Avant la fin de ce chapitre, vous saurez quand les employer et les mécanismes fondamentaux qui les mettent en application pendant

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

la compilation et l'enchaînement.

7.1 - Plus sur les décorations de nom

Dans le chapitre 4, le concept de *décoration de nom*a été présenté. Dans ce code :

```
void f();
class X { void f(); };
```

la fonction **f()**à l'intérieur de la portée de la classe X n'est pas en conflit avec la version globale de **f()**. Le compilateur réalise cette portée par différents noms internes fabriqués pour la version globale de **f()**et de **X : : f()**. Dans le chapitre 4, on a suggéré que les noms soient simplement le nom de classe « décoré » avec le nom de fonction, ainsi les noms internes que le compilateur utilise pourraient être _**f**et _**X_f**. Cependant, il s'avère que la décoration de nom pour les fonctions implique plus que le nom de classe.

Voici pourquoi. Supposez que vous vouliez surcharger deux noms de fonction :

```
void printf(char);
void printf(float);
```

Il n'importe pas qu'ils soient à l'intérieur d'une classe ou à portée globale. Le compilateur ne peut pas produire des identifiants internes uniques s'il emploie seulement la portée des noms de fonction. Vous finiriez avec _print dans les deux cas. L'idée d'une fonction surchargée est que vous employez le même nom de fonction, mais les listes d'arguments sont différentes. Ainsi, pour que la surcharge fonctionne, le compilateur doit décorer le nom de fonction avec les noms des types d'argument. Les fonctions ci-dessus, définies à la portée globale produisent des noms internes qui pourraient ressembler à quelque chose comme _print_char et _print_float. Il vaut la peine de noter qu'il n'y a aucune norme pour la manière dont les noms doivent être décorés par le compilateur, ainsi vous verrez des résultats très différents d'un compilateur à l'autre. (Vous pouvez voir à quoi ça ressemble en demandant au compilateur de générer une sortie en langage assembleur.) Ceci, naturellement, pose des problèmes si vous voulez acheter des bibliothèques compilées pour un compilateur et un éditeur de liens particuliers - mais même si la décoration de nom étaient normalisées, il y aurait d'autres barrages en raison de la manière dont les différents compilateurs produisent du code.

C'est vraiment tout ce qu'il y a dans le surchargement de fonction : vous pouvez employer le même nom de fonction pour différentes fonctions tant que les listes d'arguments sont différentes. Le compilateur décore le nom, la portée, et les listes d'argument pour produire des noms internes que lui et l'éditeur de liens emploient.

7.1.1 - Valeur de retour surchargée :

Il est commun de se demander, « pourquoi juste les portées et les listes d'arguments ? Pourquoi pas les valeurs de retour ? » Il semble à première vue qu'il soit raisonnable de décorer également la valeur de retour avec le nom interne de fonction. Alors vous pourriez surcharger sur les valeurs de retour, également :

```
void f();
int f();
```

Ceci fonctionne très bien quand le compilateur peut sans équivoque déterminer la signification du contexte, comme dans **x = f( ) ;**. Cependant, en C vous avez toujours pu appeler une fonction et ignorer la valeur de retour

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

(c'est-à-dire que vous pouvez l'appeler pour ses effets secondaires). Comment le compilateur peut-il distinguer quel appel est voulu dans ce cas-là ? La difficulté pour le lecteur de savoir quelle fonction vous appelez est probablement encore pire. La surcharge sur la valeur de retour seulement est trop subtile, et n'est donc pas permise en C++.

7.1.2 - Edition de liens sécurisée

Il y a un avantage supplémentaire à la décoration de noms. Un problème particulièrement tenace en C se produit quand un programmeur client déclare mal une fonction, ou, pire, lorsqu'il appelle une fonction sans la déclarer d'abord, et que le compilateur infère la déclaration de fonction d'après la façon dont elle est appelée. Parfois cette déclaration de fonction est correcte, mais quand elle ne l'est pas, ce peut être un bug difficile à trouver.

Puisque toutes les fonctions *doivent*être déclarées avant d'être employées en C++, le risque que ce problème se produise est considérablement diminuée. Le compilateur C++ refuse de déclarer une fonction automatiquement pour vous, ainsi il est probable que vous inclurez le fichier d'en-tête approprié. Cependant, si pour une raison quelconque vous parvenez toujours à mal déclarer une fonction, soit en la déclarant à la main soit en incluant le mauvais fichier d'en-tête (peut-être un qui est périmé), la décoration de nom fournit une sécurité qui est souvent désignée sous le nom de *édition de liens sécurisée*.

Considérez le scénario suivant, une fonction est définie dans un fichier :

```
//: C07:Def.cpp {O}
// définition de fonction
void f(int) {}
///:~
```

Dans le second fichier, la fonction est mal déclarée puis appelée :

```
//: C07:Use.cpp
//{L} Def
// Mauvaise déclaration de fonction
void f(char);

int main() {
//!  f(1); // Cause une erreur d'édition de liens
} ///:~
```

Même si vous pouvez voir que la fonction est réellement **f(int)**, le compilateur ne le sait pas parce qu'on lui a dit - par une déclaration explicite - que la fonction est **f(char)**. Ainsi, la compilation réussit. En C, l'édition de liens aurait également réussi, mais *pas*en C++. Puisque le compilateur décore les noms, la définition devient quelque chose comme **f_int**, tandis que l'utilisation de la fonction est **f_char**. Quand l'éditeur de liens essaye de trouver la référence à **f_char**, il trouve seulement **f_int**, et il envoie un message d'erreur. C'est l'édition de liens sécurisée. Bien que le problème ne se produise pas très souvent, quand cela arrive il peut être incroyablement difficile à trouver, particulièrement dans un grand projet. C'est un des cas où vous pouvez trouver facilement une erreur difficile dans un programme C simplement en le compilant avec un compilateur C++.

7.2 - Exemple de surchargement

Nous pouvons maintenant modifier les exemples vus précédemment pour employer la surcharge de fonction. Comme indiqué avant, un élément immédiatement utile à surcharger est le constructeur. Vous pouvez le voir dans la version suivante de la classe **Stash**:

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
//: C07:Stash3.h
// Fonction surchargée
#ifndef STASH3_H
#define STASH3_H

class Stash {
  int size;      // Taille pour chaque emplacement
  int quantity;  // Nombre d'espaces mémoire
  int next;      // Emplacement vide suivant
  //Tableau de bits dynamiquement alloué :
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int size); // Quantité zéro
  Stash(int size, int initQuantity);
  ~Stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH3_H ///:~
```

Le premier constructeur de **Stash()**est identique à celui d'avant, mais le second a un argument **Quantity**pour indiquer le nombre initial d'emplacements de stockage à allouer. Dans la définition, vous pouvez voir que la valeur interne de la **quantité**est placée à zéro, de même que le pointeur de **stockage**. Dans le deuxième constructeur, l'appel à **Inflate(initQuantity)**augmente **Quantity**à la taille assignée :

```
//: C07:Stash3.cpp {O}
// Fonction surchargée
#include "Stash3.h"
#include "../require.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
}

Stash::Stash(int sz, int initQuantity) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
  inflate(initQuantity);
}

Stash::~Stash() {
  if(storage != 0) {
    cout << "freeing storage" << endl;
    delete []storage;
  }
}

int Stash::add(void* element) {
  if(next >= quantity) // Suffisamment d'espace libre ?
    inflate(increment);
  // Copie l'élément dans l'emplacement mémoire,
  // Commence a l'emplacement vide suivant:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Index
}
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
void* Stash::fetch(int index) {
  require(0 <= index, "Stash::fetch (-)index");
  if(index >= next)
    return 0; // To indicate the end
  // Produit un pointeur vers l'élément désiré :
  return &(storage[index * size]);
}

int Stash::count() {
  return next; // Nombre d'éléments dans CStash
}

void Stash::inflate(int increase) {
  assert(increase >= 0);
  if(increase == 0) return;
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copie l'ancien vers le nouveau
  delete [](storage); // Libère l'ancien emplacement mémoire
  storage = b; // Pointe sur la nouvelle mémoire
  quantity = newQuantity; // Ajuste la taille
} ///:~
```

Quand vous utilisez le premier constructeur aucune mémoire n'est assignée pour le **stockage**. L'allocation se produit la première fois que vous essayez d'ajouter (**add()**en anglais, ndt) un objet et à chaque fois que le bloc de mémoire courant est excédé à l'intérieur de **add()**.

Les deux constructeurs sont illustrés dans le programme de test :

```
//: C07:Stash3Test.cpp
//{L} Stash3
// Surcharge de fonction
#include "Stash3.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize, 100);
  ifstream in("Stash3Test.cpp");
  assure(in, "Stash3Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```

L'appel du constructeur pour **stringStash**utilise un deuxième argument ; vraisemblablement vous savez quelque chose au sujet du problème spécifique que vous résolvez qui vous permet de choisir une première taille pour le **Stash**.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

7.3 - unions

Comme vous l'avez vu, la seule différence entre **struct**et **class**en C++ est que **struct**est **public**par défaut et **class**, **private**. Une **struct**peut également avoir des constructeurs et des destructeurs, comme vous pouvez vous y attendre. Mais il s'avère qu'une **union**peut aussi avoir constructeur, destructeur, fonctions membres et même contrôle d'accès. Vous pouvez encore une fois constater l'usage et le bénéfice de la surcharge dans les exemples suivants :

```
                    //: C07:UnionClass.cpp
// Unions avec constructeurs et fonctions membres
#include<iostream>
using namespace std;

union U {
private: // Contrôle d'accès également !
  int i;
  float f;
public:
  U(int a);
  U(float b);
  ~U();
  int read_int();
  float read_float();
};

U::U(int a) { i = a; }

U::U(float b) { f = b;}

U::~U() { cout << "U::~U()\n"; }

int U::read_int() { return i; }

float U::read_float() { return f; }

int main() {
  U X(12), Y(1.9F);
  cout << X.read_int() << endl;
  cout << Y.read_float() << endl;
} ///:~
```

Vous pourriez penser d'après le code ci-dessus que la seule différence entre une **union**et une **class**est la façon dont les données sont stockées (c'est-à-dire, le **int**et le **float**sont superposés sur le même espace de stockage). Cependant, une **union**ne peut pas être utilisée comme classe de base pour l'héritage, ce qui est assez limitant d'un point de vue conception orientée objet (vous étudierez l'héritage au Chapitre 14).

Bien que les fonctions membres rendent l'accès aux **union**quelque peu civilisé, il n'y a toujours aucun moyen d'éviter que le programmeur client sélectionne le mauvais type d'élément une fois que l' **union**est initialisée. Dans l'exemple ci-dessus, vous pourriez écrire **X.read_float( )**bien que ce soit inapproprié. Cependant, une **union**"sécurisée" peut être encapsulée dans une classe. Dans l'exemple suivant, observez comme **enum**clarifie le code, et comme la surcharge est pratique avec le constructeur :

```
                    //: C07:SuperVar.cpp
// Une super-variable
#include <iostream>
using namespace std;

class SuperVar {
  enum {
    character,
    integer,
    floating_point
  } vartype;  // Définit une
  union {  // Union anonyme
    char c;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    int i;
    float f;
  };
public:
  SuperVar(char ch);
  SuperVar(int ii);
  SuperVar(float ff);
  void print();
};

SuperVar::SuperVar(char ch) {
  vartype = character;
  c = ch;
}

SuperVar::SuperVar(int ii) {
  vartype = integer;
  i = ii;
}

SuperVar::SuperVar(float ff) {
  vartype = floating_point;
  f = ff;
}

void SuperVar::print() {
  switch (vartype) {
    case character:
      cout << "character: " << c << endl;
      break;
    case integer:
      cout << "integer: " << i << endl;
      break;
    case floating_point:
      cout << "float: " << f << endl;
      break;
  }
}

int main() {
  SuperVar A('c'), B(12), C(1.44F);
  A.print();
  B.print();
  C.print();
} ///:~
```

Dans le code ci-dessus, le **enum**n'a pas de nom de type (c'est une énumération sans label). C'est acceptable si vous définissez immédiatement les instances de l' **enum**, comme c'est le cas ici. Il n'y a pas besoin de faire référence au type de l' **enum**à l'avenir, si bien que le nom du type est optionnel.

L' **union**n'a aucune nom de type ni de nom de variable. Cela s'appelle une *union anonyme*, et créé de l'espace pour l' **union**mais ne requiert pas d'accéder aux éléments de l' **union**avec un nom de variable et l'opérateur point. Par exemple, si votre **union**anonyme est :

```
                    //: C07:AnonymousUnion.cpp
int main() {
  union {
    int i;
    float f;
  };
  // Accès aux membres sans utiliser de qualification :
  i = 12;
  f = 1.22;
} ///:~
```

Notez que vous accédez aux membres d'une union anonyme exactement comme s'ils étaient des variables ordinaires. La seule différence est que les deux variables occupent le même espace. Si l' **union**anonyme est dans la portée du fichier (en dehors de toute fonction ou classe) elle doit alors être déclarée **static**afin d'avoir un lien interne.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Bien que **SuperVar**soit maintenant sécurisée, son utilité est un peu douteuse car le motif pour lequel on a utilisé une **union**au début était de gagner de l'espace, et l'addition de **vartype**occupe un peu trop d'espace relativement aux données dans l' **union**, si bien que les économies sont de fait perdues. Il y a quelques alternatives pour rendre ce procédé exploitable. Vous n'auriez besoin que d'un seul **vartype**s'il contrôlait plus d'une instance d' **union**- si elles étaient toutes du même type - et il n'occuperait pas plus d'espace. Une approche plus utile est d'avoir des **#ifdef**s tout autour des codes utilisant **vartype**, ce qui peut alors garantir que les choses sont utilisées correctement durant le développement et les tests. Pour le code embarqué, l'espace supplémentaire et le temps supplémentaire peut être éliminé.

7.4 - Les arguments par défaut

Examinez les deux constructeurs de **Stash()**dans **Stash3.h**. Ils ne semblent pas si différents, n'est-ce pas ? En fait, le premier constructeur semble être un cas spécial du second avec la taille (**size**) initiale réglée à zéro. C'est un peu un gaspillage d'efforts de créer et de maintenir deux versions différentes d'une fonction semblable.

C++ fournit un remède avec les arguments par défaut. Un argument par défaut est une valeur donnée dans la déclaration que le compilateur insère automatiquement si vous ne fournissez pas de valeur dans l'appel de fonction. Dans l'exemple **Stash()**, nous pouvons remplacer les deux fonctions :

```
Stash(int size); // Quantitée zéro
Stash(int size, int initQuantity);
```

par la seule fonction :

```
Stash(int size, int initQuantity = 0);
```

La définition de **Stash(int)**est simplement enlevée -tout ce qui est nécessaire est la définition unique **Stash(int,int)**.

Maintenant, les deux définitions d'objet :

```
Stash A(100), B(100, 0);
```

Produiront exactement les mêmes résultats. Le même constructeur est appelé dans les deux cas, mais pour **A**, le deuxième argument est automatiquement substitué par le compilateur quand il voit que le premier argument est un **int**et qu'il n'y a pas de second argument. Le compilateur a vu l'argument par défaut, ainsi il sait qu'il peut toujours faire l'appel à la fonction s'il substitue ce deuxième argument, ce qui est ce que vous lui avez dit de faire en lui donnant une valeur par défaut.

Les arguments par défaut sont une commodité, comme la surcharge de fonction est une commodité. Les deux dispositifs vous permettent d'employer un seul nom de fonction dans différentes situations. La différence est qu'avec des arguments par défaut le compilateur ajoute les arguments quand vous ne voulez pas mettre vous-même. L'exemple précédent est un cas idéal pour employer des arguments par défaut au lieu de la surcharge de fonction ; autrement vous vous retrouvez avec deux fonctions ou plus qui ont des signatures semblables et des comportements semblables. Si les fonctions ont des comportements très différents, il n'est généralement pas logique d'employer des arguments par défaut (du reste, vous pourriez vous demander si deux fonctions avec des comportements très différents doivent avoir le même nom).

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Il y a deux règles que vous devez prendre en compte quand vous utilisez des arguments par défaut. En premier lieu, seul les derniers arguments peuvent être mis par défauts. C'est-à-dire que vous ne pouvez pas faire suivre un argument par défaut par un argument qui ne l'est pas. En second lieu, une fois que vous commencez à employer des arguments par défaut dans un appel de fonction particulier, tous les arguments suivants dans la liste des arguments de cette fonction doivent être par défaut (ceci dérive de la première règle).

Les arguments par défaut sont seulement placés dans la déclaration d'une fonction (typiquement placée dans un fichier d'en-tête). Le compilateur doit voir la valeur par défaut avant qu'il puisse l'employer. Parfois les gens placeront les valeurs commentées des arguments par défaut dans la définition de fonction, pour des raisons de documentation.

```
void fn(int x /* = 0 */) { // ...
```

7.4.1 - Paramètre fictif

Les arguments dans une déclaration de fonction peuvent être déclarés sans identifiants. Quand ceux-ci sont employés avec des arguments par défaut, cela peut avoir l'air un peu bizarre. Vous pouvez vous retrouver avec :

```
void f(int x, int = 0, float = 1.1);
```

En C++ vous n'avez pas besoin des identifiants dans la définition de fonction, non plus :

```
void f(int x, int, float flt) { /* ... */ }
```

Dans le corps de la fonction, on peut faire référence à **x**et **flt**, mais pas à l'argument du milieu, parce qu'il n'a aucun nom. Cependant, les appels de fonction doivent toujours fournir une valeur pour le paramètre fictif : **f(1)**ou **f(1.2.3.0)**. Cette syntaxe permet de passer l'argument comme un paramètre fictif sans l'utiliser. L'idée est que vous pourriez vouloir changer la définition de la fonction pour employer le paramètre fictif plus tard, sans changer le code partout où la fonction est appelée. Naturellement,vous pouvez accomplir la même chose en employant un argument nommé, mais si vous définissez l'argument pour le corps de fonction sans l'employer, la plupart des compilateurs vous donneront un message d'avertissement, supposant que vous avez fait une erreur logique. En omettant intentionnellement le nom d'argument, vous faite disparaître cet avertissement.

Plus important, si vous commencez à employer un argument de fonction et décidez plus tard que vous n'en avez pas besoin, vous pouvez effectivement l'enlever sans générer d'avertissements, et sans pour autant déranger de code client qui appelait la version précédente de la fonction.

7.5 - Choix entre surcharge et arguments par défaut

La surcharge et les arguments par défauts procurent tout deux un moyen pratique d'appeler les noms de fonction. Toutefois, savoir quelle technique utiliser peut être parfois peu évident. Par exemple, considérez l'outil suivant conçu pour gérer automatiquement les blocs de mémoire pour vous :

```
                    //: C07:Mem.h
#ifndef MEM_H
#define MEM_H
typedef unsigned char byte;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
class Mem {
  byte* mem;
  int size;
  void ensureMinSize(int minSize);
public:
  Mem();
  Mem(int sz);
  ~Mem();
  int msize();
  byte* pointer();
  byte* pointer(int minSize);
};
#endif // MEM_H ///:~
```

Un objet **Mem**contient un bloc de **byte**s (octets, ndt) et s'assure que vous avez suffisamment d'espace. Le constructeur par défaut n'alloue aucune place, et le second constructeur s'assure qu'il y a **sz**octets d'espace de stockage dans l'objet **Mem**. Le destructeur libère le stockage, **msize( )**vous dit combien d'octets sont présents à ce moment dans l'objet **Mem**, et **pointer( )**produit un pointeur vers l'adresse du début de l'espace de stockage (**Mem**est un outil d'assez bas niveau). Il y a une version surchargée de **pointer( )**dans laquelle les programmeurs clients peuvent dire qu'ils veulent un pointeur vers un bloc d'octets de taille minimum **minSize**, et la fonction membre s'en assure.

Le constructeur et la fonction membre **pointer( )**utilisent tout deux la fonction membre **private ensureMinSize( )**pour augmenter la taille du bloc mémoire (remarquez qu'il n'est pas sûr de retenir le résultat de **pointer( )**si la mémoire est redimensionnée).

Voici l'implémentation de la classe :

```
                    //: C07:Mem.cpp {O}
#include "Mem.h"
#include <cstring>
using namespace std;

Mem::Mem() { mem = 0; size = 0; }

Mem::Mem(int sz) {
  mem = 0;
  size = 0;
  ensureMinSize(sz);
}

Mem::~Mem() { delete []mem; }

int Mem::msize() { return size; }

void Mem::ensureMinSize(int minSize) {
  if(size < minSize) {
    byte* newmem = new byte[minSize];
    memset(newmem + size, 0, minSize - size);
    memcpy(newmem, mem, size);
    delete []mem;
    mem = newmem;
    size = minSize;
  }
}

byte* Mem::pointer() { return mem; }

byte* Mem::pointer(int minSize) {
  ensureMinSize(minSize);
  return mem;
} ///:~
```

Vous pouvez constater que **ensureMinSize( )**est la seule fonction responsable de l'allocation de mémoire, et qu'elle est utilisée par le deuxième constructeur et la deuxième forme surchargée de **pointer( )**. Dans **ensureMinSize( )**, il n'y a rien à faire si **size**est suffisamment grand. Si un espace de stockage nouveau doit être

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

alloué afin de rendre le bloc plus grand (ce qui est également le cas quand le bloc a une taille nulle après la construction par défaut), la nouvelle portion "supplémentaire" est fixée à zéro en utilisant la fonction **memset( )**de la bibliothèque C Standard, qui a été introduite au Chapitre 5. L'appel de fonction ultérieur est à la fonction **memcpy( )**de la bibliothèque C Standard, qui, dans ce cas, copie les octets existant de **mem**vers **newmem**(de manière généralement efficace). Finalement, l'ancienne mémoire est libérée et la nouvelle mémoire et la nouvelle taille sont assignées aux membres appropriés.

La classe **Mem**est conçue pour être utilisée comme un outil dans d'autres classes pour simplifier leur gestion de la mémoire (elle pourrait également être utilisée pour dissimuler un système de gestion de la mémoire plus sophistiqué, par exemple, par le système d'exploitation). On le teste ici de manière appropriée en créant une classe "string" simple :

```
                    //: C07:MemTest.cpp
// Testing the Mem class
//{L} Mem
#include "Mem.h"
#include <cstring>
#include <iostream>
using namespace std;

class MyString {
  Mem* buf;
public:
  MyString();
  MyString(char* str);
  ~MyString();
  void concat(char* str);
  void print(ostream& os);
};

MyString::MyString() {  buf = 0; }

MyString::MyString(char* str) {
  buf = new Mem(strlen(str) + 1);
  strcpy((char*)buf->pointer(), str);
}

void MyString::concat(char* str) {
  if(!buf) buf = new Mem;
  strcat((char*)buf->pointer(
    buf->msize() + strlen(str) + 1), str);
}

void MyString::print(ostream& os) {
  if(!buf) return;
  os << buf->pointer() << endl;
}

MyString::~MyString() { delete buf; }

int main() {
  MyString s("Ma chaine test");
  s.print(cout);
  s.concat(" un truc supplémentaire");
  s.print(cout);
  MyString s2;
  s2.concat("Utilise le constructeur par défaut");
  s2.print(cout);
} ///:~
```

Tout ce que vous pouvez faire avec cette classe est créer un **MyString**, concaténer du texte, et l'imprimer vers un **ostream**. La classe contient seulement un pointeur vers un **Mem**, mais notez la distinction entre le contructeur par défaut, qui initialise le pointeur à zéro, et le deuxième constructeur, qui créé un **Mem**et copie des données dedans. L'avantage du constructeur par défaut est que vous pouvez créer, par exemple, un grand tableau d'objets **MyString**vide à faible coût, comme la taille de chaque objet est seulement un pointeur et le seul délai supplémentaire du constructeur par défaut est l'assignation à zéro. Le coût d'un **MyString**commence à s'accroître seulement quand vous concaténez des données ; à ce point, l'objet **Mem**est créé, s'il ne l'a pas déjà été. Cependant, si vous utilisez le constructeur par défaut et ne concaténez jamais de données, l'appel au destructeur

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

est toujours sûr car appeler **delete**pour zéro est défini de telle façon qu'il n'essaie pas de libérer d'espace ou ne cause pas de problème autrement.

Si vous examinez ces deux constructeurs, il pourrait sembler au premier abord que c'est un bon candidat pour des arguments par défaut. Toutefois, si vous abandonnez le constructeur par défaut et écrivez le constructeur restant avec un argument par défaut :

```
MyString(char* str = "");
```

tout fonctionnera correctement, mais vous perdrez le bénéfice d'efficacité puisqu'un objet **Mem**sera systématiquement créé. Pour recouvrer l'efficacité, vous devez modifier le constructeur :

```
MyString::MyString(char* str) {
  if(!*str) { // Pointe vers une chaîne vide
    buf = 0;
    return;
  }
  buf = new Mem(strlen(str) + 1);
  strcpy((char*)buf->pointer(), str);
}
```

Ceci signifie, en effet, que la valeur par défaut devient un signal qui entraîne l'exécution d'une partie différente du code de celui utilisé pour une valeur quelconque. Bien que cela semble sans trop d'importance avec un constructeur aussi petit que celui-ci, en général cette pratique peut causer des problèmes. Si vous devez *chercher*le constructeur par défaut plutôt que l'exécuter comme une valeur ordinaire, cela devrait être une indication que vous finirez avec deux fonctions différentes dans un seul corps de fonction : une version pour le cas normal et une par défaut. Vous feriez aussi bien de la séparer en deux corps de fonction et laisser le compilateur faire la sélection. Ceci entraîne une légère (mais généralement imperceptible) augmentation d'efficacité, parce que l'argument supplémentaire n'est pas passé et le code conditionnel supplémentaire n'est pas exécuté. Plus important, vous conservez le code de deux fonctions différentes *dans*deux fonctions distinctes plutôt que de les combiner en une seule utilisant les arguments par défaut, ce qui résulte en une maintenance plus facile, surtout si les fonctions sont grandes.

D'un autre point de vue, considérez le cas de la classe **Mem**. Si vous regardez les définitions des deux constructeurs et des deux fonctions **pointer( )**, vous constatez qu'utiliser les arguments par défaut, dans les deux cas, n'entraînera aucune modification de la définition de la fonction membre. Ainsi, la classe peut devenir sans problème :

```
//: C07:Mem2.h
#ifndef MEM2_H
#define MEM2_H
typedef unsigned char byte;

class Mem {
  byte* mem;
  int size;
  void ensureMinSize(int minSize);
public:
  Mem(int sz = 0);
  ~Mem();
  int msize();
  byte* pointer(int minSize = 0);
};
#endif // MEM2_H ///:~
```

Notez qu'un appel à **ensureMinSize(0)**sera toujours assez efficace.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Bien que dans chacun de ces cas j'ai fondé une part du processus de décision sur la question de l'efficacité, vous devez faire attention à ne pas tomber dans le piège de ne penser qu'à l'efficacité (aussi fascinante soit-elle). La question la plus importante dans la conception de classes est l'interface de la classe (ses membres **public**, qui sont disponibles pour le programmeur client). Si cela produit une classe qui est facile à utiliser et réutiliser, alors c'est un succès ; vous pouvez toujours faire des réglages pour l'efficacité si nécessaire, mais l'effet d'une classe mal conçue parce que le programmeur se concentre trop sur la question de l'efficacité peut être terrible. Votre souci premier devrait être que l'interface ait un sens pour ceux qui l'utilisent et qui lisent le code résultant. Remarquez que dans **MemTest.cpp**l'utilisation de **MyString**ne change pas selon qu'on utilise un constructeur par défaut ou que l'efficacité soit grande ou non.

7.6 - Résumé

D'une manière générale, vous ne devriez pas utiliser un argument par défaut comme un signal pour l'exécution conditionnelle de code. A la place, vous devriez diviser la fonction en deux ou plusieurs fonctions surchargées si vous le pouvez. Un argument par défaut devrait être une valeur utilisée ordinairement. C'est la valeur la plus probablement utilisée parmi toutes celles possibles, si bien que les programmeurs clients peuvent généralement l'ignorer ou l'utiliser uniquement s'ils veulent lui donner une valeur différente de la valeur par défaut.

L'argument par défaut est utilisé pour rendre l'appel aux fonctions plus facile, en particulier lorsque ces fonctions ont beaucoup d'arguments avec des valeurs types. Non seulement il devient plus facile d'écrire les appels, mais il est également plus facile de les lire, spécialement si le créateur de la classe peut ordonner les arguments de telle façon que les valeurs par défauts les moins modifiées apparaissent en dernier dans la liste.

Une utilisation particulièrement importante des arguments pas défaut est quand vous commencez l'utilisation d'une fonction avec un ensemble d'arguments, et après l'avoir utilisée quelques temps, vous découvrez que vous devez utiliser plus d'arguments. En donnant une valeur par défaut à tous les nouveaux arguments, vous garantissez que tout le code client utilisant l'interface précédente n'est pas perturbé.

7.7 - Exercices

Les solutions des exercices suivants peuvent être trouvées dans le document électronique *The Thinking in C++ Annotated Solution Guide*, disponible à petit prix sur www.BruceEckel.com.

1 Créez une classe **Text**contenant un objet **string**pour stocker le texte d'un fichier. Munissez-le de deux constructeurs : un constructeur par défaut et un constructeur prenant un argument **string**qui est le nom du fichier à ouvrir. Quand le second constructeur est utilisé, ouvrez le fichier et lisez le contenu dans le membre **string**de l'objet. Ajoutez une fonction membre **contents( )**retournant la **string**pour (par exemple) qu'elle puisse être affichée. Dans le **main( )**, ouvrez un fichier en utilisant **Text**et affichez le contenu.

2 Créez une classe **Message**avec un constructeur qui prend un unique argument **string**avec une valeur par défaut. Créez un membre privé **string**, et dans le constructeur assignez simplement l'argument **string**à votre **string**interne. Créez deux fonctions membres surchargées appelées **print( )**: l'une qui ne prend aucun argument et affiche simplement le message stocké dans l'objet, et l'autre qui prend un argument **string**, lequel sera affiché en plus du message stocké. Cela a t-il plus de sens d'utiliser cette approche plutôt que celle utilisée pour le constructeur ?

3 Déterminez comment générer l'assembly avec votre compilateur, et faites des expériences pour déduire le sytème de décoration des noms.

4 Créez une classe contenant quatre fonctions membres avec 0, 1, 2 et 3 arguments **int**respectivement. Créez un **main( )**qui crée un objet de votre classe et appelle chacune des fonctions membres. Maintenant modifiez la classe pour qu'elle n'ait maintenant qu'une seule fonction membre avec tous les arguments possédant une valeur pas défaut. Cela change-t-il votre **main( )**?

5 Créez une fonction avec deux arguments et appelez-la dans le **main( )**. Maintenant rendez l'un des argument “muet” (sans identifiant) et regardez si votre appel dans **main( )**change.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

6 Modifiez **Stash3.h**et **Stash3.cpp**pour utiliser les arguments par défaut dans le constructeur. Testez le constructeur en créant deux versions différentes de l'objet **Stash**.
7 Créez une nouvelle version de la classe **Stack**(du Chapitre 6) contenant le constructeur par défaut comme précédemment, et un second constructeur prenant comme argument un tableau de pointeurs d'objets et la taille de ce tableau. Ce constructeur devra parcourir le tableau et empiler chaque pointeur sur la **Stack**. Testez la classe avec un tableau de **string**.
8 Modifiez **SuperVar**pour qu'il y ait des **#ifdef**autour de tout le code **vartype**comme décrit dans la section sur les **enum**. Faites de **vartype**une énumération régulière et privée (sans instance) et modifiez **print( )**pour qu'elle requiert un argument **vartype**pour lui dire quoi faire.
9 Implémentez **Mem2.h**et assurez-vous que la classe modifiée fonctionne toujours avec **MemTest.cpp**.
10 Utilisez la classe **Mem**pour implémenter **Stash**. Notez que parce que l'implémentation est privée et ainsi cachée au programmeur client, le code de test n'a pas besoin d'être modifié.
11 Dans la classe **Mem**, ajoutez une fonction membre **bool moved( )**qui prend le résultat d'un appel à **pointer( )**et vous indique si le pointeur a bougé (en raison d'une réallocation). Ecrivez un **main( )**qui teste votre fonction membre **moved( )**. Cela a t-il plus de sens d'utiliser quelque chose comme **moved( )**ou d'appeler simplement **pointer( )**chaque fois que vous avez besoin d'accéder à la mémoire d'un **Mem**?

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

8 - Constantes

Le concept de *constante*(exprimé par le mot-clef **const**) a été créé pour permettre au programmeur de séparer ce qui change de ce qui ne change pas. Cela assure sécurité et contrôle dans un projet C++.

Depuis son apparition, **const**a eu différentes raisons d'être. Dans l'intervalle il est passé au C où son sens a été modifié. Tout ceci peut paraître un peu confus au premier abord, et dans ce chapitre vous apprendrez quand, pourquoi et comment utiliser le mot-clef **const**. A la fin, se trouve une discussion de **volatile**qui est un proche cousin de **const**(parce qu'ils concernent tout deux le changement) et a une syntaxe identique.

La première motivation pour **const**semble avoir été d'éliminer l'usage de la directive de pré-processeur **#define**pour la substitution de valeurs. Il a depuis été utilisé pour les pointeurs, les arguments de fonction, les types retournés, les objets de classe et les fonctions membres. Tous ces emplois ont des sens légèrement différents mais conceptuellement compatibles et seront examinés dans différentes sections de ce chapitre.

8.1 - Substitution de valeurs

En programmant en C, le préprocesseur est utilisé sans restriction pour créer des macros et pour substituer des valeurs. Puisque le préprocesseur fait simplement le remplacement des textes et n'a aucune notion ni service de vérification de type, la substitution de valeur du préprocesseur présente les problèmes subtiles qui peuvent être évités en C++ en utilisant des variables **const**.

L'utilisation typique du préprocesseur pour substituer des valeurs aux noms en C ressemble à ceci :

```
#define BUFSIZE 100
```

BUFSIZEest un nom qui existe seulement pendant le prétraitement, donc il n'occupe pas d'emplacement mémoire et peut être placé dans un fichier d'en-tête pour fournir une valeur unique pour toutes les unités de traduction qui l'utilisent. Il est très important pour l'entretien du code d'utiliser la substitution de valeur au lieu de prétendus « nombres magiques. » Si vous utilisez des nombres magiques dans votre code, non seulement le lecteur n'a aucune idée d'où ces nombres proviennent ou de ce qu'ils représentent, mais si vous décidez de changer une valeur, vous devrez le faire à la main, et vous n'avez aucune façon de savoir si vous n'oubliez pas une de vos valeurs (ou que vous en changez accidentellement une que vous ne devriez pas changer).

La plupart du temps, **BUFSIZE**se comportera comme une variable ordinaire, mais pas toujours. En outre, il n'y a aucune information sur son type. Cela peut masquer des bogues qu'il est très difficile de trouver. C++ emploie **const**pour éliminer ces problèmes en déplaçant la substitution de valeur dans le domaine du compilateur. Maintenant vous pouvez dire :

```
const int bufsize = 100;
```

Vous pouvez utiliser **bufsize**partout où le compilateur doit connaître la valeur pendant la compilation. Le compilateur peut utiliser **bufsize**pour exécuter le *remplacement des constantes*, ça signifie que le compilateur ramènera une expression constante complexe à une simple valeur en exécutant les calculs nécessaires lors de la compilation. C'est particulièrement important pour des définitions de tableau :

```
char buf[bufsize];
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Vous pouvez utiliser **const**pour tous les types prédéfinis (**char**, **int**, **float**, et **double**) et leurs variantes (aussi bien que des objets d'une classe, comme vous le verrez plus tard dans ce chapitre). En raison des bogues subtiles que le préprocesseur pourrait présenter, vous devriez toujours employer **const**au lieu de la substitution de valeur **#define**.

8.1 - Constantes dans les fichiers d'en-tête

Pour utiliser **const**au lieu de **#define**, vous devez pouvoir placer les définitions **const**à l'intérieur des fichiers d'en-tête comme vous pouvez le faire avec **#define**. De cette façon, vous pouvez placer la définition d'un **const**dans un seul endroit et la distribuer aux unités de traduction en incluant le fichier d'en-tête. Un **const**en C++ est par défaut à liaison interne ; c'est-à-dire qu'il est visible uniquement dans le fichier où il est défini et ne peut pas être vu par d'autres unités de traduction au moment de l'édition de lien. Vous devez toujours affecter une valeur à un const quand vous le définissez, *excepté*quand vous utilisez une déclaration explicite en utilisant **extern**:

```
extern const int bufsize;
```

Normalement, le compilateur de C++ évite de créer un emplacement pour un **const**, mais garde plutôt la définition dans sa table de symbole. Toutefois quand vous utilisez **extern**avec **const**, vous forcez l'allocation d'un emplacement de stockage (cela vaut également pour certains autres cas, tels que si vous prennez l'adresse d'un **const**). L'emplacement doit être affecté parce que **extern**indique "utiliser la liaison externe", ce qui signifie que plusieurs unités de traduction doivent pouvoir se rapporter à l'élément, ce qui lui impose d'avoir un emplacement.

Dans le cas ordinaire, si **extern**n'est pas indiqué dans la définition, aucun emplacement n'est alloué. Quand **const**est utilisé, il subit simplement le remplacement au moment de la compilation.

L'objectif de ne jamais allouer un emplacement pour un **const**échoue également pour les structures complexes. A chaque fois que le compilateur doit allouer un emplacement, le remplacement des **const**n'a pas lieu (puisque le compilateur n'a aucun moyen pour connaître la valeur de cette variable - si il pouvait le savoir, il n'aurait pas besoin d'allouer l'emplacement).

Puisque le compilateur ne peut pas toujours éviter d'allouer l'emplacement d'un **const**, les définitions de **const** *doivent*être à liaison interne, c'est-à-dire, la liaison uniquement *dans*cette unité de compilation. Sinon, des erreurs d'édition de liens se produiraient avec les **const**s complexes parce qu'elles entraineraient l'allocation de l'emplacement dans plusieurs fichiers cpp. L'éditeur de liens verrait alors la même définition dans plusieurs fichiers, et se plaindrait. Puisqu'un **const**est à liaison interne, l'éditeur de liens n'essaye pas de relier ces définitions à travers différentes unités de compilation, et il n'y a aucune collision. Avec les types prédéfinis, qui sont employés dans la majorité de cas impliquant des expressions constantes, le compilateur peut toujours exécuter le remplacement des **const**.

8.1 - Consts de sécurité

L'utilisation du **const**n'est pas limitée à remplacer les **#define**dans des expressions constantes. Si vous initialisez une variable avec une valeur qui est produite au moment de l'exécution et vous savez qu'elle ne changera pas pour la durée de vie de cette variable, c'est une bonne pratique de programmation que de la déclarer **const**de façon à ce que le compilateur donne un message d'erreur si vous essayez accidentellement de la changer. Voici un exemple :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
//: C08:Safecons.cpp
// Usage de const pour la sécurité
#include <iostream>
using namespace std;

const int i = 100;  // constante typique
const int j = i + 10; // valeur issue d'une expression constante
long address = (long)&j; // force l'allocation
char buf[j + 10]; // encore une expression constante

int main() {
  cout << "type a character & CR:";
  const char c = cin.get(); // ne peut pas changer
  const char c2 = c + 'a';
  cout << c2;
  // ...
} ///:~
```

Vous pouvez voir que **i**est un **const**au moment de la compilation, mais **j**est calculé à partir de **i**. Cependant, parce que **i**est un **const**, la valeur calculée pour **j**vient encore d'une expression constante et est elle-même une constante au moment de la compilation. La ligne suivante exige l'adresse de **j**et force donc le compilateur à allouer un emplacement pour **j**. Pourtant ceci n'empêche pas l'utilisation de **j**dans la détermination de la taille de **buf**parce que le compilateur sait que **j**est un **const**et que la valeur est valide même si un emplacement a été alloué pour stocker cette valeur à un certain endroit du programme.

Dans le **main()**, vous voyez un genre différent de **const**avec la variable **c**parce que la valeur ne peut pas être connue au moment de la compilation. Ceci signifie que l'emplacement est exigé, et le compilateur n'essaye pas de conserver quoi que ce soit dans sa table de symbole (le même comportement qu'en C). L'initialisation doit avoir lieu à l'endroit de la définition et, une fois que l'initialisation a eu lieu, la valeur peut plus être changée. Vous pouvez voir que **c2**est calculé à partir de **c**et aussi que la portée fonctionne pour des **const**comme pour n'importe quel autre type - encore une autre amélioration par rapport à l'utilisation du **#define**.

En pratique, si vous pensez qu'une valeur ne devrait pas être changée, vous devez la rendre **const**. Ceci fournit non seulement l'assurance contre les changements accidentels, mais en plus il permet également au compilateur de produire un code plus efficace par l'élimination de l'emplacement de la variable et la suppression de lectures mémoire.

8.1 - Aggrégats

Il est possible d'utiliser **const**pour des agrégats, mais vous êtes pratiquement assurés que le compilateur ne sera pas assez sophistiqué pour maintenir un agrégat dans sa table de symbole, ainsi l'emplacement sera créé. Dans ces situations, **const**signifie « un emplacement mémoire qui ne peut pas être changé ». Cependant, la valeur ne peut pas être utilisée au moment de la compilation parce que le compilateur ne connait pas forcement la valeur au moment de la compilation. Dans le code suivant, vous pouvez voir les instructions qui sont illégales :

```
//: C08:Constag.cpp
// Constantes et aggrégats
const int i[] = { 1, 2, 3, 4 };
//! float f[i[3]]; // Illégal
struct S { int i, j; };
const S s[] = { { 1, 2 }, { 3, 4 } };
//! double d[s[1].j]; // Illégal
int main() {} ///:~
```

Dans une définition de tableau, le compilateur doit pouvoir produire du code qui déplace le pointeur de pile selon le tableau. Dans chacune des deux définitions illégales ci-dessus, le compilateur se plaint parce qu'il ne peut pas trouver une expression constante dans la définition du tableau.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

8.1 - différences avec le C

Les constantes ont été introduites dans les premières versions du C++ alors que les spécifications du standard du C étaient toujours en cours de finition. Bien que le comité du C ait ensuite décidé d'inclure **const**en C, d'une façon ou d'une autre cela prit la signification de « une variable ordinaire qui ne peut pas être changée. » En C, un **const**occupe toujours un emplacement et son nom est global. Le compilateur C ne peut pas traiter un **const**comme une constante au moment de la compilation. En C, si vous dites :

```
const int bufsize = 100;
char buf[bufsize];
```

vous obtiendrez une erreur, bien que cela semble une façon de faire rationnelle. Puisque **bufsize**occupe un emplacement quelque part, le compilateur C ne peut pas connaître la valeur au moment de la compilation. Vous pouvez dire en option :

```
const int bufsize;
```

En C, mais pas en C++, et le compilateur C l'accepte comme une déclaration indiquant qu'il y a un emplacement alloué ailleurs. Puisque C utilise la liaison externe pour les **const**s, cela à un sens. C++ utilise la liaison interne pour les **const**s ainsi si vous voulez accomplir la même chose en C++, vous devez explicitement imposer la liaison externe en utilisant **extern**:

```
extern const int bufsize; // Déclaration seule
```

Cette ligne fonctionne aussi en C.

en C++, un **const**ne crée pas nécessairement un emplacement. En C un **const**crée toujours un emplacement. Qu'un emplacement soit réservé ou non pour un **const**en C++ dépend de la façon dont il est utilisé. En général, si un **const**est simplement utilisé pour remplacer un nom par une valeur (comme vous le feriez avec **#define**), alors un emplacement n'a pas besoin d'être créé pour le **const**. Si aucun emplacement n'est créé (cela dépend de la complexité du type de données et de la sophistication du compilateur), les valeurs peuvent ne pas être intégrées dans le code pour une meilleure efficacité après la vérification de type, pas avant comme pour **#define**. Si, toutefois, vous prenez l'adresse d'un **const**(même sans le savoir, en le passant à une fonction qui prend en argument une référence) ou si vous le définissez comme **extern**, alors l'emplacement est créé pour le **const**.

En C++, un **const**qui est en dehors de toutes les fonctions possède une portée de fichier (c.-à-d., qu'il est invisible en dehors du fichier). Cela signifie qu'il est à liaison interne. C'est très différent de toutes autres identifieurs en C++ (et des **const**du C !) qui ont la liaison externe. Ainsi, si vous déclarez un **const**du même nom dans deux fichiers différents et vous ne prenez pas l'adresse ou ne définissez pas ce nom comme **extern**, le compilateur C++ idéal n'allouera pas d'emplacement pour le **const**, mais le remplace simplement dans le code. Puisque le **const**a impliqué une portée fichier, vous pouvez la mettre dans les fichiers d'en-tête C++ sans conflits au moment de l'édition de liens.

Puisqu'un **const**en C++ a la liaison interne, vous ne pouvez pas simplement définir un **const**dans un fichier et le mettre en référence **extern**dans un autre fichier. Pour donner à un **const**une liaison externe afin qu'il puisse être référencé dans un autre fichier, vous devez explicitement le définir comme **extern**, comme ceci :

```
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
extern const int x = 1;
```

Notez qu'en lui donnant une valeur d'initialisation et en la déclarant externe vous forcez la création d'un emplacement pour le **const**(bien que le compilateur a toujours l'option de faire le remplacement de constante ici). L'initialisation établit ceci comme une définition, pas une déclaration. La déclaration :

```
extern const int x;
```

En C++ signifie que la définition existe ailleurs (encore un fois, ce n'est pas nécessairement vrai en C). Vous pouvez maintenant voir pourquoi C++ exige qu'une définition de **const**possède son initialisation : l'initialisation distingue une déclaration d'une définition (en C c'est toujours une définition, donc aucune initialisation n'est nécessaire). Avec une déclaration **extern const**, le compilateur ne peut pas faire le remplacement de constante parce qu'il ne connait pas la valeur.

L'approche du C pour les **const**n'est pas très utile, et si vous voulez utiliser une valeur nommée à l'intérieur d'une expression constante (une qui doit être évalué au moment de la compilation), le C vous *oblige*presque à utiliser **#define**du préprocesseur.

8.2 - Les pointeurs

Les pointeurs peuvent être rendus **const**. Le compilateur s'efforcera toujours d'éviter l'allocation de stockage et remplacera les expressions constantes par la valeur appropriée quand il aura affaire à des pointeurs **const**, mais ces caractéristiques semblent moins utiles dans ce cas. Plus important, le compilateur vous signalera si vous essayez de modifier un pointeur **const**, ce qui améliore grandement la sécurité.

Quand vous utilisez **const**avec les pointeurs, vous avez deux options : **const**peut être appliqué à ce qui est pointé, ou bien **const**peut concerner l'adresse stockée dans le pointeur lui-même. La syntaxe, ici, paraît un peu confuse au début mais devient commode avec la pratique.

8.2.1 - Pointeur vers const

L'astuce avec la définition de pointeur, comme avec toute définition compliquée, est de la lire en partant de l'identifiant jusque vers la fin de la définition. Le mot-clef **const**est lié à l'élément dont il est le "plus proche". Donc, si vous voulez éviter tout changement à l'objet pointé, vous écrivez une définition ainsi :

```
const int* u;
```

En partant de l'identifiant, nous lisons “ **u**est un pointeur, qui pointe vers un **const int**.” Ici, il n'y a pas besoin d'initialisation car vous dites que **u**peut pointer vers n'importe quoi (c'est-à-dire qu'il n'est pas **const**), mais l'élément vers lequel il pointe ne peut pas être changé.

Voici la partie un peu déroutante. Vous pourriez pensez que pour rendre le pointeur lui-même inmodifiable, c'est-à-dire pour éviter toute modification de l'adresse contenue dans **u**, il suffit de déplacer le **const**de l'autre côté du **int**comme ceci :

```
int const* v;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Ce n'est pas du tout idiot de penser que ceci devait se lire “ **v**est un pointeur **const**vers un **int**.”. Toutefois, la vraie façon de le lire est “ **v**est un pointeur ordinaire vers un **int**qui se trouve être **const**.”. Donc, le **const**s'est lié au **int**à nouveau, et l'effet est le même que dans la définition précédente. Le fait que ces deux définitions soient les mêmes est un peu déroutant ; pour éviter cette confusion au lecteur, vous devriez probablement n'utiliser que la première forme.

8.2.2 - Pointeur const

Pour rendre le pointeur lui-même **const**, vous devez placer le mot-clef **const**à droite de l'étoile *****, comme ceci :

```
                        int d = 1;
int* const w = &amp;d;
```

*A présent*on lit : “ **w**est un pointeur de type **const**, qui pointe vers un **int**.”. Comme le pointeur lui-même est à présent le **const**, le compilateur impose qu'il lui soit assignée une valeur initiale qui restera inchangée durant toute la vie de ce pointeur. Il est possible, par contre, de modifier ce vers quoi pointe cette valeur en disant :

```
                        *w = 2;
```

Vous pouvez également créer un pointeur **const**vers un objet **const**en utilisant une de ces deux syntaxes licites :

```
                        int d = 1;
const int* const x = &d;  // (1)
int const* const x2 = &d; // (2)
```

A présent, ni le pointeur ni l'objet ne peuvent être modifiés.

Certaines personnes affirment que la deuxième forme est plus cohérente parce que le **const**est toujours placé à droite de ce qu'il modifie. Vous n'avez qu'à décider ce qui est le plus clair pour votre manière de coder.

Voici les lignes ci-dessus dans un fichier compilable :

```
                        //: C08:ConstPointers.cpp
const int* u;
int const* v;
int d = 1;
int* const w = &d;
const int* const x = &d;  // (1)
int const* const x2 = &d; // (2)
int main() {} ///:~
```

Formatage

Ce livre insiste sur le fait de ne mettre qu'une définition de pointeur par ligne, et d'initialiser chaque pointeur au point de définition lorsque c'est possible. A cause de cela, le style de formatage qui consiste à “attacher” le ‘ *****’ au type de donnée est possible :

```
                        int* u = &amp;i;
```

comme si **int***était un type *en soi*. Ceci rend le code plus facile à comprendre, mais malheureusement ce n'est pas

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

comme cela que les choses fonctionnent vraiment. Le ‘ ***’ est en fait attaché à l'identifiant, pas au type. Il peut être placé n'importe où entre le nom et l'identifiant. Vous pourriez donc écrire cela :

```
int *u = &i, v = 0;
```

qui crée un **int* u**, comme précédemment, et un **int v**qui n'est pas un pointeur. Comme les lecteurs trouvent souvent cela déroutant, il est recommandable de suivre la forme proposée dans ce livre.

8.2.3 - Assignation et vérification de type

Le C++ est très spécial en ce qui concerne la vérification de type, et ce jusqu'à l'assignation de pointeur. Vous pouvez assigner l'adresse d'un objet non- **const**à un pointeur **const**parce que vous promettez simplement de ne pas changer quelque chose qui peut se changer. Par contre, vous ne pouvez pas assigner l'adresse d'un objet **const**à un pointeur non- **const**car alors vous dites que vous pourriez changer l'objet via le pointeur. Bien sûr, vous puvez toujours utiliser la conversion de type (*cast*, ndt) pour forcer une telle assignation, mais c'est une mauvaise habitude de programmation car vous brisez la **const**ance de l'objet ainsi que la promesse de sécurité faite par le **const**. Par exemple :

```
//: C08:PointerAssignment.cpp
int d = 1;
const int e = 2;
int* u = &d; // OK -- d n'est pas const
//! int* v = &e; // Illicite -- e est const
int* w = (int*)&e; // Licite mais mauvaise habitude
int main() {} ///:~
```

Bien que C++ aide à prévenir les erreurs, il ne vous protège pas de vous-même si vous voulez enfreindre les mécanismes de sécurité.

Les tableaux de caractères littéraux

Le cas où la **const**ance stricte n'est pas imposée, est le cas des tableaux de caractères littéraux. Vous pouvez dire :

```
char* cp = "Salut";
```

et le compilateur l'acceptera sans se plaindre. C'est techniquement une erreur car un tableau de caractères littéraux (ici, **“Salut”**) est créé par le compilateur comme un tableau de caractères constant, et le résultat du tableau de caractère avec des guillemets est son adresse de début en mémoire. Modifier n'importe quel caractère dans le tableau constitue une erreur d'exécution, bien que tous les compilateurs n'imposent pas cela correctement.

Ainsi, les tableaux de caractères littéraux sont réellement des tableaux de caractères constants. Evidemment, le compilateur vous laisse vous en tirer en les traitant comme des non- **const**car il y a beaucoup de code C qui repose là-dessus. Toutefois, si vous tentez de modifier la valeur dans un tableau de caractères littéral, le comportement n'est pas défini, bien que cela fonctionnera probablement sur beaucoup de machines.

Si vous voulez être capable de modifier la chaîne, mettez-là dans un tableau :

```
char cp[] = "howdy";
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Comme les compilateurs ne font généralement pas la différence, on ne vous rappelera pas d'utiliser cette dernière forme et l'argument devient relativement subtil.

8.3 - Arguments d'une fonction & valeurs retournées

L'utilisation de **const**pour spécifier les arguments d'une fonction et les valeurs retournées peut rendre confus le concept de constantes. Si vous passez des objets *par valeur*, spécifier **const**n'a aucune signification pour le client (cela veut dire que l'argument passé ne peut pas être modifié dans la fonction). Si vous retournez par valeur un objet d'une type défini par l'utilisateur en tant que **const**, cela signifie que la valeur retournée ne peut pas être modifiée. Si vous passez et retournez des *adresses*, **const**garantit que la destination de l'adresse ne sera pas modifiée.

8.3.1 - Passer par valeur const

Vous pouvez spécifier que les arguments de fonction soient **const**quand vous les passez par valeur, comme dans ce cas

```
                        void f1(const int i) {
    i++; // Illégal - erreur à la compilation
    }
```

mais qu'est-ce que cela veut dire ? Vous faites la promesse que la valeur originale de la variable ne sera pas modifiée par la fonction **f1( )**. Toutefois, comme l'argument est passé par valeur, vous faites immédiatement une copie de la variable originale, donc la promesse au client est implicitement tenue.

Dans la fonction, **const**prend le sens : l'argument ne peut être changé. C'est donc vraiment un outil pour le créateur de la fonction, et pas pour l'appelant.

Pour éviter la confusion à l'appelant, vous pouvez rendre l'argument **const** *à l'intérieur*de la fonction, plutôt que dans la liste des arguments. Vous pourriez faire cela avec un pointeur, mais une syntaxe plus jolie est utilisable avec les *références*, sujet qui sera développé en détails au Chapitre 11. Brièvement, une référence est comme un pointeur constant qui est automatiquement dé-référencé, et qui a donc l'effet d'être un alias vers un objet. Pour créer une référence, on utilise le **&**dans la définition. Finalement, la définition de fonction claire ressemble à ceci :

```
                        void f2(int ic) {
    const int& i = ic;
    i++;  // Illégal - erreur à la compilation}
  }
```

Une fois encore, vous obtiendrez un message d'erreur, mais cette foi-ci la **const**ance de l'objet local ne fait pas partie de la signature de la fonction ; elle n'a de sens que pour l'implémentation de la fonction et est ainsi dissimulée au client.

8.3.2 - Retour par valeur const

Une réalité similaire s'applique aux valeurs retournées. Si vous dites qu'une valeur retournée par une fonction est **const**:

```
                        const int g();
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

vous promettez que la valeur originale (dans la portée de la fonction) ne sera pas modifiée. Une fois encore, comme vous la retournez par valeurs, elle est copiée si bien que la valeur originale ne pourra jamais être modifiée *via*la valeur retournée.

Au premier abord, cela peut faire paraître la déclaration **const**dénuée de sens. Vous pouvez constater le manque apparent d'effet de retourner des **const**par valeurs dans cet exemple :

```
                        //: C08:Constval.cpp
// Retourner des const par valeur
// n'a pas de sens pour les types prédéfinis

int f3() { return 1; }
const int f4() { return 1; }

int main() {
  const int j = f3(); // Marche bien
  int k = f4(); // Mais celui-ci marche bien également !
} ///:~
```

Pour les types prédéfinis, cela n'a aucune importance que vous retourniez une valeur come **const**, donc vous devriez éviter la confusion au programmeur client et oublier **const**quand vous retournez un type prédéfini par valeur.

Retourner un **const**par valeur devient important quand vous traitez des types définis par l'utilisateur. Si une fonction retourne comme **const**un objet de classe , la valeur de retour de cette fonction ne peut pas être une lvalue (c'est-à-dire, elle ne peut être assignée ou modifiée autrement). Par exemple :

```
                        //: C08:ConstReturnValues.cpp
// Constante retournée par valeur
// Le résultat ne peut être utilisé comme une lvalue

class X {
  int i;
public:
  X(int ii = 0);
  void modify();
};

X::X(int ii) { i = ii; }

void X::modify() { i++; }

X f5() {
  return X();
}

const X f6() {
  return X();
}

void f7(X& x) { // Passé par référence pas const
  x.modify();
}

int main() {
  f5() = X(1); // OK -- valeur de retour non const
  f5().modify(); // OK
//!  f7(f5()); // Provoque warning ou erreur
// Cause des erreurs de compilation :
//!  f7(f5());
//!  f6() = X(1);
//!  f6().modify();
//!  f7(f6());
} ///:~
```

f5( )renvoie un objet non- **const**, alors que **f6( )**renvoie un objet **const X**. Seule la valeur retournée qui n'est pas

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

constpeut être utilisée comme une lvalue. Ainsi, il est important d'utiliser **const**quand vous retournez un objet par valeur si vous voulez éviter son utilisation comme lvalue.

La raison pour laquelle **const**ne veut rien dire quand vous renvoyez un type prédéfini par valeur est que le compilateur empêche déjà qu'il soit utilisé comme lvalue (parce que c'est toujours une valeur, et pas une variable). C'est seulement lorsque que vous renvoyez des types définis par l'utilisateur par valeur que cela devient un problème.

La fonction **f7( )**prend son argument comme une *référence*(un moyen supplémentaire de manipuler les adresses en C++ qui sera le sujet du chapitre 11) qui n'est pas de type **const**. C'est en fait la même chose que d'utiliser un pointeur qui ne soit pas non plus de type **const**; seule la syntaxe est différente. La raison pour laquelle ce code ne compilera pas en C++ est qu'il y a création d'une variable temporaire.

Les variables temporaires

Parfois, pendant l'évaluation d'une expression, le compilateur doit créer des *objets temporaires*. Ce sont des objets comme n'importe quels autres : ils nécessitent un stockage et doivent être construits et détruits. La différence est que vous ne les voyez jamais ; le compilateur a la charge de décider s'ils sont nécessaires et de fixer les détails de leur existence. Mais il faut noter une chose en ce qui concerne les variables temporaires : elles sont automatiquement **const**. Comme vous ne serez généralement pas capable de manipuler un objet temporaire, dire de faire quelque chose qui le modifierait est presque à coup sûr une erreur parce que vous ne serez pas capable d'utiliser cette information. En rendant tous les temporaires automatiquement **const**, le compilateur vous informe quand vous commettez cette erreur.

Dans l'exemple ci-dessus, **f5( )**renvoie un objet **X**qui n'est pas **const**. Mais dans l'expression :

```
f7(f5());
```

Le compilateur doit créer un objet temporaire pour retenir la valeur de **f5( )**afin qu'elle soit passée à **f7( )**. Cela serait correct si **f7( )**prenait ses arguments par valeur ; alors, l'objet temporaire serait copié dans **f7( )**et ce qui arriverait au **X**temporaire n'aurait aucune importance. Cependant, **f7( )**prend ses arguments *par référence*, ce qui signifie dans cette exemple qu'il prend l'adresse du **X**temporaire. Comme **f7( )**ne prend pas ses arguments par référence de type **const**, elle a la permission de modifier l'objet temporaire. Mais le compilateur sait que l'objet temporaire disparaîtra dès que l'évaluation de l'expression sera achevée, et ainsi toute modification que vous ferez au **X**temporaire sera perdue. En faisant de tous les objets temporaires des **const**, cete situation produit un message d'erreur de compilation, afin que vous ne rencontriez pas un bug qui serait très difficile à trouver.

Toutefois, notez les expressions licites :

```
f5() = X(1);
f5().modify();
```

Bien que ces passages fassent appel au compilateur, ils sont problématiques. **f5( )**renvoie un objet **X**, et pour que le compilateur satisfasse les expressions ci-dessus il doit créer un objet temporaire pour retenir cette valeur temporaire. Ainsi, dans les deux expressions l'objet temporaire est modifié, et dès que l'expression est terminée, l'objet temporaire est détruit. En conséquence, les modifications sont perdues et ce code est probablement un bug ; mais le compilateur ne vous dit rien. Des expressions comme celles-ci sont suffisamment simples pour que vous détectiez le problème, mais quand les choses deviennent plus compliquées, il est possible qu'un bug se glisse à travers ces fissures.

La façon dont la **const**ance des objets d'une classe est préservée est montrée plus loin dans ce chapitre.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

8.3.3 - Passer et retourner des adresses

Si vous passez ou retournez une adresse (un pointeur ou une référence), le programmeur client peut la prendre et modifier sa valeur originale. Si vous rendez le pointeur ou la référence **const**, vous l'en empêcher, ce qui peut éviter quelques douleurs. En fait, lorsque vous passez une adresse à une fonction, vous devriez, autant que possible, en faire un **const**. Si vous ne le faites pas, vous excluez la possibilité d'utiliser cette fonction avec quelque type **const**que ce soit.

Le choix de retourner un pointeur ou une référence comme **const**dépend de ce que vous voulez autoriser le programmeur client à faire avec. Voici un exemple qui démontre l'usage de pointeurs **const**comme arguments de fonction et valeurs retournées :

```
                        //: C08:ConstPointer.cpp
// Pointeur constants comme arguments ou retournés

void t(int*) {}

void u(const int* cip) {
//!  *cip = 2; // Illicite - modifie la valeur
  int i = *cip; // OK -- copie la valeur
//!  int* ip2 = cip; // Illicite : pas const
}

const char* v() {
  // Retourne l'adresse du tableau de caractère static :
  return "result of function v()";
}

const int* const w() {
  static int i;
  return &i;
}

int main() {
  int x = 0;
  int* ip = &x;
  const int* cip = &x;
  t(ip);  // OK
//!  t(cip); // Pas bon
  u(ip);  // OK
  u(cip); // Egalement OK
//!  char* cp = v(); // Pas bon
  const char* ccp = v(); // OK
//!  int* ip2 = w(); // Pas bon
  const int* const ccip = w(); // OK
  const int* cip2 = w(); // OK
//!  *w() = 1; // Pas bon
} ///:~
```

La fonction **t( )**prend un pointeur ordinaire (non- **const**) comme argument, et **u( )**prend un pointeur **const**. Dans **u( )**vous pouvez constater qu'essayer de modifier la destination du pointeur **const**est illicite, mais vous pouvez bien sûr copier l'information dans une variable non **const**ante. Le compilateur vous empêche également de créer un pointeur non- **const**ant en utilisant l'adresse stockée dans un pointeur de type **const**.

Les fonctions **v( )**et **w( )**testent la sémantique d'une valeur de retour. **v( )**retourne un **const char***créé à partir d'un tableau de caractères. Cette déclaration produit vraiment l'adresse du tableau de caractères, après que le compilateur l'ait créé et stocké dans l'aire de stockage statique. Comme mentionné plus haut, ce tableau de caractères est techniquement une constante, ce qui est correctement exprimé par la valeur de retour de **v( )**.

La valeur retournée par **w( )**requiert que le pointeur et ce vers quoi il pointe soient des **const**. Comme avec **v( )**, la valeur retournée par **w( )**est valide après le retour de la fonction uniquement parce qu'elle est **static**. Vous ne devez jamais renvoyer des pointeurs vers des variables de pile locales parce qu'ils seront invalides après le retour de la fonction et le nettoyage de la pile. (Un autre pointeur habituel que vous pouvez renvoyer est l'adresse du

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

stockage alloué sur le tas, qui est toujours valide après le retour de la fonction).

Dans **main( )**, les fonctions sont testées avec divers arguments. Vous pouvez constater que **t( )**acceptera un pointeur non- **const**comme argument, mais si vous essayez de lui passer un pointeur vers un **const**, il n'y a aucune garantie que **t( )**laissera le pointeur tranquille, ce qui fait que le compilateur génère un message d'erreur. **u( )**prend un pointeur **const**, et acceptera donc les deux types d'arguments. Ainsi, une fonction qui prend un pointeur **const**est plus générale qu'une fonction qui n'en prend pas.

Comme prévu, la valeur de retour de **v( )**peut être assignée uniquement à un pointeur vers un **const**. Vous vous attendriez aussi à ce que le compilateur refuse d'assigner la valeur de retour de **w( )**à un pointeur non- **const**, et accepte un **const int* const**, mais il accepte en fait également un **const int***, qui ne correspond pas exactement au type retourné. Encore une fois, comme la valeur (qui est l'adresse contenue dans le pointeur) est copiée, la promesse que la variable originale ne sera pas atteinte est automatiquement tenue. Ainsi, le second **const**dans **const int* const**n'a de sens que quand vous essayez de l'utiliser comme une lvalue, auquel cas le compilateur vous en empêche.

Passage d'argument standard

En C, il est très courant de passer par valeur, et quand vous voulez passer une adresse votre seul possibilité est d'utiliser un pointeur Certaines personnes vont jusqu'à dire que *tout*en C est passé par valeur, puisque quand vous passez un pointeur une copie est réalisée (donc vous passez le pointeur par valeur). Aussi précis que cela puisse être, je pense que cela brouille la situation. . Toutefois, aucune de ces approches n'est préférée en C++. Au lieu de cela, votre premier choix quand vous passez un argument est de le passer par référence, et par référence de type **const**, qui plus est. Pour le programmeur client, la syntaxe est identique à celle du passage par valeur, et il n'y a donc aucune confusion à propos des pointeurs ; ils n'ont même pas besoin de penser aux pointeurs. Pour le créateur de la fonction, passer une adresse est pratiquement toujours plus efficace que passer un objet de classe entier, et si vous passez par référence de type **const**, cela signifie que votre fonction ne changera pas la destination de cette adresse, ce qui fait que l'effet du point de vue du programmeur client est exactement le même que de passer par valeur (c'est juste plus efficace).

A cause de la syntaxe des références (cela ressemble à un passage par valeur pour l'appelant) il est possible de passer un objet temporaire à une fonction qui prend une référence de type **const**, alors que vous ne pouvez jamais passer un objet temporaire à une fonction qui prend un pointeur ; avec un pointeur, l'adresse doit être prise explicitement. Ainsi, le passage par référence crée une nouvelle situation qui n'existe jamais en C : un objet temporaire, qui est toujours **const**, peut voir son *adresse*passée à une fonction. C'est pourquoi, pour permettre aux objets temporaires d'être passés aux fonctions par référence, l'argument doit être une référence de type **const**. L'exemple suivant le démontre :

```
                        //: C08:ConstTemporary.cpp
// Les temporaires sont des <b>const</b>

class X {};

X f() { return X(); } // Retour par valeur

void g1(X&) {} // Passage par référence de type non-const
void g2(const X&) {} // Passage par référence de type const

int main() {
  // Erreur : const temporaire créé par f()
//!  g1(f());
  // OK: g2 prend une référence const
  g2(f());
} ///:~
```

f( )retourne un objet de **class X** *par valeur*. Cela signifie que quand vous prenez immédiatement la valeur de retour de **f( )**et que vous la passez à une autre fonction comme dans l'appel de **g1( )**et **g2( )**, un objet temporaire est créé

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

et cet objet temporaire est de type **const**. Ainsi, l'appel dans **g1( )**est une erreur parce que **g1( )**ne prend pas de référence de type **const**, mais l'appel à **g2( )**est correct.

8.4 - Classes

Cette section montre de quelles façons vous pouvez utiliser **const**avec les classes. Vous pourriez vouloir créer un **const**local dans une classe pour l'utiliser dans des expressions constantes qui seront évaluées à la compilation. Toutefois, le sens de **const**est différent au sein des classes, et vous devez donc comprendre les options offertes afin de créer des données membres **const**d'une classe.

Vous pouvez aussi rendre un objet entier **const**(et comme vous venez de le voir, le compilateur crée toujours des objets temporaires **const**). Mais préserver la **const**ance d'un objet est plus complexe. Le compilateur peut garantir la **const**ance d'un type prédéfini mais il ne peut pas gérer les intrications d'une classe. Pour garantir la **const**ance d'un objet d'une classe, la fonction membre **const**est introduite : seule une fonction membre **const**peut être appelée pour un objet **const**.

8.4.1 - const dans les classes

Un des endroits où vous aimeriez utiliser un **const**pour les expressions constantes est à l'intérieur des classes. L'exemple type est quand vous créez un tableau dans une classe et vous voulez utiliser un **const**à la place d'un **#define**pour définir la taille du tableau et l'utiliser dans des calculs impliquant le tableau. La taille du tableau est quelque chose que vous aimeriez garder caché dans la classe, de telle sorte que si vous utilisiez un nom comme **taille**, par exemple, vous puissiez l'utiliser également dans une autre classe sans qu'il y ait de conflit. Le préprocesseur traite tous les **#define**comme globaux à partir du point où ils sont définis, donc ils n'auront pas l'effet désiré.

Vous pourriez supposer que le choix logique est de placer un **const**dans la classe. Ceci ne produit pas le résultat escompté. Dans une classe, **const**revient partiellement à son sens en C. Il alloue un espace de stockage dans chaque objet et représente une valeur qui ne peut être initialisée qu'une fois et ne peut changer par la suite. L'usage de **const**dans une classe signifie "Ceci est constant pour toute la durée de vie de l'objet". Toutefois, chaque objet différent peut contenir une valeur différente pour cette constante.

Ainsi, quand vous créez un **const**ordinaire (pas **static**) dans une classe, vous ne pouvez pas lui donner une valeur initiale. Cette initialisation doit avoir lieu dans le constructeur, bien sûr, mais à un endroit spécial du constructeur. Parce qu'un **const**doit être initialisé au point où il est créé, il doit déjà l'être dans le corps du constructeur. Autrement vous risqueriez d'avoir à attendre jusqu'à un certain point du constructeur et le **const**resterait non initialisé quelques temps. En outre, il n'y aurait rien qui vous empêcherait de modifier la valeur du **const**à différents endroits du corps du constructeur.

La liste d'initialisation du constructeur

Le point d'initialisation spécial est appelé *liste d'initialisation du constructeur*, et a été initialement développée pour être utilisée dans l'héritage (couvert au Chapitre 14). La liste d'initialisation du constructeur - qui, comme son nom l'indique, n'intervient que dans la définition du constructeur - est une liste "d'appels de constructeurs" qui a lieu après la liste des arguments et deux points, mais avant l'accolade ouvrante du corps du constructeur. Ceci pour vous rappeler que l'initialisation dans la liste a lieu avant que tout code du corps du constructeur ne soit exécuté. C'est là qu'il faut mettre toutes les initialisations de **const**. La forme correcte pour **const**dans une classe est montrée ici :

```
                        //: C08:ConstInitialization.cpp
// Initialiser des const dans les classes
#include <iostream>
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
using namespace std;

class Fred {
  const int size;
public:
  Fred(int sz);
  void print();
};

Fred::Fred(int sz) : size(sz) {}
void Fred::print() { cout << size << endl; }

int main() {
  Fred a(1), b(2), c(3);
  a.print(), b.print(), c.print();
} ///:~
```

La forme de la liste d'initialisation du constructeur montrée ci-dessus est déroutante au début parce que vous n'êtes pas habitués à voir un type prédéfini traité comme s'il avait un constructeur.

“Constructeurs” pour types prédéfinis

Comme le langage se développait et que plus d'effort était fourni pour rendre les types définis par l'utilisatur plus ressemblant aux types prédéfinis, il devint clair qu'il y avait des fois où il était utile de rendre des types prédéfinis semblables aux types définis par l'utilisateur. Dans la liste d'initialisation du constructeur, vous pouvez traiter un type prédéfini comme s'il avait un constructeur, comme ceci :

```
                        //: C08:BuiltInTypeConstructors.cpp
#include <iostream>
using namespace std;

class B {
  int i;
public:
  B(int ii);
  void print();
};

B::B(int ii) : i(ii) {}
void B::print() { cout << i << endl; }

int main() {
  B a(1), b(2);
  float pi(3.14159);
  a.print(); b.print();
  cout << pi << endl;
} ///:~
```

C'est critique quand on initialise des données membres **const**car elles doivent être initialisées avant l'entrée dans le corps de la fonction.

Il était logique d'étendre ce "constructeur" pour types prédéfinis (qui signifie simplement allocation) au cas général, ce qui explique pourquoi la définition **float pi(3.14159)**fonctionne dans le code ci-dessus.

Il est souvent utile d'encapsuler un type prédéfini dans une classe pour garantir son initisalisation par le constructeur. Par exemple, ici une classe **Integer**(entier, ndt) :

```
                        //: C08:EncapsulatingTypes.cpp
#include <iostream>
using namespace std;

class Integer {
  int i;
public:
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  Integer(int ii = 0);
  void print();
};

Integer::Integer(int ii) : i(ii) {}
void Integer::print() { cout << i << ' '; }

int main() {
  Integer i[100];
  for(int j = 0; j < 100; j++)
    i[j].print();
} ///:~
```

Le tableau d' **Integer**dans **main( )**est entièrement initialisé à zéro automatiquement. Cette initialisation n'est pas nécessairement plus coûteuse qu'une boucle **for**ou que **memset( )**. Beaucoup de compilateurs l'optimise très bien.

8.4.2 - Constantes de compilation dans les classes

L'utilisation vue ci-dessus de **const**est intéressante et probablement utile dans certains cas, mais elle ne résoud pas le problème initial qui était : "comment créer une constante de compilation dans une classe ?" La réponse impose l'usage d'un mot-clef supplémentaire qui ne sera pas pleinement introduit avant le Chapitre 10 : **static**. Ce mot-clef, selon les situations, signifie "une seule instante, indépendamment du nombre d'objets créés", qui est précisément ce dont nous avons besoin ici : un membre d'une classe constant et qui ne peut changer d'un objet de la classe à un autre. Ainsi, un **static const**d'un type prédéfini peut être traité comme une constante de compilation.

Il y a un aspect de **static const**, quand on l'utilise dans une classe, qui est un peu inhabituel : vous devez fournir l'initialiseur au point de définition du **static const**. C'est quelque chose qui ne se produit qu'avec **static const**; Autant que vous aimeriez le faire dans d'autres situations, cela ne marchera pas car toutes les autres données membres doivent être initialisées dans le constructeur ou dans d'autres fonctions membres.

Voici un exemple qui montre la création et l'utilisation d'un **static const**appelé **size**dans une classe qui représente une pile de pointeur vers des chaines. Au moment de la rédaction, tous les compilateurs ne supportent pas cette caractéristique.:

```
                          //: C08:StringStack.cpp
// Utilisation de static const pour créer une
// constante de compilation dans une classe
#include <string>
#include <iostream>
using namespace std;

class StringStack {
  static const int size = 100;
  const string* stack[size];
  int index;
public:
  StringStack();
  void push(const string* s);
  const string* pop();
};

StringStack::StringStack() : index(0) {
  memset(stack, 0, size * sizeof(string*));
}

void StringStack::push(const string* s) {
  if(index < size)
    stack[index++] = s;
}

const string* StringStack::pop() {
  if(index > 0) {
    const string* rv = stack[--index];
    stack[index] = 0;
    return rv;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  }
  return 0;
}

string iceCream[] = {
  "pralines & cream",
  "fudge ripple",
  "jamocha almond fudge",
  "wild mountain blackberry",
  "raspberry sorbet",
  "lemon swirl",
  "rocky road",
  "deep chocolate fudge"
};

const int iCsz =
  sizeof iceCream / sizeof *iceCream;

int main() {
  StringStack ss;
  for(int i = 0; i < iCsz; i++)
    ss.push(&iceCream[i]);
  const string* cp;
  while((cp = ss.pop()) != 0)
    cout << *cp << endl;
} ///:~
```

Comme **size**est utilisé pour déterminer la taille (size en anglais, ndt) du tableau **stack**, c'est de fait une constante de compilation, mais une qui est masquée au sein de la classe.

Remarquez que **push( )**prend un **const string***comme argument, que **pop( )**renvoie un **const string***, et **StringStack**contient **const string***. Si ce n'était pas le cas, vous ne pourriez pas utiliser un **StringStack**pour contenir les pointeurs dans **iceCream**. Toutefois, cela vous empêche également de faire quoi que ce soit qui modifierait les objets contenus dans **StringStack**. Bien sûr, tous les conteneurs ne sont pas conçus avec cette restriction.

Le “enum hack” dans le vieux code

Dans les versions plus anciennes de C++, **static const**n'était pas supporté au sein des classes. Cela signifiait que **const**était inutile pour les expressions constantes dans les classes. Toutefois, les gens voulaient toujours le faire si bien qu'une solution typique (généralement dénommée “enum hack”) consistait à utiliser un **enum**non typé et non instancié. Une énumération doit avoir toutes ses valeurs définies à la compilation, c'est local à la classe, et ses valeurs sont disponibles pour les expressions constantes. Ainsi, vous verrez souvent :

```
                        //: C08:EnumHack.cpp
#include <iostream>
using namespace std;

class Bunch {
  enum { size = 1000 };
  int i[size];
};

int main() {
  cout << "sizeof(Bunch) = " << sizeof(Bunch)
       << ", sizeof(i[1000]) = "
       << sizeof(int[1000]) << endl;
} ///:~
```

L'utilisation de **enum**ici n'occupera aucune place dans l'objet, et les énumérateurs sont tous évalués à la compilation. Vous pouvez également explicitement établir la valeur des énumérateurs :

```
                        enum { one = 1, two = 2, three };
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Avec des types **enum**intégraux, le compilateur continuera de compter à partir de la dernière valeur, si bien que l'énumérateur **three**recevra la valeur 3.

Dans l'exemple **StringStack.cpp**ci-dessus, la ligne:

```
static const int size = 100;
```

deviendrait :

```
enum { size = 100 };
```

Bien que vous verrez souvent la technique **enum**dans le code ancien, **static const**a été ajouté au langage pour résoudre précisément ce problème. Cependant, il n'y a pas de raison contraignante qui impose d'utiliser **static const**plutôt que le hack de **enum**, et dans ce livre c'est ce dernier qui sera utilisé parce qu'il est supporté par davantage de compilateurs au moment de sa rédaction.

8.4.3 - objets cont & fonctions membres

Les fonctions membres d'une classe peuvent être rendues **const**. Qu'est-ce que cela veut dire ? Pour le comprendre, vous devez d'abord saisir le concept d'objets **const**.

Un objet **const**est défini de la même façon pour un type défini par l'utilisateur ou pour un type prédéfini. Par exemple :

```
const int i = 1;
const blob b(2);
```

Ici, **b**est un objet **const**de type **blob**. Son constructeur est appelé avec l'argument 2. Pour que le compilateur impose la **const**ance, il doit s'assurer qu'aucune donnée membre de l'objet n'est changée pendant la durée de vie de l'objet. Il peut facilement garantir qu'aucune donnée publique n'est modifiée, mais comment peut-il savoir quelles fonctions membres modifieront les données et lesquelles sont "sûres" pour un objet **const**?

Si vous déclarez une fonction membre **const**, vous dites au compilateur que la fonction peut être appelée pour un objet **const**. Une fonction membre qui n'est pas spécifiquement déclarée **const**est traitée comme une fonction qui modifiera les données membres de l'objet, et le compilateur ne vous permettra pas de l'appeler pour un objet **const**.

Cela ne s'arrête pas ici, cependant. *Dire*simplement qu'une fonction membre est **const**ne garantit pas qu'elle se comportera ainsi, si bien que le compilateur vous force à définir la spécification **const**quand vous définissez la fonction. (Le **const**fait partie de la signature de la fonction, si bien que le compilateur comme l'éditeur de liens vérifient la **const**ance.) Ensuite, il garantit la **const**ance pendant la définition de la fonction en émettant un message d'erreur si vous essayez de changer un membre de l'objet *ou*d'appeler une fonction membre non- **const**. Ainsi on garantit que toute fonction membre que vous déclarez **const**se comportera de cette façon.

Pour comprendre la syntaxe de déclaration des fonctions membres **const**, remarquez tout d'abord que placer la déclaration **const**avant la fonction signifie que la valeur de retour est **const**: cela ne produit pas l'effet désiré. A la place, vous devez spécifier le **const** *après*la liste des arguments. Par exemple :

```
//: C08:ConstMember.cpp
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
class X {
  int i;
public:
  X(int ii);
  int f() const;
};

X::X(int ii) : i(ii) {}
int X::f() const { return i; }

int main() {
  X x1(10);
  const X x2(20);
  x1.f();
  x2.f();
} ///:~
```

Remarquez que le mot-clef **const**doit être répété dans la définition ou bien que le compilateur la verra comme une fonction différente. Comme **f( )**est une fonction membre **const**, si elle essaie de modifier **i**de quelle que façon que ce soit *ou*d'appeler une autre fonction membre qui ne soit pas **const**, le compilateur le signale comme une erreur.

Vous pouvez constater qu'une fonction membre **const**peut être appelée en toute sécurité que les objets soient **const**ou non. Ainsi, vous pouvez le considérer comme la forme la plus générale de fonction membre (et à cause de cela, il est regrettable que les fonctions membres ne soient pas **const**par défaut). Toute fonction qui ne modifie pas les données membres devrait être déclarée **const**, afin qu'elle puisse être utilisée avec des objets **const**.

Voici un exemple qui compare les fonctions membres **const**et non **const**:

```
                          //: C08:Quoter.cpp
// Sélection aléatoire de citation
#include <iostream>
#include <cstdlib> // Générateur de nombres aléatoires
#include <ctime> // Comme germe du générateur
using namespace std;

class Quoter {
  int lastquote;
public:
  Quoter();
  int lastQuote() const;
  const char* quote();
};

Quoter::Quoter(){
  lastquote = -1;
  srand(time(0)); // Germe du générateur de nombres aléatoires
}

int Quoter::lastQuote() const {
  return lastquote;
}

const char* Quoter::quote() {
  static const char* quotes[] = {
    "Are we having fun yet?",
    "Doctors always know best",
    "Is it ... Atomic?",
    "Fear is obscene",
    "There is no scientific evidence "
    "to support the idea "
    "that life is serious",
    "Things that make us happy, make us wise",
  };
  const int qsize = sizeof quotes/sizeof *quotes;
  int qnum = rand() % qsize;
  while(lastquote >= 0 && qnum == lastquote)
    qnum = rand() % qsize;
  return quotes[lastquote = qnum];
}

int main() {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  Quoter q;
  const Quoter cq;
  cq.lastQuote(); // OK
//!  cq.quote(); // Pas OK; fonction pas const
  for(int i = 0; i < 20; i++)
    cout << q.quote() << endl;
} ///:~
```

Ni les constructeurs ni les destructeurs ne peuvent être **const**parce qu'ils effectuent pour ainsi dire toujours des modifications dans l'objet pendant l'initialisation et le nettoyage. La fonction membre **quote( )**ne peut pas non plus être **const**parce qu'elle modifie la donnée membre **lastquote**(cf. l'instruction **return**). Toutefois, **lastQuote( )**ne réalise aucune modification et peut donc être **const**et peut être appelée par l'objet **const cq**en toute sécurité.

mutable : const logique vs. const de bit

Que se passe-t-il si vous voulez créer une fonction membre **const**mais que vous voulez toujours changer certaines données de l'objet ? Ceci est parfois appelé **const**de *bit*et **const** *logique*(parfois également **const** *de membre*). **const**de bit signifie que chaque bit dans l'objet est permanent, si bien qu'une image par bit de l'objet ne changera jamais. **const**logique signifique que, bien que l'objet entier soit conceptuellement constant, il peut y avoir des changements sur une base membre à membre. Toutefois, si l'on dit au compilateur qu'un objet est **const**, il préservera jalousement cet objet pour garantir une **const**ance de bit. Pour réaliser la **const**ance logique, il y a deux façons de modifier une donnée membre depuis une fonction membre **const**.

La première approche est historique est appelée éviction de la constance par transtypage(*casting away constnes*sen anglais, ndt) C'est réalisé de manière relativement bizarre. Vous prenez **this**(le mot-clef qui donne l'adresse de l'objet courant) et vous le transtypé vers un pointeur vers un objet du type courant. Il semble que **this**est *déjà*un pointeur de ce type. Toutefois, dans une fonction membre **const**c'est en fait un pointeur **const**. Le transtypage permet de supprimer la **const**ance pour cette opération. Voici un exemple :

```
                        //: C08:Castaway.cpp
//  Constance contournée par transtypage

class Y {
  int i;
public:
  Y();
  void f() const;
};

Y::Y() { i = 0; }

void Y::f() const {
//!  i++; // Erreur -- fonction membre const
  ((Y*)this)->i++; // OK : contourne la constance
  // Mieux : utilise  la syntaxe de transtypage expicite du :
  (const_cast<Y*>(this))->i++;
}

int main() {
  const Y yy;
  yy.f(); // Le modifie réellement !
} ///:~
```

Cette approche fonctionne et vous la verrez utilisée dans le code ancien, mais ce n'est pas la technique préférée. Le problème est que ce manque de **const**ance est dissimulé dans la définition de la fonction membre, et vous ne pouvez pas savoir grâce à l'interface de la classe que la donnée de l'objet est réellement modifiée à moins que vous n'ayez accès au code source (et vous devez suspecter que la **const**ance est contournée par transtypage, et chercher cette opération). Pour mettre les choses au clair, vous devriez utiliser le mot-clef **mutable**dans la déclaration de la classe pour spécifier qu'une donnée membre particulière peut changer dans un objet **const**:

```
                        //: C08:Mutable.cpp
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
// Le mot-clef "mutable"

class Z {
  int i;
  mutable int j;
public:
  Z();
  void f() const;
};

Z::Z() : i(0), j(0) {}

void Z::f() const {
//! i++; // Erreur -- fonction membre const
    j++; // OK: mutable
}

int main() {
  const Z zz;
  zz.f(); // Le modifie réellement !
} ///:~
```

Ainsi, l'utilisateur de la classe peut voir par la déclaration quels membres sont susceptibles d'être modifiés dans une fonction membre **const**.

ROMabilité

Si un objet est défini **const**, c'est un candidat pour être placé dans la mémoire morte (ROM = Read Only Memory, ndt), ce qui est souvent une question importante dans la programmation des systèmes embarqués. Le simple fait de rendre un objet **const**, toutefois, ne suffit pas - les conditions nécessaires pour la ROMabilité sont bien plus strictes. Bien sûr, l'objet doit avoir la **const**ance de bit, plutôt que logique. Ceci est facile à voir si la **const**ance logique est implémentée uniquement par le mot-clef **mutable**, mais probablement indétectable par le compilateur si la **const**ance est contournée par transtypage dans une fonction membre **const**. En outre,

1 La **class**e ou la **struct**ure ne doivent pas avoir de constructeur ou de destructeur définis par l'utilisateur.
2 Il ne peut pas y avoir de classe de base (cf. Chapitre 14) ou d'objet membre avec un constructeur ou un destructeur défini par l'utilisateur.

L'effet d'une opération d'écriture sur toute partie d'un obet **const**de type ROMable est indéfini. Bien qu'un objet correctement conçu puisse être placé dans la ROM, aucun nobjet n'est jamais *obligé*d'être placé dans la ROM.

8.5 - volatile

La syntaxe de **volatile**est identique à celle de **const**, mais **volatile**signifie "Cette donnée pourrait changer sans que le compilateur le sache". D'une façon ou d'une autre, l'environnement modifie la donnée (potentiellement par du multitâche, du multithreading ou des interruptions), et **volatile**dit au compilateur de ne faire aucune hypothèse à propos de cette donnée, spécialement pendant l'optimisation.

Si le compilateur dit "J'ai lu cette donnée dans un registre plus tôt, et je n'y ai pas touché", normalement, il ne devrait pas lire la donnée à nouveau. Mais si la donnée est **volatile**, le compilateur ne peut pas faire ce genre d'hypohèse parce que la donnée pourrait avoir été modifiée par un autre processus, et il doit lire à nouveau cette donnée plutôt qu'optimiser le code en supprimant ce qui serait normalement une lecture redondante.

Vous créez des objets **volatile**en utilisant la même syntaxe que vous utilisez pour créer des objets **const**. Vous pouvez aussi créer des objets **const volatile**, qui ne peuvent pas être modifiés par le programmeur client mais changent plutôt sous l'action d'organismes extérieurs. Voici un exemple qui pourrait représenter une classe associée à un fragment de communication avec le hardware :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                    //: C08:Volatile.cpp
// Le mot-clef volatile

class Comm {
  const volatile unsigned char byte;
  volatile unsigned char flag;
  enum { bufsize = 100 };
  unsigned char buf[bufsize];
  int index;
public:
  Comm();
  void isr() volatile;
  char read(int index) const;
};

Comm::Comm() : index(0), byte(0), flag(0) {}

// Juste une démonstration ; ne fonctionnera pas vraiment
// comme routine d'interruption :
void Comm::isr() volatile {
  flag = 0;
  buf[index++] = byte;
  // Repart au début du buffer :
  if(index >= bufsize) index = 0;
}

char Comm::read(int index) const {
  if(index < 0 || index >= bufsize)
    return 0;
  return buf[index];
}

int main() {
  volatile Comm Port;
  Port.isr(); // OK
//!  Port.read(0); // Erreur, read() n'est pas volatile
} ///:~
```

Comme avec **const**, vous pouvez utiliser **volatile**pour les données membres, les fonctions membres et les objets eux-mêmes. Vous pouvez appeler des fonctions membres **volatile**uniquement pour des objets **volatile**.

La raison pour laquelle **isr( )**ne peut pas vraiment être utilisée comme routine d'interruption est que dans une fonction membre, l'adresse de l'objet courant (**this**) doit être passée secrètement, et une routine d'interruption ne prend généralement aucun argument. Pour résoudre ce problème, vous pouvez faire de **isr( )**une fonction membre **static**, sujet couvert au Chapitre 10.

La syntaxe de **volatile**est identique à celle de **const**, si bien que les discussions de ces deux mots-clefs sont souvent menées ensembles. Les deux sont dénommés *qualificateurs c-v*.

8.6 - Résumé

Le mot-clef **const**vous donne la possibilité de définir comme constants des objets, des arguments de fonctions, des valeurs retournées et des fonctions membres, et d'éliminer le préprocesseur pour la substitution de valeurs sans perdre les bénéfices du préprocesseur. Tout cela fournit un moyen supplémentaire pour la vérification des types et la sécurité de vos programmes. L'usage de la technique dite de *const conformité*(l'utilisation de **const**partout où c'est possible) peut sauver la vie de projets entiers.

Bien que vous puissiez ignorer **const**et utiliser les vieilles pratiques du C, ce mot-clef est ici pour vous aider. Les Chapitres 11 et suivants commencent à utiliser abondamment les références, et ainsi vous verrez encore mieux à quel point il est critique d'utiliser **const**avec les arguments de fonction.

8.7 - Exercices

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

La solution de certains exercices sélectionnés peut se trouver dans le document électronique *The Thinking in C++ Annotated Solution Guide*, disponible pour une somme modique à www.BruceEckel.com.

1 Créer trois valeurs **const int**, puis additionez les pour produire une valeur qui détermine la taille d'un tableau dans la définition d'un tableau. Essayez de compiler le même code en C et regardez ce qu'il se produit (vous pouvez généralement forcer votre compilateur C++ à fonctionner comme un compilateur C en utilisant un argument de la ligne de commande).
2 Démontrez-vous que les compilateurs C et C++ traitent réellement les constantes différemment. Créer un **const**global et utilisez-le dans une expression globale constante, puis compilez-le en C et en C++.
3 Créez des exemples de définitions **const**pour tous les types prédéfinis et leurs variantes. Utilisez les dans des expressions avec d'autres **const**pour créer de nouvelles définitions **const**. Assurez qu'elles se compilent correctement.
4 Créez une définition const dans un fichier d'en-tête, incluez ce fichier dans deux fichiers **.cpp**, puis compiler et faites l'édition de liens de ces fichiers. Vous ne devriez avoir aucune erreur. Maintenant, faites la même chose en C.
5 Créez un **const**dont la valeur est déterminée à l'exécution en lisant l'heure à laquelle démarre le programme (vous devrez utiliser l'en-tête standard **<ctime>**). Plus loin dans le programme, essayez de lire une deuxième valeur de l'heure dans votre **const**et regardez ce qu'il se produit.
6 Créez un tableau **const**de **char**, puis essayez de changer l'un des **char**.
7 Créez une instruction **extern const**dans un fichier, et placez un **main( )**dans ce fichier qui imprime la valeur de l' **extern const**. Donnez une définition **extern const**dans un autre fichier, puis compilez et liez les deux fichiers ensemble.
8 Ecrivez deux pointeurs vers **const long**en utilisant les deux formes de déclaration. Faites pointer l'un d'entre eux vers un tableau de **long**. Montrez que vous pouvez incrémenter et décrémenter le pointeur, mais que vous ne pouvez pas changer ce vers quoi il pointe.
9 Ecrivez un pointeur **const**vers un **double**, et faites le pointer vers un tableau de **double**. Montrez que vous pouvez modifier ce vers quoi il pointe, mais que vous ne pouvez incrémenter ou décrémenter le pointeur.
10 Ecrivez un pointeur **const**vers un objet **const**. Montrez que vous ne pouvez que lire la valeur vers laquelle pointe le pointeur, mais que vous ne pouvez modifier ni le pointeur ni ce vers quoi il pointe.
11 Supprimez le commentaire sur la ligne de code générant une erreur dans **PointerAssignment.cpp**pour voir l'erreur que génère votre compilateur.
12 Créez un tableau de caractères littéral avec un pointeur qui pointe vers le début du tableau. A présent, utilisez le pointeur pour modifier des éléments dans le tableau. Est-ce que votre compilateur signale cela comme ue erreur ? Le devrait-il ? S'il ne le fait pas, pourquoi pensez vous que c'est le cas ?
13 Créez une fonction qui prend un argument par valeur comme **const**; essayez de modifier cet argument dans le corps de la fonction.
14 Créez une fonction qui prenne un **float**par valeur. Dans la fonction, liez un **const float&**à l'argument, et à partir de là, utilisez uniquement la référence pour être sûr que l'argument n'est pas modifié.
15 Modifiez **ConstReturnValues.cpp**en supprimant les commentaires des lignes provoquant des erreurs l'une après l'autre, pour voir quels messages d'erreurs votre compilateur génère.
16 Modifiez **ConstPointer.cpp**en supprimant les commentaires des lignes provoquant des erreurs l'une après l'autre, pour voir quels messages d'erreurs votre compilateur génère.
17 Faites une nouvelle version de **ConstPointer.cpp**appelée **ConstReference.cpp**qui utilise les références au lieu des pointeurs (vous aurez peut-être besoin de consulter le Chapitre 11, plus loin).
18 Modifiez **ConstTemporary.cpp**en supprimant le commentaire de la ligne provoquant une erreur, pour voir quel message d'erreur votre compilateur génère.
19 Créez une classe contenant à la fois un **float const**et non- **const**. Initialisez-les en utilisant la liste d'initialisation du constructeur.
20 Créez une classe **MyString**qui contient un **string**et possède un constructeur qui initialise le **string**, et une fonction **print( )**. Modifiez **StringStack.cpp**afin que le conteneur contienne des objets **MyString**, et **main( )**afin qu'il les affiche.
21 Créez une classe contenant un membre **const**que vous intialisez dans la liste d'initialisation du constructeur et une énumération sans label que vous utilisez pour déterminer une taille de tableau.
22 Dans **ConstMember.cpp**, supprimez l'instruction **const**sur la définition de la fonction membre, mais laissez le dans la déclaration, pour voir quel genre de message d'erreur vous obtenez du compilateur.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

23 Créez une classe avec à la fois des fonctions membres **const**et non- **const**. Créez des objets **const**et non-**const**de cette classe, et essayez d'appeler les différents types de fonctions membres avec les différents types d'objets.
24 Créez une classe avec des fonctions membres **const**et non- **const**. Essayez d'appeler une fonction membre non- **const**depuis une fonction membre **const**pour voir quel est le genre de messages d'erreur du compilateur que vous obtenez.
25 Dans **Mutable.cpp**, supprimez le commentaire de la ligne provoquant une erreur, pour voir quel message d'erreur votre compilateur génère.
26 Modifiez **Quoter.cpp**en faisant de **quote( )**une fonction membre **const**et de **lastquote**un **mutable**.
27 Créez une classe avec une donnée membre **volatile**. Créez des fonctions membres **volatile**et non-**volatile**qui modifient la donnée membre **volatile**, et voyez ce que dit le compilateur. Créez des objets de votre classe, **volatile**et non- **volatile**, et essayez d'appeler des fonctions membres **volatile**et non-**volatile**pour voir ce qui fonctionne et les messages d'erreur que produit le compilateur.
28 Créez une classe appelée **oiseau**qui peut **voler( )**et une classe **caillou**qui ne le peut pas. Créez un objet **caillou**, prenez son adresse, et assignez-là à un **void***. Maintenant, prenez le **void***et assignez-le à un **oiseau**(vous devrez utiliser le transtypage), et appelez **voler( )**grâce au pointeur. La raison pour laquelle la permission du C d'assigner ouvertement via un **void***(sans transtypage) est un "trou" dans le langage qui ne pouvait pas être propagé au C++ est-elle claire ?

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

9 - Fonctions inlines

Un des aspects importants dont le C++ hérite du C est l'efficacité. Si l'efficacité du C++ était nettement moins grande que celle du C, il y aurait un nombre significatif de programmeurs qui ne pourraient pas justifier son usage.

En C, une des façons de préserver l'efficacité est l'usage de *macros*, qui vous permet de faire ce qui ressemble à des appels de fonctions sans le temps système normalement associé à ces appels. La macro est implémentée avec le préprocesseur au lieu du compilateur proprement dit, et le préprocesseur remplace toutes les appels de macros directement par le code des macros. Ainsi, il n'y a aucun coût associé à l'empilement d'arguments sur la pile, faire un CALL en langage assembleur, retourner les arguments, et faire un RETURN en langage assembleur. Tout le travail est effectué par le préprocesseur, si bien que vous avez la commodité et la lisibilité d'un appel de fonction mais cela ne vous coûte rien.

Il y a deux problèmes liés à l'usage de macros du préprocesseur en C++. Le premier est aussi vrai en C : une macro ressemble à un appel de fonction, mais ne se comporte pas toujours comme tel. Ceci peut dissimuler des bugs difficiles à trouver. Le deuxième problème est spécifique au C++ : le préprocesseur n'a pas la permission d'accéder aux données membres des classes. Ceci signifie que les macros préprocesseurs ne peuvent pas être utilisées comme fonctions membres d'une classe.

Pour conserver l'efficacité de la macro du préprocesseur, mais y adjoindre la sécurité et la portée de classe des vraies fonctions, C++ dispose des *fonctions inlines*. Dans ce chapitre, nous examinerons les problèmes des macros du préprocesseur en C++, comment ces problèmes sont résolus avec les fonctions inline, et donneront des conseils et des aperçus sur la façon dont les fontions inline fonctionnent.

9.1 - Ecueils du préprocesseurs

La clef du problème des macros du préprocesseur est que vous pouvez être abusivement induits à penser que le comportement du préprocesseur est le même que celui du compilateur. Bien sûr, c'était voulu qu'une macro ressemble et agisse comme un appel de fonction, si bien qu'il est assez facile de tomber dans cette erreur. Les difficultés commencent quand les différences subtiles se manifestent.

Comme exemple simple, considérez le code suivant :

```
#define F (x) (x + 1)
```

A présent, si un appel est fait à **F( )**de cette façon :

```
F(1)
```

le préprocesseur le développe, de façon quelque peu inattendue, comme ceci :

```
(x) (x + 1)(1)
```

Le problème se produit à cause de l'espace entre **F**et sa parenthèse d'ouverture dans la définition de la macro. Quand cet espace est supprimé, vous pouvez de fait *appeler*la macro avec l'espace

```
F (1)
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

et elle se développera toujours correctement ainsi :

```
(1 + 1)
```

L'exemple ci-dessus est relativement trivial et le problème sera tout de suite apparent. Les vraies difficultés ont lieu quand on utilise des expressions comme argument dans les appels aux macros.

Il y a deux problèmes. Le premier est que les expressions peuvent se développer dans la macro si bien que la priorité de leur évaluation n'est pas celle que vous attendriez. Par exemple :

```
#define FLOOR(x,b) x>=b?0:1
```

A présent, si des expressions sont utilisées en arguments:

```
if(FLOOR(a&0x0f,0x07)) // ...
```

la macro se développera en :

```
if(a&0x0f>=0x07?0:1)
```

La priorité de **&**est plus basse que celle de **>=**, si bien que l'évaluation de la macro vous réserve une surprise. Une fois le problème découvert, vous pouvez le résoudre en plaçant des parenthèses autour de tous les éléments dans la définition de la macro. (C'est une bonne habitude quand vous créez des macros de préprocesseur.) Ainsi,

```
#define FLOOR(x,b) ((x)>=(b)?0:1)
```

Découvrir le problème peut être difficile, toutefois, et vous pouvez ne pas le découvrir après avoir estimé que la macro se comporte comme il faut. Dans la version sans parenthèse de la macro précédente, *la plupart*des expressions se comporteront correctement parce que la priorité de **>=**est plus basse que celle de la plupart des autres opérateurs comme +, */*, **– –**, et même les opérateurs de manipulation de bits. Ainsi, vous pouvez facilement penser que cela marche avec toutes les expressions, y compris celles qui utilisent des opérateurs logiques de bit.

Le problème précédent peut être résolu par des habitudes de programmation soigneuses : mettre tout entre parenthèses dans une macro. Toutefois, la deuxième difficulté est plus subtile. Contrairement à une fonction normale, à chaque fois que vous utilisez un argument dans une macro, cet argument est évalué. Tant que la macro est appelée seulement avec des variables ordinaires, cette évaluation est sans risque, mais si l'évaluation d'un argument a des effets secondaires, alors le résultat peut être surprenant et ne ressemblera certainement pas au comportement d'une fonction.

Par exemple, cette macro détermine si son argument est compris dans un certain intervalle :

```
#define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)
```

Tant que vous utilisez un argument "ordinaire", la macro se comporte tout à fait comme une vraie fonction. Mais dès que vous vous laissez aller et commencez à croire que *c'est*une vraie fonction, les problèmes commencent. Ainsi :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                    //: C09:MacroSideEffects.cpp
#include "../require.h"
#include <fstream>
using namespace std;

#define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)

int main() {
  ofstream out("macro.out");
  assure(out, "macro.out");
  for(int i = 4; i < 11; i++) {
    int a = i;
    out << "a = " << a << endl << '\t';
    out << "BAND(++a)=" << BAND(++a) << endl;
    out << "\t a = " << a << endl;
  }
} ///:~
```

Remarquez l'utilisation de majuscules dans le nom de la macro. C'est une pratique efficace parce qu'elle signale au lecteur que c'est une macro et pas une fonction, si bien que s'il y a des problèmes, elle agit comme un rappel.

Voici la sortie produite par le programme, qui n'est pas du tout ce à quoi vous vous seriez attendu dans le cas d'une vraie fonction :

```
                    a = 4
  BAND(++a)=0
   a = 5
a = 5
  BAND(++a)=8
   a = 8
a = 6
  BAND(++a)=9
   a = 9
a = 7
  BAND(++a)=10
   a = 10
a = 8
  BAND(++a)=0
   a = 10
a = 9
  BAND(++a)=0
   a = 11
a = 10
  BAND(++a)=0
   a = 12
```

Quand **a**vaut quatre, seule la première partie de l'expression conditionnelle a lieu, si bien que l'expression n'est évaluée qu'une fois, et l'effet secondaire de l'appel à la macro est que **a**passe à cinq, qui est ce que vous attendriez d'un appel à une fonction normale dans la même siutation. Toutefois, quand le nombre est dans l'intervalle, les deux conditions sont testées, ce qui résulte en deux incrémentations. Le résultat est produit en évaluant à nouveau l'argument, ce qui entraîne une troisième incrémentation. Une fois que le nombre sort de l'intervalle, les deux conditions sont toujours testées et vous vous retrouvez toujours avec deux incrémentations. Les effets secondaires diffèrent selon l'argument.

Ce n'est clairement pas le genre de comportement que vous voulez de la part d'une macro qui ressemble à un appel de fonction. Dans ce cas, la solution évidente est d'en faire une vraie fonction, ce qui, bien sûr, ajoute du temps système supplémentaire et peut réduire l'efficacité si vous appelez souvent cette fonction. Malheureusement, le problème peut n'être pas toujours aussi évident, et vous pouvez inconsciemment avoir une librairie qui contienne des fonctions et des macros mélangées ensemble, si bien qu'un problème comme celui-ci peut dissimuler des bugs très difficiles à découvrir. Par exemple, la macro **putc( )**dans **cstdio**peut évaluer son deuxième argument deux fois. C'est une spécification du C Standard. Ainsi, des implémentations peu soigneuses de **toupper( )**comme une macro peuvent évaluer l'argument plus d'une fois, ce qui génèrera des résultats inattendus avec **toupper(*p++)**. Andrew Koenig entre davantage dans les détails dans son livre *C Traps &*

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

(Addison-Wesley, 1989) traduit en français *Les pièges du C*(Addison-Wesley france, 1992).

9.1.1 - Les macros et l'accès

Bien sûr, un codage soigné et l'utilisation des macros du préprocesseur sont nécessaires en C, et nous pourrions certainement échapper à la même chose en C++ si un problème ne se posait pas : une macro n'a aucun concept de portée requis pour les fonctions membres. Le préprocesseur réalise simplement une substitution de texte, si bien que vous ne pouvez pas écrire quelque chose comme :

```
                        class X {
  int i;
public:
#define VAL(X::i) // Erreur
```

ou même quoi que ce soit qui s'en rapproche. En outre, il n'y aurait aucune indication de l'objet auquel vous feriez référence. Il n'y a tout simplement pas moyen d'exprimer la portée de classe dans une macro. Sans solution alternative aux macros de préprocesseur, les programmeurs seraient tentés de rendre certaines données membres **public**par souci d'efficacité, exposant ainsi l'implémentation sous-jacente et empêchant des modifications de cette implémentation, et éliminant du même coup la protection fournie par **private**.

9.2 - Fonctions inline

En résolvant le problème en C++ d'une macro ayant accès aux membres **private**d'une classe, *tous*les problèmes associés aux macros du préprocesseur ont été éliminés. Ceci a été réalisé en transportant, comme il se doit, le concept de macro sous le contrôle du compilateur. Le C++ implémente la macro comme une *fonction inline*, qui est une vraie fonction dans tous les sens du terme. Une fonction inline observe tous les comportements que vous attendez d'une fonction ordinaire. La seule différence est qu'une fonction inline est développée sur place, comme une macro du préprocesseur, si bien que le temps système de l'appel à la fonction est éliminé. Ainsi vous ne devriez (presque) jamais utiliser de macros, mais uniquement des fonctions inline.

Toute fonction définie dans le corps d'une classe est automatiquement inline, mais vous pouvez aussi rendre inline une fonction qui n'appartient pas à une classe en la faisant précéder du mot-clef **inline**. Toutefois, pour qu'il ait un effet, vous devez inclure le corps de la fonction à la déclaration, autrement le compilateur la traitera comme une déclaration de fonction ordinaire. Ainsi,

```
                    inline int plusOne(int x);
```

n'a aucun autre effet que de déclarer la fonction (qui peut avoir ou ne pas avoir une définition inline plus loin). L'approche efficace fournit le corps de la fonction :

```
                    inline int plusOne(int x) { return ++x; }
```

Remarquez que le compilateur vérifiera (comme il le fait toujours) l'usage correct de la liste des arguments de la fonction et de la valeur de retour (réalisant toute conversion nécessaire), ce dont est incapable le préprocesseur. Egalement, si vous essayez d'écrire le code ci-dessus comme une macro du préprocesseur, vous obtenez un effet secondaire non désiré.

Vous aurez presque toujours intérêt à placer les définitions inline dans un fichier d'en-tête. Quand le compilateur voit une telle fonction, il place le type de la fonction (la signature combinée avec la valeur de retour) *et*le corps de la fonction dans sa table de symboles. Quand vous utilisez la fonction, le compilateur s'assure que l'appel est

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

correct et que la valeur de retour est utilisée correctement, et substitue ensuite le corps de la fonction à l'appel, éliminant ainsi le temps système. Le code inline occupe réellement de la place, mais si la fonction est petite, il peut prendre moins de place que le code généré par un appel de fonction ordinaire (placer les arguments sur la pile et effectuer l'appel).

Une fonction inline dans un fichier d'en-tête a un statut spécial, puisque vous devez inclure le fichier d'en-tête contenant la fonction *et*sa définition dans tous les fichiers où la fonction est utilisée, mais vous ne vous retrouvez pas avec des erreurs de déclarations mutliples (toutefois, la déclaration doit être identique partout où la fonction inline est incluse).

9.2.1 - Les inline dans les classes

Pour définir une fonction inline, vous devez normalement faire précéder la définition de la fonction du mot-clef **inline**. Toutefois, ce n'est pas nécessaire dans une définition de classe. Toute fonction que vous définissez dans une définition de classe est automatiquement inline. Par exemple :

```
                        //: C09:Inline.cpp
// Inlines inside classes
#include <iostream>
#include <string>
using namespace std;

class Point {
  int i, j, k;
public:
  Point(): i(0), j(0), k(0) {}
  Point(int ii, int jj, int kk)
    : i(ii), j(jj), k(kk) {}
  void print(const string& msg = "") const {
    if(msg.size() != 0) cout << msg << endl;
    cout << "i = " << i << ", "
         << "j = " << j << ", "
         << "k = " << k << endl;
  }
};

int main() {
  Point p, q(1,2,3);
  p.print("valeur de p");
  q.print("valeur de q");
} ///:~
```

Ici, les deux constructeurs et la fonction **print( )**sont toutes inline par défaut. Remarquez dans le **main( )**que le fait que vous utilisez des fonctions inline est transparent, comme il se doit. Le comportement logique d'une fonction doit être identique indépendamment du fait qu'elle est inline (sinon, c'est votre compilateur qui est en cause). La seule différence que vous constaterez est la performance.

Bien sûr, la tentation est d'utiliser des fonctions inline partout dans les définitions de classe parce qu'elle vous éviteront l'étape supplémentaire d'écrire la définition externe de la fonction membre. Gardez à l'esprit, toutefois, que le but d'inline est de procurer au compilateur de meilleures opportunités d'optimisation. Mais rendre inline une grosse fonction dupliquera ce code partout où la fonction est appelée produisant une inflation du code qui peut réduire le bénéfice de rapidité (le seul procédé fiable est de faire des expériences pour évaluer les effets de **inline**sur votre programme avec votre compilateur).

9.2.2 - Fonctions d'accès

Une des utilisations les plus importantes de inline dans les classe est la *fonction d'accès*. C'est une petite fonction qui vous permet de lire ou de modifier des parties de l'état d'un objet - c'est-à-dire, une variable interne ou des variables. L'exemple suivant vous montre pourquoi inline est si important pour les fonctions d'accès :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                        //: C09:Access.cpp
// Fonctions d'accès inline

class Access {
  int i;
public:
  int read() const { return i; }
  void set(int ii) { i = ii; }
};

int main() {
  Access A;
  A.set(100);
  int x = A.read();
} ///:~
```

Ici, l'utilisateur de la classe n'est jamais en contact direct avec les variables d'état dans la classe, et elle peuvent être **private**, sous le contrôle du concepteur de la classe. Tous les accès aux données membres **private**peuvent être controlés grâce à l'interface de la fontion membre. En outre, l'accès est remarquablement efficace. Prenez le **read( )**, par exemple. Sans les inline, le code généré pour l'appel à **read( )**impliquerait typiquement de placer **this**sur la pile et réaliser un appel de fonction en assembleur. Pour la plupart des machines, la taille de ce code serait plus grande que celle du code créé par inline, et le temps d'exécution serait certainement plus grand.

Sans les fonctions inline, un concepteur de classe attentif à l'efficacité sera tenté de faire de **i**un membre publique, éliminant le temps système en autorisant l'utilisateur à y accéder directement. Du point de vue conception, c'est désastreux parce que **i**fait ainsi partie de l'interface publique, ce qui signifie que le concepteur de la classe ne pourra jamais le changer. Vous êtes coincés avec un **int**appelé **i**. C'est un problème parce que vous pourriez apprendre plus tard qu'il serait bien plus utile de représenter l'information d'état par un **float**plutôt que par un **int**, mais puisque **int i**fait partie de l'interface publique, vous ne pouvez le modifier. Ou bien vous pouvez vouloir réaliser des calculs supplémentaires en même temps que vous lisez ou affectez **i**, ce que vous ne pouvez pas faire s'il est **public**. Si, par contre, vous avez toujours utilisé des fonctions membres pour lire et changer l'état de l'information d'un objet, vous pouvez modifier la représentation sous-jacente de l'objet autant que vous le désirez.

En outre, l'utilisation de fonctions membres pour controler les données membres vous permet d'ajouter du code à la fontion membre pour détecter quand les données sont modifiées, ce qui peut être très utile pour déboguer. Si une donnée membre est **public**, n'importe qui peut la modifier sans que vous le sachiez.

Accesseurs et muteurs

Certaines personnes distinguent encore dans le concept de fonction d'accès les *accesseurs*(pour lire l'état de l'information d'un objet) et les *muteurs*(pour changer l'état d'un objet). De plus, la surcharge de fonction peut être utilisée pour donner le même nom à l'accesseur et au muteur ; la façon dont vous appelez la fonction détermine si vous lisez ou modifiez l'état de l'information. Ainsi,

```
                        //: C09:Rectangle.cpp
// Accesseurs & muteurs

class Rectangle {
  int wide, high;
public:
  Rectangle(int w = 0, int h = 0)
    : wide(w), high(h) {}
  int width() const { return wide; } // lit
  void width(int w) { wide = w; } // assigne
  int height() const { return high; } // lit
  void height(int h) { high = h; } // assigne
};

int main() {
  Rectangle r(19, 47);
  // Modifie width & height:
  r.height(2 * r.width());
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  r.width(2 * r.height());
} ///:~
```

Le constructeur utilise la liste d'initialisation du constructeur (brièvement introduite au Chapitre 8 et couverte en détails au Chapitre 14) pour initialiser les valeurs de **wide**et **high**(en utilisant la forme pseudoconstructeur pour type prédéfinis).

Vous ne pouvez avoir de noms de fonctions membres utilisant le même identifiant que les données membres, si bien que vous pouvez être tentés de différencier les données membres à l'aide d'un underscore en premier caractère. Toutefois, les identifiants avec underscore sont réservés et vous ne devez pas les utiliser.

Vous pourriez choisir, à la place, d'utiliser “get” et “set” pour indiquer accesseurs et muteurs :

```
                        //: C09:Rectangle2.cpp
// Accesseurs & muteurs avec "get" et "set"

class Rectangle {
  int width, height;
public:
  Rectangle(int w = 0, int h = 0)
    : width(w), height(h) {}
  int getWidth() const { return width; }
  void setWidth(int w) { width = w; }
  int getHeight() const { return height; }
  void setHeight(int h) { height = h; }
};

int main() {
  Rectangle r(19, 47);
  // Modifie width & height:
  r.setHeight(2 * r.getWidth());
  r.setWidth(2 * r.getHeight());
} ///:~
```

Bien sûr, accesseurs et muteurs ne sont pas nécessairement de simple tuyaux vers les variables internes. Parfois ils peuvent réaliser des calculs plus élaborés. L'exemple suivant utilise la libraire de fonction standard du C *time*pour produire une classe **Time**simple :

```
                        //: C09:Cpptime.h
// Une classe time simple
#ifndef CPPTIME_H
#define CPPTIME_H
#include <ctime>
#include <cstring>

class Time {
  std::time_t t;
  std::tm local;
  char asciiRep[26];
  unsigned char lflag, aflag;
  void updateLocal() {
    if(!lflag) {
      local = *std::localtime(&t);
      lflag++;
    }
  }
  void updateAscii() {
    if(!aflag) {
      updateLocal();
      std::strcpy(asciiRep,std::asctime(&local));
      aflag++;
    }
  }
public:
  Time() { mark(); }
  void mark() {
    lflag = aflag = 0;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
      std::time(&t);
    }
    const char* ascii() {
      updateAscii();
      return asciiRep;
    }
    // Différence en secondes:
    int delta(Time* dt) const {
      return int(std::difftime(t, dt->t));
    }
    int daylightSavings() {
      updateLocal();
      return local.tm_isdst;
    }
    int dayOfYear() { // Depuis le 1er janvier
      updateLocal();
      return local.tm_yday;
    }
    int dayOfWeek() { // Depuis dimanche
      updateLocal();
      return local.tm_wday;
    }
    int since1900() { // Année depuis 1900
      updateLocal();
      return local.tm_year;
    }
    int month() { // Depuis janvier
      updateLocal();
      return local.tm_mon;
    }
    int dayOfMonth() {
      updateLocal();
      return local.tm_mday;
    }
    int hour() { // Depuis minuit, 24 h pile
      updateLocal();
      return local.tm_hour;
    }
    int minute() {
      updateLocal();
      return local.tm_min;
    }
    int second() {
      updateLocal();
      return local.tm_sec;
    }
  };
  #endif // CPPTIME_H ///:~
```

Les fonctions de librairie standard du C ont plusieurs représentations du temps, et elles font toutes partie de la classe **Time**. Toutefois, il n'est pas nécessaire de les mettre toutes à jour, si bien qu'à la place **time_t t**est utilisée comme représentation de base, et **tm local**et la représentation en caractères ASCII **asciiRep**ont chacune des indicateurs pour indiquer si elles ont été mises à jour à l'heure courante **time_t**ou non. Les deux fonctions **private updateLocal( )**et **updateAscii( )**vérifient les indicateurs et effectuent une mise à jour conditionnelle.

Le constructeur appelle la fonction **mark( )**(que l'utilisateur peut également appeler pour forcer l'objet à représenter le temps courant), et cela vide les indicateurs pour signaler que le temps local et la représentation ASCII sont maintenant invalides. La fonction **ascii( )**appelle **updateAscii( )**, qui copie le résultat de la fonction de la librairie C standard **asctime( )**dans un buffer local car **asctime( )**utilise une zone de donnée statique qui est écrasée si la fonction est appelée ailleurs. La valeur de retour de la fonction **ascii( )**est l'adresse de ce buffer local.

Toutes les fonctions commençant avec **daylightSavings( )**utilisent la fonction **updateLocal( )**, qui rend les inline composites qui en résultent relativement grands. Cela ne semble pas intéressant, surtout en considérant que vous n'appelerez sans doute pas souvent ces fonctions. Cependant, cela ne signifie pas que toutes les fonctions doivent être rendues non inline. Si vous rendez d'autres fonctions non inline, conservez au moins **updateLocal( )**inline afin que son code soit dupliqué dans les fonctions non inline, éliminant des temps systèmes d'appel supplémentaires.

Voici un petit programme test :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                          //: C09:Cpptime.cpp
// Tester une classe time simple
#include "Cpptime.h"
#include <iostream>
using namespace std;

int main() {
  Time start;
  for(int i = 1; i < 1000; i++) {
    cout << i << ' ';
    if(i%10 == 0) cout << endl;
  }
  Time end;
  cout << endl;
  cout << "debut = " << start.ascii();
  cout << "fin = " << end.ascii();
  cout << "delta = " << end.delta(&start);
} ///:~
```

Un objet **Time**est créé, puis une activité consommant du temps est effectuée, puis un second objet **Time**est créé pour noter le temps de fin. On les utilise pour donner les temps de début, de fin et le temps écoulé.

9.3 - Stash & Stack avec les inlines

Armé des inlines, nous pouvons maintenant convertir les classes **Stash**et **Stack**pour être plus efficaces :

```
                      //: C09:Stash4.h
// Inline functions
#ifndef STASH4_H
#define STASH4_H
#include "../require.h"

class Stash {
  int size;      // Taille de chaque espace mémoire
  int quantity;  // Nombre d'espaces de stockage
  int next;      // Espace libre suivant
  // Tableau d'octets alloué dynamiquement:
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int sz) : size(sz), quantity(0),
    next(0), storage(0) {}
  Stash(int sz, int initQuantity) : size(sz),
    quantity(0), next(0), storage(0) {
    inflate(initQuantity);
  }
  Stash::~Stash() {
    if(storage != 0)
      delete []storage;
  }
  int add(void* element);
  void* fetch(int index) const {
    require(0 <= index, "Stash::fetch (-)index");
    if(index >= next)
      return 0; // Pour indiquer la fin
    // Produit un pointeur vers l'élément désiré:
    return &(storage[index * size]);
  }
  int count() const { return next; }
};
#endif // STASH4_H ///:~
```

Les petites fonctions fonctionnent bien de toute évidence comme inline, mais remarquez que les deux plus grandes fonctions ne sont pas transformées en inline, étant donné que cela n'entraînerait probablement aucun gain de performance:

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                    //: C09:Stash4.cpp {O}
#include "Stash4.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

int Stash::add(void* element) {
  if(next >= quantity) // Assez d'espace libre ?
    inflate(increment);
  // Copie l'élément dans l'espace de stockage,
  // à partir du prochain espace libre:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Index
}

void Stash::inflate(int increase) {
  assert(increase >= 0);
  if(increase == 0) return;
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copie l'ancien en nouveau
  delete [](storage); // Libère l'ancien espace de stockage
  storage = b; // Pointe vers la nouvelle mémoire
  quantity = newQuantity; // Ajuste la taille
} ///:~
```

Une fois de plus le programme de test vérifie que tout fonctionne correctement:

```
                    //: C09:Stash4Test.cpp
//{L} Stash4
#include "Stash4.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize, 100);
  ifstream in("Stash4Test.cpp");
  assure(in, "Stash4Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```

C'est le même programme de test que nous avons utilisé auparavant, donc la sortie devrait être fondamentalement la même.

La classe **Stack**fait même un meilleur usage des inline:

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                    //: C09:Stack4.h
// With inlines
#ifndef STACK4_H
#define STACK4_H
#include "../require.h"

class Stack {
  struct Link {
    void* data;
    Link* next;
    Link(void* dat, Link* nxt):
      data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack() {
    require(head == 0, "Stack pas vide");
  }
  void push(void* dat) {
    head = new Link(dat, head);
  }
  void* peek() const {
    return head ? head->data : 0;
  }
  void* pop() {
    if(head == 0) return 0;
    void* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};
#endif // STACK4_H ///:~
```

Remarquez que le destructeur de **Link**qui était présent mais vide dans la version précédente de **Stack**a été supprimé. Dans **pop( )**, l'expression **delete oldHead**libère simplement la mémoire utilisée par **Link**(elle ne détruit pas l'objet **data**pointé par le **Link**).

La plupart des fonctions peuvent être rendues inline aisément, en particulier pour **Link**. Même **pop( )**semble légitime, bien qu'à chaque fois que vous avez des variables conditionnelles ou locales il n'est pas évident que inline soit tellement profitable. Ici, la fonction est suffisamment petite pour que cela ne gêne probablement pas.

Si toutes vos fonctions *sont*inline, utiliser la librairie devient relativement simple parce qu'il n'y a pas besoin d'une édition de liens, comme vous pouvez le constater dans l'exemple test (remarquez qu'il n'y a pas de **Stack4.cpp**):

```
                    //: C09:Stack4Test.cpp
//{T} Stack4Test.cpp
#include "Stack4.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // Le nom de fichier est l'argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  string line;
  // Lire le fichier et stocker les lignes dans la pile :
  while(getline(in, line))
    textlines.push(new string(line));
  // Dépiler les lignes et les afficher:
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << *s << endl;
    delete s;
  }
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
} ///:~
```

Les gens écriront parfois des classes avec toutes les fonctions inline si bien que la classe sera entièrement dans le fichier d'en-tête (vous verrez dans ce livre que j'ai moi-même franchi cette limite). Pendant le développement c'est probablement sans conséquences, bien que cela puisse rendre la compilation plus longue. Une fois que le programme se stabilise un peu, vous aurez intérêt à revenir en arrière et rendre non inline les fonctions appropriées.

9.4 - Les inline & le compilateur

Pour comprendre quand la déclaration inline est efficace, il est utile de savoir ce que le compilateur fait quand il rencontre un inline. Comme avec n'importe quelle fonction, le compilateur détient le *type*de la fonction (c'est-à-dire, le prototype de la fonction incluant le nom et le type des arguments, combinés à la valeur de retour de la fontion) dans sa table des symboles. En outre, quand le compilateur voit que le type de la fonction inline *et*le corps de la fonction s'analysent sans erreur, le code du corps de la fonction est également amené dans la table des symboles. Que ce code soit stocké sous forme source, d'instructions assembleur compilées, ou tout aute représentation est du ressort du compilateur.

Quand vous faites un appel à une fonction inline, le compilateur s'assure tout d'abord que l'appel peut être correctement effectué. C'est-à-dire que soit les types des arguments passés doivent correspondre exactement aux types de la liste des arguments de la fonction, soit le compilateur doit être capable de faire une conversion vers le type correct, et la valeur de retour doit être du bon type (ou une conversion vers le bon type) dans l'expression de destination. Ce processus, bien sûr, est exactement ce que fait le compilateur pour n'importe quelle fonction et est nettement différent de ce que le préprocesseur réalise parce que le préprocesseur ne peut pas vérifier les types ou faire des conversions.

Si toutes les informations de type de la fonction s'adaptent bien au contexte de l'appel, alors le code inline est substitué directement à l'appel de la fonction, éliminant le surcoût de temps de l'appel et permettant davantage d'optimisations par le compilateur. Egalement, si le inline est une fonction membre, l'adresse de l'objet (**this**) est placé aux endroit appropriés, ce qui constitue bien sûr une autre opération dont le préprocesseur est incapable.

9.4.1 - Limitations

Il y a deux situations dans lesquelles le processeur ne peut pas réaliser l'opération inline. Dans ces cas, il revient simplement à la forme ordinaire d'une fonction en prenant la définition de la fonction inline et en créant l'espace de stockage pour la fonction exactement comme il le fait pour une fonction non inline. S'il doit le faire dans plusieurs unités de traduction (ce qui causerait normalement une erreur pour définitions multiples), l'éditeur de liens reçoit l'instruction d'ignorer les définitions multiples.

Le compilateur ne peut réaliser l'inline si la fonction est trop complexe. Ceci dépend du compilateur lui-même, mais au degré de complexité auquel la plupart des compilateurs abandonnent, le inline ne vous ferait gagner probablement aucune efficacité. En général, tout type de boucle est considéré trop complexe pour être développé en inline, et si vous y réfléchissez, les boucles nécessitent probablement beaucoup plus de temps dans la fonction que ce qui est requis en temps système par l'appel à la fonction. Si la fonction est simplement une collection d'instructions, le processeur n'aura probablement aucune difficulté à réaliser le inline, mais s'il y a beaucoup d'instructions, le temps système d'appel de la fonction sera beaucoup plus court que celui de l'exécution du corps de la fonction. Et rappelez-vous, à chaque fois que vous appelez une grosse fonction inline, tout le corps de la fonction est inséré à l'emplacement de chacun des appels, si bien que vous pouvez facilement obtenir un alourdissement du code sans amélioration notable des performances. (Remarquez que certains exemples dans ce livre peuvent excéder des tailles d'inline raisonnables afin de sauver de l'espace sur l'écran.)

Le compilateur ne peut pas non plus réaliser l'inline si l'adresse de la fonction est prise implicitement ou

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

explicitement. Si le compilateur doit fournir une adresse, alors il allouera un espace de stockage pour le code de la fonction et utilisera l'adresse qui y correspond. Toutefois, là où une adresse n'est pas requise, le compilateur réalisera probablement toujours l'inline du code.

Il est important de comprendre qu'un inline est seulement une suggestion au compilateur ; le compilateur n'est pas obligé de rendre inline quoi que ce soit. Un bon compilateur rendra inline les fonctions petites et simples et ignorera intelligemment les inline trop complexes. Ceci produira le résultat que vous recherchez - la sémantique d'un appel de fonction avec l'efficacité d'une macro.

9.4.2 - Déclarations aval

Si vous imaginez ce que fait le compilateur pour implémenter les inline, vous risquez de vous tromper en imaginant qu'il y a plus de limitations qu'il n'en existe réellement. En particulier, si un inline fait une déclaration aval vers une fonction qui n'a pas encore été déclarée dans la classe (que cette fonction soit inline ou non) il peut sembler que le compilateur ne sera pas capable de gérer le situation :

```
                          //: C09:EvaluationOrder.cpp
// Ordre d'évaluation de inline

class Forward {
  int i;
public:
  Forward() : i(0) {}
  // Appel à une fonction non déclarée:
  int f() const { return g() + 1; }
  int g() const { return i; }
};

int main() {
  Forward frwd;
  frwd.f();
} ///:~
```

Dans **f( )**, un appel est fait à **g( )**, bien que **g( )**n'ait pas encore été déclarée. Ceci fonctionne parce que la définition du langage déclare qu'aucune fonction inline dans une classe ne sera évaluée avant l'accolade de fermeture de la déclaration de classe.

Bien sûr, si **g( )**à son tour appelait **f( )**, vous finiriez avec un ensemble d'appels récursifs, ce qui serait trop complexe pour que le compilateur puisse réaliser l'inline. (Vous auriez également à réaliser des tests dans **f( )**ou **g( )**pour forcer l'une des deux à terminer ce qui risquerait d'être, autrement, une récursion infinie.)

9.4.3 - Activités cachées dans les constructeurs et les destructeurs

Les constructeurs et les destructeurs sont deux endroits ou vous pouvez être abusivement amenés à penser qu'un inline est plus efficace qu'il ne l'est réellement. Les constructeurs et les destructeurs peuvent avoir des activités cachées, si la classe contient des sous objets dont les constructeurs et les destructeurs doivent être appelés. Ces sous-objets peuvent être des objets membres, ou bien ils peuvent exister à cause de l'héritage (couvert au Chapitre 14). Voici un exemple de classe avec objets membres:

```
                          //: C09:Hidden.cpp
// Activités cachées dans les inline
#include <iostream>
using namespace std;

class Member {
  int i, j, k;
public:
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  Member(int x = 0) : i(x), j(x), k(x) {}
  ~Member() { cout << "~Member" << endl; }
};

class WithMembers {
  Member q, r, s; // Ont des constructeurs
  int i;
public:
  WithMembers(int ii) : i(ii) {} // Trivial?
  ~WithMembers() {
    cout << "~WithMembers" << endl;
  }
};

int main() {
  WithMembers wm(1);
} ///:~
```

Le constructeur pour **Member**est suffisamment simple pour être inline, puisqu'il n'y a de spécial à réaliser - par d'héritage ou d'objets membres qui causeraient des activités supplémentaires dissimulées. Mais dans **class WithMembers**, il y a plus en jeu que ce que l'on voit au premier abord. Les constructeurs et les destructeurs pour les objets membres **q**, **r**, et **s**sont appelés automatiquement, et *ces*constructeurs et destructeurs sont aussi inline, ce qui constitue une différence significative d'avec les fonctions membres ordinaires. Ceci ne signifie pas que vous devez systématiquement vous abstenir de déclarer inline les destructeurs et les constructeurs ; il y a des situations ou c'est intéressant. En outre, quand vous "esquissez" un programme en écrivant rapidement du code, il est souvent plus pratique d'utiliser des inline. Mais is vous êtes intéressés par l'efficacité, c'est quelque chose à quoi il faut prêter attention.

9.5 - Réduire le fouillis

Dans un livre comme celui-ci, la simplicité et la concision des définitions de fonctions inline dans les classes est très utile parce qu'on peut en mettre davantage sur une page ou un écran (pour un séminaire). Toutefois, Dan Saks Co-auteur avec Tom Plum de *C++ Programming Guidelines*, Plum Hall, 1991. a souligné que dans un projet réel cela a l'effet de rendre l'interface de la classe inutilement fouillis et ainsi de la rendre plus difficile à utiliser. Il fait référence aux fonctions membres définies dans les classes par l'expression latine *in situ*(sur place) et affirme que toutes les définitions devraient être maintenues en dehors de la classe pour garder l'interface propre. Il soutient que l'optimisation est un problème séparé. Si vous voulez optimiser, utilisez le mot-clef **inline**. En appliquant cette approche, **Rectangle.cpp**vu précédemment devient:

```
                    //: C09:Noinsitu.cpp
// Supprimer les fonctions in situ

class Rectangle {
  int width, height;
public:
  Rectangle(int w = 0, int h = 0);
  int getWidth() const;
  void setWidth(int w);
  int getHeight() const;
  void setHeight(int h);
};

inline Rectangle::Rectangle(int w, int h)
  : width(w), height(h) {}

inline int Rectangle::getWidth() const {
  return width;
}

inline void Rectangle::setWidth(int w) {
  width = w;
}

inline int Rectangle::getHeight() const {
  return height;
}
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
inline void Rectangle::setHeight(int h) {
  height = h;
}

int main() {
  Rectangle r(19, 47);
  // Transpose width & height:
  int iHeight = r.getHeight();
  r.setHeight(r.getWidth());
  r.setWidth(iHeight);
} ///:~
```

A présent, si vous voulez comparer les effets des fonctions inline aux fonctions non inline, vous pouvez simplement supprimer le mot-clef **inline**. (Toutefois les fonctions inline devraient normalement être placées dans des fichiers d'en-tête tandis que les fonctions non inline devraient être situées dans leur propre unité de traduction.) Si vous voulez mettre les fonctions dans la documentation, une simple opération de couper-coller suffit. Les fonctions *in situ*requièrent plus de travail et présentent plus de risques d'erreurs. Un autre argument en faveur de cette approche est que vous pouvez toujours produire un style de formatage cohérent pour les définitions de fonctions, ce qui n'est pas toujours le cas avec les fonctions *in situ*.

9.6 - Caractéristiques supplémentaires du préprocesseur

Un peu plus tôt, j'ai dit que vous avez *presque*toujours intérêt à utiliser des fonctions **inline**à la place des macros du préprocesseur. Les exceptions à cette règle sont quand vous avez besoin de trois caractéristiques spéciales du préprocesseur C (qui est également le préprocesseur du C++) : le chaînage, la concaténation de chaîne et le collage de jetons. Le chaînage, déjà introduit dans ce livre, est réalisé avec la directive #et vous permet de prendre un identifiant et de le transformer en tableau de caractères. La concaténation de chaîne a lieu quand deux tableaux de caractères adjacents n'ont pas d'intervalle de ponctuation, auquel cas ils sont combinés. Ces deux caractéristiques sont particulièrement utiles quand on écrit du code pour débugger. Ainsi,

```
#define DEBUG(x) cout << #x " = " << x << endl
```

Ceci affiche la valeur de n'importe quelle variable. Vous pouvez aussi obtenir une trace qui imprime les instructions au fur et à mesure de leur exécution :

```
#define TRACE(s) cerr << #s << endl; s
```

Le **#s**transforme l'instruction en tableau de caractères pour la sortie, et le deuxième **s**réitère l'instruction afin qu'elle soit exécutée. Bien sûr, ce genre de chose peut poser des problèmes, particulièrement dans des boucles **for**d'une ligne:

```
for(int i = 0; i < 100; i++)
  TRACE(f(i));
```

Comme il y a en fait deux instructions dans la macro **TRACE( )**, la boucle **for**d'une ligne exécute seulement la première. La solution est de remplacer le point-virgule par une virgule dans la macro.

9.6.1 - Collage de jeton

Le collage de jeton, implémenté avec la directive ##, est très utile quand vous créez du code. Il vous permet de prendre deux identifiants et de les coller ensemble pour créer automatiquement un nouvel identifiant. Par exemple,

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                        #define FIELD(a) char* a##_string; int a##_size
class Record {
  FIELD(one);
  FIELD(two);
  FIELD(three);
  // ...
};
```

Chaque appel à la macro **FIELD( )**crée un identifiant pour contenir un tableau de caractères et un deuxième pour contenir la longeur de ce tableau. Non seulement est-ce plus simple à lire, mais cela élimine des erreurs de codage et rend la maintenance plus facile.

9.7 - Vérification d'erreurs améliorée

Les fonctions de **require.h**ont été utilisées jusque là sans les définir (bien que **assert( )**ait été également utilisée pour aider à la détection des erreurs du programmeur quand approprié). Il est temps, maintenant, de définir ce fichier d'en-tête. Les fonctions inlines sont commodes ici parce qu'elles permettent de tout mettre dans un fichier d'en-tête, ce qui simplifie l'utilisation du package. Vous incluez simplement le fichier d'en-tête et vous n'avez pas à vous en faire à propos de l'édition des liens et du fichier d'implémentation.

Vous devriez noter que les exceptions (présentées en détail dans le deuxième volume de cet ouvrage) procure une manière bien plus efficace de gérer beaucoup de types d'erreurs - spécialement celles dont vous voudriez pouvoir récupérer - plutôt que de simplement arrêter le programme. Les conditions que gère **require.h**, toutefois, empêchent le programme de se poursuivre, par exemple si l'utilisateur ne fournit pas pas suffisamment d'arguments dans la ligne de commande ou si un fichier ne peut pas être ouvert. Ainsi, il est raisonnable qu'elles appellent la fonction de la librairie C standard **exit( )**.

Le fichier d'en-tête suivant se trouve dans le répertoire racine du livre et on peut donc y accéder facilement depuis tous les chapitres.

```
                    //: :require.h
// Teste les conditions d'erreur dans les programmes
// "using namespace std" local pour les vieux compilateurs
#ifndef REQUIRE_H
#define REQUIRE_H
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <string>

inline void require(bool requirement,
  const std::string& msg = "Requirement failed"){
  using namespace std;
  if (!requirement) {
    fputs(msg.c_str(), stderr);
    fputs("\n", stderr);
    exit(1);
  }
}

inline void requireArgs(int argc, int args,
  const std::string& msg =
    "Must use %d arguments") {
  using namespace std;
   if (argc != args + 1) {
     fprintf(stderr, msg.c_str(), args);
     fputs("\n", stderr);
     exit(1);
   }
}

inline void requireMinArgs(int argc, int minArgs,
  const std::string& msg =
    "Must use at least %d arguments") {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    using namespace std;
    if(argc < minArgs + 1) {
      fprintf(stderr, msg.c_str(), minArgs);
      fputs("\n", stderr);
      exit(1);
    }
  }

  inline void assure(std::ifstream& in,
    const std::string& filename = "") {
    using namespace std;
    if(!in) {
      fprintf(stderr, "Could not open file %s\n",
        filename.c_str());
      exit(1);
    }
  }

  inline void assure(std::ofstream& out,
    const std::string& filename = "") {
    using namespace std;
    if(!out) {
      fprintf(stderr, "Could not open file %s\n",
        filename.c_str());
      exit(1);
    }
  }
  #endif // REQUIRE_H ///:~
```

Les valeurs par défaut fournissent des messages raisonnables, qui peuvent être modifiés si besoin est.

Vous remarquerez qu'au lieu d'utiliser des arguments **char***, on utilise des arguments **const string&**. Ceci permet l'utilisation de **char***et de **string**comme arguments de ces fonctions, et est donc utilisable plus largement (vous pouvez avoir intérêt à adopter cette forme dans votre propre code).

Dans les définitions de **requireArgs( )**et **requireMinArgs( )**, le nombre d'arguments nécessaires sur la ligne de commande est augmenté de un parce que **argc**inclue toujours le nom du programme en cours d'exécution comme argument zéro, et a donc toujours une taille supérieure d'un argument au nombre d'arguments passés en ligne de commande.

Remarquez l'utilisation locale de “ **using namespace std**” dans chaque fonction. Ceci est dû au fait que certains compilateurs au moment de la rédaction de cet ouvrage n'incluaient pas - improprement - la bibliothèque de fonctions standards du C dans **namespace std**, si bien qu'une qualification explicite causerait une erreur à la compilation. Les déclarations locales permettent à **require.h**de travailler avec les deux types de biliothèques, correctes et incorrectes, sans ouvrir l'espace de nommage **std**à la place de quiconque incluant ce fichier en-tête.

Voici un programme simple pour tester **require.h**:

```
                    //: C09:ErrTest.cpp
  //{T} ErrTest.cpp
  // Teste require.h
  #include "../require.h"
  #include <fstream>
  using namespace std;

  int main(int argc, char* argv[]) {
    int i = 1;
    require(i, "value must be nonzero");
    requireArgs(argc, 1);
    requireMinArgs(argc, 1);
    ifstream in(argv[1]);
    assure(in, argv[1]); // Utilise le nom de fichier
    ifstream nofile("nofile.xxx");
    // Echoue :
  //!  assure(nofile); // L'argument par défaut
    ofstream out("tmp.txt");
    assure(out);
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
} ///:~
```

Vous pourriez être tenté de faire un pas de plus pour l'ouverture de fichiers et d'ajouter une macro à **require.h**:

```
#define IFOPEN(VAR, NAME) \
ifstream VAR(NAME); \
assure(VAR, NAME);
```

Qui pourrait alors être utilisée comme ceci :

```
IFOPEN(in, argv[1])
```

Au premier abord, cela peut paraître intéressant puisque cela signifie qu'il y a moins de choses à taper. Ce n'est pas trop dangereux, mais c'est un chemin sur lequel il vaut mieux éviter de s'engager. Remarquez que, encore une fois, une macro ressemble à une fonction mais se comporte différemment ; elle crée en fait un objet (**in**) dont la portée persiste au-delà de la macro. Peut-être comprenez-vous la situation, mais pour de nouveaux programmeurs et pour ceux qui maintiennent le code, cela constitue une énigme supplémentaire à résoudre. Le C++ est suffisamment compliqué pour qu'il n'y ait pas besoin d'ajouter à la confusion. Essayez donc de vous convaincre vous-même de ne pas utiliser les macros du préprocesseur si elles ne sont pas indispensables.

9.8 - Résumé

Il est critique que vous soyez capables de cacher l'implémentation sous-jacente d'une classe parce que vous pourriez vouloir modifier cette implémentation ultérieurement. Vous effectuerez ces changements pour des raisons d'efficacité, ou parce que vous aurez une meilleure compréhension du problème, ou parce qu'une nouvelle classe est devenue disponible et que vous voulez l'utiliser dans l'implémentation. Tout élément qui porte atteinte au caractère privé de l'implémentation sous-jacente réduit la flexibilité du langage. Ainsi, la fonction inline et très importante parce qu'elle élimine virtuellement le besoin de macros du préprocesseur et les problèmes qui y sont liés. Avec les inlines, les fonctions membres peuvent être aussi efficaces que les macros du préprocesseur.

On peut abuser de la fonction inline dans les définitions de classes, bien sûr. Le programmeur est tenté de le faire parce que c'est plus facile, et donc cela se produira. Toutefois, ce n'est pas un si gros problème par la suite, quand vous cherchez à réduire la taille, car vous pouvez toujours rendre les fonctions non inline sans effet sur leur fonctionnalité. La ligne conductrice de la programmation devrait être : “faites le fonctionner d'abord, optimisez-le ensuite”.

9.9 - Exercices

Les solutions à certains exercices peuvent être trouvées dans le document électronique *The Thinking in C++ Annotated Solution Guide*, disponible pour une somme modique sur *www.BruceEckel.com*.

1 Ecrivez un programme qui utilise la macro **F( )**montrée au début du chapitre et montrez qu'il ne se développe pas proprement, comme décrit dans le texte. Réparez la macro et montrez qu'elle fonctionne correctement.
2 Ecrivez un programme qui utilise la macro **FLOOR( )**montrée au début du chapitre. Montrez les conditions sous lesquelles il ne fonctionne pas proprement.
3 Modifiez **MacroSideEffects.cpp**afin que **BAND( )**fonctionne bien.
4 Créez deux fonctions identiques, **f1( )**et **f2( )**. Rendez **f1( )**inline, mais pas **f2( )**. Utilisez la fonction **clock( )**de la bibliothèque standard du C qui se trouve dans **<ctime>**pour noter les moments de début et de fin et comparez les deux fonctions pour voir laquelle est la plus rapide. Vous pourriez avoir besoin de faire plusieurs appels aux fonctions au sein de vos boucles de chronométrage pour obtenir des valeurs

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

significatives.

5 Faites varier la complexité du code dans les fonctions de l'exercice 4 pour voir si vous pouvez trouver le point d'équilibre où les deux fonctions prennent le même temps. Si vous en disposez, essayez de le faire avec différents compilateurs et notez les différences.

6 Montrez quelles fonctions inline font appel par défaut à la liaison interne.

7 Créez une classe qui contienne un tableau de **char**. Ajoutez un constructeur inline qui utilise la fonction **memset( )**de la bibliothèque standard du C pour initialiser le tableau à l'argument du constructeur (par défaut, faites en sorte que ce soit ' '), ajoutez une fonction membre inline appelée **print( )**pour afficher tous les caractères du tableau.

8 Prenez l'exemple **NestFriend.cpp**du chapitre 5 et remplacez toutes les fonctions membres avec des inline. Faites des fonctions inline qui ne soient pas *in-situ*. Convertissez également les fonctions **initialize( )**en constructeurs.

9 Modifiez **StringStack.cpp**du Chapitre 8 pour utiliser des fonctions inline.

10 Créez un **enum**appelé **Hue**contenant **red, blue**, et **yellow**. A présent, créez une classe appelée **Color**contenant une donnée membre de type **Hue**et un constructeur qui affecte le **Hue**à la valeur de son argument. Ajoutez des fonctions d'accès "get" et "set" pour lire et fixer le **Hue**. Rendez toutes ces fonctions inline.

11 Modifiez l'exercice 10 pour utiliser l'approche "accesseur" et "muteur".

12 Modifiez **Cpptime.cpp**de façon à ce qu'il mesure le temps qui sépare le démarrage du programme de l'instant où l'utilisateur tape la touche "Entrée".

13 Créez une classe avec deux fonctions inline, de façon à ce que la première fonction qui est définie dans la classe appelle la deuxième, sans qu'il y ait besoin d'une déclaration anticipée. Ecrivez un main qui crée un objet de la classe et appelle la première fonction.

14 Créez une classe **A**avec un constructeur par défaut inline qui s'annonce lui-même. Puis créez une nouvelle classe, **B**et placez un objet **A**comme membre de **B**, et donnez à **B**un constructeur inline. Créez un tableau d'objets **B**et voyez ce qu'il se produit.

15 Créez un grand nombre d'objets de l'exercice précédent et utilisez la classe **Time**pour chronométrer la différence entre constructeurs inline ou non. (Si vous disposez d'un *profiler*, essayez de l'utiliser également.)

16 Ecrivez un programme qui prend une **string**comme argument en ligne de commande. Ecrivez une boucle **for**qui supprime un caractère de la **string**à chaque passage, et utilisez la macro **DEBUG( )**de ce chapitre pour imprimer la **string**à chaque fois.

17 Corrigez la macro **TRACE( )**comme spécifié dans ce chapitre, et prouvez qu'elle focntionne correctement.

18 Modifiez la macro **FIELD( )**afin qu'elle contienne également un nombre **index**. Créez une classe dont les membres sont composés d'appels à la macro **FIELD( )**. Ajoutez une fonction membre qui vous permette de consulter un champ en utilisant son numéro d'index. Ecrivez un **main( )**pour tester la classe.

19 Modifiez la macro **FIELD( )**afin qu'elle génère automatiquement des fonctions d'accès pour chaque champ (les données devraient toutefois toujours être privées). Créez une classe dont les fonctions membres sont composées d'appels à la macro **FIELD( )**. Ecrivez un **main( )**pour tester la classe.

20 Ecrivez un programme qui prenne deux arguments en ligne de commande : le premier est un **int**et le second un nom de fichier. Utilisez **require.h**pour garantir que vous avez le bon nombre d'arguments, que le **int**est compris entre 5 et 10, et que le fichier peut être ouvert avec succès.

21 Ecrivez un programme qui utilise la macro **IFOPEN( )**pour ouvrir un fichier comme *input stream*. Notez la création de l'objet **ifstream**et sa portée.

22 (Difficile) Trouvez comment faire générer du code assembleur par votre compilateur. Créez un fichier contenant une très petite fonction et un **main( )**qui appelle la fonction. Générez le code assembleur quand la fonction est inline ou ne l'est pas, et montrez que la version inline n'a pas le surcoût de temps associé à l'appel.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

10 - Contrôle du nom

Créer des noms est une activité fondamentale en programmation, et quand un projet devient important, le nombre de noms peu aisément être accablant.

C++ vous permet un grand choix de contrôle sur la création et la visibilité des noms, sur l'endroit où sont stockés ces noms, et la liaison entre ces noms.

Le mot clef **static**était surchargé en C avant même que les gens sachent ce que le terme “surcharger (overload)” signifiait, et C++ ajoute encore un autre sens. Le concept fondamental que tout le monde utilise de **static**semble être “quelque chose qui reste à sa position ” (comme l'électricité statique), ce qui signifie une location physique dans la mémoire ou visible dans un fichier.

Dans ce chapitre, vous apprendrez comment mettre en place un contrôle de stockage et de visibilité **static**, et une façon d'améliorer le contrôle d'accès aux noms via le dispositif C++ *namespace*. Vous trouverez aussi comment utiliser des fonctions écrites et compilées en C.

10.1 - Eléments statiques issus du C

En C comme en C++ le mot-clé **static**a deux sens de base, qui malheureusement se recoupent souvent l'un avec l'autre:

1. Alloué une seul fois à une adresse fixe; c'est-à-dire, l'objet est créé dans une *zone de données statiques*spéciale plutôt que sur la pile à chaque fois qu'une fonction est appelée. C'est le concept de *stockage en mémoire statique*.
2. Local à une unité de compilation particulière (et local à la portée d'une classe en C++, ce que vous verrez plus tard). Ici, **static**contrôle la *visibilité*du nom, de manière à ce que ce nom ne puisse pas être vu en dehors de l'unité de compilation ou de la classe. Ceci décrit aussi le concept de *liaison*, qui détermine quels noms l'éditeur de liens verra.

Cette section va se pencher sur les sens de **static**présentés ci-dessus tels qu'ils ont hérités du C.

10.1.1 - Variables statiques à l'intérieur des fonctions

Quand vous créez une variable locale à l'intérieur d'une fonction, le compilateur alloue de l'emplacement mémoire pour cette variable à chaque fois que la fonction est appelée en déplaçant le pointeur de pile vers le bas autant qu'il le faut. S'il existe un initialiseur pour cette variable, l'initialisation est effectuée chaque fois que cette séquence est exécutée.

Parfois, cependant, vous voulez conserver une valeur entre les différents appels d'une fonction. Vous pourriez le faire au moyen d'une variable globale, mais alors cette variable ne serait plus sous le contrôle de cette seule fonction. C et C++ vous permettent de créer un objet **static**à l'intérieur d'une fonction; le stockage de cet objet en mémoire ne s'effectue alors plus sur la pile mais a lieu dans la zone des données statiques du programme. Cet objet est initialisé seulement une fois, la première fois que la fonction est appelée, et puis il conserve sa valeur entre chaque appel de fonction. Par exemple, la fonction ci-dessous retourne le prochain caractère du tableau à chaque fois que la fonction est appelé :

```
                        //: C10:StaticVariablesInfunctions.cpp
#include "../require.h"
#include <iostream>
using namespace std;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
char oneChar(const char* charArray = 0) {
  static const char* s;
  if(charArray) {
    s = charArray;
    return *s;
  }
  else
    require(s, "s n'est pas initialise");
  if(*s == '\0')
    return 0;
  return *s++;
}

char* a = "abcdefghijklmnopqrstuvwxyz";

int main() {
  // oneChar(); // require() echoue
  oneChar(a); // Initialise s a la valeur a
  char c;
  while((c = oneChar()) != 0)
    cout << c << endl;
} ///:~
```

La variable **static char* s**conserve sa valeur entre les appels de **oneChar( )**parce que son enregistrement en mémoire n'a pas lieu sur la pile de la fonction, mais dans la zone de données statiques du programme. Quand vous appelez **oneChar( )**avec un argument **char***, **s**est affecté à cet argument, et le premier caractère du tableau est retourné. Chaque appel suivant de **oneChar( )** *sans*argument produit la valeur zéro par défaut pour **charArray**, ce qui indique à la fonction que vous êtes toujours en train d'extraire des caractères depuis la valeur de **s**précédemment initialisée. La fonction continuera de fournir des caractères jusqu'à ce qu'elle atteigne le caractère nul de terminaison du tableau. Elle s'arrêtera alors d'incrémenter le pointeur afin de ne pas déborder la fin du tableau.

Mais qu'arrive-t-il si vous appelez **oneChar( )**sans aucun argument et sans avoir au préalable initialisé la valeur **s**? Dans la définition de **s**, vous pourriez avoir fourni un initialiseur,

```
static char* s = 0;
```

Mais si vous ne fournissez pas d'initialiseur pour une variable statique d'un type intégré, le compilateur garantit que cette variable sera initialisée à la valeur zéro (convertie dans le type qui convient) au démarrage du programme. Donc dans **oneChar( )**, la première fois que la fonction est appelée, **s**vaut zéro. Dans ces conditions, le test **if(!s)**réussira.

L'initialisation ci-dessus pour **s**est vraiment simple, mais l'initialisation des objets statiques (comme pour tous les autres objets) peut consister en des expressions arbitraires mettant en jeu des constantes ainsi que des fonctions et des variables précédemment déclarées.

Soyez conscients que la fonction ci-dessus est particulièrement vulnérable aux problèmes de multitâche ; à chaque fois que vous concevez des fonctions contenant des variables statiques vous devriez avoir les problématiques de multitâche à l'esprit.

Objets statiques à l'intérieur des fonctions

Les règles sont les mêmes pour un objet statique de type défini par l'utilisateur, y compris le fait que certaines initialisations sont requises pour cette objet. Cependant, l'affectation à zéro n'a un sens que pour les types de base ; les types définis par l'utilisateur doivent être initialisés avec l'appel du constructeur. Ainsi, si vous ne spécifiez pas d'argument au constructeur quand vous définissez un objet statique, la classe doit avoir un constructeur par défaut. Par example,

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                          //: C10:StaticObjectsInFunctions.cpp
#include <iostream>
using namespace std;

class X {
  int i;
public:
  X(int ii = 0) : i(ii) {} // Défaut
  ~X() { cout << "X::~X()" << endl; }
};

void f() {
  static X x1(47);
  static X x2; // Constructeur par défaut requis
}

int main() {
  f();
} ///:~
```

L'objet statique de type **X**à l'intérieur de **f( )**peut être initialisé soit avec la liste d'arguments du constructeur soit avec le constructeur par défaut. Cette construction se produit au premier passage sur la définition, et seulement la première fois.

Destructeur d'objet statique

Les destructeurs pour les objets statiques (c'est à dire, tous les objet avec un stockage statique, pas uniquement les objets locaux statiques comme dans l'exemple précédent) sont appelés quand on sort du **main( )**ou quand la fonction de la librairie standard du C **exit( )**est explicitement appelée. Dans la plus part des implémentations, **main( )**appelle simplement **exit( )**quand il se termine. Ce qui signifie que cela peut être dangereux d'appeler **exit( )**à l'intérieur d'un destructeur parce que vous pouvez aboutir à une récursivité infinie. Les destructeurs d'objets statiques *ne sont pas*appelés si vous sortez du programme en utilisant la fonction de la librairie standard du C **abort( )**.

Vous pouvez spécifier les actions à mettre en place quand vous quittez le **main( )**(ou quand vous appellez **exit( )**) en utilisant la fonction de la librairie standard C **atexit( )**. Dans ce cas, les fonctions enregistrées par **atexit( )**peuvent être appelées avant le destructeur de n'importe quel objet, avant de quitter le **main( )**(ou d'appeler **exit( )**).

Comme les destructions ordinaires, la destruction des objets statiques se produit dans l'ordre inverse de l'initialisation. Cependant, seuls les objets qui ont été construits sont détruits. Heureusement, les outils de développement gardent la trace de l'ordre d'initialisation et des objets qui ont été construits. Les objets globaux sont toujours construits avant de rentrer dans le **main( )**et détruits à la sortie du **main( )**, mais si une fonction contenant un objet statique local n'est jamais appelée, le constructeur pour cet objet n'est jamais exécuté, donc le destructeur n'est aussi jamais exécuté. Par exemple,

```
                          //: C10:StaticDestructors.cpp
// Destructeurs d'objets statiques
#include <fstream>
using namespace std;
ofstream out("statdest.out"); // fichier de trace

class Obj {
  char c; // Identifiant
public:
  Obj(char cc) : c(cc) {
    out << "Obj::Obj() pour " << c << endl;
  }
  ~Obj() {
    out << "Obj::~Obj() pour " << c << endl;
  }
};
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
Obj a('a'); // Global (stockage statique)
// Constructeur & destructeur sont toujours appelés

void f() {
  static Obj b('b');
}

void g() {
  static Obj c('c');
}

int main() {
  out << "début de main()" << endl;
  f(); // Appelle le constructeur statique de b
  // g() n'est pas appelé
  out << "fin de main()" << endl;
} ///:~
```

Dans **Obj**, le **char c**joue le rôle d'un identifiant afin que le constructeur et le destructeur puissent afficher des informations sur les objets avec lesquels ils travaillent. L' **Obj a**est un objet global, donc le constructeur est toujours appelé pour lui avant l'entrée dans le **main( )**, mais le constructeur pour le **static Obj b**dans **f( )**et le **static Obj c**dans **g( )**ne sont appelés que si ces fonctions sont appelées.

Pour démontrer quels constructeurs et destructeurs sont appelé, seul **f( )**est appelée. La sortie du programme est

```
                        Obj::Obj() for a
début de main()
Obj::Obj() for b
fin de main()
Obj::~Obj() for b
Obj::~Obj() for a
```

Le constructeur pour **a**est appelé avant d'entrer dans le **main( )**, et le constructeur pour **b**est appelé seulement parce que **f( )**est appelé. Quand on sort du **main( )**, les destructeurs pour les objets qui ont été construits sont appelés dans le sens inverse de l'ordre de construction. Ce qui signifie que si **g( )** *est*appelé, l'ordre dans lequel les destructeurs sont appelés pour **b**et **c**dépend de qui de **f( )**ou **g( )**a été appelé en premier.

Notez que l'objet **out**du fichier de trace **ofstream**est aussi un objet statique – puisqu'il est défini en dehors de toute fonction, il réside dans la zone de stockage statique. Il est important que sa définition (contrairement à une déclaration **extern**) apparaisse au début du fichier, avant toute utilisation possible de **out**. Sinon, vous utiliseriez un objet avant de l'avoir correctement initialisé.

En C++, le constructeur d'un objet statique global est appelé avant d'entrer dans le **main( )**, ainsi vous avez maintenant une façon simple et portable d'exécuter du code avant d'entrer dans le **main( )**et d'exécuter du code avec le destructeur après la sortie du **main( )**. En C, cela représentait une difficulté qui nécessitait toujours de manipuler le code assembleur de lancement de programmes du compilateur.

10.1.2 - Contrôle de l'édition de liens

Ordinairement, tout nom dont la portée est le *fichier*(c'est à dire qui n'est pas inclus dans une classe ou une fonction) est visible à travers toutes les unités de compilation dans le programme. Ceci est souvent appelé *liaison externe*parce qu'au moment de la liaison le nom est visible partout pour l'éditeur de liens, extérieur à cette unité de compilation. Les variables globales et les fonctions ordinaires sont à liaison externe.

Parfois, vous aimeriez limiter la visibilité d'un nom. Vous pourriez vouloir donner à une variable la portée fichier pour que toutes les fonctions dans ce fichier puissent l'utiliser, mais vous voudriez que les fonctions en dehors de ce fichier ne puissent pas voir ou accéder à cette variable, ou créer par inadvertance des conflits de noms en dehors du fichier.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Un objet ou un nom de fonction avec la portée fichier qui est explicitement déclaré **static**est local à son unité de compilation (selon les termes de ce livre, le fichier **cpp**où la déclaration a été faite). Ce nom a une *liaison interne*. Ceci signifie que vous pouvez utiliser le même nom dans d'autres unités de compilation sans conflit de noms.

Un avantage de la liaison interne est que le nom peut être placé dans l'entête du fichier sans se soucier qu'il puisse y avoir un conflit au moment de la liaison. Les noms qui sont en général placés dans les fichiers d'en-tête, comme les définitions de **const**et les fonctions **inline**, sont par défaut à liaison interne. (Toutefois, le **const**n'est à liaison interne par défaut qu'en C++ ; en C il est à liaison externe par défaut.) Notez que la liaison se réfère uniquement aux éléments qui ont une adresse au moment de l'édition de lien ou au chargement ; aussi les déclarations de classe et les variables locales n'ont pas de liaison.

Confusion

Voici un exemple de comment les deux sens de **static**peuvent se télescoper. Tous les objets globaux ont implicitement une classe de stockage statique, donc si vous dites (dans la portée du fichier),

```
int a = 0;
```

alors le stockage de **a**sera dans la zone des données statiques du programme, et l'initialisation pour **a**aura lieu une seule fois, avant d'entrer dans le **main( )**. De plus, la visibilité de **a**est globale à travers l'unité de compilation. En termes de visibilité, l'opposé de **static**(visible uniquement dans l'unité de compilation) est **extern**, ce qui signifie explicitement que le nom est visible dans toutes les unités de compilation. Donc la définition précédente est équivalente à

```
extern int a = 0;
```

Mais si vous dites au lieu de cela,

```
static int a = 0;
```

tout ce que vous avez fait est de changer la visibilité, afin que **a**ait une liaison interne. La classe de stockage n'est pas affectée – l'objet réside dans la zone des données statiques que la visibilité soit **static**ou **extern**.

Une fois que vous entrez dans des variables locales, **static**n'altère plus la visibilité et au lieu de cela affecte la classe de stockage.

Si vous déclarez **extern**ce qui apparaît comme une variable locale, cela signifie que le stockage existe ailleurs (donc la variable est de fait globale pour la fonction). Par exemple :

```
//: C10:LocalExtern.cpp
//{L} LocalExtern2
#include <iostream>

int main() {
  extern int i;
  std::cout << i;
} ///:~

//: C10:LocalExtern2.cpp {O}
int i = 5;
///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Avec les noms de fonctions (pour les fonctions non membre), **static**et **extern**peuvent seulement changer la visibilité, donc si vous dites

```
extern void f();
```

C'est la même chose que la simple déclaration

```
void f();
```

et si vous dites,

```
static void f();
```

Cela signifie que **f( )**est visible seulement dans l'unité de compilation – ceci est parfois appelé *statique au fichier*.

10.1.3 - Autre spécificateurs de classe de stockage

Vous verrez souvent **static**et **extern**. Deux autres spécificateurs de classe de stockage apparaissent moins souvent. Le spécificateur **auto**n'est presque jamais utilisé parce qu'il dit au compilateur que c'est une variable locale. **auto**est une abréviation de “automatique” et cela fait référence à la façon dont le compilateur alloue automatiquement le stockage pour la variable. Le compilateur peut toujours déterminer ceci en fonction du contexte dans lequel la variable est définie, donc **auto**est redondant.

Une variable **register**est une variable locale (**auto**), avec une indication au compilateur que cette variable sera fortement utilisée et le compilateur devrait la conserver, si possible, dans un registre. Ainsi, c'est une aide à l'optimisation. Les divers compilateurs répondent différemment à ce conseil ; ils ont la possibilité de l'ignorer. Si vous prenez l'adresse de la variable, le spécificateur **register**sera presque certainement ignoré. Vous devriez éviter d'utiliser **register**parce que le compilateur fera en général un meilleur travail d'optimisation que vous.

10.2 - Les namespaces

Bien que les noms puissent être dissimulés dans les classes, les noms des fonctions globales, des variables globales et des classes sont toujours dans espace de nommage global unique. Le mot-clef **static**vous donne un certain contrôle sur cela en vous permettant de donner aux variables et aux fonctions une convention de liaison interne (c'est-à-dire, de les rendre statique dans la portée du fichier). Mais dans un grand projet, un défaut de contrôle sur l'espace de nommage global peut causer des problèmes. Pour résoudre ces problèmes pour les classes, les vendeurs créent souvent des noms longs et compliqués qui ont peu de chance de causer de problème, mais alors vous n'avez d'autre choix que de taper ces noms. (Un **typedef**est souvent utilisé pour simplifier.) C'est une solution ni élégante, ni supportée par le langage.

Vous pouvez subdiviser l'espace de nommage global en morceaux plus faciles à gérer en utilisant la fonctionnalité **namespace**de C++. Le mot-clef **namespace**, similaire à **class**, **struct**, **enum**, et **union**, place les noms de ses membres dans un espace distinct. Alors que les autres mots-clefs ont des buts supplémentaires, la création d'un nouvel espace de nommage est le seul but de **namespace**.

10.2.1 - Créer un espace de nommage

La création d'un espace de nommage est notablement similaire à la création d'une classe :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                        //: C10:MyLib.cpp
namespace MyLib {
  // Declarations
}
int main() {} ///:~
```

Ce code produit un nouvel espace de nommage contenant les déclarations incluses. Il y a des différences significatives avec **class**, **struct**, **union**et **enum**, toutefois :

- La définition d'un espace de nommage ne peut apparaître qu'à la portée globale, ou imbriquée dans un autre espace de nommage.
- Il n'est pas nécessaire de placer un point virgule après l'accolade de fermeture de la définition d'un espace de nommage.
- La définition d'un espace de nommage peut être “continuée” sur plusieurs fichiers d'en-tête en utilisant une syntaxe qui, pour une classe, ressemblerait à une redéfinition :

```
                        //: C10:Header1.h
#ifndef HEADER1_H
#define HEADER1_H
namespace MyLib {
  extern int x;
  void f();
  // ...
}
```

```
                        #endif // HEADER1_H ///:~
//: C10:Header2.h
#ifndef HEADER2_H
#define HEADER2_H
#include "Header1.h"
// Ajoute plus de noms à MyLib
namespace MyLib { // Ceci N'est PAS une redéfinition!
  extern int y;
  void g();
  // ...
}
```

```
                        #endif // HEADER2_H ///:~
//: C10:Continuation.cpp
#include "Header2.h"
int main() {} ///:~
```

- Il est possible de créer un *alias*pour le nom d'une espace de nommage, si bien que vous n'avez pas à taper un nom peu maniable créé par un vendeur de bibliothèques de fonctions :

```
                        //: C10:BobsSuperDuperLibrary.cpp
namespace BobsSuperDuperLibrary {
  class Widget { /* ... */ };
  class Poppit { /* ... */ };
  // ...
}
// Trop à taper ! Je vais lui créer un alias :
namespace Bob = BobsSuperDuperLibrary;
int main() {} ///:~
```

- Vous ne pouvez pas créer une instance d'espace de nommage comme vous pouvez le faire avec une classe.

Espaces de nommage anonymes

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Chaque unité de traduction contient un espace de nommage anonyme auquel vous pouvez faire des ajouts en disant " **namespace**", sans identifiant :

```
                        //: C10:UnnamedNamespaces.cpp
namespace {
  class Arm  { /* ... */ };
  class Leg  { /* ... */ };
  class Head { /* ... */ };
  class Robot {
    Arm arm[4];
    Leg leg[16];
    Head head[3];
    // ...
  } xanthan;
  int i, j, k;
}
int main() {} ///:~
```

Les noms dans cet espace sont automatiquement disponibles dans cette unité de traduction sans qualification. Un espace de nommage anonyme unique est garanti pour chaque unité de traduction. Si vous mettez des noms locaux dans un espace de nommage anonyme, vous n'avez pas besoin de leur fournir de convention de liens interne en les rendant **static**.

C++ rend périmé l'utilisation de statiques de fichier en faveur des espaces de nommage anonymes.

Amis

Vous pouvez *injecter*une déclaration **friend**dans un espace de nommage en la déclarant au sein d'une classe de ce namespace :

```
                        //: C10:FriendInjection.cpp
namespace Me {
  class Us {
    //...
    friend void you();
  };
}
int main() {} ///:~
```

A présent, la fonction **you( )**est un membre de l'espace de nommage **Me**.

Si vous introduisez un ami dans une classe dans l'espace de nommage global, l'ami est injecté au niveau global.

10.2.2 - Utiliser un espace de nommage

Vous pouvez faire référence à un nom dans un espace de nommage de trois façons différentes : en spécifiant le nom en utilisant l'opérateur de résolution de portée, avec une directive **using**pour introduire tous les noms dans l'espace de nommage, ou bien avec une déclaration **using**introduisant les noms un par un.

Résolution de portée

Tout nom dans un espace de nommage peut être explicitement spécifié en utilisant l'opérateur de résolution de portée de la même façon que pour référencer les noms dans une classe :

```
                        //: C10:ScopeResolution.cpp
namespace X {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  class Y {
    static int i;
  public:
    void f();
  };
  class Z;
  void func();
}
int X::Y::i = 9;
```

```
                        class X::Z {
  int u, v, w;
public:
  Z(int i);
  int g();
};
```

```
                        X::Z::Z(int i) { u = v = w = i; }
int X::Z::g() { return u = v = w = 0; }
```

```
                        void X::func() {
  X::Z a(1);
  a.g();
}
int main(){} ///:~
```

Remarquez que la définition **X::Y::i**pourrait tout aussi facilement faire référence à une donnée membre d'une classe **Y**imbriquée dans une classe **X**au lieu d'un espace de nommage **X**.

Jusqu'ici, les espaces de nommage ressemblent beaucoup aux classes.

L'instruction using

Comme taper le nom complet d'un identifiant dans un espace de nommage peut rapidement devenir fastidieux, le mot-clef **using**vous permet d'importer un espace de nommage entier en une seule fois. Quand on l'utilise en conjonction avec le mot-clef **namespace**, on parle d'une *directive using*. Grâce à la directive **using**, les noms semblent appartenir au plus proche espace de nommage englobant, si bien que vous pouvez facilement utiliser les noms non qualifiés. Considérez un espace de nommage simple :

```
                        //: C10:NamespaceInt.h
#ifndef NAMESPACEINT_H
#define NAMESPACEINT_H
namespace Int {
  enum sign { positive, negative };
  class Integer {
    int i;
    sign s;
  public:
    Integer(int ii = 0)
      : i(ii),
        s(i >= 0 ? positive : negative)
    {}
    sign getSign() const { return s; }
    void setSign(sign sgn) { s = sgn; }
    // ...
  };
}
#endif // NAMESPACEINT_H ///:~
```

Une des utilisations de la directive **using**est de porter tous les noms de **Int**dans un autre espace de nommage, laissant ces noms imbriqués au sein de l'espace de nommage :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                        //: C10:NamespaceMath.h
#ifndef NAMESPACEMATH_H
#define NAMESPACEMATH_H
#include "NamespaceInt.h"
namespace Math {
  using namespace Int;
  Integer a, b;
  Integer divide(Integer, Integer);
  // ...
}
#endif // NAMESPACEMATH_H ///:~
```

Vous pouvez également déclarer tous les noms situés dans **Int**dans une fonction, mais laisser ces noms imbriqués dans la fonction :

```
                        //: C10:Arithmetic.cpp
#include "NamespaceInt.h"
void arithmetic() {
  using namespace Int;
  Integer x;
  x.setSign(positive);
}
int main(){} ///:~
```

Sans l'instruction **using**, tous les noms dans l'espace de nommage auraient besoin d'être complètement qualifiés.

Un des aspects de la directive **using**peut sembler un peu contr'intuitive au premier abord. La visibilité des noms introduits en utilisant une directive **using**est la portée dans laquelle la directive se trouve. Mais vous pouvez redéfinir ces noms par la directive **using**comme s'ils avaient été déclarés globalement à cette portée !

```
                        //: C10:NamespaceOverriding1.cpp
#include "NamespaceMath.h"
int main() {
  using namespace Math;
  Integer a; // Hides Math::a;
  a.setSign(negative);
  // A présent, la résolution de portée est nécessaire
  // pour sélectionner Math::a :
  Math::a.setSign(positive);
} ///:~
```

Supposez que vous avez un deuxième espace de nommage qui contient certains des noms contenus dans **namespace Math**:

```
                        //: C10:NamespaceOverriding2.h
#ifndef NAMESPACEOVERRIDING2_H
#define NAMESPACEOVERRIDING2_H
#include "NamespaceInt.h"
namespace Calculation {
  using namespace Int;
  Integer divide(Integer, Integer);
  // ...
}
#endif // NAMESPACEOVERRIDING2_H ///:~
```

Comme cet espace de nommage est également introduit à l'aide d'une directive **using**, il y a un risque de collision. Toutefois, l'ambiguïté apparaît au niveau où le nom est *utilisé*, pas à celui de la directive **using**:

```
                        //: C10:OverridingAmbiguity.cpp
#include "NamespaceMath.h"
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
#include "NamespaceOverriding2.h"
void s() {
  using namespace Math;
  using namespace Calculation;
  // Tout se passe bien jusqu'à :
  //! divide(1, 2); // Ambiguïté
}
int main() {} ///:~
```

Ainsi, il est possible d'écrire des directives **using**pour introduire des espaces de nommage avec des conflits de noms sans jamais produire d'ambiguïté.

La déclaration using

Vous pouvez injecter les noms un à la fois dans la portée courante en utilisant une *déclaration using*. Contrairement à la directive **using**, qui traitent les noms comme s'ils étaient déclarés globalement à la portée, une déclaration **using**est une déclaration dans la portée courante. Ceci signifie qu'elle peut être prioritaire sur les noms d'une directive **using**:

```
                        //: C10:UsingDeclaration.h
#ifndef USINGDECLARATION_H
#define USINGDECLARATION_H
namespace U {
  inline void f() {}
  inline void g() {}
}
namespace V {
  inline void f() {}
  inline void g() {}
}
#endif // USINGDECLARATION_H ///:~
```

```
                        //: C10:UsingDeclaration1.cpp
#include "UsingDeclaration.h"
void h() {
  using namespace U; // Directive using
  using V::f; // Déclaration using
  f(); // Calls V::f();
  U::f(); // Doit être complètement qualifié pour l'appel
}
int main() {} ///:~
```

La déclaration **using**donne uniquement le nom complètement spécifié de l'identifiant, mais pas d'information de type. Ainsi, si l'espace de nommage contient un ensemble de fonctions surchargées avec le même nom, la déclaration **using**déclare toutes les fonctions surchargées du même ensemble.

Vous pouvez placer une déclaration **using**partout où une déclaration normale peut se trouver. Une déclaration **using**fonctionne comme une déclaration normale à tous points de vue, sauf un : comme vous ne donnez pas de liste d'arguments, il est possible à une déclaration **using**de causer la surcharge d'une fonction avec les mêmes types d'arguments (ce qui est interdit dans le cas de la surcharge normale). Toutefois, cette ambiguïté ne se manifeste pas au point de la déclaration, mais plutôt au point d'utilisation.

Une déclaration **using**peut aussi apparaître dans un espace de nommage, et a le même effet que partout ailleurs - ce nom est déclaré dans l'espace de nommage :

```
                        //: C10:UsingDeclaration2.cpp
#include "UsingDeclaration.h"
namespace Q {
  using U::f;
  using V::g;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  // ...
}
void m() {
  using namespace Q;
  f(); // Calls U::f();
  g(); // Calls V::g();
}
int main() {} ///:~
```

Une déclaration **using**est un alias, et vous permet de déclarer la même fonction dans des espaces de nommage différents. Si vous vous retrouvez à re-déclarer la même fonction en important différents espaces de nommage, cela ne pose pas de problème - il n'y aura pas d'ambiguïté ni de duplication.

10.2.3 - L'utilisation des espace de nommage

Certaines des règles ci-dessus peuvent paraître un peu intimidantes à première vue, surtout si vous avez l'impression que vous les utiliserez tout le temps. En général, toutefois, vous pouvez vous en tirer avec une utilisation très simple des espaces de nommage tant que vous comprenez comment ils fonctionnent. Le point clef à se rappeler est que quand vous introduisez une directive **using**globale (via un “ **using namespace**” hors de toute portée) vous avez ouvert l'espace de nommage pour ce fichier. C'est généralement commode pour un fichier d'implémentation (un fichier “ **cpp**”) parce que la directive **using**n'a d'effet que jusqu'à la fin de la compilation de ce fichier. C'est-à-dire qu'il n'affecte aucun autre fichier, si bien que vous pouvez ajuster le contrôle de l'espace de nommage un fichier d'implémentation après l'autre. Par exemple, si vous découvrez une collision de noms à cause d'un trop grand nombre de directives **using**dans un fichier d'implémentation donné, il est facile de le modifier de façon à ce qu'il utilise la qualification explicite ou des déclarations **using**pour éliminer cette collision, sans modifier d'autres fichiers d'implémentation.

Les fichiers d'en-tête sont autre chose. Vous ne voudrez virtuellement jamais introduire une directive **using**globale dans un fichier d'en-tête, car cela signifierait que n'importe quel autre fichier qui inclut votre fichier d'en-tête aurait également le même espace de nommage ouvert (et les fichiers d'en-tête peuvent inclure d'autres fichiers d'en-tête).

Dans les fichiers d'en-tête vous devriez donc utiliser soit la qualification explicite soit des directives **using**incluses dans une portée et des déclarations **using**. C'est la technique que vous trouverez dans ce livre et en la suivant vous ne "polluerez" pas l'espace de nommage global et éviterez de vous retrouver de retour dans le monde pré-espace de nommage de C++.

10.3 - Membres statiques en C++

Il arrive que vous ayiez besoin qu'un seul emplacement de stockage soit utilisé par tous les objets d'une classe. En C, vous utiliseriez une variable globale, mais ce n'est pas très sûr. Les données globales peuvent être modifiées par tout le monde, et leur nom peuvent entrer en conflit avec d'autres dans un grand projet. L'idéal serait de pouvoir stocker les données comme si elles étaient globales, mais dissimulées dans une classe et clairement associées à cette classe.

Ceci peut se faire grâce aux données membres **static**au sein d'une classe. Il y a un seul emplacement de stockage pour une donnée membre **static**, indépendamment du nombre d'objets que vous créez de cette classe. Tous les objets partagent le même espace de stockage pour cette donnée membre **static**, ce qui en fait un moyen de "communication" entre eux. Mais la donnée **static**appartient à la classe ; la portée de son nom se réalise au sein de la classe et peut être **public**, **private**, ou **protected**.

10.3.1 - Définir le stockage pour les données membres statiques

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Comme une donnée **static**a un seul espace de stockage indépendamment du nombre d'objets créés, cet espace doit être défini dans un endroit unique. Le compilateur n'allouera pas d'espace pour vous. L'éditeur de lien rapportera une erreur si une donnée membre **static**est déclarée mais pas définie.

La définition doit avoir lieu en dehors de la classe (l'inlining n'est pas autorisé), et une seule définition est autorisée. Ainsi, il est courant de la mettre dans le fichier d'implémentation de la classe. La syntaxe pose parfois problème pour certains, mais elle est en fait assez logique. Par exemple, si vous créez une donnée membre static dans une classe comme ceci :

```
                        class A {
  static int i;
public:
  //...
};
```

Alors, vous devez définir le stockage pour cette donnée membre **static**dans le fichier de définition comme ceci :

```
                        int A::i = 1;
```

Si vous définissiez une variable globale ordinaire, vous écririez :

```
                        int i = 1;
```

mais ici, l'opérateur de résolution de portée et le nom de la classe sont utilisés pour spécifier **A::i**.

Certaines personnes ont des difficultés avec l'idée que **A::i**est **private**, et pourtant voici quelque chose qui semble le manipuler librement au grand jour. Est-ce que cela ne brise pas le mécanisme de protection ? C'est une pratique complètement sûre, pour deux raisons. Premièrement, le seul endroit où cette initialisation soit légale est dans la définition. De fait, si la donnée **static**était un objet avec un constructeur, vous appeleriez le contructeur au lieu d'utiliser l'opérateur **=**. Deuxièmement, une fois que la définition a eu lieu, l'utilisateur finale ne peut pas faire une deuxième définition - l'éditeur de liens renverrait une erreur. Et le créateur de la classe est forcer de créer la définition ou l'édition de liens du code ne se fera pas pendant le test. Ceci garantit que la définition n'a lieu qu'une fois et qu'elle est effectuée par le créateur de la classe.

L'expression d'initialisation complète pour un membre statique est dans la portée de la classe. Par exemple,

```
                        //: C10:Statinit.cpp
// Portée d'un initialiseur statique
#include <iostream>
using namespace std;

int x = 100;

class WithStatic {
  static int x;
  static int y;
public:
  void print() const {
    cout << "WithStatic::x = " << x << endl;
    cout << "WithStatic::y = " << y << endl;
  }
};

int WithStatic::x = 1;
int WithStatic::y = x + 1;
// WithStatic::x NOT ::x
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
int main() {
  WithStatic ws;
  ws.print();
} ///:~
```

Ici, le qualificatif **WithStatic::**étend la portée de **WithStatic**à la définition entière.

Initialisation de tableaux statiques

Le chapitre 8 introduit les variables **static const**qui vous permettent de définir une valeur constante dans le corps d'une classe. Il est possible également de créer des tableaux d'objets **static**, qu'ils soient **const**ou non. La syntaxe est raisonnablement cohérente :

```
                        //: C10:StaticArray.cpp
// Initialiser les tableaux statiques dans les classes
class Values {
  // Les static const sont intialisés sur place:
  static const int scSize = 100;
  static const long scLong = 100;
  // Le comptage automatique fonctionne avec les tableaux statiques.
  // Les tableaux, non intégraux et statiques non-const
  // doivent être intialisés extérieurement :
  static const int scInts[];
  static const long scLongs[];
  static const float scTable[];
  static const char scLetters[];
  static int size;
  static const float scFloat;
  static float table[];
  static char letters[];
};

int Values::size = 100;
const float Values::scFloat = 1.1;

const int Values::scInts[] = {
  99, 47, 33, 11, 7
};

const long Values::scLongs[] = {
  99, 47, 33, 11, 7
};

const float Values::scTable[] = {
  1.1, 2.2, 3.3, 4.4
};

const char Values::scLetters[] = {
  'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j'
};

float Values::table[4] = {
  1.1, 2.2, 3.3, 4.4
};

char Values::letters[10] = {
  'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j'
};

int main() { Values v; } ///:~
```

Avec les **static const**de types intégraux vous pouvez fournir la définition dans la classe, mais pour tout le reste (y compris des tableaux de types intégraux, même s'ils sont **static const**) vous devez fournir une définition unique pour le membre. Ces définitions ont des conventions de liens internes, si bien qu'elles peuvent être placées dans les fichiers d'en-tête. La syntaxe pour initialiser les tableaux statiques est la même que pour n'importe quel agrégat, y compris le comptage automatique.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Vous pouvez aussi créer des objets **static const**d'un type de classe et des tableaux de tels objets. Toutefois, vous ne pouvez pas les initialiser en utilisant la “syntaxe inline” autorisée pour les **static const**de type prédéfinis intégraux :

```
                        //: C10:StaticObjectArrays.cpp
// Tableaux statiques d'objets d'une classe
class X {
  int i;
public:
  X(int ii) : i(ii) {}
};

class Stat {
  // Ceci ne marche pas, bien que
  // vous puissiez en avoir envie :
//!  static const X x(100);
  // Les objets statiques de type const et non-const
  // doivent être initialisés à l'extérieur :
  static X x2;
  static X xTable2[];
  static const X x3;
  static const X xTable3[];
};

X Stat::x2(100);

X Stat::xTable2[] = {
  X(1), X(2), X(3), X(4)
};

const X Stat::x3(100);

const X Stat::xTable3[] = {
  X(1), X(2), X(3), X(4)
};

int main() { Stat v; } ///:~
```

L'initialisation de tableaux d'objets **const**et non **const static**doit être réalisée de la même façon, suivant la syntaxe typique de la définition **static**.

10.3.2 - Classes imbriquées et locales

Vous pouvez facilement placer des données membres statiques dans des classes imbriquées au sein d'autres classes. La définition de tels membres est une extension intuitive et évidente – vous utilisez simplement un niveau supplémentaire de résolution de portée. Toutefois, vous ne pouvez pas avoir de données membres **static**dans des classes locales (une classe locale est une classe définie dans une fonction). Ainsi,

```
                        //: C10:Local.cpp
// Membres statiques & classes locales
#include <iostream>
using namespace std;

// Une classe imbriquée PEUT avoir des données membres statiques :
class Outer {
  class Inner {
    static int i; // OK
  };
};

int Outer::Inner::i = 47;

// Une classe locale ne peut pas avoir de donné membre statique :
void f() {
  class Local {
  public:
//! static int i;  // Erreur
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    // (Comment définiriez-vous i?)
  } x;
}

int main() { Outer x; f(); } ///:~
```

Vous pouvez voir le problème immédiat avec un membre **static**dans une classe locale : Comment décrivez-vous la donnée membre à portée de fichier afin de la définir ? En pratique, les classes locales sont très rarement utilisées.

10.3.3 - Fonctions membres statiques

Vous pouvez aussi créer des fonctions membres **static**qui, comme les données membres **static**, travaille pour la classe comme un tout plutôt que pour un objet particulier de la classe. Au lieu de créer une fonction globale qui vit dans et “pollue” l'espace de nommage global ou local, vous portez la fonction dans la classe. Quand vous créez une fonction membre **static**, vous exprimez une association avec une classe particulière.

Vous pouvez appeler une fonction membre **static**de manière ordinaire, avec le point ou la flèche, en association avec un objet. Toutefois, il est plus courant d'appeler une focntion membre **static**elle-même, sans objet spécifique, en utilisant l'opérateur de résolution de portée comme ceci :

```
                        //: C10:SimpleStaticMemberFunction.cpp
class X {
public:
  static void f(){};
};

int main() {
  X::f();
} ///:~
```

Quand vous voyez des fonctions membres statiques dans une classe, souvenez-vous que le concepteur a voulu que cette fonction soit conceptuellement associée avec la classe dans son ensemble.

Une fonction membre **static**ne peut pas accéder aux données membres ordinaires, mais uniquement aux données membres **static**. Elle peut appeler uniquement d'autres fonctions membres **static**. Normalement, l'adresse de l'objet courant (**this**) est calmement passée quand n'importe quelle fonction membre est appelée, mais un membre **static**n'a pas de **this**, ce qui explique pourquoi il ne peut accéder aux membres ordinaires. Ainsi, vous profitez de la légère augmentation de vitesse que peut fournir une fonction globale parce qu'une fonction membre **static**n'a pas besoin du temps système supplémentaire nécessaire pour passer **this**. En même temps vous avez les bénéfices du fait d'avoir la fonction dans la classe.

Pour les données membres, **static**indique qu'un seul espace de stockage existe pour tous les objets d'une classe. Ceci est analogue à l'utilisation de **static**pour définir les objets *dans*une fonction pour signifier qu'une seule copie d'une variable locale est utilisée pour tous les appels de cette fonction.

Voici un exemple montrant un usage simultané de données et fonctions membres **static**:

```
                        //: C10:StaticMemberFunctions.cpp
class X {
  int i;
  static int j;
public:
  X(int ii = 0) : i(ii) {
     // Une fonction membre non statique peut accéder aux
     // fonctions ou données membres statiques :
    j = i;
  }
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  int val() const { return i; }
  static int incr() {
    //! i++; // Erreur : les fonctions membres statiques
    // ne peuvent accéder aux données membres non statiques
    return ++j;
  }
  static int f() {
    //! val(); // Erreur : les fonctions membres statiques
    // ne peuvent accéder aux fonction membres non statiques
    return incr(); // OK -- appelle une statique
  }
};

int X::j = 0;

int main() {
  X x;
  X* xp = &x;
  x.f();
  xp->f();
  X::f(); // Ne fonctionne qu'avec les membres statiques
} ///:~
```

Parce qu'elles n'ont pas de pointeur **this**, les fonctions membres **static**ne peuvent ni accéder aux données membres non **static**, ni appeler des fonctions membres non **static**.

Remarquez dans **main( )**qu'un membre **static**peut être sélectionné en utilisant la syntaxe habituelle (point ou flèche), associant cette fonction à un objet, mais également avec aucun objet (parce qu'un membre **static**est associé avec une classe, pas un objet particulier), en utilisant le nom de la classe et l'opérateur de résolution de portée.

Voici un aspect intéressant : A cause de la façon dont l'initialisation a lieu pour les objets membres **static**, vous pouvez placer une donnée membre **static**de la même classe *au sein*de cette classe. Voici un exemple qui n'autorise l'existence que d'un seul objet de type **Egg**(*Oeuf*, ndt) en rendant le constructeur privé. Vous pouvez accéder à cet objet, mais vous ne pouvez pas créer de nouvel objet **Egg**:

```
                        //: C10:Singleton.cpp
// Membres statiques de même type, garantissent que
// seulement un objet de ce type existe.
// Egalement dénommés technique "singleton".
#include <iostream>
using namespace std;

class Egg {
  static Egg e;
  int i;
  Egg(int ii) : i(ii) {}
  Egg(const Egg&); // Evite la copie de construction
public:
  static Egg* instance() { return &e; }
  int val() const { return i; }
};

Egg Egg::e(47);

int main() {
//!  Egg x(1); // Erreur -- ne peut créer un Egg
  // Vous pouvez accéder à la seule instance :
  cout << Egg::instance()->val() << endl;
} ///:~
```

L'initialisation de **E**a lieu après que la déclaration de la classe soit terminée, si bien que le compilateur a toutes les informations dont il a besoin pour allouer le stockage et faire appel au constructeur.

Pour éviter complètement la création de tout autre objet, quelque chose d'autre a été ajouté : un deuxième constructeur privé appelé **constructeur-copie**. A ce point du livre, vous ne pouvez pas savoir pourquoi c'est

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

nécessaire du fait que le constructeur-copie ne sera pas introduit avant le prochain chapitre. Toutefois, comme premier aperçu, si vous supprimiez le constructeur-copie défini dans l'exemple ci-dessus, vous seriez capable de créer un objet **Egg**comme ceci :

```
                        Egg e = *Egg::instance();
Egg e2(*Egg::instance());
```

Ces deux codes utilisent le constructeur-copie, afin d'éviter cette possibilité le constructeur-copie est déclaré privé (aucune définition n'est nécessaire parce qu'il n'est jamais appelé). Une grande partie du chapitre suivant est consacrée au constructeur-copie ce qui vous permettra de mieux comprendre ce dont il est question ici.

10.4 - Dépendance de l'initialisation statique

Au sein d'une unité de traduction spécifique, il est garanti que l'ordre d'initialisation des objets statiques sera le même que celui dans lequel les définitions des objets apparaissent dans cette unité de traduction. L'ordre de destruction aura la garantie de s'effectuer dans le sens inverse de l'initialisation.

Toutefois, il n'y a aucune garantie concernant l'ordre d'initialisation des objets statiques *entre*unités de traduction, et le langage ne fournit aucun moyen de spécifier cet ordre. Ceci peut être la source d'importants problèmes. Voici un exemple de désastre immédiat (qui arrêtera les systèmes d'exploitation primitifs et tuera le processus sur ceux qui sont plus sophistiqués), si un fichier contient :

```
                    //: C10:Out.cpp {O}
// Premier fichier
#include <fstream>
std::ofstream out("out.txt"); ///:~
```

et un autre fichier utilise l'objet **out**dans un de ses initialisateurs

```
                    //: C10:Oof.cpp
// Deuxième fichier
//{L} Out
#include <fstream>
extern std::ofstream out;
class Oof {
public:
  Oof() { std::out << "oups"; }
} oof;
```

```
                    int main() {} ///:~
```

le programme peut marcher ou ne pas marcher. Si l'environnement de programmation construit le programme de telle sorte que le premier fichier est initialisé avant le second, alors il n'y a aucun problème. Toutefois, si le second est initialisé avant le premier, le constructeur de **Oof**dépend de l'existence de **out**, qui n'a pas encore été construit ce qui produit une situation chaotique.

Ce problème ne se produit qu'avec les initialisations d'objets statiques *qui dépendent*les uns des autres. Les statiques dans une unité de traduction sont initialisés avant le premier appel à une fonction dans cette unité – mais ce peut être après **main( )**. Vous ne pouvez avoir de certitude sur l'ordre d'initialisation d'objets statiques s'ils se trouvent dans différents fichiers.

Un exemple plus subtil se trouve dans l'ARM. Bjarne Stroustrup et Margaret Ellis, *The Annotated C++ Reference*

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Manual, Addison-Wesley, 1990, pp. 20-21. Dans un fichier vous avez, à portée globale :

```
                    extern int y;
int x = y + 1;
```

et dans un second fichier vous avez, à portée globale également :

```
                    extern int x;
int y = x + 1;
```

Pour tous les objets statiques, le mécanisme d'édition de liens et de chargement garantit une initialisation statique à zéro avant que l'initialisation dynamique spécifiée par le programmeur ait lieu. Dans l'exemple précédent, la mise à zéro de l'espace de stockage occupé par l'objet **fstream out**n'a pas de signification spéciale, si bien qu'il est réellement indéfini jusqu'à ce que le constructeur soit appelé. Toutefois, avec les types prédéfinis, l'initialisation à zéro a *réellement*un sens, et si les fichiers sont initialisés dans l'ordre montré ci-dessus, **y**commence initialisé statiquement à zéro, si bien que **x**prend la valeur un, et **y**est dynamiquement initialisé à deux. Toutefois, si les fichiers sont initialisés dans l'ordre inverse, **x**est statiquement initialisé à zéro, **y**est dynamiquement initialisé à un, et **x**prend alors la valeur deux.

Les programmeurs doivent en être conscients parce qu'ils peuvent créer un programme avec des dépendances à l'initialisation statique qui fonctionne sur une plateforme, mais qui, après avoir été déplacé dans un autre environnement de compilation, cesse brutalement et mystérieusement de fonctionner.

10.4.1 - Que faire

Il y a trois approches pour traiter ce problème :

1. Ne le faites pas. Eviter les dépendances à l'initialisation statique est la meilleure solution.
2. Si vous devez le faire, placez les définitions d'objets statiques critiques dans un fichier unique, afin que vous puissiez contrôler de manière portable leur initialisation en les plaçant dans le bon ordre.
3. Si vous êtes convaincus qu'il est inévitable de disperser des objets statiques dans vos unités de traduction – comme dans le cas d'une bibliothèque de fonctions, où vous ne pouvez controller le programmeur qui l'utilise – il y a deux techniques de programmation pour résoudre ce problème.

Première technique

Le pionnier de cette technique est Jerry Schwarz, alors qu'il créait la bibliothèque iostream (parce que les définitions de **cin**, **cout**, et **cerr**sont **static**et se trouvent dans un fichier différent). Elle est en fait inférieure à la deuxième technique mais elle est utilisée depuis un moment et vous pourriez donc croiser du code qui l'utilise ; il est donc important que vous compreniez comment elle fonctionne.

Cette technique requiert une classe suplémentaire dans votre fichie d'en-tête. Cette classe est responsable de l'initialisation dynamique des objets statiques de votre bibliothèque. Voici un exemple simple :

```
                        //: C10:Initializer.h
// Technique d'initialisation statique
#ifndef INITIALIZER_H
#define INITIALIZER_H
#include <iostream>
extern int x; // Déclarations, non pas définitions
extern int y;

class Initializer {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  static int initCount;
public:
  Initializer() {
    std::cout << "Initializer()" << std::endl;
    // Initialise la première fois seulement
    if(initCount++ == 0) {
      std::cout << "performing initialization"
                << std::endl;
      x = 100;
      y = 200;
    }
  }
  ~Initializer() {
    std::cout << "~Initializer()" << std::endl;
    // Clean up last time only
    if(--initCount == 0) {
      std::cout << "performing cleanup"
                << std::endl;
      // Tout nettoyage nécessaire ici
    }
  }
};

// Ce qui suit crée un objet dans chaque
// fichier où Initializer.h est inclu, mais cet
// objet n'est visible que dans ce fichier :
static Initializer init;
#endif // INITIALIZER_H ///:~
```

Les déclarations de **x**et **y**annoncent seulement que ces objets existent, mais elles n'allouent pas d'espace de stockage pour les objets. Toutefois, la définition pour le **Initializer init**alloue l'espace pour cet objet dans tous les fichiers où le fichier d'en-tête est inclu. Mais parce que le nom est **static**(contrôlant cette fois la visibilité, pas la façon dont le stockage est alloué ; le stockage est à portée de fichier par défaut), il est visible uniquement au sein de cette unité de traduction, si bien que l'éditeur de liens ne se plaindra pas au sujet d'erreurs liées à des définitions multiples.

Voici le fichier contenant les définitions pour **x**, **y**et **initCount**:

```
                        //: C10:InitializerDefs.cpp {O}
// Définitions pour Initializer.h
#include "Initializer.h"
// L'initialisation statique forcera
// toutes ces valeurs à zéro :
int x;
int y;
int Initializer::initCount;
///:~
```

(Bien sûr, une instance statique fichier d' **init**est également placée dans ce fichier quand le fichier d'en-tête est inclu.) Suppposez que deux autres fichiers soient créés par l'utilisateur de la bibliothèque :

```
                        //: C10:Initializer.cpp {O}
// Initialisation statique
#include "Initializer.h"
///:~
```

and

```
                        //: C10:Initializer2.cpp
//{L} InitializerDefs Initializer
// Initialisation statique
#include "Initializer.h"
using namespace std;

int main() {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  cout << "inside main()" << endl;
  cout << "leaving main()" << endl;
} ///:~
```

Maintenant, l'ordre d'initialisation des unités de traduction n'a plus d'importance. La première fois qu'une unité de traduction contenant **Initializer.h**est initialisée, **initCount**vaudra zéro si bien que l'initialisation sera réalisée. (Ceci dépend fortement du fait que l'espace de stockage statique est fixé à zéro avant que l'initialisation dynamique n'ait lieu.) Pour le reste des unités de traduction, **initCount**ne vaudra pas zéro et l'initialisation sera omise. Le nettoyage se déroule dans l'ordre inverse, et **~Initializer( )**garantit qu'il ne se produira qu'une fois.

Cet exemple a utilisé des types prédéfinis comme les objets statiques globaux. La technique fonctionne aussi avec les classes, mais ces objets doivent alors être initialisés dynamiquement par la classe **Initializer**. Une façon de le faire est de créer les classes sans constructeurs ni destructeurs, mais à la place avec des fonctions initialisation et nettoyage utilisant des noms différents. Une approche plus habituelle, toutefois, est d'avoir des pointeurs vers des objets et de les créer en utilisant **new**dans **Initializer( )**.

Deuxième technique

Longtemps après que la première technique ait été utilisée, quelqu'un (je ne sais pas qui) a trouvé la technique expliquée dans cette section, qui est beaucoup plus simple et propre que la première. Le fait qu'il ait fallu si longtemps pour la découvrir est un hommage à la complexité du C++.

Cette technique repose sur le fait que les objets statiques dans les fonctions sont initialisés la première fois (seulement) que la fonction est appelée. Gardez à l'esprit que le problème que nous essayons de résoudre ici n'est pas *quand*les objets statiques sont initialisés (ceci peut être contrôlé séparément) mais plutôt s'assurer que l'initialisation se déroule correctement.

Cette technique est très claire et très ingénieuse. Quelle que soit la dépendance d'initialisation, vous placez un objet statique dans une fonction qui renvoie une référence vers cet objet. Ainsi, la seule façon d'accéder à cet objet statique est d'appeler la fonction, et si cet objet a besoin d'accéder à d'autres objets statiques dont il dépend il doit appeler *leur*fonction. Et la première fois qu'une fonction est appelée, cela force l'initialisation. L'ordre correct d'initialisation statique est garanti par la structure du code, pas par un ordre arbitraire établi par l'éditeur de liens.

Pour constuire un exemple, voici deux classes qui dépendent l'une de l'autre. La première contient un **bool**qui est initialisé uniquement par le constructeur, si bien que vous pouvez dire si le constructeur a été appelé pour une instance statique de la classe (l'aire de stockage statique est initialisée à zéro au démarrage du programme, ce qui produit une valeur **false**pour le **bool**si le constructeur n'a pas été appelé) :

```
                        //: C10:Dependency1.h
#ifndef DEPENDENCY1_H
#define DEPENDENCY1_H
#include <iostream>

class Dependency1 {
  bool init;
public:
  Dependency1() : init(true) {
    std::cout << "Dependency1 construction"
              << std::endl;
  }
  void print() const {
    std::cout << "Dependency1 init: "
              << init << std::endl;
  }
};
#endif // DEPENDENCY1_H ///:~
```

Le constructeur signale aussi quand il est appelé, et vous pouvez afficher (**print( )**) l'état de l'objet pour voir s'il a

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

été initialisé.

La deuxième classe est initialisée à partir d'un objet de la première, ce qui causera la dépendance:

```
                        //: C10:Dependency2.h
#ifndef DEPENDENCY2_H
#define DEPENDENCY2_H
#include "Dependency1.h"

class Dependency2 {
  Dependency1 d1;
public:
  Dependency2(const Dependency1& dep1): d1(dep1){
    std::cout << "Dependency2 construction ";
    print();
  }
  void print() const { d1.print(); }
};
#endif // DEPENDENCY2_H ///:~
```

Le constructeur s'annonce et affiche l'état de l'objet **d1**afin que vous puissiez voir s'il a été initialisé au moment où le constructeur est appelé.

Pour montrer ce qui peut mal se passer, le fichier suivant commence par mettre les définitions statiques dans le mauvais ordre, comme elles se produiraient si l'éditeur de liens initialisait l'objet **Dependency2**avant l'objet **Dependency1**. Puis l'ordre est inversé pour montrer comment il fonctionne correctement si l'ordre se trouve être le “bon”. Finalement, la deuxième technique est démontrée.

Pour fournir une sortie plus lisible, la fonction **separator( )**est créée. Le truc est que vous ne pouvez pas appeler une fonction globalement à moins que cette fonction ne soit utilisée pour réaliser l'initialisation de la variable, si bien que **separator( )**renvoie une valeur factice qui est utilisée pour initialiser deux variables globales.

```
                        //: C10:Technique2.cpp
#include "Dependency2.h"
using namespace std;

// Renvoie une valeur si bien qu'elle peut être appelée comme
// un initialiseur global :
int separator() {
  cout << "--------------------" << endl;
  return 1;
}

// Simule le problème de dépendence :
extern Dependency1 dep1;
Dependency2 dep2(dep1);
Dependency1 dep1;
int x1 = separator();

// Mais s'il se produit dans cet ordre, cela marche bien :
Dependency1 dep1b;
Dependency2 dep2b(dep1b);
int x2 = separator();

// Englober les objets statiques dans des fonctions marche bien
Dependency1& d1() {
  static Dependency1 dep1;
  return dep1;
}

Dependency2& d2() {
  static Dependency2 dep2(d1());
  return dep2;
}

int main() {
  Dependency2& dep2 = d2();
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
} ///:~
```

Les fonctions **d1( )**et **d2( )**englobent les instances statiques des objets **Dependency1**et **Dependency2**. A présent, la seule façon d'accéder aux objets statiques est d'appeler les fonctions et cela force l'initialisation statique lors du premier appel de la fonction. Ceci signifie que l'initialisation est garantie correcte, ce que vous verrez lorsque vous exécuterez le programme et regarderez la sortie.

Voici comment vous organiseriez réellement le code pour utiliser la technique. Normalement, les objets statiques seraient définis dans des fichiers différents (parce que vous y êtes forcés pour une raison quelconque; souvenez-vous que définir les objets statiques dans différents fichiers est la cause du problème), et à la place vous définissez les fonctions englobantes dans des fichiers séparés. Mais elles devront être déclarées dans des fichiers d'en-tête :

```
                        //: C10:Dependency1StatFun.h
#ifndef DEPENDENCY1STATFUN_H
#define DEPENDENCY1STATFUN_H
#include "Dependency1.h"
extern Dependency1& d1();
#endif // DEPENDENCY1STATFUN_H ///:~
```

En fait, le “extern” est redondant pour la déclaration de fonction. Voici le deuxième fichier d'en-tête :

```
                        //: C10:Dependency2StatFun.h
#ifndef DEPENDENCY2STATFUN_H
#define DEPENDENCY2STATFUN_H
#include "Dependency2.h"
extern Dependency2& d2();
#endif // DEPENDENCY2STATFUN_H ///:~
```

Dans les fichiers d'implémentation où vous auriez auparavant placé les définitions des objets statiques, vous mettez maintenant à la place les définitions des fonctions englobantes :

```
                        //: C10:Dependency1StatFun.cpp {O}
#include "Dependency1StatFun.h"
Dependency1& d1() {
  static Dependency1 dep1;
  return dep1;
} ///:~
```

A priori, du code supplémentaire peut également être placé dans ces fichiers. Voici l'autre fichier :

```
                        //: C10:Dependency2StatFun.cpp {O}
#include "Dependency1StatFun.h"
#include "Dependency2StatFun.h"
Dependency2& d2() {
  static Dependency2 dep2(d1());
  return dep2;
} ///:~
```

Ainsi, il y a maintenant deux fichiers qui pourraient être liés dans n'importe quel ordre et s'ils contenaient des objets statiques ordinaires pourraient supporter n'importe quel ordre d'initialisation. Mais comme ils contiennent les fonctions englobantes, il n'y a aucun risque d'initialisation impropre :

```
                        //: C10:Technique2b.cpp
//{L} Dependency1StatFun Dependency2StatFun
#include "Dependency2StatFun.h"
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
int main() { d2(); } ///:~
```

Quand vous exécuterez ce programme vous verrez que l'initialisation de l'objet statique **Dependency1**a toujours lieu avant l'initialisation de l'objet statique **Dependency2**. Vous pouvez aussi constater que c'est une approche beaucoup plus simple que la première technique.

Vous pourriez être tentés d'écrire **d1( )**et **d2( )**comme des fonctions inline dans leur fichier d'en-tête respectif, mais c'est quelque chose qu'il faut absolument éviter. Une fonction inline peut être dupliquée dans tous les fichiers où elle apparaît – et cette duplication *inclut*la définition des objets statiques. Comme les fonctions inline sont automatiquement par défaut en convention de liaison interne, ceci entrainerait la présence de plusieurs objets statiques parmi les différentes unités de traduction, ce qui causerait certainement des problèmes. Vous devez donc vous assurer qu'il n'y a qu'une seule définition de chaque fonction englobante, et ceci veut dire qu'il ne faut pas les rendre inline.

10.5 - Spécification alternative des conventions de liens

Que se passe-t-ilsi vous écrivez un programme en C++ et que vous voulez utiliser une bibliothèque de fonctions C ? Si vous faites la déclaration de fonction C,

```
float f(int a, char b);
```

le compilateur C++ décorera ce nom en quelque chose comme **_f_int_char**pour supporter la surcharge de fonction (et la convention de liens sécurisée au niveau des types). Toutefois, le compilateur C qui a compilé votre bibliothèque C *n*'a plus que probablement *pas*décoré le nom, si bien que son nom interne sera **_f**. Ainsi, l'éditeur de liens ne sera pas capable de résoudre vos appels en C++ à **f( )**.

Le mécanisme d'échappement fourni en C++ est la *spécification alternative des conventions de liens*, qui a été produite dans le langage en surchargeant le mot-clef **extern**. **extern**est suivi par une chaîne qui spécifie la convention de liens que vous désirez pour la déclaration, suivie par la déclaration :

```
extern "C" float f(int a, char b);
```

Ceci dit au compilateur de fournir une convention de lien C à **f( )**afin que le compilateur ne décore pas le nom. Les deux seuls types de spécification de convention de liens supportées par le standard sont **"C"**et **"C++,"**mais les vendeurs de compilateurs ont la possibilité de supporter d'autres langages de la même façon.

Si vous avez un groupe de déclarations avec une convention de liens alternative, mettez-les entre des accolades, comme ceci :

```
extern "C" {
  float f(int a, char b);
  double d(int a, char b);
}
```

Ou, pour un fichier d'en-tête,

```
extern "C" {
#include "Myheader.h"
}
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

La plupart des vendeurs de compilateurs C++ gèrent la spécification de convention de liens alternative dans leurs fichiers d'en-tête qui fonctionnent à la fois en C et en C++, et vous n'avez donc pas à vous en inquiéter.

10.6 - Sommaire

Le mot clef **static**peut être confus parce que dans beaucoup de situations, il contrôle la position du stockage, et dans d'autre cas il contrôle la visibilité et le lien des noms.

Avec l'introduction des espaces de noms (namespaces) en C++, vous avez une alternative améliorée et plus flexible pour commander la prolifération des noms dans de grands projets.

L'utilisation de **static**à l'intérieur des classes est une possibilité supplémentaire de contrôler les noms dans un programme. Les noms ne peuvent pas être en conflit avec des noms globaux, et la visibilité et l'accès est gardé dans le programme, donnant un meilleur contrôle dans l'entretien de votre code.

10.7 - Exercices

La solution de certains exercices sélectionnés peut se trouver dans le document électronique *The Thinking in C++ Annotated Solution Guide*, disponible pour une somme modique à www.BruceEckel.com.

1 Créez une fonction avec une variable statique qui est un pointeur (avec un argument par défaut à zéro). Quand l'appelant fournit une valeur pour cet argument, elle est utilisée pour pointer au début d'un tableau de **int**. Si vous appelez la fonction avec un argument à zéro (utilisant l'argument par défaut), la fonction retourne la prochaine valeur dans le tableau, jusqu'à ce qu'il voit une valeur “-1” dans le tableau (pour agir comme indicateur de fin de tableau). Testez cette fonction dans le **main( )**.

2 Créez une fonction qui retourne la prochaine valeur d'une séquence de Fibonacci à chaque fois que vous l'appelez. Ajoutez un argument de type **bool**avec une valeur par défaut à **false**tels que quand vous donnez l'argument à **true**il “réinitialise” la fonction au début de la séquence de Fibonacci. Testez cette fonction dans le **main( )**.

3 Créez une classe avec un tableau de **int**s. Définissez la taille du tableau en utilisant **static const int**dans la classe. Ajoutez une variable **const int**, et initialisez la dans la liste d'initialisation du constructeur ; rendez le constructeur **inline**. Ajoutez un membre **static int**et initialisez le avec une valeur spécifique. Ajoutez une fonction membre **static**qui affiche la donnée du membre **static**. Ajoutez une fonction membre **inline**appelant **print( )**pour afficher sur la sortie toutes les valeurs du tableau et pour appeler la fonction membre **static**. Testez cette fonction dans le **main( )**.

4 Créez une classe appelée **Monitor**qui garde la trace du nombre de fois qu'une fonction membre **incident( )**a été appelée. Ajoutez une fonction membre **print( )**qui affiche le nombre d'incidents. Maintenant créez une fonction globale (pas une fonction membre) contenant un objet **static Monitor**. Chaque fois que vous appelez cette fonction, elle appellera **incident( )**,puis **print( )**pour afficher le compteur d'incidents. Testez cette fonction dans le **main( )**.

5 Modifiez la classe **Monitor**de l'exercice 4 pour que vous puissiez décrémenter (**decrement( )**) le compteur d'incidents. Fabriquez une classe **Monitor2**qui prend en argument du constructeur un pointeur sur un **Monitor1**, et qui stocke ce pointeur et appelle **incident( )**et **print( )**. Dans le destructeur de **Monitor2**, appelez **decrement( )**et **print( )**. Maintenant fabriquez un objet **static Monitor2**dans une fonction. A l'intérieur du **main( )**, expérimentez en appelant et en n'appelant pas la fonction pour voir ce qui arrive avec le destructeur de **Monitor2**.

6 Fabriquez un objet global **Monitor2**et voyez ce qui arrive.

7 Créez une classe avec un destructeur qui affiche un message et qui appelle **exit( )**. Créez un objet global de cette classe et regardez ce qui arrive.

8 Dans **StaticDestructors.cpp**, expérimentez avec l'ordre d'appel des constructeurs et destructeurs en appelant **f( )**et **g( )**dans le **main( )**dans des ordres différents. Votre compilateur le fait-il correctement ?

9 Dans **StaticDestructors.cpp**, testez la gestion des erreurs par défaut de votre implémentation en changeant

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

la définition originale de **out**en une déclaration **extern**et mettez la définition actuelle après la définition de **a**(à qui le constructeur d' **Obj**envoie des informations à **out**). Assurez vous qu'il n'y ait rien d'important qui tourne sur votre machine quand vous lancez le programme ou alors que votre machine gère solidement les plantages.

10 Prouvez que les variables statique de fichier dans un fichier d'en-tête ne sont pas en désaccord les unes avec les autres quand elles sont incluses dans plus d'un fichier **cpp**.

11 Créez une classe simple contenant un **int**, un constructeur qui initialise le **int**depuis son argument, une fonction membre pour affecter l' **int**avec son argument, et une fonction **print( )**qui affiche ce **int**. Mettez votre classe dans un fichier d'en-tête, et incluez le fichier d'en-tête dans deux fichiers **cpp**. Dans un fichier **cpp**faites une instance de votre classe, et dans l'autre déclarez un identifiant **extern**et testez le dans le **main( )**. Rappelez vous que vous devez lier les deux fichiers objet ou sinon le linker ne pourra pas trouver l'objet.

12 Rendez **statique**l'instance de l'objet de l'exercice 11 et vérifiez que il ne peut pas être trouvé par le linker pour cette raison.

13 Déclarez une fonction dans un fichier d'en-tête. Définissez la fonction dans un fichier **cpp**et appelez la dans le **main( )**dans un second fichier **cpp**. Compilez et vérifiez que ça marche. Maintenant changez la définition de la fonction de façon à la rendre **static**et vérifiez que le linker ne peut pas le trouver.

14 Modifiez **Volatile.cpp**du chapitre 8 pour faire de **comm::isr( )**quelque chose qui pourrait en réalité marcher comme une routine de service d'interruption. Conseil : une routine de service d'interruption ne prend aucun argument.

15 Ecrivez et compilez un programme simple qui utilise les mots clef **auto**et **register**.

16 Créez un fichier d'en-tête contenant un **namespace**. A l'intérieur de ce **namespace**créez plusieurs déclarations de fonctions. Maintenant créez un second fichier d'en-tête qui inclut le premier et qui continue le **namespace**, en ajoutant plusieurs autres déclarations de fonctions. Maintenant créez un fichier **cpp**qui inclut le second fichier d'en-tête. Renommez votre namespace avec un autre diminutif (plus petit). Dans la définition d'une fonction, appelez une de vos fonctions utilisant la résolution de portée. Dans une définition de fonction séparée, écrivez une directive **using**permettant d'insérer votre namespace dans la portée de cette fonction, et montrez que vous n'avez pas besoin de résolution de portée pour appeler la fonction depuis votre namespace.

17 Créez un fichier d'en-tête avec un namespace anonyme. Inclure l'en-tête dans deux fichiers **cpp**séparés et montrez que un espace anonyme est unique pour chaque unité de traduction.

18 En utilisant le fichier d'en-tête de l'exercice 17, montrez que les noms dans le namespace anonyme sont automatiquement disponibles dans l'unité de traduction sans les qualifier.

19 Modifiez **FriendInjection.cpp**pour ajouter une définition pour la fonction amie et appelez la fonction dans le **main( )**.

20 Dans **Arithmetic.cpp**, démontrez que la directive **using**ne s'étend pas en dehors de la portée la fonction dans laquelle la directive fut créée.

21 Réparez le problème dans **OverridingAmbiguity.cpp**, d'abord avec la résolution de portée, puis à la place de cela avec une déclaration **using**qui force le compilateur à choisir l'une des fonctions ayant des noms identique.

22 Dans deux fichiers d'en-tête, créez deux namespaces, chaque un contenant une classe (avec toutes les définitions inline) avec un nom identique à celui dans l'autre namespace. Créez un fichier **cpp**qui inclut les deux fichiers d'en-tête. Créez une fonction, et à l'intérieur de la fonction utilisez la directive **using**pour introduire les deux namespace. Essayez de créer un objet de la classe et de voir ce qui arrive. Rendez la directive **using**globale (en dehors de la fonction) pour voir si cela fait une différence. Réparez le problème utilisant la résolution de portée, et créez des objets de chaque classe.

23 Réparez le problème de l'exercice 22 avec une déclaration **using**qui force le compilateur à choisir un nom des classes identiques.

24 Extrayez la déclaration de namespace dans **BobsSuperDuperLibrary.cpp**et **UnnamedNamespaces.cpp**et mettez les dans des fichiers d'en-tête séparés, en donnant un nom au namespace anonyme dans ce processus. Dans un troisième fichier d'en-tête créez un nouveau namespace qui combine les éléments des deux autres namespace avec des déclarations **using**. Dans le **main( )**, introduisez votre nouveau namespace avec une directive **using**et accédez à tous les éléments de votre namespace.

25 Créez un fichier d'en-tête qui inclut **<string>**et **<iostream>**mais sans utiliser aucune directive **using**ou déclaration **using**. Ajoutez les “gardes d'inclusion” comme vous l'avez vu dans le chapitre sur les fichier

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

d'en-tête dans ce livre. Créez une classe avec toutes ses fonctions inline qui contient un membre **string**, avec un constructeur qui initialise ce **string**depuis son argument et une fonction **print( )**qui affiche le **string**. Créez un fichier **cpp**et testez votre classe dans le **main( )**.

26 Créez une classe contenant un **static double**et **long**. Ecrivez une fonction membre **static**qui affiche les valeurs.

27 Créez une classe contenant un **int**, un constructeur qui initialise le **int**depuis son argument, et une fonction **print( )**pour afficher le **int**. Maintenant créez une seconde classe qui contient un objet **static**de la première. Ajoutez une fonction membre **static**qui appelle la fonction **print( )**de l'objet **static**. Testez votre classe dans le **main( )**.

28 Créez une classe contenant deux tableaux **static**de **int**l'un **const**et l'autre non- **const**. Ecrivez des méthodes **static**pour afficher les tableaux. testez votre classe dans le **main( )**.

29 Créez une class contenant une **string**, avec un constructeur qui initialise la **string**depuis son argument, et une fonction **print( )**pour afficher la **string**. Créez une autre classe qui contient deux tableaux d'objet **static const**et non- **const**de la première classe, et des méthodes **static**pour afficher ces tableaux. Testez cette seconde classe dans le **main( )**.

30 Créez une **struct**qui contient un **int**et un constructeur par défaut qui initialise le **int**à zéro. Rendez cette **struct**locale à une fonction. A l'intérieur de cette fonction, créez un tableau d'objets de votre **struct**et démontrez que chaque **int**dans le tableaux a automatiquement était initialisé à zéro.

31 Créez une classe qui représente une connexion d'imprimante, et qui ne vous autorise qu'une seule imprimante.

32 Dans un fichier d'en-tête, créez une classe **Mirror**qui contient deux données membre : un pointeur sur un objet **Mirror**et un **bool**. Donnez lui deux constructeurs : le constructeur par défaut initialise le **bool**à **true**et le pointeur de **Mirror**à zéro. Le second constructeur prend en argument un pointeur sur un objet **Mirror**, qu'il affecte au pointeur interne de l'objet ; il met le **bool**à **false**. Ajoutez une fonction membre **test( )**: si le pointeur de l'objet n'est pas zéro, il retourne la valeur de **test( )**appelé à travers le pointeur. Si le pointeur est zéro, il retourne le **bool**. Maintenant créez cinq fichiers **cpp**, chacun incluant le fichier d'en-tête de **Mirror**. Le premier fichier **cpp**définit un objet global **Mirror**utilisant le constructeur par défaut. Le second fichier déclare l'objet du premier fichier comme **extern**, et définit un objet global **Mirror**utilisant le second constructeur, avec un pointeur sur le premier objet. Continuez à faire cela jusqu'au dernier fichier, qui contiendra aussi une définition d'objet global. Dans ce fichier, **main( )**devrait appeler la fonction **test( )**et reporter le résultat. Si le résultat est **true**, trouvez comment changer l'ordre des liens sur votre linker et changez le jusqu'à ce que le résultat soit **false**.

33 Réparez le problème de l'exercice 32 en utilisant la technique numéro un montrée dans ce livre.

34 Réparez le problème de l'exercice 32 en utilisant la technique numéro deux montrée dans ce livre.

35 Sans inclure un fichier d'en-tête, déclarez la fonction **puts( )**depuis la librairie standard C. Appelez cette fonction depuis le **main( )**.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

11 - Références & le constructeur de copie

Les références sont comme des pointeurs constants qui sont automatiquement déréférencés par le compilateur.

Bien que les références existent aussi en Pascal, la version C++ est issue du langage Algol. Ils sont essentiels en C++ pour maintenir la syntaxe de la surcharge d'opérateur (voir le chapitre 12), mais ils sont aussi une convenance générale pour contrôler les arguments qui sont passés à l'intérieur et à l'extérieur des fonctions.

Ce chapitre regardera d'abord brièvement la différence entre les pointeurs en C et C++, puis introduira les références. Mais la majeure partie du chapitre fouillera dans une notion un peu confuse pour les programmeurs C++ novices : le constructeur de copie, un constructeur spécial (les références sont requises) qui fabriquent un nouvel objet à partir d'un objet existant du même type. Le constructeur de copie est utilisé par le compilateur pour donner et récupérer des objets *par valeur*à l'intérieur et à l'extérieur des fonctions.

Finalement, les notions de *pointeur sur membre*qui sont obscures en C++ seront éclairées.

11.1 - Les pointeurs en C++

La différence la plus importante entre les pointeurs en C et en C++ est que C++ est un langage plus fortement typé. Ceci ressort quand **void***est concerné. Le C ne vous laisse pas négligemment assigner un pointeur d'un type à un autre, mais il vous permet *de fait*de la réaliser à l'aide d'un **void***. Ainsi,

```
                    bird* b;
rock* r;
void* v;
v = r;
b = v;
```

Comme cette particularité du C vous permet de traiter tranquillement n'importe quel type comme n'importe quel autre, elle crée une faille dans le système des types. Le C++ ne le permet pas ; le compilateur émet un message d'erreur, et si vous voulez vraiment traiter un type comme s'il en était un autre, vous devez le rendre explicite, au compilateur comme au lecteur, en utilisent le transtypage. (Le chapitre 3 a introduit la syntaxe “explicite” améliorée de transtypage du C++.)

11.2 - Les références en C++

Une *reference*(**&**) est comme un pointeur constant qui est automatiquement déréférencé. Elles sont généralement utilisées pour les listes d'arguments des fonctions et les valeurs retournées par celles-ci. Mais vous pouvez également créer une référence autonome. Par exemple,

```
                    //: C11:FreeStandingReferences.cpp
#include <iostream>
using namespace std;

// Référence ordinaire autonome :
int y;
int& r = y;
// Quand une référence est créée, elle doit
// être initialisée vers un objet vivant.
// Toutefois, vous pouvez aussi dire :
const int& q = 12;  // (1)
// Les références sont liées au stockage de quelqu'un d'autre :
int x = 0;          // (2)
int& a = x;         // (3)
int main() {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    cout << "x = " << x << ", a = " << a << endl;
    a++;
    cout << "x = " << x << ", a = " << a << endl;
  } ///:~
```

En ligne (1), le compilateur alloue un espace de stockage, l'initialise avec la valeur 12, et lie la référence à cette espace de stockage. Toute la question est que toute référence doit être liée à l'espace de stockage de quelqu'un *d'autre*. Quand vous accédez à une référence, vous accédez à cet espace. Ainsi, si vous écrivez des lignes comme les lignes (2) et (3), alors incrémenter **a**revient à incrémenter **x**, comme le montre le **main( )**. Une fois encore, la manière la plus facile de se représenter une référence est un pointeur sophistiqué. Un avantage de ce “pointeur” est que vous n'avez jamais à vous demander s'il a été intialisé (le compilateur l'impose) et comment le déréférencer (le compilateur le fait).

Il y a certaines règles quand on utilise des références :

1 Une référence doit être initialisée quand elle est créée. (Les pointeurs peuvent être initialisés n'importe quand.)
2 Une fois qu'une référence a été initialisée vers un objet, elle ne peut être modifiée afin de pointer vers un autre objet. (Les pointeurs peuvent pointer vers un autre objet à tout moment.)
3 Vous ne pouvez pas avoir de références NULLES. Vous devez toujours pouvoir supposer qu'une référence est connectée à un espace de stockage légitime.

11.2.1 - Les références dans les fonctions

L'endroit le plus courant où vous verrez des références est dans les arguments de fonctions et les valeurs de retour. Quand une référence est utilisée comme argument de fonction, toute modification de la référence *dans*la fonction causera des changements à l'argument *en dehors*de la fonction. Bien sûr, vous pourriez faire la même chose en passant un pointeur, mais une référence a une syntaxe beaucoup plus propre. (Si vous voulez, vous pouvez considérez une référence comme une simple commodité de syntaxe.)

Si vous renvoyez une référence depuis une fonction, vous devez prendre les mêmes soins que si vous retourniez un pointeur depuis une fonction. Quel que soit l'élément vers lequel pointe une référence, il ne devrait pas disparaître quand la fonction effectue son retour, autrement vous feriez référence à un espace de mémoire inconnu.

Voici un exemple :

```
                          //: C11:Reference.cpp
// Références simples en C++

int* f(int* x) {
  (*x)++;
  return x; // Sûr, x est en dehors de cette portée
}

int& g(int& x) {
  x++; // Même effet que dans f()
  return x; // Sûr, en dehors de cette portée
}

int& h() {
  int q;
//!  return q;  // Erreur
  static int x;
  return x; // Sûr, x vit en dehors de cette portée
}

int main() {
  int a = 0;
  f(&a); // Moche (mais explicite)
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  g(a);  // Propre (mais caché)
} ///:~
```

L'appel à **f( )**n'a pas la commodité ni la propreté de l'utilisation de références, mais il est clair qu'une adresse est passée. Dans l'appel à **g( )**, une adresse est passée (par référence), mais vous ne le voyez pas.

Références const

L'argument référence dans **Reference.cpp**fonctionne seulement quand l'argument n'est pas un objet **const**. Si c'est un objet **const**, la fonction **g( )**n'acceptera pas l'argument, ce qui est en fait une bonne chose, parce que la fonction modifie *réellement*l'argument externe. Si vous savez que la fonction respectera la **const**ance d'un objet, faire de l'argument une référence **const**permettra à la fonction d'être utilisée dans toutes les situations. Ceci signifie que, pour les types prédéfinis, la fonction ne modifiera pas l'argument, et pour les types définis par l'utilisateur, la fonction appelera uniquement des fonctions membres **const**, et ne modifiera aucune donnée membre **public**.

L'utilisation de références **const**dans les arguments de fonctions est particulièrement important parce que votre fonction peut recevoir un objet temporaire. Il a pu être créé comme valeur retour d'une autre fonction ou bien explicitement par l'utilisateur de votre fonction. Les objets temporaires sont toujours **const**, donc si vous n'utilisez pas une référence **const**, cet argument ne sera pas accepté par le compilateur. Comme exemple très simple,

```
                        //: C11:ConstReferenceArguments.cpp
// Passer une référence comme const

void f(int&) {}
void g(const int&) {}

int main() {
//!  f(1); // Erreur
  g(1);
} ///:~
```

L'appel à **f(1)**provoque une erreur de compilation parce que le compilateur doit d'abord créer une référence. Il le fait en allouant du stockage pour un **int**, en l'initialisant à un et en produisant l'adresse pour le lier à la référence. Le stockage *doit*être un **const**parce que le modifier n'aurait aucun sens – vous ne pourrez jamais l'atteindre à nouveau. Avec tous les objets temporaires vous devez faire la même supposition : qu'ils sont inaccessibles. Cela vaut la peine pour le compilateur de vous dire quand vous modifiez de telles données parce que le résultat serait une perte d'information.

Les références vers pointeur

En C, si vous voulez modifier le *contenu*du pointeur plutôt que ce vers quoi il pointe, votre déclaration de fonction ressemble à :

```
                        void f(int**);
```

et vous aurez à prendre l'adresse du pointeur quand vous le passez :

```
                        int i = 47;
int* ip = &amp;i;
f(&amp;ip);
```

Avec les références en C++, la syntaxe est plus propre. L'argument de la fonction devient une référence vers un pointeur, et vous n'avez plus besoin de prendre l'adresse de ce pointeur. Ainsi,

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                        //: C11:ReferenceToPointer.cpp
#include <iostream>
using namespace std;

void increment(int*& i) { i++; }

int main() {
  int* i = 0;
  cout << "i = " << i << endl;
  increment(i);
  cout << "i = " << i << endl;
} ///:~
```

En exécutant ce programme, vous verrez par vous-même que le pointeur est décrémenté, pas ce vers quoi il pointe.

11.2.2 - Indications sur le passage d'arguments

Votre habitude quand vous passez un argument à une fonction devrait être de le passer par référence **const**. Bien qu'à première vue celui puisse ne paraître que comme une question d'efficacité (et vous n'avez normalement pas à vous inquiéter de réglages d'efficacité tant que vous concevez et assemblez votre programme), il y a plus en jeu : comme vous le verrez dans la suite du chapitre, un constructeur de copie est requis pour passer un objet par valeur, et il n'est pas toujours disponible.

Les économies d'efficacité peuvent être substantiels pour une habitude aussi simple : passer un argument par valeur requiert un appel à un constructeur et un destructeur, mais si vous n'allez pas modifier l'argument alors le passer par référence **const**nécessite seulement qu'une adresse soit poussée sur la pile.

En fait, pratiquement, la seule fois où passer une adresse *n'est pas*préférable est quand vous allez faire de tels dommages à un objet que le passer par valeur est la seule approche sûre (plutôt que de modifier l'objet extérieur, quelque chose que l'appelant n'attend pas en général). C'est le sujet de la section suivante.

11.3 - Le constructeur par recopie

Maintenant que vous comprenez les notions de base des références en C++, vous êtes prêts à aborder l'un des concepts les plus déroutant du langage : le constructeur par recopie, souvent appelé **X(X&)**(“X de ref de X”). Ce constructeur est essentiel pour le contrôle du passage et du renvoi par valeur de types définis par l'utilisateur lors des appels de fonction. Il est tellement important, en fait, que le compilateur synthétisera automatiquement un constructeur par recopie si vous n'en fournissez pas un vous-même, comme nous allons le voir.

11.3.1 - Passer & renvoyer par valeur

Pour comprendre la nécessité du constructeur par recopie, considérez la façon dont le C gère le passage et le retour de variables par valeur lors des appels de fonction. Si vous déclarez une fonction et effectuez un appel à celle-ci,

```
                        int f(int x, char c);
int g = f(a, b);
```

comment le compilateur sait-il comment passer et renvoyer ces variables ? Il le sait, tout simplement ! La variété des types avec laquelle il doit travailler est si petite – **char**, **int**, **float**, **double**, et leurs variations – que cette information est pré-définie dans le compilateur.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Si vous devinez comment générer du code assembleur avec votre compilateur et déterminez les instructions générées par l'appel à la fonction **f( )**, vous obtiendrez l'équivalent de :

```
                        push  b
push  a
call  f()
add  sp,4
mov  g, register a
```

Ce code a été significativement nettoyé pour le rendre général ; les expressions pour **b**et **a**seront différentes selon que les variables sont globales (auquel cas elles seront **_b**et **_a**) ou locales (le compilateur les indexera avec le pointeur de pile). Ceci est également vrai de l'expression pour **g**. L'allure de l'appel à **f( )**dépendra de votre schéma de décoration des noms, et “register a” dépend de la façon dont les registres du processeur sont nommés dans votre assembleur. La logique sous-jacente au code, toutefois, restera la même.

En C et C++, les arguments sont tout d'abord poussés sur la pile de la droite vers la gauche, puis l'appel à la fonction est effectué. Le code appelant est responsable du nettoyage des arguments sur la pile (ce qui explique le **add sp,4**). Mais remarquez que pour passer les arguments par valeur, le compilateur pousse simplement des copies sur la pile – il connaît leur taille et sait que pousser ces arguments en réalise des copies fidèles.

La valeur de retour de **f( )**est placée dans un registre. Encore une fois, le compilateur sait tout ce qu'il faut savoir à propos du type de la valeur à retourner parce que ce type est prédéfini dans le langage, si bien que le compilateur peut le retourner en le plaçant dans un registre. Avec les types de données primitifs en C, la simple action de recopier les bits de la valeur est équivalent à copier l'objet.

Passer & retourner de grands objets

Mais à présent, considérez les types définis par l'utilisateur. Si vous créez une classe et que vous voulez passer un objet de cette classe par valeur, comment le compilateur est-il supposé savoir ce qu'il faut faire ? Ce n'est pas un type prédéfini dans le compilateur ; c'est un type que vous avez créé.

Pour étudier cela, vous pouvez commencer avec une structure simple, qui est clairement trop large pour revenir via les registres :

```
                        //: C11:PassingBigStructures.cpp
struct Big {
  char buf[100];
  int i;
  long d;
} B, B2;

Big bigfun(Big b) {
  b.i = 100; // Fait quelque chose à l'argument
  return b;
}

int main() {
  B2 = bigfun(B);
} ///:~
```

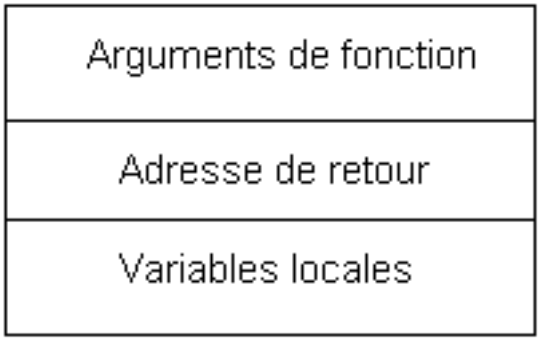
Décoder le code assembleur résultant est un petit peu plus compliqué ici parce que la plupart des compilateurs utilisent des fonctions “assistantes” plutôt que de mettre toutes les fonctionnalités inline. Dans **main( )**, l'appel à **bigfun( )**commence comme vous pouvez le deviner – le contenu entier de **B**est poussé sur la pile. (Ici, vous pouvez voir certains compilateurs charger les registres avec l'adresse du **Big**et sa taille, puis appeler une fonction assistante pour pousser le **Big**sur la pile.)

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Dans le fragment de code précédent, pousser les arguments sur la pile était tout ce qui était requis avant de faire l'appel de fonction. Dans **PassingBigStructures.cpp**, toutefois, vous verrez une action supplémentaire : l'adresse de **B2**est poussée avant de faire l'appel, même si ce n'est de toute évidence pas un argument. Pour comprendre ce qu'il se passe ici, vous devez saisir les contraintes qui s'exercent sur le compilateur quand il fait un appel de fonction.

Cadre de pile de l'appel de fonction

Quand le compilateur génère du code pour un appel de fonction, il pousse d'abord tous les arguments sur la pile, puis fait l'appel. Dans la fonction, du code est généré pour déplacer le pointeur sur la pile encore plus bas pour fournir un espace de stockage pour les variables locales. (“Bas” est relatif ici ; votre machine peut aussi bien incrémenter que décrémenter le pointeur sur la pile pendant un push.) Mais pendant le CALL en langage assembleur, le processeur pousse l'adresse dans le code programme d'où *provenait*l'appel à la fonction, afin que le RETURN en langage assembleur puisse utiliser cette adresse pour revenir à l'endroit de l'appel. Cette adresse est bien sûr sacrée, parce que sans elle votre programme sera complètement perdu. Voici à quoi ressemble la cadre de pile après le CALL et l'allocation de l'emplacement pour les variables locales dans la fonction :



Le code généré pour le reste de la fonction s'attend à ce que la mémoire soit disposée exactement de cette façon, afin qu'il puisse délicatement choisir parmi les arguments de la fonction et les variables locales, sans toucher l'adresse de retour. J'appellerai ce bloc de mémoire, qui est tout ce qu'une fonction utilise dans le processus d'appel à la fonction, la *cadre de fonction*.

Vous pourriez penser raisonnable d'essayer de retourner les valeurs sur la pile. Le compilateur pourrait simplement les pousser, et la fonction pourrait retourner un offset pour indiquer où commence la valeur retour dans la pile.

Re-entrée

Le problème se produit parce que les fonctions en C et en C++ supportent les interruptions ; c'est-à-dire que les langages sont *ré-entrants*. Ils supportent également les appels de fonctions récursifs. Ceci signifie qu'à tout endroit de l'exécution d'un programme une interruption peut se produire sans ruiner le programme. Bien sûr, la personne qui écrit la routine de service d'interruption (ISR - pour *interrupt service routine*, ndt) est responsable de la sauvegarde et de la restauration de tous les registres qui sont utilisés dans l'ISR, mais si l'ISR a besoin d'utiliser de la mémoire plus bas dans la pile, elle doit pouvoir le faire sans risque. (Vous pouvez considérer une ISR comme une fonction ordinaire sans argument et une valeur de retour **void**qui sauve et restaure l'état du processeur. Un appel à une fonction ISR est provoqué par un événement hardware plutôt que par un appel explicite depuis un programme.)

A présent imaginez ce qui se produirait si une fonction ordinaire essayait de retourner des valeurs sur la pile. Vous ne pouvez toucher aucune partie de la pile qui se trouve au-dessus de l'adresse de retour, donc la fonction devrait pousser les valeurs sous l'adresse de retour. Mais quand le RETURN en langage assembleur est exécuté, le pointeur de la pile doit pointer l'adresse de retour (ou juste en dessous, selon le type de machine). Ainsi juste avant le RETURN, la fonction doit déplacer le pointeur sur la pile vers le haut, nettoyant ainsi toutes ses variables

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

locales. Si vous essayez de retourner des valeurs sur la pile sous l'adresse de retour, vous êtes vulnérable à ce moment parce qu'une interruption pourrait intervenir. L'ISR déplacerait le pointeur sur la pile vers le bas pour retenir son adresse de retour et ses variables locales et écraser votre valeur de retour.

Pour résoudre ce problème, l'appelant *pourrait*être responsable de l'allocation de stockage supplémentaire sur la pile pour les valeurs de retour avant d'appeler la fonction. Toutefois, le C n'a pas été conçu ainsi, et le C++ doit être compatible. Comme vous verrez dans peu de temps, le compilateur C++ utilise un procédé plus efficace.

Votre idée suivante pourrait être de retourner la valeur dans une zone de donnée globale, mais cela ne marche pas non plus. Réentrer signifie que n'importe quelle fonction peut être une routine d'interruption pour n'importe quelle autre, *y compris la fonction dans laquelle vous vous trouvez à ce moment*. Ainsi, si vous placez la valeur de retour dans une zone globale, vous pourriez revenir dans la même fonction, ce qui écraserait cette valeur de retour. La même logique s'applique à la récursion.

Le seul endroit sûr pour retourner les valeurs est dans les registres, si bien que vous vous retrouvez à nouveau confronté au problème de ce qu'il faut faire quand les registres ne sont pas assez grands pour contenir la valeur de retour. La réponse est de pousser l'adresse de la destination de la valeur de retour sur la pile comme l'un des arguments de la fonction, et laisser la fonction copier l'information de retour directement dans la destination. Ceci ne résoud pas seulement tous les problèmes, c'est plus efficace. C'est aussi la raison pour laquelle, dans **PassingBigStructures.cpp**, le compilateur pousse l'adresse de **B2**avant l'appel à **bigfun( )**dans **main( )**. Si vous regardez le code assembleur résultant pour **bigfun( )**, vous pouvez voir qu'il attend cet argument caché et réalise la copie vers la destination *dans*la fonction.

Copie de bit versus initialisation

Jusque là, tout va bien. Il y a une procédure exploitable pour passer et retourner de grandes structures simples. Mais notez que tout ce dont vous disposez est un moyen de recopier les bits d'un endroit vers un autre, ce qui fonctionne certainement bien pour la manière primitive qu'a le C de considérer les variables. Mais en C++ les objets peuvent être beaucoup plus sophistiqués qu'un assemblage de bits ; ils ont une signification. Cette signification peut ne pas réagir correctement à la recopie de ses bits.

Considérez un exemple simple : une classe qui sait combien d'objets de son type existe à chaque instant. Depuis le chapitre 10, vous connaissez la façon de le faire en incluant une donnée membre **static**:

```
                        //: C11:HowMany.cpp
// Une classe qui compte ses objets
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany.out");

class HowMany {
  static int objectCount;
public:
  HowMany() { objectCount++; }
  static void print(const string& msg = "") {
    if(msg.size() != 0) out << msg << ": ";
    out << "objectCount = "
        << objectCount << endl;
  }
  ~HowMany() {
    objectCount--;
    print("~HowMany()");
  }
};

int HowMany::objectCount = 0;

// Passe et renvoie PAR VALEURS
HowMany f(HowMany x) {
  x.print("argument x dans f()");
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  return x;
}

int main() {
  HowMany h;
  HowMany::print("après construction de h");
  HowMany h2 = f(h);
  HowMany::print("après appel de f()");
} ///:~
```

La classe **HowMany**contient un **static int objectCount**et une fonction membre **static print( )**pour afficher la valeur de cet **objectCount**, accompagnée d'un argument message optionnel. Le constructeur incrémente le compte à chaque fois qu'un objet est créé, et le destructeur le décrémente.

La sortie, toutefois, n'est pas ce à quoi vous vous attendriez :

```
                        après construction de h: objectCount = 1
argument x dans f(): objectCount = 1
~HowMany(): objectCount = 0
après l'appel à f(): objectCount = 0
~HowMany(): objectCount = -1
~HowMany(): objectCount = -2
```

Après que **h**soit créé, le compte d'objets vaut un, ce qui est correct. Mais après l'appel à **f( )**vous vous attendriez à avoir le compte des objets à deux, parce que **h2**est maintenant également dans la portée. A la place, le compte vaut zéro, ce qui indique que quelque chose a terriblement échoué. Ceci confirmé par le fait que les deux destructeurs à la fin rendent le compte d'objets négatif, ce qui ne devrait jamais se produire.

Regardez l'endroit dans **f( )**, après que l'argument ait été passé par valeur. Ceci signifie que **h**, l'objet original, existe en dehors du cadre de fonction, et il y a un objet supplémentaire **dans**le cadre de fonction, qui est la copie qui a été passée par valeur. Toutefois, l'argument a été passé en utilisant la notion primitive de copie de bits du C, alors que la classe C++ **HowMany**requiert une vraie initialisation pour maintenir son intégrité, et la recopie de bit par défaut échoue donc à produire l'effet désiré.

Quand l'objet local sort de la portée à la fin de l'appel à **f( )**, le destructeur est appelé, et il décrémente **objectCount**, si bien qu'en dehors de la fonction **objectCount**vaut zéro. La création de **h2**est également réalisée en utilisant une copie de bits, si bien que le constructeur n'est pas appelé ici non plus, et quand **h**et **h2**sortent de la portée, leur destructeur rendent **objectCount**négatif.

11.3.2 - Construction par recopie

Le problème a lieu parce que le compilateur fait une supposition sur la manière de créer *un nouvel objet à partir d'un objet existant*. Quand vous passez un objet par valeur, vous créez un nouvel objet, l'objet passé dans le cadre de fonction, à partir d'un objet existant, l'objet original hors du cadre de fonction. C'est également souvent vrai quand un objet est renvoyé par une fonction. Dans l'expression

```
                        HowMany h2 = f(h);
```

h2, un objet qui n'a pas été construit auparavant, est créé à partir de la valeur de retour de **f( )**, si bien qu'un nouvel objet est encore créé à partir d'un autre existant.

La supposition du compilateur est que vous voulez réaliser cette création par recopie de bits, et dans bien des cas cela peut fonctionner correctement, mais dans **HowMany**ce n'est pas le cas parce que la signification de l'initialisation va au-delà d'une simple recopie. Un autre exemple habituel a lieu si la classe contient des pointeurs –

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

vers quoi pointent-ils, et devriez-vous les recopier ou bien devraient-il être connectés à un nouvel espace mémoire ?

Heureusement, vous pouvez intervenir dans ce processus et empêcher le compilateur de faire une recopie de bits. Vous faites cela en définissant votre propre fonction à utiliser lorsque le compilateur doit faire un nouvel objet à partir d'un objet existant. En toute logique, vous créez un nouvel objet, donc cette fonction est un constructeur, et en toute logique également, le seul argument de ce constructeur à rapport à l'objet à partir duquel vous contruisez. Mais cet objet ne peut être passé dans le constructeur par valeur, parce que vous essayez de *définir*la fonction qui gère le passage par valeur, et syntaxiquement cela n'a aucun sens de passer un pointeur parce que, après tout, vous créez le nouvel objet à partir d'un objet existant. Les références viennent ici à notre secours, donc vous prenez la référence de l'objet source. Cette fonction est appelée le *constructeur par recopie*et est souvent désigné **X(X&)**, ce qui est sa forme pour une classe nommée **X**.

Si vous créez un constructeur par recopie, le compilateur ne fera pas une recopie de bits quand il créera un nouvel objet à partir d'un objet existant. Il appelera toujours votre constructeur par recopie. Donc, si vous ne créez pas un constructeur par recopie, le compilateur fera quelque chose de raisonnable, mais vous avez la possiblité de contrôler complètement le processus.

A présent, il est possible de résoudre le problème dans **HowMany.cpp**:

```
                        //: C11:HowMany2.cpp
// Le constructeur par recopie
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany2.out");

class HowMany2 {
  string name; // Identifiant d'objet
  static int objectCount;
public:
  HowMany2(const string& id = "") : name(id) {
    ++objectCount;
    print("HowMany2()");
  }
  ~HowMany2() {
    --objectCount;
    print("~HowMany2()");
  }
  // Le constructeur par recopie :
  HowMany2(const HowMany2& h) : name(h.name) {
    name += " copie";
    ++objectCount;
    print("HowMany2(const HowMany2&)");
  }
  void print(const string& msg = "") const {
    if(msg.size() != 0)
      out << msg << endl;
    out << '\t' << name << ": "
        << "objectCount = "
        << objectCount << endl;
  }
};

int HowMany2::objectCount = 0;

// Passe et renvoie PAR VALEUR:
HowMany2 f(HowMany2 x) {
  x.print("argument x dans f()");
  out << "Retour de f()" << endl;
  return x;
}

int main() {
  HowMany2 h("h");
  out << "avant appel à f()" << endl;
  HowMany2 h2 = f(h);
  h2.print("h2 après appel à f()");
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  out << "Appel de f(), pas de valeur de retour" << endl;
  f(h);
  out << "après appel à f()" << endl;
} ///:~
```

Il y a un certain nombre de nouvelles astuces dans ce code afin que vous puissiez avoir une meilleure idée de ce qu'il se produit. Tout d'abord, le **string name**agit comme un identifiant d'objet quand l'information concernant cet objet est affichée. Dans le constructeur, vous pouvez placer une chaîne identifiante (généralement le nom de l'objet) qui est copiée vers **name**en utilisant le constructeur de **string**. Le **= ""**par défaut crée une **string**vide. Le constructeur incrémente l' **objectCount**comme précédemment, et le destructeur le décrémente.

Après cela vient le constructeur par recopie **HowMany2(const HowMany2&)**. Il ne peut créer un nouvel objet qu'à partir d'un autre objet existant, si bien que le nom de l'objet existant est copié vers **name**, suivi par le mot “copie” afin que vous puissiez voir d'où il vient. Si vous regardez attentivement, vous verrez que l'appel **name(h.name)**dans la liste d'initialisation du constructeur appelle en fait le constructeur par recopie de **string**.

Dans le constructeur par recopie, le compte d'objet est incrémenté exactement comme dans le constructeur normal. Ceci signifie que vous aurez à présent un compte d'objet exact quand vous passez et retournez par valeur.

La fonction **print( )**a été modifiée pour afficher un message, l'identifiant de l'objet, et le compte de l'objet. Elle doit à présent accéder à la donnée **name**d'un objet particulier, et ne peut donc plus être une fonction membre **static**.

Dans **main( )**, vous pouvez constater qu'un deuxième appel à **f( )**a été ajouté. Toutefois, cet appel utilise l'approche commune en C qui consiste à ignorer la valeur retour. Mais maintenant que vous savez comment la valeur est retournée (c'est-à-dire que du code *au sein*de la fonction gère le processus de retour, plaçant le résultat dans une destination dont l'adresse est passée comme un argument caché), vous pouvez vous demander ce qu'il se passe quand la valeur retour est ignorée. La sortie du programme éclairera la question.

Avant de montrer la sortie, voici un petit programme qui utilise les iostreams pour ajouter des numéros de ligne à n'importe quel fichier :

```
                          //: C11:Linenum.cpp
//{T} Linenum.cpp
// Ajoute les numéros de ligne
#include "../require.h"
#include <vector>
#include <string>
#include <fstream>
#include <iostream>
#include <cmath>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1, "Usage: linenum file\n"
    "Ajoute des numéros de ligne à un fihier");
  ifstream in(argv[1]);
  assure(in, argv[1]);
  string line;
  vector<string> lines;
  while(getline(in, line)) // Lit le fichier entièrement
    lines.push_back(line);
  if(lines.size() == 0) return 0;
  int num = 0;
  // Le nombre de lignes dans le fichier détermine la valeur de width :
  const int width =
    int(log10((double)lines.size())) + 1;
  for(int i = 0; i < lines.size(); i++) {
    cout.setf(ios::right, ios::adjustfield);
    cout.width(width);
    cout << ++num << ") " << lines[i] << endl;
  }
} ///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Le fichier entier est lu dans un **vector<string>**, utilisant le code que vous avez vu plus tôt dans le livre. Quand nous affichons les numéros de ligne, nous aimerions que toutes les lignes soient alignées, et ceci requiert d'ajuster le nombre de lignes dans le fichier afin que la largeur autorisée pour le nombre de lignes soit homogène. Nous pouvons facilement déterminer le nombre de lignes en utilisant **vector::size( )**, mais ce qu'il nous faut vraiment savoir est s'il y a plus de 10 lignes, 100 lignes, 1000 lignes etc. Si vous prenez le logarithme en base 10 du nombre de lignes dans le fichier, que vous le tronquez en **int**et ajoutez un à la valeur, vous trouverez la largeur maximum qu'aura votre compte de lignes.

Vous remarquerez une paire d'appels étranges dans la boucle **for**: **setf( )**et **width( )**. Ce sont des appels **ostream**qui vous permettent de contrôler, dans ce cas, la justification et la largeur de la sortie. Toutefois, ils doivent être appelés à chaque fois qu'une ligne est sortie et c'est pourquoi ils se trouvent dans la boucle **for**. Le deuxième volume de ce livre contient un chapitre entier expliquant les iostreams qui vous en dira plus à propos de ces appels ainsi que d'autres moyens de contrôler les iostreams.

Quand **Linenum.cpp**est appliqué à **HowMany2.out**, cela donne

```
                        1) HowMany2()
 2)   h: objectCount = 1
 3) avant appel à f()
 4) HowMany2(const HowMany2&)
 5)   h copie: objectCount = 2
 6) argument x dans f()
 7)   h copie: objectCount = 2
 8) Retour de f()
 9) HowMany2(const HowMany2&)
10)   h copie copie: objectCount = 3
11) ~HowMany2()
12)   h copie: objectCount = 2
13) h2 après appel à f()
14)   h copie copie: objectCount = 2
15) appel à f(), pas de valeur de retour
16) HowMany2(const HowMany2&)
17)   h copie: objectCount = 3
18) argument x dans f()
19)   h copie: objectCount = 3
20) Retour de f()
21) HowMany2(const HowMany2&)
22)   h copie copie: objectCount = 4
23) ~HowMany2()
24)   h copie: objectCount = 3
25) ~HowMany2()
26)   h copie copie: objectCount = 2
27) après appel à f()
28) ~HowMany2()
29)   h copie copie: objectCount = 1
30) ~HowMany2()
31)   h: objectCount = 0
```

Comme vous pouvez vous y attendre, la première chose qui se produit est que le constructeur normal est appelé pour **h**, ce qui incrémente le compte d'objets à un. Mais ensuite, comme on entre dans **f( )**, le constructeur par recopie est tranquillement appelé par le compilateur pour réaliser le passage par valeur. Un nouvel objet est créé, qui est la copie de **h**(d'où le nom "h copie") dans le cadre de fonction de **f( )**, si bien que le compte d'objet passe à deux, par la grâce du constructeur par recopie.

La ligne huit indique le début du retour de **f( )**. Mais avant que la variable locale "h copie" puisse être détruite (elle sort de la portée à la fin de la fonction), elle doit être copiée dans la valeur de retour, qui se trouve être **h2**. Un objet précédemment non construit (**h2**) est créé à partir d'un objet existant (la variable locale dans **f( )**), donc bien sûr le constructeur par recopie est utilisé à nouveau ligne neuf. A présent, le nom devient "h copie copie" pour l'identifiant de **h2**parce qu'il est copié à partir de la copie qu'est l'objet local dans **f( )**. Après que l'objet soit retourné, mais avant la fin de la fonction, le compte d'objet passe temporairement à trois, mais ensuite l'objet local "h copie" est détruit. Après que l'appel à **f( )**ne se termine ligne 13, il n'y a que deux objets, **h**et **h2**, et vous pouvez voir que **h2**finit de fait comme "h copie copie".

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Objets temporaires

A la ligne 15 commence l'appel à **f(h)**, ignorant cette fois-ci la valeur de retour. Vous pouvez voir ligne 16 que le constructeur par recopie est appelé exactement comme avant pour passer l'argument. Et, également comme auparavant, la ligne 21 montre que le constructeur par recopie est appelé pour la valeur de retour. Mais le constructeur par recopie doit disposer d'une adresse pour l'utiliser comme sa destination (un pointeur **this**). D'où provient cette adresse ?

Il s'avère que le compilateur peut créer un objet temporaire à chaque fois qu'il en a besoin pour évaluer proprement une expression. Dans ce cas il en crée un que vous ne voyez même pas pour agir comme la destination de la valeur de retour ignorée de **f( )**. La durée de vie de cet objet temporaire est aussi courte que possible afin que le paysage ne se retrouve pas encombré avec des objets temporaires attendant d'être détruits et occupant de précieuses ressources. Dans certains cas, le temporaire peut être immédiatement passé à une autre fonction, mais dans ce cas il n'est plus nécessaire après l'appel à la fonction, si bien que dès que la fonction se termine en appelant le destructeur pour l'objet local (lignes 23 et 24), l'objet temporaire est détruit (lignes 25 et 26).

Finalement, lignes 28-31, l'objet **h2**est détruit, suivi par **h**, et le compte des objets revient correctement à zéro.

11.3.3 - Constructeur par recopie par défaut

Comme le constructeur par recopie implémente le passage et le retour par valeur, il est important que le compilateur en crée un pour vous dans le cas de structures simples – en fait, la même chose qu'il fait en C. Toutefois, tout ce que vous avez vu jusqu'ici est le comportement primitif par défaut : la copie de bits.

Quand des types plus complexes sont impliqués, le compilateur C++ créera encore automatiquement un constructeur par recopie si vous n'en faites pas un. A nouveau, toutefois, une copie de bit n'a aucun sens, parce qu'elle n'implémente pas nécessairement la bonne signification.

Voici un exemple pour montrer l'approche plus intelligente que prend le compilateur. Supposez que vous créez une classe composée d'objets de plusieurs classes existantes. On appelle cela, judicieusement, *composition*, et c'est une des manières de créer de nouvelles classes à partir de classes existantes. A présent, endossez le rôle d'un utilisateur naïf qui essaie de résoudre rapidement un problème en créant une nouvelle classe de cette façon. Vous ne connaissez rien au sujet des constructeurs par recopie, et vous n'en créez donc pas. L'exemple démontre ce que le compilateur fait en créant le constructeur par recopie par défaut pour votre nouvelle classe :

```
                        //: C11:DefaultCopyConstructor.cpp
// Création automatique du constructeur par recopie
#include <iostream>
#include <string>
using namespace std;

class WithCC { // Avec constructeur par recopie
public:
  // Constructeur par défaut explicite requis :
  WithCC() {}
  WithCC(const WithCC&) {
    cout << "WithCC(WithCC&)" << endl;
  }
};

class WoCC { // Sans constructeur par recopie
  string id;
public:
  WoCC(const string& ident = "") : id(ident) {}
  void print(const string& msg = "") const {
    if(msg.size() != 0) cout << msg << ": ";
    cout << id << endl;
  }
};
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
class Composite {
  WithCC withcc; // Objets embarqués
  WoCC wocc;
public:
  Composite() : wocc("Composite()") {}
  void print(const string& msg = "") const {
    wocc.print(msg);
  }
};

int main() {
  Composite c;
  c.print("Contenu de c");
  cout << "Appel du constructeur par recopie de Composite"
       << endl;
  Composite c2 = c;  // Appelle le constructeur par recopie
  c2.print("Contenu de c2");
} ///:~
```

La classe **WithCC**contient un constructeur par recopie, qui annonce simplement qu'il a été appelé, et ceci révèle un problème intéressant. Dans la classe **Composite**, un objet **WithCC**est créé en utilisant un constructeur par défaut. S'il n'y avait pas de construteur du tout dans **WithCC**, le compilateur créerait automatiquement un constructeur par défaut, qui ne ferait rien dans le cas présent. Toutefois, si vous ajoutez un constructeur par recopie, vous avez dit au compilateur que vous allez gérer la création du constructeur, si bien qu'il ne crée plus de constructeur par défaut pour vous et se plaindra à moins que vous ne créiez explicitement un constructeur par défaut comme cela a été fait pour **WithCC**.

La classe **WoCC**n'a pas de constructeur par recopie, mais son constructeur stockera un message dans une **string**interne qui peut être affiché en utilisant **print( )**. Ce constructeur est appelé explicitement dans la liste d'initialisation du constructeur de **Composite**(brièvement introduite au Chapitre 8 et couverte en détails au Chapitre 14). La raison en deviendra claire plus loin.

La classe **Composite**a des objets membres de **WithCC**et de **WoCC**(remarquez que l'objet inclus **wocc**est initialisé dans la liste d'initialisation du constructeur, comme il se doit), et pas de constructeur par recopie explicitement défini. Toutefois, dans **main( )**un objet est créé en utilisant le constructeur par recopie dans la définition :

```
Composite c2 = c;
```

Le constructeur par recopie pour **Composite**est créé automatiquement par le compilateur, et la sortie du programme révèle la façon dont il est créé :

```
Contenu de c: Composite()
Appel du constructeur par recopie de Composite
WithCC(WithCC&)
Contenu de c2: Composite()
```

Pour créer un constructeur par recopie pour une classe qui utilise la composition (et l'héritage, qui est introduit au Chapitre 14), le compilateur appelle récursivement le constructeur par recopie de tous les objets membres et de toutes les classes de base. C'est-à-dire que si l'objet membre contient également un autre objet, son constructeur par recopie est aussi appelé. Dans le cas présent, le compilateur appelle le constructeur par recopie de **WithCC**. La sortie montre ce constructeur en train d'être appelé. Comme **WoCC**n'a pas de constructeur par recopie, le compilateur en crée un pour lui qui réalise simplement une copie de bits, et l'appelle dans le constructeur par recopie de **Composite**. L'appel à **Composite::print( )**dans le main montre que cela se produit parce que le contenu de **c2.wocc**est identique au contenu de **c.wocc**. Le processus que suit le compilateur pour synthétiser un constructeur par recopie est appelé *initialisation par les membres (memberwise initialisation ndt)*.

Il vaut toujours mieux créer votre propre constructeur par recopie au lieu de laisser le compilateur le faire pour

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

vous. Cela garantit que vous le contôlerez.

11.3.4 - Alternatives à la construction par recopie

A ce stade votre esprit flotte peut-être, vous pouvez vous demander comment vous avez bien pu écrire une classe qui fonctionne sans connaître le constructeur par recopie. Mais rappelez-vous : vous n'avez besoin d'un constructeur par recopie que si vous allez passer un objet de votre classe *par valeur*. Si cela ne se produit jamais, vous n'avez pas besoin d'un constructeur par recopie.

Eviter le passage par valeur

“Mais,” dites-vous, “si je ne fais pas de constructeur par recopie, le compilateur en créera un pour moi. Donc comment puis-je savoir qu'un objet ne sera jamais passé par valeur ?”

Il y a une technique simple pour éviter le passage par valeur : déclarer un constructeur par recopie **private**. Vous n'avez même pas besoin de créer une définition, à moins qu'une de vos fonctions membres ou une fonction **friend**ait besoin de réaliser un passage par valeur. Si l'utilisateur essaye de passer ou de retourner l'objet par valeur, le compilateur produit un message d'erreur parce que le constructeur par recopie est **private**. Il ne peut plus créer de constructeur par recopie par défaut parce que vous avez explicitement déclaré que vous vous occupiez de ce travail.

Voici un exemple :

```
                          //: C11:NoCopyConstruction.cpp
// Eviter la contruction par recopie

class NoCC {
  int i;
  NoCC(const NoCC&); // Pas de définition
public:
  NoCC(int ii = 0) : i(ii) {}
};

void f(NoCC);

int main() {
  NoCC n;
//! f(n); // Erreur : constructeur par recopie appelé
//! NoCC n2 = n; // Erreur : c-c appelé
//! NoCC n3(n); // Erreur : c-c appelé
} ///:~
```

Remarquez l'utilisation de la forme plus générale

```
                          NoCC(const NoCC&amp;);
```

utilisant **const**.

Fonctions modifiant les objets exterieurs

La syntaxe de référence est plus agréable à utiliser que celle des pointeurs, cependant elle obscurcit le sens pour le lecteur. Par exemple, dans la bibliothèque iostreams une fonction surchargée de la fonction **get( )**prend un **char&**comme argument, et tout l'enjeu de la fonction est de modifier son argument en insérant le résultat du **get( )**. Toutefois, quand vous lisez du code utilisant cette fonction ce n'est pas immédiatement évident pour vous que l'objet extérieur est modifié :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                    char c;
cin.get(c);
```

A la place, l'appel à la fonction ressemble à un passage par valeur, qui suggère que l'objet extérieur *n'est pas*modifié.

A cause de cela, il est probablement plus sûr d'un point de vue de la maintenance du code d'utiliser des pointeurs quand vous passez l'adresse d'un argument à modifier. Si vous passez *toujours*les adresses comme des références **const** *sauf*quand vous comptez modifier l'objet extérieur via l'adresse, auquel cas vous passez par pointeur non- **const**, alors votre code est beaucoup plus facile à suivre pour le lecteur.

11.4 - Pointeurs sur membre

Un pointeur est une variable qui contient l'adresse d'un emplacement. Vous pouvez changer ce que sélectionne un pointeur pendant l'exécution, et la destination du pointeur peut être des données ou une fonction. Le *pointeur sur membre*du C++ utilise ce même concept, à l'exception que ce qu'il sélectionne est un emplacement dans une classe. Le dilemme ici est que le pointeur a besoin d'une adresse, mais il n'y a pas d'"adresse" à l'intérieur d'une classe; sélectionner un membre d'une classe signifie un décalage dans cette classe. Vous ne pouvez pas produire une adresse effective jusqu'à ce que vous combiniez ce décalage avec le début de l'adresse d'un objet particulier. La syntaxe d'un pointeur sur un membre requiert que vous sélectionniez un objet au même moment que vous déréférencez le pointeur sur le membre.

Pour comprendre cette syntaxe, considérez une structure simple, avec un pointer **sp**et un objet **so**pour cette structure. Vous pouvez sélectionnez les membres avec la syntaxe suivante :

```
                    //: C11:SimpleStructure.cpp
struct Simple { int a; };
int main() {
  Simple so, *sp = &so;
  sp->a;
  so.a;
} ///:~
```

Maintenant supposez que vous avez un pointeur ordinaire sur un entier, **ip**. Pour accéder à ce que **ip**pointe, vous déréférencez le pointeur avec une ' ***'**:

```
                    *ip = 4;
```

Finalement, considérez ce qui arrive si vous avez un pointeur qui pointe sur quelque chose à l'intérieur d'une classe, même si il représente en fait un décalagepar rapport à l'adresse de l'objet. Pour accéder à ce qui est pointé, vous devez le déréférencer avec ***. Mais c'est un excentrage dans un objet, donc vous devriez aussi vous référez à cette objet particulier. Ainsi, l' ***est combiné avec un objet déréférencé. Donc la nouvelle syntaxe devient **–>***pour un pointeur sur un objet, et **.***pour l'objet ou une référence, comme ceci :

```
                    objectPointer->*pointerToMember = 47;
object.*pointerToMember = 47;
```

Maintenant, quelle est la syntaxe pour définir un **pointeur sur membre**? Comme n'importe quel pointeur, vous devez dire de quel type est le pointeur, et vous utilisez une ***dans la définition. La seule différence est que vous devez dire quelle classe d'objet est utilisée par ce pointeur sur membre. Bien sûr, ceci est accompli avec le nom de la classe et l'opérateur de résolution de portée. Ainsi,

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
int ObjectClass::*pointerToMember;
```

définit une variable pointeur sur membre appelé **pointerToMember**qui pointe sur n'importe quel **int**à l'intérieur de **ObjectClass**. Vous pouvez aussi initialiser le pointeur sur membre quand vous le définissez (ou à n'importe quel autre moment) :

```
int ObjectClass::*pointerToMember = &ObjectClass::a;
```

Il n'y a actuellement aucune “adresse” de **ObjectClass::a**parce que vous venez juste de référencer la classe et non pas un objet de la classe. Ainsi, **&ObjectClass::a**peut être utilisé seulement comme une syntaxe de pointeur sur membre.

Voici un exemple qui montre comment créer et utiliser des pointeurs sur des données de membre :

```
//: C11:PointerToMemberData.cpp
#include <iostream>
using namespace std;

class Data {
public:
  int a, b, c;
  void print() const {
    cout << "a = " << a << ", b = " << b
         << ", c = " << c << endl;
  }
};

int main() {
  Data d, *dp = &d;
  int Data::*pmInt = &Data::a;
  dp->*pmInt = 47;
  pmInt = &Data::b;
  d.*pmInt = 48;
  pmInt = &Data::c;
  dp->*pmInt = 49;
  dp->print();
} ///:~
```

Evidement, c'est trop maladroit pour être utilisé n'importe où excepté pour des cas spéciaux (ce qui est exactement ce pour lequel ils ont étés prévus).

Aussi, les pointeurs sur membre sont tous à fait limités : ils peuvent être assignés seulement à une localisation spécifique à l'intérieur d'une classe. Vous ne pouvez pas, par exemple, les incrémenter ou les comparer comme vous pourriez avec des pointeurs ordinaires.

11.4.1 - Fonctions

Un exercice similaire produit la syntaxe du pointeur sur membre pour les fonctions membre. Un pointeur sur un fonction (introduit à la fin du chapitre 3) est défini comme cela:

```
int (*fp)(float);
```

Les parenthèses autour de **(*fp)**sont nécessaires pour forcer le compilateur à évaluer proprement la définition. Faute de quoi, ceci semblerait être une fonction qui retourne un **int***.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Les parenthèses jouent aussi un rôle important pour définir et utiliser des pointeurs sur fonction membre. Si vous avez une fonction à l'intérieur d'une classe, vous définissez un pointeur sur cette fonction membre en insérant le nom de la classe et un opérateur de résolution de portée dans une définition de pointeur d'une fonction ordinaire :

```
                        //: C11:PmemFunDefinition.cpp
class Simple2 {
public:
  int f(float) const { return 1; }
};
int (Simple2::*fp)(float) const;
int (Simple2::*fp2)(float) const = &Simple2::f;
int main() {
  fp = &Simple2::f;
} ///:~
```

Dans la définition pour **fp2**vous pouvez voir qu'un pointeur sur fonction membre peut aussi être initialisé quand il est créé, ou à n'importe quel moment. Au contraire des fonctions non membres, le **&** *n'est pas*optionnel quand il prend l'adresse d'une fonction membre. Cependant, vous pouvez donner l'identificateur de fonction sans liste d'arguments parce que tout peut être résolu au moyen du nom de classe et de l'opérateur de résolution de portée..

Un exemple

L'intérêt d'un pointeur est que vous pouvez changer la valeur pointée au moment de l'exécution, ce qui produit une flexibilité importante dans votre programmation parce que à travers un pointeur vous pouvez sélectionner ou changer le *comportement*durant l'exécution. Un pointeur de membre n'est pas différent ; il vous permet de choisir un membre durant l'exécution. Typiquement, vos classes peuvent seulement avoir des fonctions membres avec une portée publique (D'habitude, les données membres sont considérées comme faisant partie de l'implémentation sous-jacente), donc l'exemple suivant sélectionne les fonctions membres durant l'exécution.

```
                        //: C11:PointerToMemberFunction.cpp
#include <iostream>
using namespace std;

class Widget {
public:
  void f(int) const { cout << "Widget::f()\n"; }
  void g(int) const { cout << "Widget::g()\n"; }
  void h(int) const { cout << "Widget::h()\n"; }
  void i(int) const { cout << "Widget::i()\n"; }
};

int main() {
  Widget w;
  Widget* wp = &w;
  void (Widget::*pmem)(int) const = &Widget::h;
  (w.*pmem)(1);
  (wp->*pmem)(2);
} ///:~
```

Bien sûr, ce n'est pas particulièrement raisonnable de s'attendre à ce que l'utilisateur occasionnel crée de telles expressions compliquées. Si l'utilisateur doit directement manipuler un pointeur sur membre, alors un **typedef**est préférable. Pour vraiment comprendre cela, vous pouvez utiliser le pointeur sur membre comme une part du mécanisme interne de l'implémentation. Le précédent exemple utilisait un pointeur sur membre *à l'intérieur*de la classe. Tout ce que l'utilisateur a besoin de faire c'est de passer un nombre dans la sélection d'une fonction. Merci à Owen Mortensen pour cette exemple

```
                        //: C11:PointerToMemberFunction2.cpp
#include <iostream>
using namespace std;

class Widget {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    void f(int) const { cout << "Widget::f()\n"; }
    void g(int) const { cout << "Widget::g()\n"; }
    void h(int) const { cout << "Widget::h()\n"; }
    void i(int) const { cout << "Widget::i()\n"; }
    enum { cnt = 4 };
    void (Widget::*fptr[cnt])(int) const;
  public:
    Widget() {
      fptr[0] = &Widget::f; // Full spec required
      fptr[1] = &Widget::g;
      fptr[2] = &Widget::h;
      fptr[3] = &Widget::i;
    }
    void select(int i, int j) {
      if(i < 0 || i >= cnt) return;
      (this->*fptr[i])(j);
    }
    int count() { return cnt; }
  };

  int main() {
    Widget w;
    for(int i = 0; i < w.count(); i++)
      w.select(i, 47);
  } ///:~
```

Dans l'interface de classe et dans le **main( )**, vous pouvez voir que l'implémentation entière, incluant les fonctions, a été cachée ailleurs. Le code doit même demander le **count( )**des fonctions. De cette manière, l'implémenteur de la classe peut changer la quantité de fonctions dans l'implémentation sous-jacente sans affecter le code où la classe est utilisée.

L'initialisation du pointeur sur membre dans le constructeur peut sembler trop spécifié. Ne pouvez vous pas dire

```
                        fptr[1] = &amp;g;
```

parce que le nom **g**se trouve dans la fonction membre, qui est automatiquement dans la portée de la classe ? Le problème est que ce n'est pas conforme à la syntaxe du pointeur sur membre, qui est requise pour que chacun, spécialement le compilateur, puisse voir ce qui se passe. De même, quand le pointeur sur membre est déréférencé, il semble que

```
                        (this->*fptr[i])(j);
```

soit aussi trop spécifié ; **this**semble redondant. De nouveau, la syntaxe requise est qu'un pointeur sur membre est toujours lié à un objet quand il est déréférencié.

11.5 - Résumé

Les pointeurs en C++ sont presque identiques aux pointeurs en C, ce qui est une bonne chose. Autrement, beaucoup de code C ne serait pas correctement compilé en C++. Les seules erreurs de compilation que vous produirez ont lieu lors d'affectations dangereuses. Si, en fait, celles-ci sont intentionnelles, l'erreur de compilation peut être supprimée à l'aide d'un simple transtypage (explicite, qui plus est !).

Le C++ ajoute également les *références*venant de l'Algol et du Pascal, qui sont comme des pointeurs constants automatiquement déréférencés par le compilateur. Une référence contient une adresse, mais vous la traitez comme un objet. Les références sont essentielles pour une syntaxe propre avec surcharge d'opérateur (sujet du prochain chapitre), mais elles ajoutent également une commodité de syntaxe pour passer et renvoyer des objets pour les fonctions ordinaires.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Le constructeur-copie prend une référence à un objet existant du même type que son argument, et il est utilisé pour créer un nouvel objet à partir d'un existant. Le compilateur appelle automatiquement le constructeur-copie quand vous passez ou renvoyez un objet par valeur. Bien que le compilateur crée automatiquement un constructeur-copie pour vous, si vous pensez qu'il en faudra un pour votre classe, vous devriez toujours le définir vous-même pour garantir que le comportement correct aura lieu. Si vous ne voulez pas que l'objet soit passé ou renvoyé par valeur, vous devriez créer un constructeur-copie privé.

Les pointeurs-vers-membres ont la même fonctionnalité que les pointeurs ordinaires : vous pouvez choisir une région de stockage particulière (données ou fonctions) à l'exécution. Il se trouve que les pointeurs-vers-membres travaillent avec les membres d'une classe au lieu de travailler les données ou les fonctions globales. Vous obtenez la flexibilité de programmation qui vous permet de changer de comportement à l'exécution.

11.6 - Exercices

Les solutions à certains exercices peuvent être trouvées dans le document électronique *The Thinking in C++ Annotated Solution Guide*, disponible pour une somme modique sur www.BruceEckel.com.

1 Transformez le fragment de code “bird & rock” au début de ce chapitre en un programme C (en utilisant des **struct**pour le type de données), et montrez que cela compile. Ensuite, essayez de le compiler avec le compilateur C++ et voyez ce qui se produit.
2 Prenez les fragments de code au début de la section “Les références en C++” et placez les dans un **main( )**. Ajoutez des instructions pour afficher ce qui convient pour que vous vous prouviez à vous-même que les références sont comme des pointeurs qui sont automatiquement déréférencés.
3 Ecrivez un programme dans lequel vous essayez de (1) Créer une référence qui ne soit pas initialisée à sa création. (2) Modifier une référence pour qu'elle pointe vers un autre objet après son initialisation. (3) Créer une référence NULL.
4 Ecrivez une fonction qui prend un pointeur comme argument, modifie ce vers quoi il pointe, puis renvoie la destination du pointeur en tant que référence.
5 Créez une classe avec quelques fonctions membres, et faites-en l'objet qui soit pointé par l'argument de l'exercice 4. Faites du pointeur un **const**et faites de certaines des fonctions membres des **const**et montrez que vous ne pouvez appeler que ces fonctions membre **const**dans votre fonction. Faites de l'argument de votre fonction une référence au lieu d'un pointeur.
6 Prenez les fragments de code au début de la section “Les références vers pointeur” et transformez-les en programme.
7 Créez une fonction qui prenne en argument une référence vers un pointeur de pointeur et modifie cet argument. Dans **main( )**appelez cette fonction.
8 Créez une fonction qui prenne un **char&**comme argument et le modifie. Dans **main( )**, affichez une variable **char**, appelez votre fonction avec cette variable, et affichez-la encore pour vous prouvez à vous-même qu'elle a été modifié. Comment cela affecte-t-il la lisibilité du code ?
9 Ecrivez une classe qui a une fonction membre **const**et une fonction membre non- **const**. Ecrivez trois fonctions qui prennent un objet de cette classe comme argument ; la première le prend par valeur, la deuxième par référence, et la troisième par référence **const**. Dans les fonctions, essayez d'appeler les deux fonctions membres de votre classe et expliquez le résultat.
10 (Un peu difficile) Ecrivez une fonction simple qui prend un **int**comme argument, incrémente la valeur, et la retourne. Dans **main( )**, appelez votre fonction. A présent, trouvez comment votre compilateur génère du code assembleur et parcourez les instructions assembleur pour comprendre comment les arguments sont passés et retournés, et comment les variables locales sont indexées depuis la pile.
11 Ecrivez une fonction qui prenne comme argument un **char**, un **int**, un **float**, et un **double**. Générez le code assembleur avec votre compilateur et trouvez les instructions qui poussent les arguments sur la pile avant l'appel de fonction.
12 Ecrivez une fonction qui retourne un **double**. Générez le code assembleur et déterminez comment la valeur est retournée.
13 Produisez le code assembleur pour **PassingBigStructures.cpp**. Parcourez-le et démystifiez la façon dont votre compilateur génère le code pour passer et retourner de grandes structures.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

14 Ecrivez une fonction récursive simple qui décrémente son argument et retourne zéro si l'argument passe à zéro ou autrement s'appelle elle-même. Générez le code assembleur pour cette fonction et expliquez comment la façon dont le code assembleur est créé par le compilateur supporte la récursion.
15 Ecrivez du code pour prouver que le compilateur synthétise automatiquement un constructeur de copie si vous n'en créez pas un vous-même. Prouvez que le constructeur de copie synthétisé réalise une copie bit à bit des types primitifs et appelle le constructeur de copie des types définis par l'utilisateur.
16 Ecrivez une classe avec un constructeur de copie qui s'annonce dans **cout**. Maintenant, créez une fonction qui passe un objet de votre nouvelle classe par valeur et une autre qui crée un objet local de votre nouvelle classe et le retourne par valeur. Appelez ces fonctions pour montrer que le constructeur de copie est, de fait, discrètement appelé quand on passe et retourne des objets par valeur.
17 Créez une classe qui contienne un **double***. Le constructeur initialise le **double***en appelant **new double**et affecte une valeur à l'espace résultant à partir de l'argument du constructeur. Le destructeur affiche la valeur qui est pointée, affecte cette valeur à -1, appelle **delete**pour cet espace de stockage, puis affecte le pointeur à zéro. Ensuite, créez une fonction qui prend un objet de votre classe par valeur, et appelez cette fonction dans le **main( )**. Que se passe-t-il ? Réglez le problème en écrivant un constructeur de copie.
18 Créez une classe avec un constructeur qui ressemble à un constructeur de copie, mais qui possède un argument supplémentaire avec une valeur par défaut. Montrez qu'il est toujours utilisé comme constructeur de copie.
19 Créez une classe avec un constructeur de copie qui s'annonce lui-même. Faites une deuxième classe contenant un objet membre de la première classe, mais ne créez pas de constructeur de copie. Montrez que le constructeur de copie synthétisé dans la deuxième classe appelle automatiquement le constructeur de copie de la première classe.
20 Créez une classe très simple, et une fonction qui retourne un objet de cette classe par valeur. Créez une deuxième fonction qui prenne une référence vers un objet de votre classe. Appelez la première fonction comme argument de la première fonction, et montrez que la deuxième fonction doit utiliser une référence **const**comme argument.
21 Créez une classe simple sans constructeur de copie, et une fonction simple qui prend un objet de cette classe par valeur. A présent, modifiez votre classe en ajoutant une déclaration **private**(uniquement) pour le constructeur de copie. Expliquez ce qu'il se passe lors de la compilation de votre fonction.
22 Cet exercice crée une alternative à l'utilisation du constructeur de copie. Créez une classe **X**et déclarez (mais ne définissez pas) un constructeur de copie **private**. Faites une fonction membre **const**publique **clone( )**qui retourne une copie de l'objet qui est créé en utilisant **new**. A présent, écrivez une fonction qui prenne comme argument un **const X&**et clone une copie locale qui peut être modifiée. L'inconvénient de cette approche est que vous êtes responsable de la destruction explicite de l'objet cloné (en utilisant **delete**) quand vous en avez fini avec lui.
23 Expliquez ce qui ne va pas avec **Mem.cpp**et **MemTest.cpp**du chapitre 7. Corrigez le problème.
24 Créez une classe contenant un **double**et une fonction **print( )**qui affiche le **double**. Dans **main( )**, créez des pointeurs ves membres pour, à la fois, la donnée membre et la fonction membre de votre classe. Créez un objet de votre classe et un pointeur vers cet objet, et manipulez les deux éléments de la classe via vos pointeurs vers membres, en utilisant à la fois l'objet et le pointeur vers l'objet.
25 Créez une classe contenant un tableau de **int**. Pouvez vous le parcourir en utilisant un pointeur vers membre ?
26 Modifiez **PmemFunDefinition.cpp**en ajoutant une fonction membre surchargée **f( )**(vous pouvez déterminez la liste d'argument qui cause la surcharge). Ensuite, créez un deuxième pointeur vers membre, affectez-lui la version surchargée de **f( )**, et appelez la fonction via ce pointeur. Comment se déroule la résolution de la surcharge dans ce cas ?
27 Commencez avec **FunctionTable.cpp**du Chapitre 3. Créez une classe qui contienne un **vector**de pointeurs vers fonctions, ainsi que des fonctions membres **add( )**et **remove( )**pour, respectivement, ajouter et enlever des pointeurs vers fonctions. Ajoutez une fonction **run( )**qui se déplace dans le **vector**et appelle toutes les fonctions.
28 Modifiez l'exercice 27 ci-dessus afin qu'il fonction avec des pointeurs vers fonctions membres à la place.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

12 - Surcharges d'opérateurs

La surcharge d'opérateurs n'est rien d'autre qu'une “douceur syntaxique,” ce qui signifie qu'il s'agit simplement d'une autre manière pour vous de faire un appel de fonction .

La différence est que les arguments de cette fonction n'apparaissent pas entre parenthèses , mais qu'au contraire ils entourent ou sont près de symboles dont vous avez toujours pensé qu'ils étaient immuables.

Il y a deux différences entre l'utilisation d'un opérateur et un appel de fonction ordinaire. La syntaxe est différente; un opérateur est souvent “appelé” en le plaçant entre ou parfois après les arguments. La seconde différence est que le compilateur détermine quelle “fonction” appeler. Par exemple, si vous utilisez l'opérateur **+**avec des arguments flottants, le compilateur “appelle” la fonction pour effectuer l'addition des flottants (cet “appel” consiste principalement en l'insertion de code en-ligne (substitution de code à l'appel) , ou bien une instruction machine du prépocesseur arithmétique). Si vous utilisez l'opérateur **+**avec un flottant et un entier,le compilateur “appelle” une fonction spéciale pour convertir l' **int**en un **float**, puis après “appelle” le code d'addition des flottants.

Mais en C++, il est possible de définir de nouveaux opérateurs qui fonctionnent avec les classes. Cette définition n'est rien d'autre qu'une définition de fonction ordinaire sauf que le nom de la fonction est constitué du mot-clé **operator**suivi par l'opérateur. C'est la seule différence, et cela devient une fonction comme toute autre fonction, que le compilateur appelle lorsqu'il voit le modèle correspondant.

12.1 - Soyez avertis et rassurés

Il est tentant de s'enthousiasmer plus que nécessaire avec la surcharge des opérateurs. Au début c'est un joujou. Mais gardez à l'esprit que ce n'est *qu'une*douceur syntaxique, une autre manière d'appeler une fonction. En voyant les choses comme ça, vous n'avez aucune raison de surcharger un opérateur sauf s'il rend le code concernant votre classe plus facile à écrire et surtout plus facile à *lire*. (Souvenez vous que le code est beaucoup plus souvent lu qu'il n'est écrit) Si ce n'est pas le cas, ne vous ennuyez pas avec ça.

La panique est une autre réponse habituelle à la surcharge des opérateurs; soudain, les opérateurs C n'ont plus leur sens usuel. "Tout est changé et tout mon code C va faire des choses différentes !" Ce n'est pas vrai. Tous les opérateurs utilisés dans des expressions ne contenant que des types prédéfinis ne peuvent pas être changés. Vous ne pouvez jamais surcharger des opérateurs de telle manière que:

```
1 &lt;&lt; 4;
```

se comporte différemment, ou que

```
1.414 &lt;&lt; 2;
```

ait un sens. Seule une expression contenant un type défini par l'utilisateur peut contenir un opérateur surchargé.

12.2 - Syntaxe

Définir un opérateur surchargé c'est comme définir une fonction, mais le nom de cette fonction est **operator@**, où @représente l'opérateur qui est surchargé. Le nombre d'arguments dans la liste des arguments de l'opérateur dépend de deux facteurs:

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

1 Si c'est un opérateur unaire (un argument) ou un opérateur binaire (deux arguments).
2 Si l'opérateur est défini comme une fonction globale (un argument si unaire, deux si binaire) ou bien une fonction membre (zero argument si unaire, un si binaire – l'objet devenant alors l'argument de gauche).

Voici une petite classe qui montre la surcharge d'opérateurs:

```
                    //: C12:OperatorOverloadingSyntax.cpp
#include <iostream>
using namespace std;

class Integer {
  int i;
public:
  Integer(int ii) : i(ii) {}
  const Integer
  operator+(const Integer& rv) const {
    cout << "operator+" << endl;
    return Integer(i + rv.i);
  }
  Integer&
  operator+=(const Integer& rv) {
    cout << "operator+=" << endl;
    i += rv.i;
    return *this;
  }
};

int main() {
  cout << "built-in types:" << endl;
  int i = 1, j = 2, k = 3;
  k += i + j;
  cout << "user-defined types:" << endl;
  Integer ii(1), jj(2), kk(3);
  kk += ii + jj;
} ///:~
```

Les deux opérateurs surchargés sont définis comme des fonctions membres 'inline' qui avertissent quand elles sont appelées. L'argument unique est ce qui apparaît à droite de l'opérateur pour les opérateurs binaires. Les opérateurs unaires n'ont pas d'arguments lorsqu'ils sont définis comme des fonctons membres. La fonction membre est appelée pour l'objet qui se trouve à gauche de l'opérateur.

Pour les opérateurs non conditionnels (les conditionnels retournent d'habitude un booléen), vous souhaiterez presque toujours retourner un objet ou une référence du même type que ceux avec lesquels vous travaillez si les deux arguments sont du même type. (S'ils ne sont pas du même type, l'interprétation du résultat est à votre charge.) De la sorte, des expressons complexes peuvent être formées:

```
                    kk += ii + jj;
```

L' **operator+**produit un nouvel **Entier**(temporaire) qui est utilisé comme l'argument **rv**pour l' **operator+=**. Cette valeur temporaire est détruite aussitôt qu'elle n'est plus nécessaire.

12.3 - Opérateurs surchargeables

Bien que vous puissiez surcharger tous les opérateurs disponibles en C, l'utilisation de la surcharge d'opérateurs comporte des restrictions notoires. En particulier, vous ne pouvez faire des combinaisons d'opérateurs qui, en l'état actuel des choses n'ont aucun sens en C (comme **pour représenter l'exponentiation), vous ne pouvez changer la priorité des opérateurs, et vous ne pouvez changer l'arité (nombre d'arguments requis) pour un opérateur. Tout cela a une raison – toutes ces actions engendreraient des opérateurs plus susceptibles d'apporter de la confusion que de la clarté.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Les deux sous-sections suivantes donnent des exemples de tous les opérateurs “réguliers”, surchargés sous la forme que vous êtes le plus susceptible d'utiliser.

12.3.1 - Opérateurs unaires

L'exemple suivant montre la syntaxe pour surcharger tous les opérateurs unaires, à la fois sous la forme de fonctions globales (non-membres **amies**) et comme fonctions membres. Il étendra la classe **Integer**montrée auparavant et ajoutera une nouvelle classe **byte**. La signification de vos opérateurs particuliers dépendra de la façon dont vous voulez les utiliser, mais prenez en considération le programmeur client avant de faire quelque chose d'inattendu.

Voici un catalogue de toutes les fonctions unaires:

```
                        //: C12:OverloadingUnaryOperators.cpp
#include <iostream>
using namespace std;

// fonctions non membres:
class Integer {
  long i;
  Integer* This() { return this; }
public:
  Integer(long ll = 0) : i(ll) {}
  // pas d'effet de bord prend un argument const&
  friend const Integer&
    operator+(const Integer& a);
  friend const Integer
    operator-(const Integer& a);
  friend const Integer
    operator~(const Integer& a);
  friend Integer*
    operator&(Integer& a);
  friend int
    operator!(const Integer& a);
  // Pour les effets de bord argument non-const& :
  // Préfixé :
  friend const Integer&
    operator++(Integer& a);
  // Postfixé :
  friend const Integer
    operator++(Integer& a, int);
  // Préfixé :
  friend const Integer&
    operator--(Integer& a);
  // Postfixé :
  friend const Integer
    operator--(Integer& a, int);
};

// Opérateurs globaux:
const Integer& operator+(const Integer& a) {
  cout << "+Integer\n";
  return a; // le + unaire n'a aucun effet
}
const Integer operator-(const Integer& a) {
  cout << "-Integer\n";
  return Integer(-a.i);
}
const Integer operator~(const Integer& a) {
  cout << "~Integer\n";
  return Integer(~a.i);
}
Integer* operator&(Integer& a) {
  cout << "&Integer\n";
  return a.This(); // &a est récursif!
}
int operator!(const Integer& a) {
  cout << "!Integer\n";
  return !a.i;
}
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  // Préfixé ; retourne une valeur incrémentée :
  const Integer& operator++(Integer& a) {
    cout << "++Integer\n";
    a.i++;
    return a;
  }
  // Postfixé ; retourne la valeur avant l'incrémentation :
  const Integer operator++(Integer& a, int) {
    cout << "Integer++\n";
    Integer before(a.i);
    a.i++;
    return before;
  }
  // Préfixé ; return la valeur décrémentée :
  const Integer& operator--(Integer& a) {
    cout << "--Integer\n";
    a.i--;
    return a;
  }
  // Postfixé ; retourne la valeur avant décrémentation :
  const Integer operator--(Integer& a, int) {
    cout << "Integer--\n";
    Integer before(a.i);
    a.i--;
    return before;
  }

  // Montre que les opérateurs surchargés fonctionnent:
  void f(Integer a) {
    +a;
    -a;
    ~a;
    Integer* ip = &a;
    !a;
    ++a;
    a++;
    --a;
    a--;
  }

  // Fonctions membres ( "this" implicite):
  class Byte {
    unsigned char b;
  public:
    Byte(unsigned char bb = 0) : b(bb) {}
    // Pas d'effet de bord : fonction membre const :
    const Byte& operator+() const {
      cout << "+Byte\n";
      return *this;
    }
    const Byte operator-() const {
      cout << "-Byte\n";
      return Byte(-b);
    }
    const Byte operator~() const {
      cout << "~Byte\n";
      return Byte(~b);
    }
    Byte operator!() const {
      cout << "!Byte\n";
      return Byte(!b);
    }
    Byte* operator&() {
      cout << "&Byte\n";
      return this;
    }
    // Effets de bord : fonction membre non-const :
    const Byte& operator++() { // Préfixé
      cout << "++Byte\n";
      b++;
      return *this;
    }
    const Byte operator++(int) { // Postfixé
      cout << "Byte++\n";
      Byte before(b);
      b++;
      return before;
    }
    const Byte& operator--() { // Préfixé
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    cout << "--Byte\n";
    --b;
    return *this;
  }
  const Byte operator--(int) { // Postfixé
    cout << "Byte--\n";
    Byte before(b);
    --b;
    return before;
  }
};

void g(Byte b) {
  +b;
  -b;
  ~b;
  Byte* bp = &b;
  !b;
  ++b;
  b++;
  --b;
  b--;
}

int main() {
  Integer a;
  f(a);
  Byte b;
  g(b);
} ///:~
```

Les fonctions sont regroupées suivant la façon dont leurs arguments sont passés. Des conseils pour passer et retourner des arguments seront donnés plus loin. Les formes ci-dessus (et celles qui suivent dans la section suivante) correspondent typiquement à ce que vous utiliserez, aussi commencez en les prenant comme modèles quand vous commencerez à surcharger vos propres opérateurs.

Incrémentation et décrémentation

Les surcharges de **++**et **––**font l'objet d'un dilemne parce que vous voulez pouvoir appeler différentes fonctions selon qu'elles apparaissent avant (préfixé) ou après (postfixé) l'objet sur lequel elles agissent. La solution est simple, mais les gens trouvent parfois cela un peu embrouillé au premier abord. Lorsque le compilateur voit, par exemple, **++a**(une pré-incrémentation), il provoque un appel à **operator++(a)**; mais lorsque il voit **a++,**il provoque un appel à **operator++(a, int)**. Ce qui signifie que, le compilateur différencie les deux formes en faisant des appels à différentes fonctions surchargées. Dans **OverloadingUnaryOperators.cpp**pour les versions des fonctions membres, si le compilateur voit **++b**, il provoque un appel à **B::operator++( )**; s'il voit **b++**il appelle **B::operator++(int)**.

Tout ce que l'utilisateur voit c'est qu'une fonction différente est appelée pour les versions préfixée et postfixée. Dans le fond des choses, toutefois, les deux appels de fonctions ont des signatures différentes, ainsi ils font des liaisons avec des corps de fonctions différents. Le compilateur passe une valeur constante factice pour l'argument **int**(que l'on ne nomme jamais puisque la valeur n'est jamais utilisée) pour générer la signature spécifique pour la version postfixée.

12.3.2 - Opérateurs binaires

Le listing qui suit reproduit l'exemples de **OverloadingUnaryOperators.cpp**pour les opérateurs binaires de sorte que vous ayez un exemple de tous les opérateurs que vous pourriez vouloir surcharger. Cette fois encore, nous donnons les versions fonctions globales et fonctions-membres.

```
                        //: C12:Integer.h
// surcharge non-membres
#ifndef INTEGER_H
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
#define INTEGER_H
#include <iostream>

// fonctions non-membres:
class Integer {
  long i;
public:
  Integer(long ll = 0) : i(ll) {}
  // opérateurs créant une valeur nouvelle modifiée :
  friend const Integer
    operator+(const Integer& left,
              const Integer& right);
  friend const Integer
    operator-(const Integer& left,
              const Integer& right);
  friend const Integer
    operator*(const Integer& left,
              const Integer& right);
  friend const Integer
    operator/(const Integer& left,
              const Integer& right);
  friend const Integer
    operator%(const Integer& left,
              const Integer& right);
  friend const Integer
    operator^(const Integer& left,
              const Integer& right);
  friend const Integer
    operator&(const Integer& left,
              const Integer& right);
  friend const Integer
    operator|(const Integer& left,
              const Integer& right);
  friend const Integer
    operator<<(const Integer& left,
               const Integer& right);
  friend const Integer
    operator>>(const Integer& left,
               const Integer& right);
  // Affectations combinées & retourne une lvalue :
  friend Integer&
    operator+=(Integer& left,
               const Integer& right);
  friend Integer&
    operator-=(Integer& left,
               const Integer& right);
  friend Integer&
    operator*=(Integer& left,
               const Integer& right);
  friend Integer&
    operator/=(Integer& left,
               const Integer& right);
  friend Integer&
    operator%=(Integer& left,
               const Integer& right);
  friend Integer&
    operator^=(Integer& left,
               const Integer& right);
  friend Integer&
    operator&=(Integer& left,
               const Integer& right);
  friend Integer&
    operator|=(Integer& left,
               const Integer& right);
  friend Integer&
    operator>>=(Integer& left,
                const Integer& right);
  friend Integer&
    operator<<=(Integer& left,
                const Integer& right);
  // Opérateurs conditionnels retournent true/false :
  friend int
    operator==(const Integer& left,
               const Integer& right);
  friend int
    operator!=(const Integer& left,
               const Integer& right);
  friend int
    operator<(const Integer& left,
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                const Integer& right);
  friend int
    operator>(const Integer& left,
              const Integer& right);
  friend int
    operator<=(const Integer& left,
               const Integer& right);
  friend int
    operator>=(const Integer& left,
               const Integer& right);
  friend int
    operator&&(const Integer& left,
               const Integer& right);
  friend int
    operator||(const Integer& left,
               const Integer& right);
  // Ecrit le contenu dans un ostream :
  void print(std::ostream& os) const { os << i; }
};
#endif // INTEGER_H ///:~
```

```
                        //: C12:Integer.cpp {O}
// Implementation des opérateurs surchargés
#include "Integer.h"
#include "../require.h"

const Integer
  operator+(const Integer& left,
            const Integer& right) {
  return Integer(left.i + right.i);
}
const Integer
  operator-(const Integer& left,
            const Integer& right) {
  return Integer(left.i - right.i);
}
const Integer
  operator*(const Integer& left,
            const Integer& right) {
  return Integer(left.i * right.i);
}
const Integer
  operator/(const Integer& left,
            const Integer& right) {
  require(right.i != 0, "division par zéro");
  return Integer(left.i / right.i);
}
const Integer
  operator%(const Integer& left,
            const Integer& right) {
  require(right.i != 0, "modulo zéro");
  return Integer(left.i % right.i);
}
const Integer
  operator^(const Integer& left,
            const Integer& right) {
  return Integer(left.i ^ right.i);
}
const Integer
  operator&(const Integer& left,
            const Integer& right) {
  return Integer(left.i & right.i);
}
const Integer
  operator|(const Integer& left,
            const Integer& right) {
  return Integer(left.i | right.i);
}
const Integer
  operator<<(const Integer& left,
             const Integer& right) {
  return Integer(left.i << right.i);
}
const Integer
  operator>>(const Integer& left,
             const Integer& right) {
  return Integer(left.i >> right.i);
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
}
// Affectations modifient & renvoient une lvalue :
Integer& operator+=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* auto-affectation */}
   left.i += right.i;
   return left;
}
Integer& operator-=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* auto-affectation */}
   left.i -= right.i;
   return left;
}
Integer& operator*=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* auto-affectation */}
   left.i *= right.i;
   return left;
}
Integer& operator/=(Integer& left,
                    const Integer& right) {
   require(right.i != 0, "division par zéro");
   if(&left == &right) {/*auto-affectation */}
   left.i /= right.i;
   return left;
}
Integer& operator%=(Integer& left,
                    const Integer& right) {
   require(right.i != 0, "modulo zéro");
   if(&left == &right) {/* auto-affectation */}
   left.i %= right.i;
   return left;
}
Integer& operator^=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* auto-affectation */}
   left.i ^= right.i;
   return left;
}
Integer& operator&=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* auto-affectation */}
   left.i &= right.i;
   return left;
}
Integer& operator|=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* auto-affectation */}
   left.i |= right.i;
   return left;
}
Integer& operator>>=(Integer& left,
                     const Integer& right) {
   if(&left == &right) {/* auto-affectation */}
   left.i >>= right.i;
   return left;
}
Integer& operator<<=(Integer& left,
                     const Integer& right) {
   if(&left == &right) {/* auto-affectation */}
   left.i <<= right.i;
   return left;
}
// les opérateurs conditionnels renvoient true/false :
int operator==(const Integer& left,
               const Integer& right) {
    return left.i == right.i;
}
int operator!=(const Integer& left,
               const Integer& right) {
    return left.i != right.i;
}
int operator<(const Integer& left,
              const Integer& right) {
    return left.i < right.i;
}
int operator>(const Integer& left,
              const Integer& right) {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    return left.i > right.i;
}
int operator<=(const Integer& left,
               const Integer& right) {
    return left.i <= right.i;
}
int operator>=(const Integer& left,
               const Integer& right) {
    return left.i >= right.i;
}
int operator&&(const Integer& left,
               const Integer& right) {
    return left.i && right.i;
}
int operator||(const Integer& left,
               const Integer& right) {
    return left.i || right.i;
} ///:~
```

```
                        //: C12:IntegerTest.cpp
//{L} Integer
#include "Integer.h"
#include <fstream>
using namespace std;
ofstream out("IntegerTest.out");

void h(Integer& c1, Integer& c2) {
  // Une expression complexe :
  c1 += c1 * c2 + c2 % c1;
  #define TRY(OP) \
    out << "c1 = "; c1.print(out); \
    out << ", c2 = "; c2.print(out); \
    out << ";  c1 " #OP " c2 produit "; \
    (c1 OP c2).print(out); \
    out << endl;
  TRY(+) TRY(-) TRY(*) TRY(/)
  TRY(%) TRY(^) TRY(&) TRY(|)
  TRY(<<) TRY(>>) TRY(+=) TRY(-=)
  TRY(*=) TRY(/=) TRY(%=) TRY(^=)
  TRY(&=) TRY(|=) TRY(>>=) TRY(<<=)
  // Conditionelles:
  #define TRYC(OP) \
    out << "c1 = "; c1.print(out); \
    out << ", c2 = "; c2.print(out); \
    out << ";  c1 " #OP " c2 produit "; \
    out << (c1 OP c2); \
    out << endl;
  TRYC(<) TRYC(>) TRYC(==) TRYC(!=) TRYC(<=)
  TRYC(>=) TRYC(&&) TRYC(||)
}

int main() {
  cout << "fonctions amies" << endl;
  Integer c1(47), c2(9);
  h(c1, c2);
} ///:~
```

```
                        //: C12:Byte.h
// opérateurs surchargés par des membres
#ifndef BYTE_H
#define BYTE_H
#include "../require.h"
#include <iostream>
// fonctions membres ("this" implicite) :
class Byte {
  unsigned char b;
public:
  Byte(unsigned char bb = 0) : b(bb) {}
  // Pas d'effet de bord: fonction membre const:
  const Byte
    operator+(const Byte& right) const {
    return Byte(b + right.b);
  }
  const Byte
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  operator-(const Byte& right) const {
  return Byte(b - right.b);
}
const Byte
  operator*(const Byte& right) const {
  return Byte(b * right.b);
}
const Byte
  operator/(const Byte& right) const {
  require(right.b != 0, "division par zéro");
  return Byte(b / right.b);
}
const Byte
  operator%(const Byte& right) const {
  require(right.b != 0, "modulo zéro");
  return Byte(b % right.b);
}
const Byte
  operator^(const Byte& right) const {
  return Byte(b ^ right.b);
}
const Byte
  operator&(const Byte& right) const {
  return Byte(b & right.b);
}
const Byte
  operator|(const Byte& right) const {
  return Byte(b | right.b);
}
const Byte
  operator<<(const Byte& right) const {
  return Byte(b << right.b);
}
const Byte
  operator>>(const Byte& right) const {
  return Byte(b >> right.b);
}
// Les affectations modifient & renvoient une lvalue.
// operator= ne peut être qu'une fonction membre :
Byte& operator=(const Byte& right) {
  // traite l' auto-affectation:
  if(this == &right) return *this;
  b = right.b;
  return *this;
}
Byte& operator+=(const Byte& right) {
  if(this == &right) {/* auto-affectation */}
  b += right.b;
  return *this;
}
Byte& operator-=(const Byte& right) {
  if(this == &right) {/* auto-affectation */}
  b -= right.b;
  return *this;
}
Byte& operator*=(const Byte& right) {
  if(this == &right) {/* auto-affectation */}
  b *= right.b;
  return *this;
}
Byte& operator/=(const Byte& right) {
  require(right.b != 0, "division par zéro");
  if(this == &right) {/* auto-affectation */}
  b /= right.b;
  return *this;
}
Byte& operator%=(const Byte& right) {
  require(right.b != 0, "modulo zéro");
  if(this == &right) {/* auto-affectation */}
  b %= right.b;
  return *this;
}
Byte& operator^=(const Byte& right) {
  if(this == &right) {/* auto-affectation */}
  b ^= right.b;
  return *this;
}
Byte& operator&=(const Byte& right) {
  if(this == &right) {/* auto-affectation */}
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    b &= right.b;
    return *this;
  }
  Byte& operator|=(const Byte& right) {
    if(this == &right) {/* auto-affectation */}
    b |= right.b;
    return *this;
  }
  Byte& operator>>=(const Byte& right) {
    if(this == &right) {/* auto-affectation */}
    b >>= right.b;
    return *this;
  }
  Byte& operator<<=(const Byte& right) {
    if(this == &right) {/* auto-affectation */}
    b <<= right.b;
    return *this;
  }
  // les opérateurs conditionnels renvoient true/false :
  int operator==(const Byte& right) const {
      return b == right.b;
  }
  int operator!=(const Byte& right) const {
      return b != right.b;
  }
  int operator<(const Byte& right) const {
      return b < right.b;
  }
  int operator>(const Byte& right) const {
      return b > right.b;
  }
  int operator<=(const Byte& right) const {
      return b <= right.b;
  }
  int operator>=(const Byte& right) const {
      return b >= right.b;
  }
  int operator&&(const Byte& right) const {
      return b && right.b;
  }
  int operator||(const Byte& right) const {
      return b || right.b;
  }
  // Ecrit le contenu dans un ostream:
  void print(std::ostream& os) const {
    os << "0x" << std::hex << int(b) << std::dec;
  }
};
#endif // BYTE_H ///:~
```

```
                        //: C12:ByteTest.cpp
#include "Byte.h"
#include <fstream>
using namespace std;
ofstream out("ByteTest.out");

void k(Byte& b1, Byte& b2) {
  b1 = b1 * b2 + b2 % b1;

  #define TRY2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ";  b1 " #OP " b2 produit "; \
    (b1 OP b2).print(out); \
    out << endl;

  b1 = 9; b2 = 47;
  TRY2(+) TRY2(-) TRY2(*) TRY2(/)
  TRY2(%) TRY2(^) TRY2(&) TRY2(|)
  TRY2(<<) TRY2(>>) TRY2(+=) TRY2(-=)
  TRY2(*=) TRY2(/=) TRY2(%=) TRY2(^=)
  TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)
  TRY2(=) // opérateur d'affectation

  // Conditionelles:
  #define TRYC2(OP) \
    out << "b1 = "; b1.print(out); \
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
      out << ", b2 = "; b2.print(out); \
      out << ";  b1 " #OP " b2 produit "; \
      out << (b1 OP b2); \
      out << endl;

    b1 = 9; b2 = 47;
    TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)
    TRYC2(>=) TRYC2(&&) TRYC2(||)

    // affectations en série:
    Byte b3 = 92;
    b1 = b2 = b3;
  }

  int main() {
    out << "fonctions membres :" << endl;
    Byte b1(47), b2(9);
    k(b1, b2);
  } ///:~
```

Vous pouvez voir que **operator=**ne peut être qu'une fonction membre. Cela est expliqué plus loin.

Notez que tous les opérateurs d'affectation ont une portion de code pour vérifier l'auto-affectation ; c'est une règle générale. Dans certains cas ce n'est pas nécessaire ; par exemple, avec **operator+=**vous *voulez*souvent dire **A+=A**et vouloir que **A**s'ajoute à lui-même. L'endroit le plus important à vérifier pour l'auto-affectation est **operator=**parce qu'avec des objets complexes les conséquences peuvent être désastreuses. (Dans certains cas ça passe, mais vous devez toujours garder cela à l'esprit en écrivant **operator=**.)

Tous les opérateurs montrés dans les deux exemples précédents sont surchargés pour manipuler un type unique. Il est également possible de surcharger des opérateurs pour manipuler en même temps des types distincts, pour vous permettre d'additionner des pommes et des oranges, par exemple. Avant de vous lancer dans une surcharge exhaustive des opérateurs, toutefois, vous devriez jeter un coup d'oeil à la section consacrée aux conversions automatiques de types plus loin dans ce chapitre. Souvent, une conversion de type au bon endroit peut vous faire faire l'économie d'un grand nombre de surcharges d'opérateurs.

12.3.3 - Arguments & valeurs de retour

Cela peut sembler un peu troublant tout d'abord quand vous regardez **OverloadingUnaryOperators.cpp**, **Integer.h**et **Byte.h**et que vous voyez toutes les différentes façons dont les arguments sont passés et renvoyés. Bien que vous *puissiez*passer et retourner des arguments comme vous l'entendez, les choix dans ces exemples n'ont pas été faits au hasard. Ils suivent un schéma logique, le même que celui que vous voudrez utiliser dans la plupart de vos choix.

1. Comme avec n'importe quel argument de fonction, si vous avez seulement besoin de lire l'argument sans le changer, choisissez plutôt de le passer comme une référence **const**. Les opératons arithmétiques ordinaires (comme **+**et **–**, etc.) et les booléens ne changeront pas leurs arguments, aussi, les passer par référence **const**est la technique que vous utiliserez dans la plupart des cas. Lorsque la fonction est un membre de classe, ceci revient à en faire une fonction membre **const**. Seulement avec les opérateurs d'affectation (comme **+=**) et **operator=**, qui changent l'argument de gauche, l'argument de gauche n'est *pas*une constante, mais il est toujours passé par adresse parce qu'il sera modifié.
2. Le type de la valeur de retour que vous devez choisir dépend de la signification attendue pour l'opérateur. (Je le répète, vous pouvez faire tout ce que vous voulez avec les arguments et les valeurs de retour.) Si l'effet de l'opérateur consiste à générer une nouvelle valeur, il se peut que vous ayez besoin de générer un nouvel objet comme valeur de retour. Par exemple, **Integer::operator+**doit produire un objet **Integer**qui est la somme des opérandes. Cet objet est retourné par valeur comme un **const**, ainsi la résultat ne peut être modifié comme 'lvalue'.
3. Tous les opérateurs d'affectation modifient la 'lvalue'. Pour permettre au résultat de l'affectation d'être utilisé dans des expressions chaînées, comme **a=b=c**, on s'attend à ce que vous retourniez une référence identique à cette même 'lvalue' qui vient d'être modifiée. Mais cette référence doit-elle être **const**ou non **const**? Bien

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

que vous lisiez **a=b=c**de gauche à droite, le compilateur l'analyse de droite à gauche, de sorte que vous n'êtes pas forcé de retourner un non **const**pour supporter l'affectation en chaîne. Toutefois, les gens s'attendent parfois à être capables d'effectuer une opération sur la chose venant juste d'être affectée, comme **(a=b).func( );**pour appeler **func( )**sur **a**après affectation de **b**sur lui. Dans ce cas la valeur de retour pour tous les opérateurs d'affectation devrait être une référence non **const**à la 'lvalue'.

4 Pour les opérateurs logiques, tout le monde s'attend à récupérer au pire un **int**, et au mieux un **bool**. (Les librairies développées avant que la plupart des compilateurs ne supportent le type prédéfini C++'s **bool**utiliseront **int**ou un **typedef**équivalent.)

Les opérateurs d'incrémentaton et de décrémentation sont la source d'un dilemne à cause des versions préfixée et postfixée. Les deux versions modifient l'objet, et ne peuvent, de la sorte, traiter l'objet comme un **const**. La version préfixée retourne la valeur de l'objet après qu'il eut été changé, vous espérez donc récupérer l'objet ayant été modifié. Ainsi, avec la version préfixée vous pouvez simplement retourner ***this**en tant que référence. La version postfixée est supposée retourner la valeur *avant*qu'elle ne soit changée, aussi vous êtes obligés de créer un objet séparé pour représenter cette valeur et de le retourner. Ainsi avec la version postfixée, vous devez retourner par valeur si vous voulez préserver la sémantique attendue (Notez que vous trouverez parfois les opérateurs d'incrémentation et de décrémentation renvoyant un **int**ou un **bool**pour indiquer, par exemple, si un objet conçu pour se déplacer à l'intérieur d'une liste est parvenu à la fin de cette liste). Maintenant la question est : ces valeurs doivent elles être renvoyées **const**ou non **const**? Si vous donnez la permission de modifier l'objet et que quelqu'un écrit **(++a).func( )**, **func( )**travaillera sur **a**lui-même, mais avec **(a++).func( )**, **func( )**travaille sur l'objet temporaire retourné par l'opérateur postfixé **operator++**. Les objets temporaires sont automatiquement **const**, de sorte que ceci sera repéré par le compilateur, mais pour des raisons de cohérence il est plus logique de les rendre tous deux **const**, comme ce qui a été fait ici. Sinon vous pouvez choisir de rendre la version préfixée non- **const**et la postfixée **const**. A cause de la variété des significations que vous pouvez vouloir donner aux opérateurs d'incrément et de décrément, il faut faire une étude au cas par cas.

Retour par valeur en tant que const

Retourner par valeur en **const**peut sembler, à première vue, un peu subtil, cela mérite donc un peu plus d'explications. Considerez l'opérateur binaire **operator+**. Si vous l'utilisez dans une expression telle que **f(a+b)**, le résultat de **a+b**devient un objet temporaire qui est utilisé dans l'appel à **f( )**. Parce qu'il est temporaire, il est automatiquement **const**, ainsi le fait de rendre la valeur de retour explicitement **const**ou le contraire n'a aucun effet.

Cependant, il vous est aussi possible d'envoyer un message à la valeur de retour de **a+b**, plutôt que de la passer en argument à la fonction. Par exemple, vous pouvez dire **(a+b).g( )**, où **g( )**est quelque fonction membre de **Integer**dans ce cas. En rendant la valeur de retour **const**, vous stipulez que seule une fonction membre **const**peut être appelée pour cette valeur de retour. C'est ' **const**-correct', parce que cela vous empêche de stocker une information potentiellement utile dans un objet qui sera selon toute vraisemblance perdu.

L'optimisation de la valeur de retour

Lorsque de nouveaux objets sont créés pour un retour par valeur, remarquez la forme utilisée. Dans **operator+**, par exemple:

```
return Integer(left.i + right.i);
```

Ceci peut ressembler, à première vue, à, un “appel à un constructeur”, mais ce n'en est pas un. La syntaxe est celle d'un objet temporaire ; l'instruction dit “fabrique un objet **Integer**temporaire et retourne le.” A cause de cela, vous pourriez penser que le résultat est le même que de créer un objet local nommé et le retourner. Cependant, c'est tout à fait différent. Si au lieu de cela vous deviez dire :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                        Integer tmp(left.i + right.i);
  return tmp;
```

Il se passerait trois choses. D'abord, l'objet **tmp**est créé incluant un appel à son constructeur. Ensuite, le constructeur de copie reproduit **tmp**à l'emplacement de la valeur de retour vers l'appelant. Troisièmement, le destructeur est appelé pour **tmp**à la fin de la portée.

Au contraire, l'approche “retourner un temporaire” opère de façon sensiblement différente. Quand le compilateur vous voit faire cela, il sait que vous n'avez pas de besoin ultérieur de l'objet qu'il crée autre que le renvoyer. Le compilateur tire avantage de celà en bâtissant l'objet *directement*à l'endroit de la valeur de retour vers l'appelant. Ceci ne nécessite qu'un simple appel de constructeur (pas besoin du constructeur de copie) et il n'y a pas d'appel de destructeur parce que vous ne créez jamais effectivement un objet local. Ainsi, alors que cela ne coûte rien de plus que la compétence du programmeur, c'est sensiblement plus efficace. On appelle souvent cela *optimisation de la valeur de retour*.

12.3.4 - opérateurs inhabituels

Plusieurs autres opérateurs ont une syntaxe sensiblement différente pour la surcharge .

L'opérateur d'indexation, **operator[ ]**, doit être une fonction membre et il nécessite un argument unique. Parce que **operator[ ]**implique que l'objet pour lequel il est appelé agisse comme un tableau, vous ferez souvent retourner une référence par cet opérateur, de façon qu'il puisse être facilement utilisé comme partie gauche d'une affectation. Cet opérateur est fréquemment surchargé ; vous verrez des exemples dans le reste de l'ouvrage.

Les opérateurs **new**et **delete**contrôlent l'allocation dynamique de mémoire et peuvent être surchargés d'un grand nombre de manières. Ce sujet est couvert dans le chapitre 13.

Opérateur virgule

L'opérateur virgule est appelé lorsqu'il apparaît à côté d'un objet du type pour lequel la virgule est définie. Toutefois, “ **operator,**” *n'est pas*appelé pour des listes d'arguments de fonctions, seulement pour des objets qui sont à l'extérieur, séparés par des virgules. Il ne semble pas y avoir beaucoup d'utilisations pratiques de cet opérateur; il existe dans le langage pour des raisons de cohérence. Voici un exemple montrant comment l'opérateur virgule peut être appelé quand la virgule apparaît *avant*un objet, aussi bien qu'après :

```
                        //: C12:OverloadingOperatorComma.cpp
  #include <iostream>
  using namespace std;

  class After {
  public:
    const After& operator,(const After&) const {
      cout << "After::operator,()" << endl;
      return *this;
    }
  };

  class Before {};

  Before& operator,(int, Before& b) {
    cout << "Before::operator,()" << endl;
    return b;
  }

  int main() {
    After a, b;
    a, b;  // Appel de l'opérateur virgule
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    Before c;
    1, c;  // Appel de l'opérateur virgule
} ///:~
```

La fonction globale permet à la virgule d'être placée avant l'objet en question. L'usage montré est relativement obscur et discutable. Bien que vous puissiez probablement utiliser une liste séparée par des virgules comme partie d'une expression plus complexe, c'est trop subtil à utiliser dans la plupart des cas.

Operator->

L' **opérateur–>**est généralement utilisé quand vous voulez faire apparaître un objet comme un pointeur. Etant donné qu'un tel objet comporte plus de “subtilités” qu'il n'en existe pour un pointeur usuel, un tel objet est souvent appelé un *pointeur intelligent*. Ils sont particulièrement utiles si vous voulez “envelopper” une classe autour d'un pointeur pour rendre ce pointeur sécurisé, ou bien dans le rôle habituel d'un *itérateur*, qui est un objet générique qui se déplace dans une *collection / conteneur*d'autres objets et les choisit un par un, sans procurer d'accès direct à l'implémentation du conteneur (Vous trouverez souvent des conteneurs et des itérateurs dans les librairies de classes, comme dans la Librairie Standard C++, décrite dans le Volume 2 de ce livre).

Un opérateur de déréférencement de pointeur doit être une fonction membre. Il possède des contraintes supplémentaires atypiques : Il doit retourner un objet (ou une référerence à un objet) qui possède aussi un opérateur de déréférencement de pointeur, ou il doit retourner un pointeur qui peut être utilisé pour sélectionner ce que la flèche de déréférencement de pointeur pointe. Voici un exemple simple:

```
                          //: C12:SmartPointer.cpp
#include <iostream>
#include <vector>
#include "../require.h"
using namespace std;

class Obj {
  static int i, j;
public:
  void f() const { cout << i++ << endl; }
  void g() const { cout << j++ << endl; }
};

// définitions de membres statiques :
int Obj::i = 47;
int Obj::j = 11;

// Conteneur :
class ObjContainer {
  vector<Obj*> a;
public:
  void add(Obj* obj) { a.push_back(obj); }
  friend class SmartPointer;
};

class SmartPointer {
  ObjContainer& oc;
  int index;
public:
  SmartPointer(ObjContainer& objc) : oc(objc) {
    index = 0;
  }
  // La valeur de retour indique la fin de liste:
  bool operator++() { // Préfixé
    if(index >= oc.a.size()) return false;
    if(oc.a[++index] == 0) return false;
    return true;
  }
  bool operator++(int) { // Postfixé
    return operator++(); // Utilise la version préfixée
  }
  Obj* operator->() const {
    require(oc.a[index] != 0, "Valeur nulle "
      "renvoyée par SmartPointer::operator->()");
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    return oc.a[index];
  }
};

int main() {
  const int sz = 10;
  Obj o[sz];
  ObjContainer oc;
  for(int i = 0; i < sz; i++)
    oc.add(&o[i]); // Le remplit
  SmartPointer sp(oc); // Crée un itérateur
  do {
    sp->f(); // Appel de déréférencement de pointeur
    sp->g();
  } while(sp++);
} ///:~
```

La classe **Obj**définit les objets qui sont manipulés par le programme. Les fonctions **f( )**et **g( )**ne font qu'afficher les valeurs intéressantes des données membres **static**. Les pointeurs vers ces objets sont stockés dans des conteneurs du type **ObjContainer**en utilisant sa fonction **add( )**. **ObjContainer**ressemble à un tableau de pointeurs, mais vous remarquerez qu'il n'existe aucun moyen d'en retirer les pointeurs. Toutefois, **SmartPointer**est déclarée comme une classe **friend**, de sorte qu'elle a le droit de regarder à l'intérieur du conteneur. La classe **SmartPointer**class ressemble beaucoup à un pointeur intelligent – vous pouvez le déplacer en utilisant **operator++**(vous pouvez également définir un **operator– –**), il n'ira pas au delà du conteneur dans lequel il pointe, et il restitue (au moyen de l'opérateur de déréférencement) la valeur vers laquelle il pointe. Notez que **SmartPointer**est une spécialisation pour le conteneur pour lequel il est créé ; à la différence d'un pointeur ordinaire, il n'existe aucun pointeur intelligent “à tout faire”. Vous en aprendrez plus sur les pointeurs intelligents appelés “itérateurs” dans le dernier chapitre de ce livre et dans le Volume 2 (téléchargeable depuis *www.BruceEckel.com*).

Dans **main( )**, un fois que le conteneur **oc**est rempli avec des objets **Obj**, un **SmartPointer sp**est créé. Les appels au pointeur intelligent surviennent dans les expressions :

```
    sp->f(); // appels au pointeur intelligent
sp->g();
```

Ici, même si **sp**n'a pas, en fait de fonctions membres **f( )**et **g( )**, l'opérateur de déréférencement appelle automatiquement ces fonctions pour le **Obj***qui est retourné par **SmartPointer::operator–>**. Le compilateur effectue tout le travail de vérification pour s'assurer que l'appel de fonction marche correctement.

Bien que la mécanique sous-jacente de l'opérateur de déréférencement de pointeur soit plus complexe que pour les autres opérateurs, le but est exactement le même: procurer une syntaxe plus pratique pour les utilisateurs de vos classes

Un itérateur imbriqué

Il est plus commun de voir une classe “smart pointer” ou “iterator” imbriquée dans la classe qu'elle sert. L'exemple précédent peut être réécrit pour imbriquer **SmartPointer**à l'intérieur de **ObjContainer**comme ceci :

```
    //: C12:NestedSmartPointer.cpp
#include <iostream>
#include <vector>
#include "../require.h"
using namespace std;

class Obj {
  static int i, j;
public:
  void f() { cout << i++ << endl; }
  void g() { cout << j++ << endl; }
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
};

// définitions des membres statiques :
int Obj::i = 47;
int Obj::j = 11;

// Conteneur:
class ObjContainer {
  vector<Obj*> a;
public:
  void add(Obj* obj) { a.push_back(obj); }
  class SmartPointer;
  friend class SmartPointer;
  class SmartPointer {
    ObjContainer& oc;
    unsigned int index;
  public:
    SmartPointer(ObjContainer& objc) : oc(objc) {
      index = 0;
    }
    // La valeur de retour signale la fin de la liste :
    bool operator++() { // Préfixé
      if(index >= oc.a.size()) return false;
      if(oc.a[++index] == 0) return false;
      return true;
    }
    bool operator++(int) { // Postfixé
      return operator++(); // Utilise la version préfixée
    }
    Obj* operator->() const {
      require(oc.a[index] != 0, "valeur Zéro "
        "renvoyée par SmartPointer::operator->()");
      return oc.a[index];
    }
  };
  // Fonction qui fournit un pointeur intelligent
  // pointant au début de l'ObjContainer:
  SmartPointer begin() {
    return SmartPointer(*this);
  }
};

int main() {
  const int sz = 10;
  Obj o[sz];
  ObjContainer oc;
  for(int i = 0; i < sz; i++)
    oc.add(&o[i]); // Remplissage
  ObjContainer::SmartPointer sp = oc.begin();
  do {
    sp->f(); // Appel à l'opérateur de déréférencement
    sp->g();
  } while(++sp);
} ///:~
```

A côté de l'imbrication effective de la classe, il y a ici deux différences. La première est dans la déclaration de la classe de sorte qu'elle puisse être **amie**:

```
                        class SmartPointer;
friend SmartPointer;
```

Le compilateur doit d'abord savoir que la classe existe avant de pouvoir lui dire que c'est une **amie**.

La seconde différence est dans la fonction membre **begin( )**de **ObjContainer**, qui fournit un **SmartPointer**qui pointe au début de la suite **ObjContainer**. Bien que ce soit en fait seulement une disposition d'ordre pratique, cela a une valeur universelle parce qu'elle se conforme à la règle utilisée dans la Librairie Standard C++.

Operator->*

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

L' **opérateur–>***est un opérateur binaire qui se comporte commme tous les autres opérateurs binaires. Il est proposé pour les situations où vous voulez imiter le comportement induit par la syntaxe *pointeur sur membre*, décrite dans le chapitre précédent.

Tout comme **operator->**, l'opérateur de déréférencement pointeur sur membre est généralement utilisé avec un type d'objet qui représente un “pointeur intelligent,” bien que l'exemple montré ici soit plus simple de façon à être compréhensible. L'astuce en définissant **operator->***est qu'il doit retourner un objet pour lequel **operator( )**puisse être appelé avec les arguments de la fonction membre que vous appelez.

L' *opérateur d'appel de fonction* **operator( )**doit être une fonction membre, et il est unique en ce qu'il permet un nombre quelconque d'arguments. Il fait en sorte que votre objet ressemble effectivement à une fonction. Bien que vous puissiez définir plusieurs fonctions **operator( )**avec différents arguments, il est souvent utilisé pour des types ayant une seule opération, ou au moins une particulièrement dominante. Vous verrez dans le volume 2 que la Librairie Standard C++ utilise l'opérateur d'appel de fonction afin de créer des “objets fonction.”

Pour créer un **operator->***vous devez commencer par créer une classe avec un **operator( )**qui est le type d'objet que **operator->***retournera. Cette classe doit, d'une manière ou d'une autre, encapsuler l'information nécessaire pour que quand l' **operator( )**est appelé (ce qui se produit automatiquement), le pointeur-sur-membre soit déréférencé pour l'objet. Dans l'exemple suivant, le constructeur **FunctionObject**capture et stocke à la fois le pointeur sur l'objet et le pointeur sur la fonction membre, et alors l' **operator( )**utilise ceux-ci pour faire l'appel effectif pointeur-sur-membre

```
                        //: C12:PointerToMemberOperator.cpp
#include <iostream>
using namespace std;

class Dog {
public:
  int run(int i) const {
    cout << "court\n";
    return i;
  }
  int eat(int i) const {
     cout << "mange\n";
     return i;
  }
  int sleep(int i) const {
    cout << "ZZZ\n";
    return i;
  }
  typedef int (Dog::*PMF)(int) const;
  // l'opérateur->* doit retourner un objet
  // ayant un operator():
  class FunctionObject {
    Dog* ptr;
    PMF pmem;
  public:
    // Enregistrer le pointeur sur objet et le pointeur sur membre
    FunctionObject(Dog* wp, PMF pmf)
      : ptr(wp), pmem(pmf) {
      cout << "constructeur FunctionObject\n";
    }
    // Faire l'appel en utilisant le pointeur sur objet
    // et le pointeur membre
    int operator()(int i) const {
      cout << "FunctionObject::operator()\n";
      return (ptr->*pmem)(i); // Faire l'appel
    }
  };
  FunctionObject operator->*(PMF pmf) {
    cout << "operator->*" << endl;
    return FunctionObject(this, pmf);
  }
};

int main() {
  Dog w;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    Dog::PMF pmf = &Dog::run;
    cout << (w->*pmf)(1) << endl;
    pmf = &Dog::sleep;
    cout << (w->*pmf)(2) << endl;
    pmf = &Dog::eat;
    cout << (w->*pmf)(3) << endl;
} ///:~
```

Doga trois fonctions membres, chacune d'elles prend un argument **int**et retourne un **int**. **PMF**est un **typedef**pour simplifier la définition d'un pointeur-sur-membre sur les fonctions membres de **Dog**.

Un **FunctionObject**est créé et retourné par **operator->***. Notez que **operator->***connaît à la fois l'objet pour lequel le pointeur-sur-membre est appelé (**this**) et le pointeur-sur-membre, et il les passe au constructeur de **FunctionObject**qui conserve les valeurs. Lorsque **operator->***est appelé, le compilateur le contourne immédiatement et appelle **operator( )**pour la valeur de retour de **operator->***, en passant les arguments qui étaient donnés à **operator->***. Le **FunctionObject::operator( )**prend les arguments et ensuite déréférence le pointeur-sur-membre “reél” en utilisant les pointeurs sur objet et pointeur-sur-membre enregistrés.

Remarquez que ce que vous êtes en train de faire là, tout comme avec **operator->**, consiste à vous insérer au beau milieu de l'appel à **operator->***. Ceci vous donne la possibilité d'accomplir certaines opératons supplémentaires si le besoin s'en fait sentir.

Le mécanisme de l' **operator->***implémenté ici ne fonctionne que pour les fonctions membres prenant un argument **int**et retournant un **int**. C'est une limitation, mais si vous essayez de créer des mécanismes surchargés pour chaque possibilité différente, cela paraît une tâche prohibitive. Heureusement, le mécanisme **template**de C++ (décrit dans le dernier chapitre de ce livre, et dans le Volume 2) est conçu pour traiter exactement ce genre de problème.

12.3.5 - Opérateurs que vous ne pouvez pas surcharger

Il y a certains opérateurs, dans le jeu disponible, qui ne peuvent être surchargés. La raison générale invoquée pour cela est la sécurité. Si ces opérateurs étaient surchargeables, cela saboterait ou réduirait à néant les mécanismes de sécurité, rendraient les choses plus difficiles, ou jetterait le trouble sur les pratiques existantes.

- L'opérateur de sélection de membre **operator.**. Actuellement, le point a une signification pour tout membre d'une classe, mais si vous permettez qu'on le surcharge, alors il ne vous serait plus possible d'accéder aux membres de la façon habituelle ; Il vous faudrait à la place un pointeur et la flèche **operator->**.
- Le déréférencement de pointeur-sur-membre **operator.***, pour les mêmes raisons que **operator.**.
- Il n'y a pas d'opérateur d'exponentiation. Le candidat le plus populaire pour ça était **operator****deFortran, mais cela soulevait des questions difficiles pour l'analyseur syntaxique. C'est pourquoi, C n'a pas d'opérateur d'exponentiation, et de même C++ ne semble pas en avoir besoin lui-même parce que vous pouvez toujours effectuer un appel de fonction. Un opérateur d'exponentiation ajouterait une notation pratique, mais aucune nouvelle fonctionnalité en rapport avec la nouvelle complexité induite pour le compilateur.
- Il n'y a pas d'opérateurs définis par l'utilisateur. Ce qui signifie que vous ne pouvez créer de nouveaux opérateurs qui ne sont pas dans le jeu standard. Une partie du problème réside dans la détermination des règles de priorité, une autre dans un un besoin insuffisant au vu du dérangement introduit.
- Vous ne pouvez changer les règles de priorité. Elles sont assez difficiles comme ça, pour ne pas laisser les gens jouer avec.

12.4 - Opérateurs non membres

Dans quelques uns des exemples précédents, les opérateurs peuvent être membres ou non membres, et cela ne semble pas faire une grande différence. Cela soulève habituellement la question, “Lequel dois-je choisir?” En général, si cela ne fait aucune différence, il faudrait opter pour les membres, pour mettre en relief l'association

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

entre l'opérateur et sa classe. Lorsque le membre de gauche est toujours un objet de la classe courante, cela marche bien.

Cependant, quelquefois vous voulez que l'opérande de gauche soit un objet de quelque autre classe. Un endroit courant où vous verrez cela est avec les opérateurs **<<**et **>>**surchargés pour les iostreams. Etant donné que les iostreams constituent une librairie C++ fondamentale, vous voudrez certainement surcharger ces opérateurs pour la plupart de vos classes, aussi cela vaut la peine de mémoriser le processus :

```
//: C12:IostreamOperatorOverloading.cpp
// Example of non-member overloaded operators
#include "../require.h"
#include <iostream>
#include <sstream> // "flux de chaînes"
#include <cstring>
using namespace std;

class IntArray {
  enum { sz = 5 };
  int i[sz];
public:
  IntArray() { memset(i, 0, sz* sizeof(*i)); }
  int& operator[](int x) {
    require(x >= 0 && x < sz,
      "IntArray::operator[] index hors limites");
    return i[x];
  }
  friend ostream&
    operator<<(ostream& os, const IntArray& ia);
  friend istream&
    operator>>(istream& is, IntArray& ia);
};

ostream&
operator<<(ostream& os, const IntArray& ia) {
  for(int j = 0; j < ia.sz; j++) {
    os << ia.i[j];
    if(j != ia.sz -1)
      os << ", ";
  }
  os << endl;
  return os;
}

istream& operator>>(istream& is, IntArray& ia){
  for(int j = 0; j < ia.sz; j++)
    is >> ia.i[j];
  return is;
}

int main() {
  stringstream input("47 34 56 92 103");
  IntArray I;
  input >> I;
  I[4] = -1; // Utilise l'operator[] surchargé
  cout << I;
} ///:~
```

Cette classe contient aussi un opérateur **[ ]**, qui retourne une référence vers une valeur valide du tableau. Parce qu'on retourne une référence, l'expression

```
I[4] = -1;
```

non seulement a l'air plus civilisée que si on utilisait des pointeurs, mais elle accomplit aussi l'effet désiré.

Il est important que les opérateurs de décalage surchargés passent et retournent *par référence*, de sorte que les actions affectent les objets externes. Dans les définitions de fonctions, des expressions comme

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                os << ia.i[j];
```

ont pour effet que les fonctions surchargées d'opérateurs *existantes*sont appelées (c'est à dire, celles définies dans **<iostream>**). Dans ce cas, la fonction appelée est **ostream& operator<<(ostream&, int)**parce que **ia.i[j]**est résolu comme un **int**.

Une fois que toutes les actions sont effectuées sur le **istream**ou le **ostream**, il est retourné, de sorte qu'il peut être réutilisé dans une expression plus complexe.

Dans **main( )**, un nouveau type de **iostream**est utilisé : le **stringstream**(déclaré dans **<sstream>**). C'est une classe qui prend un **string**(qu'il peut créer à partir d'un tableau de **char**, comme c'est montré ici) et le transforme en un **iostream**. Dans l'exemple ci-dessus, cela signifie que les opérateurs de décalage peuvent être testés sans ouvrir un fichier ou taper des données en ligne de commande.

La forme montrée dans cet exemple pour l'inserteur et l'extracteur est standard. Si vous voulez créer ces opérateurs pour votre propre classe, copiez les signatures des fonctions ainsi que les types de retour ci-dessus et suivez la forme du corps.

12.4.1 - Conseils élémentaires

Murray Rob Murray, *C++ Strategies & Tactics*, Addison-Wesley, 1993, page 47. suggère ces directives pour choisir entre membres et non membres :

12.5 - Surcharge de l'affectation

Une source habituelle de confusion pour les programmeurs C++ novices est l'affectation. C'est sans doute parce que le signe **=**est une opération aussi fondamentale en programmation, que de copier un registre au niveau machine. De plus, le constructeur de copie (décrit dans le Chapitre 11) est aussi quelquefois invoqué quand le symbole **=**est utilisé :

```
                MyType b;
MyType a = b;
a = b;
```

Dans la seconde ligne, l'objet **a**est *défini*. Un nouvel objet est créé là où aucun n'existait auparavant. Parce que vous savez maintenant combien le compilateur C++ est sourcilleux pour ce qui concerne l'initialisation d'objets, vous savez qu'un constructeur doit toujours être appelé à l'endroit où un objet est défini. Mais quel constructeur? **a**est créé à partir d'un objet **MyType**existant (**b**, à droite du signe égal), de sorte qu'il n'y a qu'un seul choix: le constructeur de copie. Quand bien même un signe égal est invoqué, le constructeur de copie est appelé.

Dans la troisième ligne, les choses sont différentes. A gauche du signe égal, il y a un objet ayant déjà été initialisé précédemment. Il est clair que vous n'appelez pas un constructeur pour un objet ayant déjà été créé. Dans ce cas **MyType::operator=**est appelé pour un **a**, prenant pour argument tout qui se peut se trouver à droite. (Vous pouvez avoir plusieurs fonctions **operator=**prenant différents types d'arguments de droite.)

Ce comportement n'est pas réservé au constructeur de copie. A chaque fois que vous initialisez un objet en utilisant **=**au lieu de l'appel ordinaire sous forme de fonction d'un constructeur, le compilateur va chercher un constructeur qui accepte ce qui se trouve à droite.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                    //: C12:CopyingVsInitialization.cpp
class Fi {
public:
  Fi() {}
};

class Fee {
public:
  Fee(int) {}
  Fee(const Fi&) {}
};

int main() {
  Fee fee = 1; // Fee(int)
  Fi fi;
  Fee fum = fi; // Fee(Fi)
} ///:~
```

Lorsqu'on utilise le signe **=**, il est important de conserver à l'esprit cette distinction : Si l'objet n'a pas encore été créé, l'initialisation est nécessaire ; autrement l'opérateur d'affectation **operator=**est utilisé.

Il est même préférable d'éviter d'écrire du code qui utilise le **=**pour l'initialisation ; au lieu de cela, utilisez toujours la forme explicite du constructeur. Les deux constructions avec le signe **=**deviennent alors :

```
                    Fee fee(1);
Fee fum(fi);
```

De la sorte vous éviterez de semer la confusion chez vos lecteurs.

12.5.1 - Comportement de operator=

Dans **Integer.h**et **Byte.h**, vous avez vu que **operator=**ne peut être qu'une fonction membre. Il est intimement connecté à l'objet qui se trouve à gauche de ‘ **=**’. S'il était possible de définir **operator=**globalement, alors vous pourriez essayer de redéfinir le signe ‘ **=**’ prédéfini :

```
                        int operator=(int, MyType); // Global = interdit!
```

Le compilateur contourne entièrement ce problème en vous forçant à faire de **operator=**une fonction membre.

Lorsque vous créez un **operator=**, vous devez copier toute l'information nécessaire de l'objet de droite dans l'objet courant (C'est à dire, l'objet sur lequel l' **operator=**est appelé) pour accomplir tout ce que vous considérez une “affectation” pour votre classe. Pour des objets simples, c'est évident:

```
                        //: C12:SimpleAssignment.cpp
// Simple operator=()
#include <iostream>
using namespace std;

class Value {
  int a, b;
  float c;
public:
  Value(int aa = 0, int bb = 0, float cc = 0.0)
    : a(aa), b(bb), c(cc) {}
  Value& operator=(const Value& rv) {
    a = rv.a;
    b = rv.b;
    c = rv.c;
    return *this;
  }
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  friend ostream&
  operator<<(ostream& os, const Value& rv) {
    return os << "a = " << rv.a << ", b = "
      << rv.b << ", c = " << rv.c;
  }
};

int main() {
  Value a, b(1, 2, 3.3);
  cout << "a: " << a << endl;
  cout << "b: " << b << endl;
  a = b;
  cout << "a après affectation: " << a << endl;
} ///:~
```

Ici l'objet à la gauche du signe **=**copie tous les éléments de l'objet de droite, puis retourne une référence sur lui-même, ce qui permet la création d'une expression plus complète.

Cet exemple comporte une erreur commune. Lorsque vous affectez deux objets du même type, vous devez toujours commencer par vérifier l'auto-affectation : l'objet est-il affecté à lui-même ? Dans certains cas, comme celui-ci, c'est sans danger de réaliser cette affectation tout de même, mais si des changements sont faits à l'implémentation de la classe, cela peut faire une différence, et si vous ne le faites pas de manière routinière, vous pouvez l'oublier et provoquer des bogues difficiles à identifier.

Les pointeurs dans les classes

Que se passe-t-il si l'objet n'est pas si simple ? Par exemple, que se passe-t-il si l'objet contient des pointeurs vers d'autres objets ? Le fait de simplement copier un pointeur signifie que vous vous retrouverez avec deux objets pointant sur un même emplacement mémoire. Dans des situations de ce genre, il vous faut faire votre propre comptabilité.

Il y a deux approches communes à ce problème. La technique la plus simple consiste à recopier tout ce que le pointeur référence lorsque vous faites une affectation ou une construction par copie. C'est immédiat :

```
                        //: C12:CopyingWithPointers.cpp
// Résolution du problème d'aliasing des pointeurs
// en dupliquant ce qui est pointé durant
// l'affectation et la construction par copie.
#include "../require.h"
#include <string>
#include <iostream>
using namespace std;

class Dog {
  string nm;
public:
  Dog(const string& name) : nm(name) {
    cout << "Creation de Dog: " << *this << endl;
  }
  // Le constructeur de copie & l'operator=
  // synthétisés sont corrects.
  // création d'un Dog depuis un pointeur sur un Dog :
  Dog(const Dog* dp, const string& msg)
    : nm(dp->nm + msg) {
    cout << "Copie d'un dog " << *this << " depuis "
         << *dp << endl;
  }
  ~Dog() {
    cout << "Destruction du Dog: " << *this << endl;
  }
  void rename(const string& newName) {
    nm = newName;
    cout << "Dog renommé : " << *this << endl;
  }
  friend ostream&
  operator<<(ostream& os, const Dog& d) {
    return os << "[" << d.nm << "]";
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  }
};

class DogHouse {
  Dog* p;
  string houseName;
public:
  DogHouse(Dog* dog, const string& house)
   : p(dog), houseName(house) {}
  DogHouse(const DogHouse& dh)
    : p(new Dog(dh.p, " copie-construit")),
      houseName(dh.houseName
        + " copie-construit") {}
  DogHouse& operator=(const DogHouse& dh) {
    // vérification de l'auto-affectation :
    if(&dh != this) {
      p = new Dog(dh.p, " affecté");
      houseName = dh.houseName + " affecté";
    }
    return *this;
  }
  void renameHouse(const string& newName) {
    houseName = newName;
  }
  Dog* getDog() const { return p; }
  ~DogHouse() { delete p; }
  friend ostream&
  operator<<(ostream& os, const DogHouse& dh) {
    return os << "[" << dh.houseName
      << "] contient " << *dh.p;
  }
};

int main() {
  DogHouse fidos(new Dog("Fido"), "Niche de Fido");
  cout << fidos << endl;
  DogHouse fidos2 = fidos; // Construction par copie
  cout << fidos2 << endl;
  fidos2.getDog()->rename("Spot");
  fidos2.renameHouse("Niche de Spot");
  cout << fidos2 << endl;
  fidos = fidos2; // Affectation
  cout << fidos << endl;
  fidos.getDog()->rename("Max");
  fidos2.renameHouse("Niche de Max");
} ///:~
```

Dogest une classe simple qui ne contient qu'un **string**qui conserve le nom du chien. Cependant, vous serez généralement averti que quelque chose arrive à un **Dog**parce que les constructeurs et les destructeurs affichent des informations lorsqu'ils sont invoqués. Notez que le second constructeur est un peu comme un constructeur de copie excepté qu'il prend un pointeur sur un **Dog**au lieu d'une référence, et qu'il a un second argument qui est un message qui est concaténé à l'argument nom du **Dog**. Cela est utilisé pour aider à tracer le comportement du programme.

Vous pouvez voir qu'à chaque fois qu'une fonction membre affiche de l'information, elle n'accède pas à cette information directement mais au contraire envoie ***this**à **cout**. Ceci appelle ensuite **ostream operator<<**. C'est une bonne façon de faire parce que si vous voulez reformater la manière dont l'information de **Dog**est affichée (comme je l'ai fait en ajoutant '[' et ']') vous n'avez besoin de le faire qu'à un seul endroit.

Un **DogHouse**contient un **Dog***et illustre les quatre fonctions qu'il vous faudra toujours définir quand votre classe contient des pointeurs : tous les constructeurs ordinaires nécessaires, le constructeur de copie, **operator=**(définissez-le ou interdisez-le), et un destructeur. L'opérateur **operator=**vérifie l'auto-affectation, cela va sans dire, même si ce n'est pas strictement nécessaire ici. Ceci élimine presque totalement la possibilité que vous puissiez oublier cette vérification si vous *modifiez*le code de sorte que ce point devienne important.

Le comptage de références

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Dans l'exemple ci-dessus, le constructeur de copie et **operator=**font une nouvelle copie de ce que le pointeur référence, et le destructeur le libère. Toutefois, si votre objet nécessite beaucoup de mémoire ou bien un lourd travail d'initialisation, il se peut que vous ayez envie d'éviter cette copie. Une approche habituelle de ce problème est appelée *comptage de références*. Vous donnez de l'intelligence à l'objet pointé de sorte qu'il sache combien d'objets pointent sur lui. Alors construction par copie ou affectation signifient attacher un autre pointeur à un objet existant et incrémenter le compteur de références. La destruction signifie diminuer le compteur de référence et détruire l'objet si ce compteur atteint zéro.

Mais que se passe-t-il si vous voulez écrire dans l'objet(le **Dog**dans l'exemple précédent) ? Plus d'un objet peuvent utiliser ce **Dog**, de sorte que vous allez modifier le **Dog**de quelqu'un d'autre en même temps que le votre ce qui ne semble pas très amical. Pour résoudre ce problème d' “aliasing”, une technique supplémentaire appelée *copie à l'écriture*(copy-on-write) est utilisée. Avant d'écrire dans un bloc de mémoire, vous vous assurez que personne d'autre ne l'utilise. Si le compteur de références est supérieur à un, vous devez vous créer une copie personnelle de ce bloc avant d'écrire dessus, de la sorte vous ne piétinez les plates-bandes de personne. Voici un exemple simple de comptage de références et de copie à l'écriture :

```
                        //: C12:ReferenceCounting.cpp
// Comptage de références, copie à l'écriture
#include "../require.h"
#include <string>
#include <iostream>
using namespace std;

class Dog {
  string nm;
  int refcount;
  Dog(const string& name)
    : nm(name), refcount(1) {
    cout << "Creation du Dog: " << *this << endl;
  }
  // Empêcher l'affectation
  Dog& operator=(const Dog& rv);
public:
  // Les Dog ne peuvent être créés que sur le tas :
  static Dog* make(const string& name) {
    return new Dog(name);
  }
  Dog(const Dog& d)
    : nm(d.nm + " copie"), refcount(1) {
    cout << "Dog constructeur copie : "
         << *this << endl;
  }
  ~Dog() {
    cout << "Destruction du Dog : " << *this << endl;
  }
  void attach() {
    ++refcount;
    cout << "Attachement du Dog : " << *this << endl;
  }
  void detach() {
    require(refcount != 0);
    cout << "Detachement du Dog : " << *this << endl;
    // Détruit l'objet si personne ne s'en sert:
    if(--refcount == 0) delete this;
  }
  // Copy conditionnelle de ce 'Dog'.
  // Appel avant de modifier, affectation
  // pointeur résultant vers votre Dog*.
  Dog* unalias() {
    cout << "Unaliasing du Dog: " << *this << endl;
    // Pas de duplication si pas d'alias:
    if(refcount == 1) return this;
    --refcount;
    // Utiliser le constructeur de copie pour dupliquer :
    return new Dog(*this);
  }
  void rename(const string& newName) {
    nm = newName;
    cout << "Dog renommé : " << *this << endl;
  }
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  friend ostream&
  operator<<(ostream& os, const Dog& d) {
    return os << "[" << d.nm << "], rc = "
      << d.refcount;
  }
};

class DogHouse {
  Dog* p;
  string houseName;
public:
  DogHouse(Dog* dog, const string& house)
   : p(dog), houseName(house) {
    cout << "Creation du DogHouse : "<< *this << endl;
  }
  DogHouse(const DogHouse& dh)
    : p(dh.p),
      houseName("copie construit " +
        dh.houseName) {
    p->attach();
    cout << "DogHouse copie construction : "
         << *this << endl;
  }
  DogHouse& operator=(const DogHouse& dh) {
    // Vérification de l'auto-affectation:
    if(&dh != this) {
      houseName = dh.houseName + " affecté";
      // Nettoyer d'abord ce que vous utilisez:
      p->detach();
      p = dh.p; // Comme le constructeur de copie
      p->attach();
    }
    cout << "DogHouse operator= : "
         << *this << endl;
    return *this;
  }
  // Décrémente refcount, destruction ocnditionelle
  ~DogHouse() {
    cout << "DogHouse destructeur : "
         << *this << endl;
    p->detach();
  }
  void renameHouse(const string& newName) {
    houseName = newName;
  }
  void unalias() { p = p->unalias(); }
  // Copie à l'écriture. A chaque fois que vous modifiez
  // le contenu du pointeur vous devez d'abord
  // faire en sorte qu'il ne soit plus partagé :
  void renameDog(const string& newName) {
    unalias();
    p->rename(newName);
  }
  // ... ou quand vous permettez l'accès à quelqu'un d'autre:
  Dog* getDog() {
    unalias();
    return p;
  }
  friend ostream&
  operator<<(ostream& os, const DogHouse& dh) {
    return os << "[" << dh.houseName
      << "] contient " << *dh.p;
  }
};

int main() {
  DogHouse
    fidos(Dog::make("Fido"), "Niche de Fido"),
    spots(Dog::make("Spot"), "Niche de Spot");
  cout << "Avant copie construction" << endl;
  DogHouse bobs(fidos);
  cout << "Après copie construction de bobs" << endl;
  cout << "fidos:" << fidos << endl;
  cout << "spots:" << spots << endl;
  cout << "bobs:" << bobs << endl;
  cout << "Avant spots = fidos" << endl;
  spots = fidos;
  cout << "Après spots = fidos" << endl;
  cout << "spots:" << spots << endl;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    cout << "Avant l'auto affectation" << endl;
    bobs = bobs;
    cout << "Après l'auto affectation" << endl;
    cout << "bobs:" << bobs << endl;
    // Commentez les lignes suivantes :
    cout << "Avant rename("Bob")" << endl;
    bobs.getDog()->rename("Bob");
    cout << "Après rename("Bob")" << endl;
  } ///:~
```

La classe **Dog**est l'objet pointé par un **DogHouse**. Il contient un compteur de références et des fonctions pour contrôler et lire le compteur de références. Un constructeur de copie est disponible de sorte que vous pouvez fabriquer un nouveau **Dog**à partir d'un autre existant déjà.

La fonction **attach( )**incrémente le compteur de référence d'un **Dog**pour indiquer qu'un autre objet l'utilise. **detach( )**décrémente le compteur de références. Si le compteur de références arrive à zéro, alors personne ne l'utilise plus, alors la fonction membre détruit son propre objet en disant **delete this**.

Avant de faire toute modification (comme renommer un **Dog**), vous devez vous assurer que vous ne modifiez pas un **Dog**qu'un autre objet utilise. Vous faites cela en appelant **DogHouse::unalias( )**, qui à son tour appelle **Dog::unalias( )**. Cette dernière fonction retournera le pointeur sur **Dog**existant si le compteur de référence vaut un (signifiant que personne d'autre ne pointe sur ce **Dog**), mais il dupliquera le **Dog**si le compteur de références est supérieur à un.

Le constructeur de copie, au lieu de créer sa propre zone mémoire, affecte **Dog**au **Dog**de l'objet source. Ensuite, parce qu'il y a maintenant un objet supplémentaire utilisant ce bloc de mémoire, il incrémente le compte de références en appelant **Dog::attach( )**.

L'opérateur **operator=**traite un objet ayant déjà été créé et qui se trouve à gauche de **=**, de sorte qu'il doit déjà nettoyer par un appel à **detach( )**pour ce **Dog**, qui détruira l'ancien **Dog**si personne ne l'utilise. Ensuite **operator=**répète le comportement du constructeur de copie. Notez qu'il vérifie d'abord si vous affectez l'objet à lui-même.

Le destructeur appelle **detach( )**pour détruire conditionelement le **Dog**.

Pour implémenter la copie à l'écriture, vous devez contrôler toutes les actions qui écrivent sur votre bloc de mémoire. Par exemple, la fonction membre **renameDog( )**vous permet de changer les valeurs dans le bloc de mémoire. Mais d'abord, il utilise **unalias( )**pour empêcher la modification d'un **Dog**'aliasé' (un **Dog**ayant plus d'un **DogHouse**pointant sur lui). Et si vous avez besoin de générer un pointeur vers un **Dog**depuis un **DogHouse**, vous appelez **unalias( )**sur ce pointeur d'abord.

main( )teste les différentes fonctions qui fonctionnent correctement pour implémenter le comptage de références : le constructeur, le constructeur de copie, **operator=**, et le destructeur. Il teste également la copie à l'écriture en appelant **renameDog( )**.

Voici la sortie (après une petite mise en forme):

```
                        Creation du Dog : [Fido], rc = 1
Creation du DogHouse : [FidoHouse]
  contient [Fido], rc = 1
Creation du Dog: [Spot], rc = 1
Creation du DogHouse : [SpotHouse]
  contient [Spot], rc = 1
Avant copie construction
Attachement du Dog : [Fido], rc = 2
DogHouse copie construction :
  [copie construit FidoHouse]
    contient [Fido], rc = 2
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
Après copie construction bobs
fidos:[FidoHouse] contient [Fido], rc = 2
spots:[SpotHouse] contient [Spot], rc = 1
bobs:[copie construit FidoHouse]
  contient [Fido], rc = 2
Avant spots = fidos
Detachement du Dog : [Spot], rc = 1
Destruction du Dog : [Spot], rc = 0
Attachement du Dog : [Fido], rc = 3
DogHouse operator= : [FidoHouse affecté]
  contient [Fido], rc = 3
Après spots = fidos
spots:[FidoHouse affecté] contient [Fido],rc = 3
Avant auto affectation
DogHouse operator= : [copie construit FidoHouse]
  contient [Fido], rc = 3
Après auto affectation
bobs :[copie construit FidoHouse]
  contient [Fido], rc = 3
avant rename("Bob")
après rename("Bob")
DogHouse destruction : [copie construit FidoHouse]
  contient [Fido], rc = 3
Detachement de Dog : [Fido], rc = 3
DogHouse destruction : [FidoHouse affecté]
  contient [Fido], rc = 2
Detachement du Dog : [Fido], rc = 2
DogHouse destruction : [FidoHouse]
  contient [Fido], rc = 1
Detachement du Dog : [Fido], rc = 1
Destruction du Dog: [Fido], rc = 0
```

En étudiant la sortie, en tracant dans le code source, et en vous livrant à des expériences à partir de ce programme, vous approfonderez votre compréhension de ces techniques.

creation automatique de l'operator=

Parce qu'affecter un objet à partir d'un autre objet *du même type*est une opération que la plupart des gens s'attendent à être possible, le compilateur créera automatiquement un **type::operator=(type)**si vous n'en créez pas un vous-même. Le comportement de cet opérateur imite celui du constructeur de copie créé automatiquement ; Si la classe contient des objets (ou dérive d'une autre classe), l'opérateur **operator=**pour ces objets est appelé récursvement. On appelle cela *affectation membre à membre*. Par exemple,

```
                          //: C12:AutomaticOperatorEquals.cpp
#include <iostream>
using namespace std;

class Cargo {
public:
  Cargo& operator=(const Cargo&) {
    cout << "dans Cargo::operator=()" << endl;
    return *this;
  }
};

class Truck {
  Cargo b;
};

int main() {
  Truck a, b;
  a = b; // Affiche: "dans Cargo::operator=()"
} ///:~
```

L'opérateur **operator=**généré automatiquement pour **Truck**appelle **Cargo::operator=**.

En général, vous ne voudrez pas laisser le compilateur faire cela pour vous. Avec des classes de toute complexité (spécialement si elles contiennent des pointeurs !) vous voudrez créer explicitement un **operator=**. Si vous ne

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

voulez réellement pas que les gens effectuent des affectations, déclarez **operator=**comme une fonction **private**. (Vous n'avez pas besoin de le définir à moins que vous ne l'utilisiez dans la classe).

12.6 - Conversion de type automatique

En C et en C++, si le compilateur voit une expression ou un appel de fonction utilisant un type n'est pas exactement celui qu'il attend, il peut souvent effectuer une conversion de type automatique du type qu'il a vers le type qu'il veut. En C++, vous pouvez provoquer ce même effet pour les types définis par l'utilisateur en définissant des fonctions de conversion automatique. Ces fonctions se présentent sous deux formes : un type particulier de constructeur et un opérateur surchargé.

12.6.1 - Conversion par constructeur

Si vous définissez un constructeur qui prend comme seul argument un objet (ou une référence) d'un autre type, ce constructeur permet au compilateur d'effectuer une conversion de type automatique. Par exemple

```
                        //: C12:AutomaticTypeConversion.cpp
// Conversion de type par constructeur
class One {
public:
  One() {}
};

class Two {
public:
  Two(const One&) {}
};

void f(Two) {}

int main() {
  One one;
  f(one); // Veut un Two, utilise un One
} ///:~
```

Quand le compilateur voit **f( )**appelé avec un objet de type **One**, il regarde la déclaration de **f( )**et note qu'elle attend un **Two**. Ensuite il regarde s'il y a un moyen d'obtenir un **Two**partant d'un **One**, et trouve le constructeur **Two::Two(One)**, qu'il appelle silencieusement. L'objet **Two**résultant est passé à **f( )**.

Dans ce cas, la conversion automatique de type vous a épargné la peine de définir deux versions surchargées de **f( )**. Cependant, le coût est l'appel caché du constructeur de **Two**, ce qui peut importer si vous êtes concerné par l'efficacité des appels de **f( )**.

Eviter la conversion par constructeur

Il y a des fois où la conversion automatique de type au travers du constructeur peut poser problèmes. Pour la désactiver, modifiez le constructeur en le préfaçant avec le mot clé **explicit**(qui ne fonctionne qu'avec les constructeurs). Utilisé pour modifier le constructeur de la classe **Two**dans l'exemple qui suit :

```
                        //: C12:ExplicitKeyword.cpp
// Using the "explicit" keyword
class One {
public:
  One() {}
};

class Two {
public:
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  explicit Two(const One&) {}
};

void f(Two) {}

int main() {
  One one;
//!  f(one); // conversion automatique non autorisée
  f(Two(one)); // OK -- l'utilisateur effectue la conversion
} ///:~
```

En rendant le constructeur de **Two**explicite, le compilateur est prévenu qu'il ne doit pas effectuer la moindre conversion automatique en utilisant ce constructeur particulier (les autres constructeurs non- **explicit**dans cette classe peuvent toujours effectuer des conversions automatiques). Si l'utilisateur veut qu'une conversion survienne, le code doit être écrit. Dans le code ci-dessous, **f(Two(one))**crée un objet temporaire de type **Two**à partir de **one**, exactement comme le faisait le compilateur dans la version précédente.

12.6.2 - Conversion par opérateur

La seconde manière de produire une conversion de type automatique passe par la surcharge d'opérateur. Vous pouvez créer une fonction membre qui prend le type courant et le convertit dans le type désiré en utilisant le mot clé **operator**suivi par le type dans lequel vous voulez convertir. Cette forme de surcharge d'opérateur est unique parce que vous ne semblez pas spécifier de type de retour – Le type de retour est le *nom*de l'opérateur que vous surchargez. En voici un exemple :

```
                          //: C12:OperatorOverloadingConversion.cpp
class Three {
  int i;
public:
  Three(int ii = 0, int = 0) : i(ii) {}
};

class Four {
  int x;
public:
  Four(int xx) : x(xx) {}
  operator Three() const { return Three(x); }
};

void g(Three) {}

int main() {
  Four four(1);
  g(four);
  g(1);  // Appelle Three(1,0)
} ///:~
```

Avec la technique basée sur le constructeur, la classe de destination effectue la conversion, mais avec les opérateurs, c'est la classe source qui effectue la conversion. La valeur de la technique baséee sur le constructeur est que vous pouvez ajouter un moyen de conversion à un système existant quand vous créez une nouvelle classe. Cependant, la création d'un constructeur mono-argument définit *toujours*une conversionde type automatique (ainsi que s'il y a plus d'un argument, si le reste des arguments disposent de valeurs par défaut), qui peut ne pas être ce que vous voulez (dans ce cas, vous désactivez la conversion en utilisant **explicit**). De plus, il n'y a aucun moyen d'utiliser un constructeur de conversion d'un type utilisateur à un type intégré ; Ce n'est possible qu'avec la surcharge d'opérateur.

Réflexivité

L'une des raisons les plus commodes à l'utilisation d'opérateurs surchargés globaux par rapport aux opérateurs membres est que dans les versions globales, la conversion de type peut s'appliquer à l'un ou l'autre opérande, alors qu'avec les membres objet, l'opérande de gauche doit être du type adéquat. Si vous voulez convertir les deux

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

opérandes, les versions globales peuvent économiser beaucoup de codage. Voici un petit exemple :

```
                        //: C12:ReflexivityInOverloading.cpp
class Number {
  int i;
public:
  Number(int ii = 0) : i(ii) {}
  const Number
  operator+(const Number& n) const {
    return Number(i + n.i);
  }
  friend const Number
    operator-(const Number&, const Number&);
};

const Number
  operator-(const Number& n1,
            const Number& n2) {
    return Number(n1.i - n2.i);
}

int main() {
  Number a(47), b(11);
  a + b; // OK
  a + 1; // le deuxième argument est converti en Number
//! 1 + a; // Mauvais ! le premier argument n'est pas de type Number
  a - b; // OK
  a - 1; // Le second argument est converti en Number
  1 - a; // Le premier argument est converti en Number
} ///:~
```

La classe **Number**a à la fois un **operator+**et un **operator–**ami (ndt **friend**). Comme il y a un constructeur qui prend un seul argument **int**, un **int**peut être automatiquement converti en un **Number**, mais uniquement dans les bonnes conditions. Dans **main( )**, vous pouvez remarquer qu'ajouter un **Number**à un autre **Number**fonctionne correctement parce que cela correspond exactement à l'opérateur surchargé. De même, qund le compilateur voit un **Number**suivi d'un **+**et d'un **int**, il peut trouver la fonction membre **Number::operator+**et convertir l'argument **int**en un **Number**en utilisant le constructeur. Mais quand il voit un **int**, un **+**et un **Number**, il ne sait pas que faire parce que tout ce qu'il a c'est **Number::operator+**qui exoge que l'opérande de gauche soit déjà un objet **Number**. Le compilateur nous reporte donc une erreur.

Avec l' **operator– friend**, les choses sont différentes. Le compilateur doit remplir les arguments comme il peut ; il n'est pas restreint à avoir un **Number**comme argument de gauche. Auusi, si il voit

```
                        1 &#8211; a
```

il peut convertir le premier argument en **Number**en utilisant le constructeur.

Vous voulez parfois pouvoir restreindre l'utilisation de vos opérateurs en les rendant membres. Par exemple, quand vous multipliez une matrice par un vecteur, le vecteur doit aller à droite. Mais si vous voulez que vos opérateurs puissent convertir l'un ou l'autre argument, transformez l'opérateur en fonction amie.

Heureusement, le compilateur ne va pas prendre **1 – 1**et convertir chaque argument en objet **Number**et ensuite appeler l' **operator–**. Cela voudrait dire que le code C existant pourrait soudainement commencer à fonctionner différemment. Le compilateur sélectionne la possibilité “la plus simple” en premier, ce qui est l'opérateur intégré pour l'expression **1 – 1**.

12.6.3 - Exemple de conversion de type

Un exemple dans lequel la conversion automatique de type est extrèmement utile survient avec toute classe qui

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

encapsule des chaînes de caractères (dans ce cas, nous allons juste implémenter la classe en utilisant la classe Standard C++ **string**parce que c'est simple). Sans conversion automatique de type, si vous voulez utiliser toutes les fonctions de chaîne de bibliothèque standard C, vous devez créer une fonction membre pour chacune, comme ceci :

```
                        //: C12:Strings1.cpp
// Pas de conversion de type automatique
#include "../require.h"
#include <cstring>
#include <cstdlib>
#include <string>
using namespace std;

class Stringc {
  string s;
public:
  Stringc(const string& str = "") : s(str) {}
  int strcmp(const Stringc& S) const {
    return ::strcmp(s.c_str(), S.s.c_str());
  }
  // ... etc., pour chaque fonction dans string.h
};

int main() {
  Stringc s1("hello"), s2("there");
  s1.strcmp(s2);
} ///:~
```

Ici, seule la fonction strcmp( ) a été créé, mais vous auriez à créer un fonction correspondant à chacune de celles dans **<cstring>**qui pourrait s'avérer utile. Par chance, vous pouvez fournir une conversion automatique de type permettant d'accéder à toutes les fonction dans **<cstring>**:

```
                        //: C12:Strings2.cpp
// Avec conversion de type automatique
#include "../require.h"
#include <cstring>
#include <cstdlib>
#include <string>
using namespace std;

class Stringc {
  string s;
public:
  Stringc(const string& str = "") : s(str) {}
  operator const char*() const {
    return s.c_str();
  }
};

int main() {
  Stringc s1("hello"), s2("there");
  strcmp(s1, s2); // fonction Standard C
  strspn(s1, s2); // n'importe quelle fonction de chaîne !
} ///:~
```

Maintenant, toute fonction qui prend un argument **char***peut aussi prendre un argument **Stringc**parce que le compilateur sait comment créer un **char***depuis un **Stringc**.

12.6.4 - Les pièges de la conversion de type automatique

Du fait que le compilateur doit choisir comment effectuer silencieusement une conversion de type, il peut être troublé si vous ne concevez pas correctement vos conversions. Une situation simple et évidente est celle qui survient avec une classe **X**qui peut se convertir en un objet de classe **Y**avec un **operator Y( )**. Si la classe **Y**dispose d'un constructeur prenant un seul argument de type **X**, cela représente la même conversion. Le

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

compilateur a maintenant deux moyens d'aller de **X**à **Y**, donc il va générer une erreur d'ambiguité quand cette conversion se produit :

```
                        //: C12:TypeConversionAmbiguity.cpp
class Orange; // Class declaration

class Apple {
public:
  operator Orange() const; // Convertit Apple en Orange
};

class Orange {
public:
  Orange(Apple); // Convertit Apple en Orange
};

void f(Orange) {}

int main() {
  Apple a;
//! f(a); // Erreur : conversion ambiguë
} ///:~
```

La solution évidente est de ne pas le faire. Fournissez simplement un moyen unique de convertir automatiquement un type en un autre.

Un problème plus difficile à résoudre apparait parfois quand vous fournissez des conversions automatiques vers plusieurs types. C'est parfois appelé *fan-out*:

```
                        //: C12:TypeConversionFanout.cpp
class Orange {};
class Pear {};

class Apple {
public:
  operator Orange() const;
  operator Pear() const;
};

// Overloaded eat():
void eat(Orange);
void eat(Pear);

int main() {
  Apple c;
//! eat(c);
  // Erreur : Apple -> Orange ou Apple -> Pear ???
} ///:~
```

La classe **Apple**peut être convertie automatiquement aussi bien en **Orange**qu'en **Pear**. Ce qui est incidieux ici est qu'il n'y a pas de problème jusqu'à ce que quelqu'un vienne innocemment créer deux versions surchargées de **eat( )**. (Avec une seule version, le code qui se trouve dans **main( )**fonctionne bien.)

Encore une fois la solution – et le mot d'ordre général sur la conversion automatique de type – est de ne fournir qu'une seule conversion automatique d'un type vers un autre. Vous pouvez avoir des conversions vers d'autres types ; elles ne devraient simplement pas être *automatiques*. Vous pouvez créer des appels de fonction explicites avec des noms commes **makeA( )**et **makeB( )**.

Les activités cachées

La conversion automatique de type peut introduire des activités plus fondamentales que vous pouvez l'envisager. Comme petite énigme, voyez cette modification de **CopyingVsInitialization.cpp**:

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                          //: C12:CopyingVsInitialization2.cpp
class Fi {};

class Fee {
public:
  Fee(int) {}
  Fee(const Fi&) {}
};

class Fo {
  int i;
public:
  Fo(int x = 0) : i(x) {}
  operator Fee() const { return Fee(i); }
};

int main() {
  Fo fo;
  Fee fee = fo;
} ///:~
```

Il n'y a pas de constructeur pour créer **Fee fee**à partir d'un objet **Fo**. Cependant, **Fo**a une conversion automatique de type en **Fee**. Il n'y a pas de constructeur par recopie pour créer un **Fee**à partir d'un **Fee**, mais c'est l'une des fonctions spéciales que le compilateur peut créer pour vous. (le constructeur par défaut, le constructeur par recopie, l' **operator=**et le destructeur peuvent être automatiquement synthétisés par le compilateur.) Ainsi, pour la déclaration relativement innocente

```
                          Fee fee = fo;
```

l'opérateur de conversion automatique de type est appelé, et un constructeur par recopie est créé.

Utilisez la conversion automatique de type avec prudence. Comme pour toutes les surcharges d'opérateur, c'est excellent quand cela réduit le travail de codage de manière significative, mais cela ne vaut généralement pas la peine de l'utiliser gratuitement.

12.7 - Résumé

L'unique raison de la surcharge d'opérateur réside dans les situations où elle rend la vie plus facile. Il n'y a rien de particulièrement magique à ce propos ; les opérateurs surchargés ne sont que des fonctions aux noms amusants, et les appels de fonctions sont faits pour vous par le compilateur lorsqu'il détecte la construction syntaxique associée. Mais si la surcharge d'opérateurs ne procure pas un bénéfice appréciable à vous (le créateur de la classe) ou à l'utilisateur, ne compliquez pas les choses en en l'ajoutant.

12.8 - Exercices

Les solutions aux exercices peuvent être trouvées dans le document électronique *Le Guide annoté des solutions Penser en C++*, disponible à un coût modeste depuis www.BruceEckel.com.

1 Créer une classe simple avec un opérateur surchargé **operator++**. Essayez d'appeler cet opérateur en forme préfixée et postfixée et voyez quel type d'avertissement vous recevez du compilateur.
2 Créez une classe simple contenant un **int**et surcharger **operator+**comme fonction membre. Prévoir aussi une fonction membre **print( )**qui prend un **ostream&**comme argument et écrit dans cet **ostream&**. Testez votre classe pour montrer qu'elle fonctionne correctement.
3 Ajouter un **operator-**binaire à l'exercice 2 comme fonction membre. Démontrez que vous pouvez utiliser vos objets dans des expressions complexes telles que **a + b – c**.
4 Ajoutez un **operator++**et un **operator--**à l'exercice 2, à la fois en version préfixée et postfixée, de sorte qu'ils

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

retournent l'objet incrémenté ou décrémenté. Assurez vous que les versions postfixées retournent les valeurs correctes.

5 Modifiez les opérateurs d'incrémentation et de décrémentation dans l'exercice 4 de sorte que les versions préfixées retournent une référencenon- **const**et les versions postfixées retournent un objet **const**. Montrez qu'ils fonctionnent correctement et expliquez pourquoi il serait fait ainsi dans la pratique.

6 Changez la fonction **print( )**dans l'exercice 2 de sorte que ce soit **operator<<**surchargé comme dans **IostreamOperatorOverloading.cpp**.

7 Modifiez l'exercice 3 de façon que **operator+**et **operator-**soient des fonctions non membres. Démontrez qu'elles fonctionnent encore correctement.

8 Ajoutez l'opérateur unaire **operator-**à l'exercice 2 et démontrez qu'il fonctionne correctement.

9 Créez une classe qui contient un unique **private char**. Surchargez les opérateurs **<<**et **>>**d'iostream (comme dans **IostreamOperatorOverloading.cpp**) et testez les. Vous pouvez les tester avec **fstreams**, **stringstream**s, et **cin**et **cout**.

10 Déterminez la valeur constante factice que votre compilateur passe pour les opérateurs postfixés **operator++**et **operator--**.

11 Ecrivez une classe **Number**qui comporte une donnée **double**, et ajoutez des opérateurs surchargés pour **+, –, *, /**, et l'affectation. Choisissez les valeurs de retour de ces fonctions de sorte que les expressions puissent être chaînées entre elles, et pour une meilleure efficacité. Ecrivez un opérateur de conversion automatique **operator double( )** .

12 Modifiez l'exercice 11 de sorte que *l'optimisation de la valeur de retour*soit utilisée, si vous ne l'avez pas déjà fait.

13 Créez une classe qui contient un pointeur, et démontrez que si vous permettez au compilateur de synthétiser l' **operator=**le résultat de l'utilisation de cet opérateur sera des pointeurs qui pointent la même zone mémoire. Résolvez maintenant le problème en écrivant votre propre **operator=**et démontrez qu'il corrige ce défaut. Assurez vous de vérifier l'auto-affectation et traitez ce cas correctement.

14 Ecrivez une classe appelée **Bird**qui contient un membre **string**et un **static int**. Dans le constructeur par défaut, utilisez le **int**pour générer automatiquement un identificateur que vous construisez dans le **string**, en association avec le nom de la classe (**Bird #1**, **Bird #2**, etc.). Ajoutez un **operator<<**pour **ostream**s pour afficher les objets **Bird**. Ecrivez un opérateur d'affectation **operator=**et un constructeur de recopie. Dans **main( )**, vérifiez que tout marche correctement.

15 Ecrivez une classe nommée **BirdHouse**qui contient un objet, un pointeur, et une référence pour la classe **Bird**de l'exercice 14. Le constructeur devrait prendre les trois **Bird**s comme arguments. Ajoutez un **operator<<**pour **ostream**s pour **BirdHouse**. Interdisezl'opérateur affectation **operator=**et le constructeur de recopie. Dans **main( )**, vérifiez que tout marche correctement.

16 Ajoutez une donnée membre **int**à la fois à **Bird**et à **BirdHouse**dans l'exercice 15. Ajoutez des opérateurs membres **+**, **–**, *****, et **/**qui utilise les membres **int**pour effectuer les opérations sur les membres respectifs. Vérifiez que ceux ci fonctionnent.

17 Refaire l'exercice 16 en utilisant des opérateurs non membres.

18 Ajoutez un **operator--**à **SmartPointer.cpp**et **NestedSmartPointer.cpp**.

19 Modifiez **CopyingVsInitialization.cpp**de façon que tous les constructeurs affichent un message qui vous dit ce qui se passe. Vérifiez maintenant que les deux formes d'appel au constructeur de recopie (l'affectation et la forme parenthésée) sont équivalentes.

20 Essayez de créer un opérateur non membre **operator=**pour une classe et voyez quel genre de messages vous recevez du compilateur.

21 Créez une classe avec un opérateur d'affectationqui a un second argument, un **string**qui a une valeur par défaut qui dit “ op=call.” Créez une fonction qui affecteun objet de votre classe à un autreet montrez que votre opérateur d'affectationest appelé correctement.

22 Dans **CopyingWithPointers.cpp**, enlevez l' **operator=**dans **DogHouse**et montrez que l'opérateur **operator=**synthétisé par le compilateur copie correctement le **string**mais ne fait qu'un alias du pointeur de **Dog**.

23 Dans **ReferenceCounting.cpp**, ajoutez un **static int**et un **int**ordinaire comme données membres à **Dog**et à **DogHouse**. Dans tous les constructeurs pour les deux classes incrémentez le **static int**et affectez le résultat à l' **int**ordinaire pour garder une trace du nombre d'objets qui ont été créés. Faites les modifications nécessaires de sorte que toutes les instructions d'affichage donneront les identificateurs **int**des objets impliqués.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

24 Créez une classe contenant un **string**comme donnée membre. Initialisez le **string**dans le constructeur, mais ne créez pas de constructeur de recopie ni d'opérateur affectation **operator=**. Faites une seconde classe qui a un membre objet de votre première classe ; ne créez pas non plus de constructeur de recopie, ni d' **operator=**pour cette classe. Démontrez que le constructeur de recopie et l' **operator=**sont correctement synthétisés par le compilateur.
25 Combinez les classes dans **OverloadingUnaryOperators.cpp**et **Integer.cpp**.
26 Modifiez **PointerToMemberOperator.cpp**en ajoutant deux nouvelles fonctions membres à **Dog**qui ne prennent aucun argument et qui retournent **void**. Créez et testez un **operator->***surchargé qui fonctionne avec vos deux nouvelles fonctions.
27 Ajoutez un **operator->***à **NestedSmartPointer.cpp**.
28 Créez deux classes, **Apple**et **Orange**. Dans **Apple**, créez un constructeur qui prend un **Orange**comme argument. Créez une fonction qui prend un **Apple**et appelez cette fonction avec un **Orange**pour montrer que cela fonctionne. Rendez maintenant le constructeur **Apple explicite**pour démontrer que la conversion automatique est de la sorte empêchée. Modifiez l'appel à votre fonction de sorte que la conversion soit faite explicitement et ainsi fonctionne.
29 Ajouter un **operator***global à **ReflexivityInOverloading.cpp**et démontrez qu'il est réflexif.
30 Créez deux classes et créez un **operator+**et les fonctions de conversion de sorte que l'addition soit réflexive pour les deux classes.
31 Corrigez **TypeConversionFanout.cpp**en créant une fonction explicite à appeler pour réaliser la conversion de type, à la place de l'un des opérateurs de conversion automatique.
32 Ecrivez un code simple qui utilise les opérateurs **+**, **–**, *****, et **/**pour les **double**s. Essayez de vous représenter comment votre compilateur génère le code assembleur et regardez l'assembleur généré pour découvrir et expliquer ce qui se passe sous le capot.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

13 - Création d'Objets Dynamiques

Parfois vous connaissez l'exacte quantité, le type, et la durée de vie des objets dans votre programme. Mais pas toujours.

Combien d'avions un système de contrôle du traffic aérien aura-t-il à gérer? Combien de formes différentes utilisera un système de DAO? Combien y aura-t-il de noeuds dans un réseau?

Pour résoudre ce problème général de programmation, il est essentiel d'être capable de créer et de détruire les objets en temps réel (en cours d'exécution). Bien entendu, C a toujours proposé les fonctions d' *allocation de mémoire dynamique* **malloc( )**et **free( )**(ainsi que quelques variantes de **malloc( )**) qui allouent de la mémoire sur le *tas*(également appelé *espace de stockage libre*) au moment de l'éxecution.

Cependant, ceci ne marchera tout simplement pas en C++. Le constructeur ne vous permet pas de manipuler l'adresse de la mémoire à initialiser ,et pour une bonne raison. Si vous pouviez faire cela, vous pourriez:

1 Oublier. Dans ce cas l'initialisation des objets en C++ ne serait pas garantie.
2 Accidentellement faire quelque chose à l'objet avant de l'initialiser, espérant que la chose attendue se produira.
3 Installer un objet d'une taille inadéquate.

Et bien sûr, même si vous avez tout fait correctement, toute personne qui modifie votre programme est susceptible de faire les mêmes erreurs. Une initialisation incorrecte est responsable d'une grande part des problèmes de programmation, de sorte qu'il est spécialement important de garantir des appels de constructeurs pour les objets créés sur le tas.

Aussi comment C++ réussit-il à garantir une initialisation correcte ainsi qu'un nettoyage, tout en vous permettant de créer des objets dynamiquement sur le tas?

La réponse est: en apportant la création d'objet dynamique au coeur du langage. **malloc( )**et **free( )**sont des fonctions de bibliothèque, elles se trouvent ainsi en dehors du contrôle direct du compilateur. Toutefois, si vous avez un *opérateur*pour réaliser l'action combinée d'allocation dynamique et et d'initialisation et un autre opérateur pour accomplir l'action combinée de nettoyage et de restitution de mémoire, le compilateur peut encore guarantir que les constructeurs et les destructeurs seront appelés pour tous les objets.

Dans ce chapitre, vous apprendrez comment les **new**et **delete**de C++ résolvent élégamment ce problème en créant des objets sur le tas en toute sécurité.

13.1 - Création d'objets

Lorsqu'un objet C++ est créé, deux événements ont lieu:

1 Un espace mémoire est alloué pour l'objet.
2 Le constructeur est appelé pour initialiser cette zone.

Maintenant vous devriez croire que la seconde étape a *toujours*lieu. C++ l'impose parce que les objets non initialisés sont une source majeure de bogues de programmes. Où et comment l'objet est créé n'a aucune importance – le constructeur est toujours appelé.

La première étape, cependant, peut se passer de plusieurs manières, ou à différents moments :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

1 L'espace peut être alloué avant que le programme commence, dans la zone de stockage statique. Ce stockage existe pour toute la durée du programme.
2 L'allocation peut être créée sur la pile à chaque fois qu'un point d'exécution particulier est atteint (une accolade ouvrante). Ce stockage est libéré automatiquement au point d'exécution complémentaire (l'accolade fermante). Ces opérations d'allocation sur la pile font partie de la logique câblée du jeu d'instructions du processeur et sont très efficaces. Toutefois, il vous faut connaître exactement de combien de variables vous aurez besoin quand vous êtes en train d'écrire le programme de façon que le compilateur puisse générer le code adéquat.
3 L'espace peut être alloué depuis un segment de mémoire appelé le 'tas' (aussi connu comme " *free store*"). Ceci est appelé l'allocation dynamique de la mémoire. Pour allouer cette mémoire, une fonction est appelée au moment de l'exécution ; cela signifie que vous pouvez décider à n'importe quel moment que vous voulez de la mémoire et combien vous en voulez. Vous êtes également responsable de déterminer quand libérer la mémoire, ce qui signifie que la durée de vie de cette mémoire peut être aussi longue que vous le désirez – ce n'est pas déterminé par la portée.

Souvent ces trois régions sont placées dans un seul segment contigu de mémoire physique : la zone statique, la pile, et le tas (dans un ordre déterminé par l'auteur du compilateur). Néammoins, il n'y a pas de règles. La pile peut être dans un endroit particulier, et le tas peut être implémenté en faisant des appels à des tronçons de mémoire depuis le système d'exploitation. En tant que programmeur, ces choses sont normallement protégées de vous, aussi la seule chose à laquelle vous avez besoin de penser est que la mémoire est là quand vous la demandez.

13.1.1 - L'approche du C au tas

Pour allouer de la mémoire dynamiquement au moment de l'exécution, C fournit des fonctions dans sa bibliothèque standard : **malloc( )**et ses variantes **calloc( )**et **realloc( )**pour produire de la mémoire du tas, et **free( )**pour rendre la mémoire au tas. Ces fonctions sont pragmatiques mais primitives et nécessitent de la compréhension et du soin de la part du programmeur. Pour créer une instance d'une classe sur le tas en utilisant les fonctions de mémoire dynamique du C, il vous faudrait faire quelque chose comme ceci :

```
                        //: C13:MallocClass.cpp
// Malloc avec des classes
// Ce qu'il vous faudrait faire en l'absence de "new"
#include "../require.h"
#include <cstdlib> // malloc() & free()
#include <cstring> // memset()
#include <iostream>
using namespace std;

class Obj {
  int i, j, k;
  enum { sz = 100 };
  char buf[sz];
public:
  void initialize() { // impossible d'utiliser un constructeur
    cout << "initialisation de Obj" << endl;
    i = j = k = 0;
    memset(buf, 0, sz);
  }
  void destroy() const { // impossible d'utiliser un destructeur
    cout << "destruction de Obj" << endl;
  }
};

int main() {
  Obj* obj = (Obj*)malloc(sizeof(Obj));
  require(obj != 0);
  obj->initialize();
  // ... un peu plus tard :
  obj->destroy();
  free(obj);
} ///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Vous pouvez voir l'utilisation de **malloc( )**pour créer un stockage pour l'objet à la ligne :

```
Obj* obj = (Obj*)malloc(sizeof(Obj));
```

Ici, l'utilisateur doit déterminer la taille de l'objet (une occasion de se tromper). **malloc( )**retourne un **void***parce qu'il produit un fragment de mémoire, pas un objet. C++ ne permet pas à un **void***d'être affecté à tout autre pointeur, il doit donc être transtypé.

Parce que **malloc( )**peut échouer à trouver de la mémoire (auquel cas elle retourne zéro), vous devez vérifier le pointeur retourné pour vous assurer que l'opération a été un succès.

Mais le pire problème est cette ligne :

```
Obj->initialize();
```

A supposer que les utilisateurs aient tout fait correctement jusque là, ils doivent se souvenir d'initialiser l'objet avant qu'il soit utilisé. Notez qu'un constructeur n'a pas été utilisé parce qu'un constructeur ne peut pas être appelé explicitement Il y a une syntaxe spéciale appelée *placement new*qui vous permet d'appeler un constructeur pour une zone de mémoire pré-allouée. Ceci est introduit plus tard dans le chapitre. – il est appelé pour vous par le compilateur quand un objet est créé. Le problème ici est que l'utilisateur a maintenant la possibilité d'oublier d'accomplir l'initialisation avant que l'objet soit utilisé, réintroduisant ainsi une source majeure d'erreurs.

Il s'avère également que de nombreux programmeurs semblent trouver les fonctions de mémoire dynamique du C trop peu claires et compliquées ; il n'est pas rare de trouver des programmeurs C qui utilisent des machines à mémoire virtuelle allouant de très grands tableaux de variables dans la zone statique pour éviter de réfléchir à l'allocation dynamique. Parce que C++ essaie de rendre l'utilisation de bibliothèques sûre et sans effort pour le programmeur occasionnel, l'approche de C à la mémoire dynamique n'est pas acceptable.

13.1.2 - l'operateur new

La solution en C++ est de combiner toutes les actions nécessaires pour créer un objet en un unique opérateur appelé **new**. Quand vous créez un objet avec **new**(en utilisant une *expression new*), il alloue suffisamment d'espace sur le tas pour contenir l'objet et appelle le constructeur pour ce stockage. Ainsi, si vous dites

```
MyType *fp = new MyType(1,2);
```

à l'exécution, l'équivalent de **malloc(sizeof(MyType))**est appelé (souvent, c'est littéralement un appel à **malloc( )**), et le constructeur pour **MyType**est appelé avec l'adresse résultante comme pointeur **this**, utilisant **(1,2)**comme liste d'arguments. A ce moment le pointeur est affecté à **fp**, c'est un objet vivant, initialisé ; vous ne pouvez même pas mettre vos mains dessus avant cela. C'est aussi automatiquement le type **MyType**adéquat de sorte qu'aucun transtypage n'est nécessaire.

l'opérateur **new**par défaut vérifie pour s'assurer que l'allocation de mémoire est réussie avant de passer l'adresse au constructeur, de sorte que vous n'avez pas à déterminer explicitement si l'appel a réussi. Plus tard dans ce chapitre, vous découvrirez ce qui se passe s'il n'y a plus de mémoire disponible.

Vous pouvez créer une expression-new en utilisant n'importe quel constructeur disponible pour la classe. Si le constructeur n'a pas d'arguments, vous écrivez l'expression-new sans liste d'argument du constructeur:

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
MyType *fp = new MyType;
```

Remarquez comment le processus de création d'objets sur le tas devient simple ; une simple expression, avec tout les dimensionnements, les conversions et les contrôles de sécurité integrés. Il est aussi facile de créer un objet sur le tas que sur la pile.

13.1.3 - l'opérateur delete

Le complément de l'expression-new est l' *expression-delete*, qui appelle d'abord le destructeur et ensuite libère la mémoire (souvent avec un appel à **free( )**). Exactement comme une expression-new retourne un pointeur sur l'objet, une expression-delete nécessite l'adresse d'un objet.

```
delete fp;
```

Ceci détruit et ensuite libère l'espace pour l'objet **MyType**alloué dynamiquement qui a été créé plus tôt.

deletepeut être appelé seulement pour un objet créé par **new**. Si vous **malloc(ez)( )**(ou **calloc(ez)( )**ou **realloc(ez)( )**) un objet et ensuite vous le **delete(z)**, le comportement est indéfini. Parce que la plupart des implémentations par défaut de **new**et **delete**utilisent **malloc( )**et **free( )**, vous finirez probablement par libérer la mémoire sans appeler le destructeur.

Si le pointeur que vous détruisez vaut zéro, rien ne se passera. Pour cette raison, les gens recommandent souvent de mettre un pointeur à zero immédiatemment après un 'delete', pour empêcher de le détruire deux fois. Détruire un objet plus d'une fois est assurement une mauvaise chose à faire, et causera des problèmes.

13.1.4 - Un exemple simple

Cet exemple montre que l'initialisation a lieu :

```
//: C13:Tree.h
#ifndef TREE_H
#define TREE_H
#include <iostream>

class Tree {
  int height;
public:
  Tree(int treeHeight) : height(treeHeight) {}
  ~Tree() { std::cout << "*"; }
  friend std::ostream&
  operator<<(std::ostream& os, const Tree* t) {
    return os << "La hauteur de l'arbre est : "
              << t->height << std::endl;
  }
};
#endif // TREE_H ///:~
```

```
//: C13:NewAndDelete.cpp
// Démo simple de new & delete
#include "Tree.h"
using namespace std;

int main() {
  Tree* t = new Tree(40);
  cout << t;
  delete t;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
} ///:~
```

Vous pouvez prouver que le consructeur est appelé en affichant la valeur de **Tree**. Ici, c'est fait en surchargeant l'opérateur **operator<<**pour l'utiliser avec un **ostream**et un **Tree***. Notez, toutefois, que même si la fonction est déclarée comme **friend**, elle est définie 'inline' ! C'est pour des raisons d'ordre purement pratique– la définition d'une fonction **amie**d'une classe comme une 'inline' ne change pas le statut d' **amie**ou le fait que c'est une fonction globale et non une fonction membre de classe. Notez également que la valeur de retour est le résultat de toute l'expression de sortie, qui est un **ostream&**(ce qu'il doit être, pour satisfaire le type de valeur de retour de la fonction).

13.1.5 - Le surcoût du gestionnaire de mémoire

Lorsque vous créez des objets automatiques sur la pile, la taille des objets et leur durée de vie sont intégrées immédiatement dans le code généré, parce que le compilateur connaît le type exact, la quantité, et la portée. La création d'objets sur le tas implique un surcoût à la fois dans le temps et dans l'espace. Voici un scenario typique. (Vous pouvez remplacer **malloc( )**par **calloc( )**ou **realloc( )**.)

Vous appelez **malloc( )**, qui réclame un bloc de mémoire dans le tas. (Ce code peut, en réalité, faire partie de **malloc( )**.)

Une recherche est faite dans le tas pour trouver un bloc de mémoire suffisamment grand pour satisfaire la requête. Cela est fait en consultant une carte ou un répertoire de quelque sorte montrant quels blocs sont actuellement utilisés et quels blocs sont disponibles. C'est un procédé rapide, mais qui peut nécessiter plusieurs essais, aussi il peut ne pas être déterministe – c'est à dire que vous ne pouvez pas compter sur le fait que **malloc( )**prenne toujours le même temps pour accomplir son travail.

Avant qu'un pointeur sur ce bloc soit retourné, la taille et l'emplacement du bloc doivent être enregistrés de sorte que des appels ultérieurs à **malloc( )**ne l'utiliseront pas, et de sorte que lorsque vous appelez **free( )**, le système sache combien de mémoire libérer.

La façon dont tout cela est implémenté peut varier dans de grandes proportions. Par exemple, rien n'empêche que des primitives d'allocations de mémoire soit implémentées dans le processeur. Si vous êtes curieux, vous pouvez écrire des programmes de test pour essayer de deviner la façon dont votre **malloc( )**est implémenté. Vous pouvez aussi lire le code source de la bibliothèque, si vous l'avez (les sources GNU sont toujours disponibles).

13.2 - Exemples précédents revus

En utilisant **new**et **delete**, l'exemple **Stash**introduit précédemment dans ce livre peut être réécrit en utilisant toutes les fonctionalités présentées dans ce livre jusqu'ici. Le fait d'examiner le nouveau code vous donnera également une revue utile de ces sujets .

A ce point du livre, ni la classe **Stash**ni la classe **Stack**ne “posséderont” les objets qu'elles pointent ; c'est à dire que lorsque l'objet **Stash**ou **Stack**sort de la portée, il n'appellera pas **delete**pour tous les objets qu'il pointe. La raison pour laquelle cela n'est pas possible est que, en tentant d'être génériques, ils conservent des pointeurs **void**. Si vous detruisez un pointeur **void**, la seule chose qui se produit est la libération de la mémoire, parce qu'il n'y a pas d'informaton de type et aucun moyen pour le compilateur de savoir quel destructeur appeler.

13.2.1 - detruire un void* est probablement une erreur

Cela vaut la peine de relever que si vous appelez **delete**sur un **void***, cela causera presque certainement une erreur dans votre programme à moins que la destination de ce pointeur ne soit très simple ; en particulier, il ne doit

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

pas avoir de destructeur. Voici un exemple pour vous montrer ce qui se passe :

```
                        //: C13:BadVoidPointerDeletion.cpp
// detruire des pointeurs void peut provoquer des fuites de mémoire
#include <iostream>
using namespace std;

class Object {
  void* data; // un certain stockage
  const int size;
  const char id;
public:
  Object(int sz, char c) : size(sz), id(c) {
    data = new char[size];
    cout << "Construction de l'objet " << id
         << ", taille = " << size << endl;
  }
  ~Object() {
    cout << "Destruction de l'objet " << id << endl;
    delete []data; // OK libérer seulement la donnée,
    // aucun appel de destructeur nécessaire
  }
};

int main() {
  Object* a = new Object(40, 'a');
  delete a;
  void* b = new Object(40, 'b');
  delete b;
} ///:~
```

La classe **Object**contient un **void***qui est initialisé pour une donnée “brute” (il ne pointe pas sur des objets ayant un destructeur). Dans le destructeur de **Object**, **delete**est appelé pour ce **void***avec aucun effet néfaste, parce que la seule chose que nous ayons besoin qu'il se produise est que cette mémoire soit libérée.

Cependant, dans **main( )**vous pouvez voir qu'il est tout à fait nécessaire que **delete**sache avec quel type d'objet il travaille. Voici la sortie :

```
                        Construction de l'objet a, taille = 40
Destruction de l'objet a
Construction de l'objet b, taille = 40
```

Parce que **delete a**sait que **a**pointe sur un **Object**, le destructeur est appelé et le stockage alloué pour **data**est libérée. Toutefois, si vous manipulez un objet à travers un **void***comme dans le cas de **delete b**, la seule chose qui se produit est que la mémoire pour l' **Object**est libérée ; mais le destructeur n'est pas appelé de sorte qu'il n'y a pas libération de la mémoire que **data**pointe. A la compilation de ce programme, vous ne verrez probablement aucun message d'avertissement ; le compilateur suppose que vous savez ce que vous faites. Aussi vous obtenez une fuite de mémoire très silencieuse.

Si vous avez une fuite de mémoire dans votre programme, recherchez tous les appels à **delete**et vérifiez le type de pointeur qui est détruit. Si c'est un **void***alors vous avez probablement trouvé une source de votre fuite de mémoire (C++ offre, cependant, d'autres possibilités conséquentes de fuites de mémoire).

13.2.2 - La responsabilité du nettoyage avec les pointeurs

Pour rendre les conteneurs **Stash**et **Stack**flexibles (capables de contenir n'importe quel type d'objet), ils conserveront des pointeurs **void**. Ceci signifie que lorsqu'un pointeur est retourné par l'objet **Stash**ou **Stack**, vous devez le transtyper dans le type convenablebavant de pouvoir l'utiliser ; comme vu auparavant, vous devez également le transtyper dans le type convenable avant de le detruire ou sinon vous aurez une fuite de mémoire.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

L'autre cas de fuite de mémoire concerne l'assurance que **delete**est effectivement appelé pour chaque pointeur conservé dans le conteneur. Le conteneur ne peut pas “posséder” le pointeur parce qu'il le détient comme un **void***et ne peut donc pas faire le nettoyage correct. L'utilisateur doit être responsable du nettoyage des objets. Ceci produit un problème sérieux si vous ajoutez des pointeurs sur des objets créés dans la pile *et*des objets créés sur le tas dans le même conteneur parce qu'une expression-delete est dangereuse pour un pointeur qui n'a pas été créé sur le tas. (Et quand vous récupérez un pointeur du conteneur, comment saurez vous où son objet a été alloué ?) Ainsi, vous devez vous assurer que les objets conservés dans les versions suivantes de **Stash**et **Stack**ont été créés seulement sur le tas, soit par une programmation méticuleuse ou en créant des classes ne pouvant être construites que sur le tas.

Il est également important de s'assurer que le programmeur client prend la responsabilité du nettoyage de tous les pointeurs du conteneur. Vous avez vu dans les exemples précédents comment la classe **Stack**vérifie dans son destructeur que tous les objets **Link**ont été dépilés. Pour un **Stash**de pointeurs, toutefois, une autre approche est nécessaire.

13.2.3 - Stash pour des pointeurs

Cette nouvelle version de la classe **Stash**, nommée **PStash**, détient des *pointers*sur des objets qui existent eux-mêmes sur le tas, tandis que l'ancienne **Stash**dans les chapitres précédents copiait les objets par valeur dans le conteneur **Stash**. En utilisant **new**et **delete**, il est facile et sûr de conserver des pointeurs vers des objets qui ont été créés sur le tas.

Voici le fichier d'en-tête pour le “ **Stash**de pointeurs”:

```
                        //: C13:PStash.h
// Conserve des pointeurs au lieu d'objets
#ifndef PSTASH_H
#define PSTASH_H

class PStash {
  int quantity; //Nombre d'espaces mémoire
  int next; // Espace vide suivant
   // Stockage des pointeurs:
  void** storage;
  void inflate(int increase);
public:
  PStash() : quantity(0), storage(0), next(0) {}
  ~PStash();
  int add(void* element);
  void* operator[](int index) const; // Récupération
  // Enlever la référence de ce PStash:
  void* remove(int index);
  // Nombre d'éléments dans le Stash:
  int count() const { return next; }
};
#endif // PSTASH_H ///:~
```

Les éléments de données sous-jacents sont assez similaires, mais maintenant le **stockage**est un tableau de pointeurs **void**, et l'allocation de mémoire pour ce tableau est réalisé par **new**au lieu de **malloc( )**. Dans l'expression

```
                        void** st = new void*[quantity + increase];
```

le type d'objet alloué est un **void***, ainsi l'expression alloue un tableau de pointeurs **void**.

Le destructeur libère la mémoire où les pointeurs **void**sont stockés plutôt que d'essayer de libérer ce qu'ils pointent (ce qui, comme on l'a remarqué auparavant, libérera leur stockage et n'appellera pas les destructeurs parce qu'un

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

pointeur **void**ne comporte aucune information de type).

L'autre changement est le remplacement de la fonction **fetch( )**par **operator[ ]**, qui est syntaxiquement plus expressif. De nouveau, toutefois, un **void***est retourné, de sorte que l'utilisateur doit se souvenir des types rassemblés dans le conteneur et transtyper les pointeurs au fur et à mesure de leur récupération (un problème auquel nous apporterons une solution dans les chapitres futurs).

Voici les définitions des fonctions membres :

```
                        //: C13:PStash.cpp {O}
// définitions du Stash de Pointeurs
#include "PStash.h"
#include "../require.h"
#include <iostream>
#include <cstring> // fonctions 'mem'
using namespace std;

int PStash::add(void* element) {
  const int inflateSize = 10;
  if(next >= quantity)
    inflate(inflateSize);
  storage[next++] = element;
  return(next - 1); // Indice
}

// Pas de propriété:
PStash::~PStash() {
  for(int i = 0; i < next; i++)
    require(storage[i] == 0,
      "PStash n'a pas été nettoyé");
  delete []storage;
}

// Surcharge d'opérateur pour accès
void* PStash::operator[](int index) const {
  require(index >= 0,
    "PStash::operator[] indice négatif");
  if(index >= next)
    return 0; // Pour signaler la fin
  // Produit un pointeur vers l'élément désiré:
  return storage[index];
}

void* PStash::remove(int index) {
  void* v = operator[](index);
  // "Enlève" le pointeur:
  if(v != 0) storage[index] = 0;
  return v;
}

void PStash::inflate(int increase) {
  const int psz = sizeof(void*);
  void** st = new void*[quantity + increase];
  memset(st, 0, (quantity + increase) * psz);
  memcpy(st, storage, quantity * psz);
  quantity += increase;
  delete []storage; // Ancien emplacement
  storage = st; // Pointe sur un nouvel emplacement
} ///:~
```

La fonction **add( )**est effectivement la même qu'avant, sauf qu'un pointeur est stocké plutôt qu'une copie de l'objet entier.

Le code de **inflate( )**est modifié pour gérer l'allocation d'un tableau de **void***à la différence de la conception précédente, qui ne fonctionnait qu'avec des octets 'bruts'. Ici, au lieu d'utiliser l'approche précédente de copier par indexation de tableau, la fonction de la librairie standard C **memset( )**est d'abord utilisée pour mettre toute la nouvelle mémoire à zéro (ceci n'est pas strictement nécessaire, puisqu'on peut supposer que le **PStash**gère toute la mémoire correctement– mais en général cela ne fait pas de mal de prendre quelques précautions

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

supplémentaires). Ensuite **memcpy( )**déplace les données existantes de l'ancien emplacement vers le nouveau. Souvent, des fonctions comme **memset( )**et **memcpy( )**ont été optimisées avec le temps, de sorte qu'elles peuvent être plus rapides que les boucles montrées précédemment. Mais avec une fonction comme **inflate( )**qui ne sera probablement pas utilisée très souvent il se peut que vous ne perceviez aucune différence de performance. Toutefois, le fait que les appels de fonction soient plus concis que les boucles peut aider à empêcher des erreurs de codage.

Pour placer la responsabilité du nettoyage d'objets carrément sur les épaules du programmeur client, il y a deux façons d'accéder aux pointeurs dans le **PStash**: l' **operator[]**, qui retourne simplement le pointeur mais le conserve comme membre du conteneur, et une seconde fonction membre **remove( )**, qui retourne également le pointeur, mais qui l'enlève également du conteneur en affectant zéro à cette position. Lorsque le destructeur de **PStash**est appelé, il vérifie pour s'assurer que tous les pointeurs sur objets ont été enlevés ; dans la négative, vous êtes prévenu pour pouvoir empêcher une fuite de mémoire (des solutions plus élégantes viendront à leur tour dans les chapitres suivants).

Un test

Voici le vieux programme de test pour **Stash**récrit pour **PStash**:

```
                        //: C13:PStashTest.cpp
//{L} PStash
// Test du Stash de pointeurs
#include "PStash.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  PStash intStash;
  // 'new' fonctionnne avec des types prédéfinis, également. Notez
  // la syntaxe "pseudo-constructor":
  for(int i = 0; i < 25; i++)
    intStash.add(new int(i));
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash[" << j << "] = "
         << *(int*)intStash[j] << endl;
  // Nettoyage :
  for(int k = 0; k < intStash.count(); k++)
    delete intStash.remove(k);
  ifstream in ("PStashTest.cpp");
  assure(in, "PStashTest.cpp");
  PStash stringStash;
  string line;
  while(getline(in, line))
    stringStash.add(new string(line));
  // Affiche les chaînes :
  for(int u = 0; stringStash[u]; u++)
    cout << "stringStash[" << u << "] = "
         << *(string*)stringStash[u] << endl;
  // Nettoyage :
  for(int v = 0; v < stringStash.count(); v++)
    delete (string*)stringStash.remove(v);
} ///:~
```

Commes auparavant, les **Stash**s sont créés et remplis d'information, mais cette fois l'information est constituée des pointeurs résultant d'expressions- **new**. Dans le premier cas, remarquez la ligne:

```
                        intStash.add(new int(i));
```

L'expression **new int(i)**utilise la forme pseudo-constructeur, ainsi le stockage pour un nouvel objet **int**est créé sur

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

le tas, et le **int**est initialisé avec la valeur **i**.

Pendant l'affichage, la valeur retournée par **PStash::operator[]**doit être transtypée dans le type adéquat; ceci est répété pour le reste des autres objets **PStash**dans le programme. C'est un effet indésirable d'utiliser des pointeurs **void**comme représentation sous-jacente et ce point sera résolu dans les chapitres suivants.

Le second test ouvre le fichier du code source et le lit ligne par ligne dans un autre **PStash**. Chaque ligne est lue dans un objet **string**en utilisant **getline( )**, ensuite un **nouveau string**est créé depuis **line**pour faire une copie indépendante de cette ligne. Si nous nous étions contentés de passer l'adresse de **line**à chaque fois, nous nous serions retrouvés avec un faisceau de pointeurs pointant tous sur **line**, laquelle contiendrait la dernière ligne lue du fichier.

Lorsque vous récupérez les pointeurs, vous voyez l'expression :

```
*(string*)stringStash[v]
```

Le pointeur retourné par **operator[]**doit être transtypé en un **string***pour lui donner le type adéquat. Ensuite, le **string***est déréférencé de sorte que l'expression est évaluée comme un objet, à ce moment le compilateur voit un objet **string**à envoyer à **cout**.

Les objets créés sur le tas doivent être détruits par l'utilisation de l'instruction **remove( )**ou autrement vous aurez un message au moment de l'exécution vous disant que vous n'avez pas complètement nettoyé les objets dans le **PStash**. Notez que dans le cas des pointeurs sur **int**, aucun transtypage n'est nécessaire parce qu'il n'y a pas de destructeur pour un **int**et tout ce dont nous avons besoin est une libération de la mémoire :

```
delete intStash.remove(k);
```

Néammoins, pour les pointeurs sur **string**, si vous oubliez de transtyper vous aurez une autre fuite de mémoire (silencieuse), de sorte que le transtypage est essentiel :

```
delete (string*)stringStash.remove(k);
```

Certains de ces problèmes (mais pas tous) peuvent être résolus par l'utilisation des 'templates' (que vous étudierez dans le chapitre 16).

13.3 - new & delete pour les tableaux

En C++, vous pouvez créer des tableaux d'objets sur la pile ou sur le tas avec la même facilité, et (bien sûr) le constructeur est appelé pour chaque objet dans le tableau. Il y a une contrainte, toutefois : il doit y avoir un constructeur par défaut, sauf pour l'initialisation d'agrégat sur la pile (cf. Chapitre 6), parce qu'un constructeur sans argument doit être appelé pour chaque objet.

Quand vous créez des tableaux d'objets sur le tas en utilisant **new**, vous devez faire autre chose. Voici un exemple d'un tel tableau :

```
MyType* fp = new MyType[100];
```

Ceci alloue suffisamment d'espace de stockage sur le tas pour 100 objets **MyType**et appelle le constructeur pour

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

chacun d'eux. Cependant, à présent vous ne disposez que d'un **MyType***, ce qui est exactement la même chose que vous auriez eu si vous aviez dit :

```
MyType* fp2 = new MyType;
```

pour créer un seul objet. Parce que vous avez écrit le code, vous savez que **fp**est en fait l'adresse du début d'un tableau, et il paraît logique de sélectionner les éléments d'un tableau en utilisant une expression comme **fp[3]**. Mais que se passe-t-il quand vous détruisez le tableau ? Les instructions

```
delete fp2; // OK
delete fp;  // N'a pas l'effet désiré
```

paraissent identiques, et leur effet sera le même. Le destructeur sera appelé pour l'objet **MyType**pointé par l'adresse donnée, et le stockage sera libéré. Pour **fp2**cela fonctionne, mais pour **fp**cela signifie que 99 appels au destructeur ne seront pas effectués. La bonne quantité d'espace de stockage sera toujours libérée, toutefois, parce qu'elle est allouée en un seul gros morceau, et la taille de ce morceau est cachée quelque part par la routine d'allocation.

La solution vous impose de donner au compilateur l'information qu'il s'agit en fait de l'adresse du début d'un tableau. Ce qui est fait avec la syntaxe suivante :

```
delete []fp;
```

Les crochets vides disent au compilateur de générer le code qui cherche le nombre d'objets dans le tableau, stocké quelque part au moment de la création de ce dernier, et appelle le destructeur pour ce nombre d'objets. C'est en fait une version améliorée de la syntaxe ancienne, que vous pouvez toujours voir dans du vieux code :

```
delete [100]fp;
```

qui forçait le programmeur à inclure le nombre d'objets dans le tableau et introduisait le risque que le programmeur se trompe. Le fait de laisser le compilateur gérer cette étape consommait un temps système supplémentaire très bas, et il a été décidé qu'il valait mieux ne préciser qu'en un seul endroit plutôt qu'en deux le nombre d'objets.

13.3.1 - Rendre un pointeur plus semblable à un tableau

A côté de cela, le **fp**défini ci-dessus peut être modifié pour pointer sur n'importe quoi, ce qui n'a pas de sens pour l'adresse de départ d'un tableau. Il est plus logique de le définir comme une constante, si bien que toute tentative de modifier le pointeur sera signalée comme une erreur. Pour obtenir cet effet, vous pouvez essayer

```
int const* q = new int[10];
```

ou bien

```
const int* q = new int[10];
```

mais dans les deux cas, le **const**sera lié au **int**, c'est-à-dire *ce qui est*pointé, plutôt qu'à la qualité du pointeur lui-même. Au lieu de cela, vous devez dire :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
int* const q = new int[10];
```

A présent, les éléments du tableau dans **q**peuvent être modifiés, mais tout changement de **q**(comme **q++**) est illégal, comme c'est le cas avec un identifiant de tableau ordinaire.

13.4 - Manquer d'espace de stockage

Que se passe-t-il quand l' **operator new ()** ne peut trouver de block de mémoire contigü suffisamment grand pour contenir l'objet désiré ? Une fonction spéciale appelée le *gestionnaire de l'opérateur new*est appelée. Ou plutôt, un pointeur vers une fonction est vérifié, et si ce pointeur ne vaut pas zéro la fonction vers laquelle il pointe est appelée.

Le comportement par défaut pour ce gestionnaire est *de lancer une exception*, sujet couvert dans le deuxième Volume. Toutefois, si vous utilisez l'allocation sur le tas dans votre programme, il est prudent de remplacer au moins le gestionnaire de 'new' avec un message qui dit que vous manquez de mémoire et ensuite termine le programme. Ainsi, pendant le débugage, vous aurez une idée de ce qui s'est produit. Pour le programme final vous aurez intérêt à utiliser une récupération plus robuste.

Vous remplacez le gestionnaire de 'new' en incluant **new.h**et en appelant ensuite **set_new_handler( )**avec l'adresse de la fonction que vous voulez installer :

```
//: C13:NewHandler.cpp
// Changer le gestionnaire de new
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

int count = 0;

void out_of_memory() {
  cerr << "memory exhausted after " << count
    << " allocations!" << endl;
  exit(1);
}

int main() {
  set_new_handler(out_of_memory);
  while(1) {
    count++;
    new int[1000]; // Epuise la mémoire
  }
} ///:~
```

La fonction gestionnaire de 'new' ne doit prendre aucun argument et avoir une valeur de retour **void**. La boucle **while**continuera d'allouer des objets **int**(et de jeter leurs adresses de retour) jusqu'à ce que le stockage libre soit épuisé. A l'appel à **new**qui suit immédiatement, aucun espace de stockage ne peut être alloué, et le gestionnaire de 'new' sera appelé.

Le comportement du gestionnaire de 'new' est lié à **operator new ( )** , donc si vous surchargez **operator new ( )** (couvert dans la section suivante) le gestionnaire de 'new' ne sera pas appelé par défaut. Si vous voulez toujours que le gestionnaire de 'new' soit appelé vous devrez écrire le code pour ce faire dans votre **operator new ( )** surchargé.

Bien sûr, vous pouvez écrire des gestionnaires de 'new' plus sophistiqués, voire même un qui essaye de réclamer de la mémoire (généralement connu sous le nom de *ramasse-miettes*(*garbage collector*en anglais, ndt)). Ce n'est pas un travail pour un programmeur novice.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

13.5 - Surcharger new & delete

Quand vous créez une expression **new**, deux choses se produisent. D'abord, l'espace de stockage est alloué en utilisant **operator new( )**, puis le constructeur est appelé. Dans une expression **delete**, le destructeur est appelé, puis le stockage est libéré en utilisant **operator delete( )**. Les appels au constructeur et au destructeur ne sont jamais sous votre contrôle (autrement vous pourriez les corrompre accidentellement), mais vous *pouvez*modifier les fonctions d'allocation de stockage **operator new( )**et **operator delete( )**.

Le système d'allocation de mémoire utilisé par **new**et **delete**est conçu pour un usage général. Dans des situations spéciales, toutefois, il ne convient pas à vos besoin. La raison la plus commune pour modifier l'allocateur est l'efficacité : vous pouvez créer et détruire tellement d'objets d'une classe donnée que cela devient goulet d'étranglement en terme de vitesse. Le C++ vous permet de surcharger **new**et **delete**pour implémenter votre propre schéma d'allocation de stockage, afin que vous puissiez gérer ce genre de problèmes.

Un autre problème est la fragmentation du tas. En allouant des objets de taille différente, il est possible de fragmenter le tas si bien que vous vous retrouvez à cours de stockage. En fait, le stockage peut être disponible, mais à cause de la fragmentation aucun morceau n'est suffisamment grand pour satisfaire vos besoins. En créant votre propre allocateur pour une classe donnée, vous pouvez garantir que cela ne se produira jamais.

Dans les systèmes embarqués et temps-réel, un programme peut devoir tourner pendant longtemps avec des ressources limitées. Un tel système peut également nécessiter que l'allocation de mémoire prennent toujours le même temps, et il n'y a aucune tolérance à l'épuisement du tas ou à sa fragmentation. Un allocateur de mémoire sur mesure est la solution ; autrement, les programmeurs éviteront carrémnt d'utiliser **new**et **delete**dans ce genre de situations et manqueront un des atouts précieux du C++.

Quand vous surchargez **operator new( )**et **operator delete( )**, il est important de se souvenir que vous modifiez uniquement la façon dont *l'espace de stockage brut est alloué*. Le compilateur appellera simplement votre **new**au lieu de la version par défaut pour allouer le stockage, puis appellera le constructeur pour ce stockage. Donc, bien que le compilateur alloue le stockage *et*appelle le constructeur quand il voit **new**, tout ce que vous pouvez changer quand vous surchargez **new**est le volet allocation de la mémoire. (**delete**a la même limitation.)

Quand vous surchargez **operator new( )**, vous remplacez également le comportement quand il en vient à manquer de mémoire, vous devez donc décider que faire dans votre **operator new( )**: renvoyer zéro, écrire une boucle pour appeler le gestionnaire de new et réessayer l'allocation, ou (typiquement) de lancer une exception **bad_alloc**(traitée dans le Volume 2, disponible sur *www.BruceEckel.com*en anglais et bientôt sur *www.developpez.com*en français).

Surcharger **new**et **delete**c'est comme surcharger n'importe quel autre opérateur. Toutefois, vous avez le choix de surcharger l'allocateur global ou d'utiliser un allocateur différent pour une classe donnée.

13.5.1 - La surcharge globale de new & delete

C'est l'approche extrême, quand les versions globales de **new**et **delete**ne sont pas satisfaisantes pour l'ensemble du système. Si vous surchargez les versions globales, vous rendez les versions par défaut complètement innaccessibles – vous ne pouvez même pas les appeler depuis vos redéfinitions.

Le **new**surchargé doit prendre un argument de type **size_t**(le type standard pour les tailles en C standard). Cet argument est généré et vous est passé par le compilateur et représente la taille de l'objet que vous avez la charge d'allouer. Vous devez renvoyer un pointeur soit vers un objet de cette taille (ou plus gros, si vous avez des raisons pour ce faire), ou zéro si vous ne pouvez pas trouver la mémoire (auquel cas le constructeur *n'est pas*appelé !). Toutefois, si vous ne pouvez pas trouver la mémoire, vous devriez probablement faire quelque chose de plus informatif que simplement renvoyer zéro, comme d'appeler new-handler ou de lancer une exception, pour signaler

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

qu'il y a un problème.

La valeur de retour de **operator new( )**est un **void***, *pas*un pointeur vers un quelconque type particulier. Tout ce que vous avez fait est allouer de la mémoire, pas un objet fini – cela ne se produit pas tant que le constructeur n'est pas appelé, une action que le compilateur garantit et qui échappe à votre contrôle.

operator delete( )prend un **void***vers la mémoire qui a été allouée par **operator new**. C'est un **void***parce que **operator delete**ne reçoit le pointeur qu' *après*que le destructeur ait été appelé, ce qui efface le caractère objet du fragment de stockage. Le type de retour est **void**.

Voici un exemple simple montrant comment surcharger le **new**et le **delete**globaux :

```
                        //: C13:GlobalOperatorNew.cpp
// Surcharge new/delete globaux
#include <cstdio>
#include <cstdlib>
using namespace std;

void* operator new(size_t sz) {
  printf("operateur new: %d octets\n", sz);
  void* m = malloc(sz);
  if(!m) puts("plus de memoire");
  return m;
}

void operator delete(void* m) {
  puts("operateur delete");
  free(m);
}

class S {
  int i[100];
public:
  S() { puts("S::S()"); }
  ~S() { puts("S::~S()"); }
};

int main() {
  puts("creation & destruction d'un int");
  int* p = new int(47);
  delete p;
  puts("creation & destruction d'un s");
  S* s = new S;
  delete s;
  puts("creation & destruction de S[3]");
  S* sa = new S[3];
  delete []sa;
} ///:~
```

Ici vous pouvez voir la forme générale de la surcharge de **new**et **delete**. Ceux-ci utilisent les fonctions de librairie du C standard **malloc( )**et **free( )**pour les allocateurs (qui sont probablement ce qu'utilisent également les **new**et **delete**par défaut !). Toutefois, ils affichent également des messages à propos de ce qu'ils font. Remarquez que **printf( )**et **puts( )**sont utilisés plutôt que **iostreams**. C'est parce que quand un objet **iostream**est créé (comme les **cin**, **cout**et **cerr**globaux), il appelle **new**pour allouer de la mémoire. Avec **printf( )**vous ne vous retrouvez pas dans un interblocage parce qu'il n'appelle pas **new**pour s'initialiser.

Dans **main( )**, des objets de type prédéfinis sont créés pour prouver que le **new**et le **delete**surchargés sont également appelés dans ce cas. Puis un unique objet de type **S**est créé, suivi par un tableau de **S**. Pour le tableau, vous verrez par le nombre d'octets réclamés que de la mémoire supplémentaire est allouée pour stocker de l'information (au sein du tableau) à propos du nombre d'objets qu'il contient. Dans tous les cas, les versions surchargées globales de **new**et **delete**sont utilisées.

13.5.2 - Surcharger new & delete pour une classe

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Bien que vous n'ayez pas à dire explicitement **static**, quand vous surchargez **new**et **delete**pour une classe, vous créez des fonctions membres **static**. Comme précédemment, la syntaxe est la même que pour la surcharge de n'importe quel opérateur. Quand le compilateur voit que vous utilisez **new**pour créer un objet de votre classe, il choisit le membre **operator new( )**de préférence à la définition globale. Toutefois, les versions globales de **new**et **delete**sont utilisées pour tous les autres types d'objets (à moins qu'ils n'aient leur propres **new**et **delete**).

Dans l'exemple suivant, un système d'allocation primitif est créé pour la classe **Framis**. Un tronçon de mémoire est réservé dans la zone de données statiques au démarrage du programme, et cette mémoire est utilisée pour allouer de l'espace pour les objets de type **Framis**. Pour déterminer quels blocs ont été alloués, un simple tableau d'octets est utilisé, un octet pour chaque bloc :

```
                        //: C13:Framis.cpp
// Surcharge locale de new & delete
#include <cstddef> // Size_t
#include <fstream>
#include <iostream>
#include <new>
using namespace std;
ofstream out("Framis.out");

class Framis {
  enum { sz = 10 };
  char c[sz]; // Pour occuper de l'espace, pas utilisé
  static unsigned char pool[];
  static bool alloc_map[];
public:
  enum { psize = 100 };  // nombre de framis autorisés
  Framis() { out << "Framis()\n"; }
  ~Framis() { out << "~Framis() ... "; }
  void* operator new(size_t) throw(bad_alloc);
  void operator delete(void*);
};
unsigned char Framis::pool[psize * sizeof(Framis)];
bool Framis::alloc_map[psize] = {false};

// La taille est ignorée -- suppose un objet Framis
void*
Framis::operator new(size_t) throw(bad_alloc) {
  for(int i = 0; i < psize; i++)
    if(!alloc_map[i]) {
      out << "utilise le bloc " << i << " ... ";
      alloc_map[i] = true; // le marquer utilisé
      return pool + (i * sizeof(Framis));
    }
  out << "plus de memoire" << endl;
  throw bad_alloc();
}

void Framis::operator delete(void* m) {
  if(!m) return; // Vérifie si le pointeur est nul
  // Suppose qu'il a été créé dans la réserve
  // Calcule le numéro du bloc :
  unsigned long block = (unsigned long)m
    - (unsigned long)pool;
  block /= sizeof(Framis);
  out << "libère le bloc " << block << endl;
  // le marque libre :
  alloc_map[block] = false;
}

int main() {
  Framis* f[Framis::psize];
  try {
    for(int i = 0; i < Framis::psize; i++)
      f[i] = new Framis;
    new Framis; // Plus de mémoire
  } catch(bad_alloc) {
    cerr << "Plus de mémoire !" << endl;
  }
  delete f[10];
  f[10] = 0;
  // Utilise la mémoire libérée :
  Framis* x = new Framis;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    delete x;
    for(int j = 0; j < Framis::psize; j++)
      delete f[j]; // Delete f[10] OK
  } ///:~
```

La réserve de mémoire pour le tas de **Framis**est créé en allouant un tableau d'octets suffisamment grand pour contenir **psize**objets **Framis**. La carte d'allocation fait **psize**éléments de long, et il y a donc un **bool**pour chaque bloc. Toutes les valeurs dans la carte d'allocation sont initialisées à **false**en utilisant l'astuce de l'initialisation d'agrégats qui consiste à affecter le premier élément afin que le compilateur initialise automatiquement tout le reste à leur valeur par défaut (qui est **false**, dans le cas des **bool**).

L' **operator new( )**local a la même syntaxe que le global. Tout ce qu'il fait est de chercher une valeur **false**dans la carte d'allocation, puis met cette valeur à **true**pour indiquer qu'elle a été allouée et renvoie l'adresse du bloc de mémoire correspondant. S'il ne peut trouver aucune mémoire, il émet un message au fichier trace et lance une exception **bad_alloc**.

C'est le premier exemple d'exceptions que vous avez vu dans ce livre. Comme la discussion détaillée est repoussée au Volume 2, c'en est un usage très simple. Dans l' **operator new( )**il y a deux utilisations de la gestion d'exceptions. Premièrement, la liste d'arguments de la fonction est suivie par **throw(bad_alloc)**, qui dit au compilateur et au lecteur que cette fonction peut lancer une exception de type **bad_alloc**. Deuxièmement, s'il n'y a plus de mémoire la fonction lance effectivement l'exception dans l'instruction **throw bad_alloc**. Quand une exception est lancée, la fonction cesse de s'exécuter et le contrôle est passé à un *gestionnaire d'exécution*, qui est exprimé par une clause **catch**.

Dans **main( )**, vous voyez l'autre partie de la figure, qui est la clause *try-catch*. Le bloc **try**est entouré d'accolades et contient tout le code qui peut lancer des exceptions – dans ce cas, tout appel à **new**qui invoque des objets de type **Framis**. Immédiatement après le bloc **try**se trouvent une ou plusieurs clauses **catch**, chacune d'entre elles spécifiant le type d'exception qu'elles capturent. Dans ce cas, **catch(bad_alloc)**signifie que les exceptions **bad_alloc**seront interceptées ici. Cette clause **catch**particulière est exécutée uniquement quand une exception **bad_alloc**est lancée, et l'exécution continue à la fin de la dernière clause **catch**du groupe (il n'y en a qu'une seule ici, mais il pourrait y en avoir plusieurs).

Dans cet exemple, on peut utiliser iostreams parce que les opérateurs **operator new( )**et **delete( )**globaux ne sont pas touchés.

operator delete( )suppose que l'adresse de **Framis**a été créée dans la réserve. C'est une supposition logique, parce que l' **operator new( )**local sera appelé à chaque fois que vous créez un objet **Framis**unique sur le tas – mais pas un tableau : l'opérateur **new**global est appelé pour les tableaux. Ainsi, l'utilisateur peut avoir accidentellement appelé l' **operator delete( )**sans utiliser la syntaxe avec les crochets vides pour indiquer la destruction d'un tableau. Ceci poserait un problème. Egalement, l'utilisateur pourrait detruire un pointeur vers un objet créé sur la pile. Si vous pensez que ces choses pourraient se produire, vous pourriez vouloir ajouter une ligne pour être sûr que l'adresse est dans la réserve et sur une frontière correcte (peut-être commencez-vous à voir le potentiel des **new**et **delete**surchargés pour trouver les fuites de mémoire).

L' **operator delete( )**calcule le bloc de la réserve que représente ce pointeur, puis fixe l'indicateur de la carte d'allocation de ce bloc à faux pour indiquer que ce bloc a été libéré.

Dans **main( )**, suffisamment d'objets **Framis**sont alloués dynamiquement pour manquer de mémoire ; ceci teste le comportement au manque de mémoire. Puis un des objets est libéré, et un autre est créé pour montrer que la mémoire libérée est réutilisée.

Puisque ce schéma d'allocation est spécifique aux objets **Framis**, il est probablement beaucoup plus rapide que le schéma d'allocation générique utilisé pour les **new**et **delete**par défaut. Toutefois, vous devriez noter que cela ne marche pas automatiquement si l'héritage est utilisé (l'héritage est couvert au Chapitre 14).

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

13.5.3 - Surcharger new & delete pour les tableaux

Si vous surchargez les opérateurs **new**et **delete**pour une classe, ces opérateurs sont appelés à chaque fois que vous créez un objet de cette classe. Toutefois, si vous créez un *tableau*d'objets de cette classe, l' **operator new( )**global est appelé pour allouer suffisamment d'espace de stockage pour tout le tableau d'un coup, et l' **operator delete( )**global est appelé pour libérer ce stockage. Vous pouvez contrôler l'allocation de tableaux d'objets en surchargeant la version spéciale tableaux de **operator new[ ]**et **operator delete[ ]**pour la classe. Voici un exemple qui montre quand les deux versions différentes sont appelées :

```
                          //: C13:ArrayOperatorNew.cpp
// Operateur new pour tableaux
#include <new> // Définition de size_t
#include <fstream>
using namespace std;
ofstream trace("ArrayOperatorNew.out");

class Widget {
  enum { sz = 10 };
  int i[sz];
public:
  Widget() { trace << "*"; }
  ~Widget() { trace << "~"; }
  void* operator new(size_t sz) {
    trace << "Widget::new: "
          << sz << " octets" << endl;
    return ::new char[sz];
  }
  void operator delete(void* p) {
    trace << "Widget::delete" << endl;
    ::delete []p;
  }
  void* operator new[](size_t sz) {
    trace << "Widget::new[]: "
          << sz << " octets" << endl;
    return ::new char[sz];
  }
  void operator delete[](void* p) {
    trace << "Widget::delete[]" << endl;
    ::delete []p;
  }
};

int main() {
  trace << "new Widget" << endl;
  Widget* w = new Widget;
  trace << "\ndelete Widget" << endl;
  delete w;
  trace << "\nnew Widget[25]" << endl;
  Widget* wa = new Widget[25];
  trace << "\ndelete []Widget" << endl;
  delete []wa;
} ///:~
```

Ici, les versions globales de **new**et **delete**sont appelées si bien que l'effet est le même que de ne pas avoir de version surchargée de **new**et **delete**sauf qu'une information de traçage est ajoutée. Bien sûr, vous pouvez utliser n'importe quel schéma d'allocation de mémoire dans les **new**et **delete**surchargés.

Vous pouvez constater que la syntaxe de **new**et de **delete**pour tableaux est la même que pour leur version destinée aux objets individuels sauf qu'on y a ajouté des crochets. Dans les deux cas, on vous passe la taille de la mémoire que vous devez allouer. La taille passée à la version tableaux sera celle du tableau entier. Cela vaut la peine de garder à l'esprit que la *seule*chose que l' **operator new( )**surchargé soit obligé de faire est de renvoyer un pointeur vers un bloc mémoire suffisamment grand. Bien que vous puissiez faire une initialisation de cette mémoire, c'est normalement le travail du contructeur qui sera automatiquement appelé pour votre mémoire par le compilateur.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Le constructeur et le destructeur affichent simplement des caractères afin que vous puissiez voir quand ils ont été appelés. Voici à quoi ressemble le fichier de traces pour un compilateur :

```
                        new Widget
Widget::new: 40 octets
*
delete Widget
~Widget::delete

new Widget[25]
Widget::new[]: 1004 octets
************************
delete []Widget
~~~~~~~~~~~~~~~~~~~~~~~~Widget::delete[]
```

Créer un objet individuel requiert 40 octet, comme vous pourriez prévevoir. (Cette machine utilise quatre octets pour un **int**.) L' **operator new( )**est appelé, puis le constructeur (indiqué par *****). De manière complémentaire, appeler **delete**entraîne l'appel du destructeur, puis de l' **operator delete( )**.

Comme promis, quand un tableau d'objets **Widget**est créé, la version tableau de l' **operator new( )**est utilisée. Mais remarquez que la taille demandée dépasse de quatre octets la valeur attendue. Ces quatre octets supplémentaires sont l'endroit où le système conserve de l'information à propos du tableau, en particulier, le nombre d'objets dans le tableau. Ainsi, quand vous dites :

```
                        delete []Widget;
```

Les crochets disent au compilateur que c'est un tableau d'objets, afin que le compilateur génère le code pour chercher le nombre d'objets dans le tableau et appeler le destructeur autant de fois. Vous pouvez voir que même si les versions tableau de **operator new( )**et **operator delete( )**ne sont appelées qu'une fois pour tout le bloc du tableau, les constructeur et destructeur par défaut sont appelés pour chaque objet du tableau.

13.5.4 - Appels au constructeur

En considérant que

```
                        MyType* f = new MyType;
```

appelle **new**pour allouer un espace de stockage de taille adaptée à un objet **MyType**, puis invoque le constructeur de **MyType**sur cet espace, que se passe-t-il si l'allocation de mémoire dans **new**échoue ? Dans ce cas, le constructeur n'est pas appelé, aussi, bien que vous ayez un objet créé sans succès, au moins vous n'avez pas appelé le contructeur et vous ne lui avez pas passé un poiteur **this**nul. Voici un exemple pour le prouver :

```
                        //: C13:NoMemory.cpp
// Le constructeur n'est pas appelé si new échoue
#include <iostream>
#include <new> // définition de bad_alloc
using namespace std;

class NoMemory {
public:
  NoMemory() {
    cout << "NoMemory::NoMemory()" << endl;
  }
  void* operator new(size_t sz) throw(bad_alloc){
    cout << "NoMemory::operator new" << endl;
    throw bad_alloc(); // "Plus de memoire"
  }
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
};

int main() {
  NoMemory* nm = 0;
  try {
    nm = new NoMemory;
  } catch(bad_alloc) {
    cerr << "exception plus de mémoire" << endl;
  }
  cout << "nm = " << nm << endl;
} ///:~
```

Quand le programme s'exécute, il n'affiche pas le message du constructeur, uniquement le message de l' **operator new( )**et le message dans le gestionnaire d'exception. Puisque **new**ne retourne jamais, le constructeur n'est pas appelé si bien que son message n'est pas affiché.

Il est important que **nm**soit initialisé à zéro parce que l'expression **new**n'aboutit jamais, et le pointeur devrait être à zéro pour être sûr que vous n'en ferez pas mauvais usage. Toutefois, vous devriez en fait faire plus de choses dans le gestionnaire d'exceptions que simplement afficher un message et continuer comme si l'objet avait été créé avec succès. Idéalement, vous ferez quelque chose qui permettra au programme de se remettre de ce problème, ou au moins de se terminer après avoir enregistré une erreur.

Dans les versions de C++ antérieures, c'était la pratique standard que **new**renvoie zéro si l'allocation du stockage échouait. Cela évitait que la construction se produise. Toutefois, si vous essayez de faire renvoyer zéro à **new**avec un compilateur conforme au standard, il devrait vous dire que vous êtes supposés lancer une excpetion **bad_alloc**à la place.

13.5.5 - new & delete de placement

Il y a deux autres usages, moins courants, pour la surcharge de l' **operator new( )**.

1. Vous pouvez avoir envie de placer un objet dans un emplacement spécifique de la mémoire. Ceci est particulièrement important pour les systèmes embarqués orientés matériel où un objet peut être synonyme d'un élément donnée du matériel.
2. Vous pouvez vouloir être capable de choisir entre différents allocateurs quand vous appelez **new**.

Ces deux situations sont résolues par le même mécanisme : l' **operator new( )**surchargé peut prendre plus d'un argument. Comme vous l'avez vu auparavant, le premier argument est toujours la taille de l'objet, qui est calculée en secret et passée par le compilateur. Mais les autres arguments peuvent être tout ce que vous voulez – l'adresse où vous voulez que l'objet soit placé, une référence vers une fonction ou un objet d'allocation de mémoire, ou quoique ce soit de pratique pour vous.

La façon dont vous passez les arguments supplémentaires à **operator new( )**pendant un appel peut sembler un peu curieuse à première vue. Vous placez la liste d'arguments (*sans*l'argument **size_t**, qui est géré par le compilateur) après le mot-clef **new**et avant le nom de la classe de l'objet que vous êtes en train de créer. Par exemple,

```
X* xp = new(a) X;
```

Passera **a**comme deuxième argument à **operator new( )**. Bien sûr, ceci ne peut fonctionner que si un tel **operator new( )**a été déclaré.

Voic un exemple montrant comment vous pouvez placer un objet à un endroit donné :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                        //: C13:PlacementOperatorNew.cpp
// Placement avec l'opérateur new()
#include <cstddef> // Size_t
#include <iostream>
using namespace std;

class X {
  int i;
public:
  X(int ii = 0) : i(ii) {
    cout << "this = " << this << endl;
  }
  ~X() {
    cout << "X::~X(): " << this << endl;
  }
  void* operator new(size_t, void* loc) {
    return loc;
  }
};

int main() {
  int l[10];
  cout << "l = " << l << endl;
  X* xp = new(l) X(47); // X à l'emplacement l
  xp->X::~X(); // Appel explicite au destructeur
  // Utilisé UNIQUEMENT avec le placement !
} ///:~
```

Remarquez que **operator new**ne renvoie que le pointeur qui lui est passé. Ainsi, l'appelant décide où l'objet va résider, et le contructeur est appelé pour cette espace mémoire comme élément de l'expression new.

Bien que cet exemple ne montre qu'un argument additionnel, il n'y a rien qui vous empêche d'en ajouter davantage si vous en avez besoin pour d'autres buts.

Un dilemne apparaît lorsque vous voulez détruire l'objet. Il n'y a qu'une version de **operator delete**, et il n'y a donc pas moyen de dire, “Utilise mon dé-allocateur spécial pour cet objet.” Vous voulez appeler le destructeur, mais vous ne voulez pas que la mémoire soit libérée le mécanisme de mémoire dynamique parce qu'il n'a pas été alloué sur la pile.

La réponse est une syntaxe très spéciale. Vous pouvez explicitement appeler le destructeur, comme dans

```
xp->X::~X(); // Appel explicite au destructeur
```

Un avertissement sévère est justifié ici. Certaines personnes voient cela comme un moyen de détruire des objets à un certain moment avant la fin de la portée, plutôt que soit d'ajuster la portée ou (manière plus correcte de procéder) en utilisant la création dynamique d'objets s'ils veulent que la durée de vie de l'objet soit déterminée à l'exécution. Vous aurez de sérieux problèmes si vous appelez le destructeur ainsi pour un objet ordinaire créé sur la pile parce que le destructeur sera appelé à nouveau à la fin de la portée. Si vous appelez le destrcuteur ainsi pour un objet qui a été créé sur le tas, le destructeur s'exécutera, mais la mémoire ne sera pas libérée, ce qui n'est probablement pas ce que vous désirez. La seule raison pour laquelle le destructeur peut être appelé explicitement ainsi est pour soutenir la syntaxe de placement de **operator new**.

Il y a également un **operator delete**de placement qui est appelé uniquement si un constructeur pour une expression de placement **new**lance une exception (afin que la mémoire soit automatiquement nettoyée pendant l'exception). L' **operator delete**de placement a une liste d'arguments qui correspond à l' **operator new**de placement qui est appelé avant que le constucteur ne lance l'exception. Ce sujet sera couvert dans le chapitre de gestion des exceptions dans le Volume 2.

13.6 - Résumé

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Il est commode et optimal du point de vue de l'efficacité de créer des objets sur la pile, mais pour résoudre le problème de programmation général vous devez être capables de créer et de détruire des objets à tout moment durant l'exécution d'un programme, spécialement pour réagir à des informations provenant de l'extérieur du programme. Bien que l'allocation dynamique de mémoire du C obtienne du stockage sur le tas, cela ne fournit pas la facilité d'utilisation ni la garantie de construction nécessaire en C++. En portant la création dynamique d'objets au coeur du langage avec **new**et **delete**, vous pouvez créer des objets sur le tas aussi facilement que sur la pile. En outre, vous profitez d'une grande flexibilité. Vous pouvez modifier le comportement de **new**et **delete**s'ils ne correspondent pas à vos besoins, particulièrement s'ils ne sont pas suffisamment efficaces. Vous pouvez aussi modifier ce qui se produit quand l'espace de stockage du tas s'épuise.

13.7 - Exercices

Les solutions aux exercices choisis peuvent être trouvées dans le document électronique *The Thinking in C++ Annotated Solution Guide*, disponible à un coût modeste sur www.BruceEckel.com.

1 Créez une **classe Counted**qui contient un **int id**et un **static int count**. Le constructeur par défaut devrait commencer ainsi : **Counted( ) : id(count++) {**. Il devrait aussi afficher son **id**et qu'il est en train d'être créé. Le destructeur devrait afficher qu'il est en train d'être détruit et son **id**. Testez votre classe.

2 Prouvez vous à vous-même que **new**et **delete**appellent toujours les constructeurs et les destructeurs en créant un objet de la **class Counted**(de l'exercice 1) avec **new**et en le détruisant avec **delete**. Créez et détruisez également un tableau de ces objets sur le tas.

3 Créez un objet **PStash**et remplissez le avec de nouveaux objets de l'exercice 1. Observez ce qui se passe quand cet objet **PStash**disparaît de la portée et que son destructeur est appelé.

4 Créez un **vector<Counted*>**et remplissez le avec des pointeurs vers des nouveaux objets **Counted**(de l'exercice 1). Parcourez le **vector**et affichez les objets **Counted**, ensuite parcourez à nouveau le **vector**et detruisez chacun d'eux.

5 Répétez l'exercice 4, mais ajoutez une fonction membre **f( )**à **Counted**qui affiche un message. Parcourez le **vector**et appelez **f( )**pour chaque objet.

6 Répétez l'exercice 5 en utilisant un **PStash**.

7 Répétez l'exercice 5 en utilisant **Stack4.h**du chapitre 9.

8 Créez dynamiquement un tableau d'objets **class Counted**(de l'exercice 1). Appelez **delete**pour le pointeur résultant, *sans les crochets*. Expliquez les résultats.

9 Créez un objet de **class Counted**(de l'exercice 1) en utilisant **new**, transtypez le pointeur résultant en un **void***, et détruisez celui là. Expliquez les résultats.

10 Executez **NewHandler.cpp**sur votre machine pour voir le compte résultant. Calculez le montant d'espace libre (free store ndt) disponible pourvotre programme.

11 Créez une classe avec des opérateurs **new**et **delete**surchargés, à la fois les versions 'objet-unique' et les versions 'tableau'. Démontrez que les deux versions fonctionnent.

12 Concevez un test pour **Framis.cpp**pour vous montrer approximativement à quel point les versions personnalisées de **new**et **delete**fonctionnent plus vite que les **new**et **delete**globaux.

13 Modifiez **NoMemory.cpp**de façon qu'il contienne un tableau de **int**et de façon qu'il alloue effectivement de la mémoire au lieu de lever l'exception **bad_alloc**. Dans **main( )**, mettez en place une boucle **while**comme celle dans **NewHandler.cpp**pour provoquer un épuisement de la mémoire and voyez ce qui se passe si votre **operator new**ne teste pas pour voir si la mémoire est allouée avec succès. Ajoutez ensuite la vérification à votre **operator new**et jetez **bad_alloc**.

14 Créez une classe avec un **new**de placement avec un second argument de type **string**. La classe devrait contenir un **static vector<string>**où le second argument de **new**est stocké. The **new**de placement devrait allouer de l'espace comme d'habitude. Dans **main( )**, faites des appels à votre **new**de placement avec des arguments **string**qui décrivent les appels (Vous pouvez vouloir utiliser les macros du préprocesseur __**FILE**__et __**LINE**__).

15 Modifiez **ArrayOperatorNew.cpp**en ajoutant un **static vector<Widget*>**qui ajoute chaque adresse **Widget**qui est allouée dans **operator new( )**et l'enlève quand il est libéré via l' **operator delete( )**. (Vous pourriez avoir besoin de chercher de l'information sur **vector**dans la documentation de votre Librairie Standard C++ ou dans le second volume de ce livre, disponible sur le site Web.) Créez une seconde classe

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

appelée **MemoryChecker**ayant un destructeur qui affiche le nombre de pointeurs **Widget**dans votre **vector**. Créez un programme avec une unique instance globale de **MemoryChecker**et dans **main( )**, allouez dynamiquement et détruisez plusieurs objets et tableaux de **Widget**. Montrez que **MemoryChecker**révèle des fuites de mémoire.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

14 - Héritage & composition

Une des caractéristiques les plus contraignantes du C++ est la réutilisation du code. Mais pour être révolutionnaire, il vous faut être capable de faire beaucoup plus que copier du code et le modifier.

C'est l'approche du C, et ça n'a pas très bien marché. Comme avec presque tout en C++, la solution tourne autour de la classe. Vous réutilisez le code en créant de nouvelles classes, mais au lieu de les créer à partir de rien, vous utilisez des classes existantes que quelqu'un d'autre a écrites et déboguées.

Le truc consiste à utiliser les classes sans polluer le code existant. Dans ce chapitre, vous verrez deux façons d'accomplir cela. La première est tout à fait immédiate : Vous créez simplement des objets de vos classes existantes dans la nouvelle classe. Ceci est appelé *composition*parce que la nouvelle classe est composée d'objets de classes existantes.

La seconde approche est plus subtile. Vous créez une nouvelle classe comme une *sorte de*classe existante. Vous prenez littéralement la forme de la classe existante et vous lui ajoutez du code, sans modifier la classe existante. Ce processus magique est appelé *héritage*, et la majorité du travail est fait par le compilateur. L'héritage est l'une des pierres angulaires de la programmation orientée objet et a d'autres implications qui seront explorées dans le chapitre 15.

Il se trouve que l'essentiel de la syntaxe et du comportement sont similaires pour la composition et l'héritage (ce qui est raisonnable ; ce sont deux manières de fabriquer de nouveaux types à partir de types existants). Dans ce chapitre, vous en apprendrez plus sur ces mécanismes de réutilisation de code.

14.1 - Syntaxe de la composition

En fait vous avez toujours utilisé la composition pour créer des classes. Vous avez simplement composé des classes de façon primaire avec des types pré-définis (et parfois des objets **string**). Il s'avère être presque aussi facile d'utiliser la composition avec des types personnalisés.

Considérez une classe qui est intéressante pour quelque raison :

```
                    //: C14:Useful.h
// Une classe à réutiliser
#ifndef USEFUL_H
#define USEFUL_H

class X {
  int i;
public:
  X() { i = 0; }
  void set(int ii) { i = ii; }
  int read() const { return i; }
  int permute() { return i = i * 47; }
};
#endif // USEFUL_H ///:~
```

Les données membres sont **private**dans cette classe, de la sorte il est sans danger d'inclure un objet de type **X**en temps qu'objet **public**dans une nouvelle classe, ce qui rend l'interface immédiate :

```
                    //: C14:Composition.cpp
// Réutilisation de code par composition
#include "Useful.h"

class Y {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  int i;
public:
  X x; // Objet embarqué
  Y() { i = 0; }
  void f(int ii) { i = ii; }
  int g() const { return i; }
};

int main() {
  Y y;
  y.f(47);
  y.x.set(37); // Accès à l'objet embarqué
} ///:~
```

L'accession aux fonctions membres de l'objet embarqué (désigné comme un *sous-objet*) ne nécessite qu'une autre sélection de membre.

Il est plus habituel de rendre les objets embarqués **private**, de sorte qu'ils deviennent partie de l'implémentation sous-jacente (ce qui signifie que vous pouvez changer l'implémentation si vous voulez). Les fonctions d'interface **public**pour votre nouvelle classe impliquent alors l'utilisation de l'objet embarqué, mais elles n'imitent pas forcément l'interface de l'objet :

```
                    //: C14:Composition2.cpp
// Objets privés embarqués
#include "Useful.h"

class Y {
  int i;
  X x; // Objet embarqué
public:
  Y() { i = 0; }
  void f(int ii) { i = ii; x.set(ii); }
  int g() const { return i * x.read(); }
  void permute() { x.permute(); }
};

int main() {
  Y y;
  y.f(47);
  y.permute();
} ///:~
```

Ici, la fonction **permute( )**est transportée dans l'interface de la nouvelle classe, mais les autres fonctions membres de **X**sont utilisés dans les membres de **Y**.

14.2 - Syntaxe de l'héritage

La syntaxe pour la composition est évidente, mais pour réaliser l'héritage il y a une forme différente et nouvelle.

Lorsque vous héritez, vous dites, “Cette nouvelle classe est comme l'ancienne.” Vous posez cela en code en donnant le nom de la classe comme d'habitude, mais avant l'accolade ouvrante du corps de la classe, vous mettez un ':' et le nom de la *classe de base*(ou des *classes de base*, séparées par virgules, pour l'héritage multiple). Lorsque vous faites cela, vous obtenez automatiquement toutes les données membres et toutes les fonctions membres dans la classe de base. Voici un exemple:

```
                    //: C14:Inheritance.cpp
// Héritage simple
#include "Useful.h"
#include <iostream>
using namespace std;

class Y : public X {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  int i; // Différent du i de X
public:
  Y() { i = 0; }
  int change() {
    i = permute(); // Appel avec nom différent
    return i;
  }
  void set(int ii) {
    i = ii;
    X::set(ii); // Appel avec homonyme
  }
};

int main() {
  cout << "sizeof(X) = " << sizeof(X) << endl;
  cout << "sizeof(Y) = "
       << sizeof(Y) << endl;
  Y D;
  D.change();
  // l'interface fonctionelle de X intervient:
  D.read();
  D.permute();
  // Les fonctions redéfinies masquent les anciennes:
  D.set(12);
} ///:~
```

Vous pouvez voir **Y**héritant de **X**, comme signifiant que **Y**contiendra toutes les données de **X**et toutes les fonctions membres de **X**. En fait, **Y**contient un sous-objet de type **X**juste comme si vous aviez créé un objet membre de type **X**à l'intérieur de **Y**au lieu d'hériter de **X**. A la fois les objets membres et les classes de bases sont désignés comme sous-objets.

Tous les éléments **private**de **X**sont toujours **private**dans **Y**; c'est à dire que, le simple fait de dériver **Y**de **X**ne signifie pas que **Y**peut briser le mécanisme de protection . Les éléments **private**de **X**sont toujours là, ils occupent de l'espace – vous ne pouvez pas y accéder directement.

Dans **main( )**vous pouvez voir que les données membres de **Y**sont combinés avec ceux de **X**parce que **sizeof(Y)**est deux fois plus gros que **sizeof(X)**.

Vous remarquerez que le nom de la classe de base est précédé de **public**. Pour l'héritage, tout est par défaut fixé à **private**. Si la classe de base n'était pas précédée par **public**, cela signifierait que tous les membres **public**de la classe de base seraient **private**dans la classe dérivée. Ce n'est presque jamais ce que vous souhaitez En Java, le compilateur ne vous laissera pas décroître l'accès à un membre pendant le processus d'héritage.; le résultat désiré est de conserver tous les membres **public**de la classe de base comme **public**dans la classe dérivée. Vous faites cela en utilisant le mot-clé **public**pendant l'héritage.

Dans **change( )**, la fonction de la classe de base **permute( )**est appelée. La classe dérivée à un accès direct à toutes les fonctions de la classe de base **public**.

La fonction **set( )**dans la classe dérivée *redéfinit*la fonction **set( )**dans la classe de base. C'est à dire que, si vous appelez les fonctions **read( )**et **permute( )**pour un objet de type **Y**, vous aurez les versions de la classe de base; de ces fonctions (vous pouvez voir cela se produire dans **main( )**). Mais si vous appelez **set( )**pour un objet **Y**, vous obtenez la version redéfinie. Cecla signifie que si vous n'aimez pas la version d'une fonction que vous obtenez par héritage, vous pouvez changer ce qu'elle fait. (Vous pouvez aussi ajouter complètement de nouvelles fonctions comme **change( )**.)

Toutefois, quand vous redéfinissez une fonction, vous pouvez toujours vouloir appeler la version de la classe de base. Si, à l'intérieur de **set( )**,vous appelez simplement **set( )**vous obtiendrez pour la version locale de la fonction – un appel récursif. Pour appeler la version de la classe de base, vous devez explicitement nommer la classe de base en utilisant l'opérateur de résolution de portée.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

14.3 - La liste d'initialisation du construteur

Vous avez vu comme il est important en C++ de garantir une initialisation correcte, et ce n'est pas différent pour la composition et l'héritage. Quand un objet est créé, le compilateur garantit que les constructeurs pour tous ses sous-objets sont appelés. Dans les exemples jusqu'ici, tous les sous-objets ont des constructeurs par défaut, et c'est ce que le compilateur appelle automatiquement. Mais que se passe-t-il si vos sous-objets n'ont pas de constructeurs par défaut, ou si vous voulez changer un argument par défaut dans un constructeur ? C'est un problème parce que le constructeur de la nouvelle classe n'a pas la permission d'accéder aux éléments de données **private**du sous-objet, de sorte qu'il ne peut les initialiser directement.

La solution est simple: Appeler le constructeur pour le sous-objet. C++ propose une syntaxe spéciale pour cela, la *liste de l'initialisation du constructeur*. La forme de la liste d'initialisation du constructeur se fait l'écho du processus d'héritage. Avec l'héritage, vous placez les classes de base après un ':' et avant l'accolade ouvrante du corps de la classe. Dans la liste d'initialisation du constructeur, vous placez les appels aux constructeurs des sous-objets après la liste d'arguments du constructeur et un ':', mais avant l'accolade ouvrante du corps de la fonction. Pour une classe **MyType**, dérivant de **Bar**, cela peut ressembler à ceci :

```
MyType::MyType(int i) : Bar(i) { // ...
```

si **Bar**a un constructeur qui prend un unique argument **int**.

14.3.1 - Initialisation d'un objet membre

Il s'avère que vous utilisez exactement la même syntaxe pour l'initialisation d'objets membres quand vous utilisez la composition. Pour la composition, vous donnez les noms des objets au lieu de noms de classes. Si vous avez plus d'un appel de constructeur dans une liste d'initialisation, vous séparez les appels avec des virgules :

```
MyType2::MyType2(int i) : Bar(i), m(i+1) { // ...
```

C'est le début d'un constructeur pour la classe **MyType2**, qui est hérité de **Bar**et contient un objet membre appelé **m**. Notez que tandis que vous pouvez voir le type de la classe de base dans la liste d'initialisation du constructeur, vous ne voyez que les identificateurs des objets membres.

14.3.2 - Types prédéfinis dans la liste d'initialisation

La liste d'initialisation du constructeur vous permet d'appeler explicitement les constructeurs pour les objets membres. En fait,il n'y a pas d'autre façon d'appeler ces constructeurs. L'idée est que les constructeurs sont tous appelés avant d'entrer dans le corps du constructeur de la nouvelle classe. De la sorte, tous les appels que vous faites à des fonctions membres de sous-objets iront toujours à des objets initialisés. Il n'y a aucun moyen d'accéder à la parenthèse ouvrante du constructeur sans qu'un appel soit fait à *quelque*constructeur pour tous les objets membres et tous les objets des classes de base, même si le compilateur doit faire un appel caché à un constructeur par défaut. C'est un renforcement supplémentaire de la garantie que C++ donne qu'aucun objet (ou partie d'un objet) ne peut prendre le départ sans que son constructeur n'ait été appelé.

L'idée que tous les objets membres sont initialisés au moment où on atteint l'accolade ouvrante est également une aide pratique à la programmation. Une fois que vous atteignez l'accolade ouvrante, vous pouvez supposez que tous les sous-objets sont correctement initialisés et vous concentrer sur les tâches spécifiques que vous voulez voir réalisées par le constructeur. Cependant, il y a un accroc : qu'en est-il des objets membres de types prédéfinis, qui n'ont *pas*de constructeurs ?

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Pour rendre la syntaxe cohérente, vous avez l'autorisation de traiter les types prédéfinis comme s'ils avaient un constructeur unique, qui prend un simple argument : une variable du même type que la variable que vous initialisez. De la sorte vous pouvez dire

```
                          //: C14:PseudoConstructor.cpp
class X {
  int i;
  float f;
  char c;
  char* s;
public:
  X() : i(7), f(1.4), c('x'), s("allo") {}
};

int main() {
  X x;
  int i(100);  // Appliqué aux définitions ordinaires
  int* ip = new int(47);
} ///:~
```

L'action de ces “appels à des pseudo-construteurs” est de réaliser une simple affectation. C'est une technique pratique et un bon style de codage, aussi vous le verrez souvent utilisé.

Il est même possible d'utiliser la syntaxe du pseudo-constructeur pour créer à l'extérieur d'une classe une variable d'un type prédéfini :

```
                          int i(100);
int* ip = new int(47);
```

Cela fait que les types prédéfinis se comportent un peu plus comme des objets. Souvenez vous, cependant, que ce ne sont pas de vrais constructeurs. En particulier, si vous ne faites pas explicitement un appel à un pseudo-constructeur, aucune initialisation n'a lieu.

14.4 - Combiner composition & héritage

Bien sûr, vous pouvez utiliser la composition et l'héritage ensemble. L'exemple suivant montre la création d'une classe plus complexe utilisant l'un et l'autre.

```
                    //: C14:Combined.cpp
// Héritage & composition

class A {
  int i;
public:
  A(int ii) : i(ii) {}
  ~A() {}
  void f() const {}
};

class B {
  int i;
public:
  B(int ii) : i(ii) {}
  ~B() {}
  void f() const {}
};

class C : public B {
  A a;
public:
  C(int ii) : B(ii), a(ii) {}
  ~C() {} // Appelle ~A() and ~B()
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  void f() const {  // Redéfinition
    a.f();
    B::f();
  }
};

int main() {
  C c(47);
} ///:~
```

Chérite de **B**et a un objet membre (“est composé de ”) de type **A**. Vous pouvez voir que la liste d'initialisation du constructeur contient des appels au constructeur de la classe de base et le constructeur du membre objet.

La fonction **C::f( )**redéfinit **B::f( )**, de laquelle elle hérite, et appelle également la version de la classe de base. De plus, elle appelle **a.f( )**. Notez que le seul moment où vous pouvez parler de la redéfinition de fonctions est pendant l'héritage ; avec un objet membre vous ne pouvez que manipuler l'interface publique de l'objet, pas la redéfinir. De plus, appeler **f( )**pour un objet de classe **C**n'appelerait pas **a.f( )**si **C::f( )**n'avait pas été défini, tandis que cela *appelerait* **B::f( )**.

Appels au destructeur automatique

Bien qu'il vous faille souvent faire des appels explicites aux constructeurs dans la liste d'initialisation, vous n'avez jamais besoin de faire des appels explicites aux destructeurs parce qu'il n'y a qu'un seul destructeur pour toute classe, et il ne prend aucun argument. Toutefois le compilateur assure encore que tous les destructeurs sont appelés, et cela signifie tous les destructeurs dans toute la hiérarchie, en commençant avec le destructeur le plus dérivé et en remontant à la racine.

Il est important d'insister sur le fait que les constructeurs et les destructeurs sont tout à fait inhabituels en cela en cela que chacun dans la hiérarchie est appelé, tandis qu'avec une fonction membre normale seulement cette fonction est appelée, mais aucune des versions des classes de base. Si vous voulez appeler aussi la version de la classe de base d'une fonction membre normale que vous surchargez, vous devez le faire explicitement.

14.4.1 - Ordre des appels des constructeurs & et des destructeurs

Il est intéressant de connaître l'ordre des appels de constructeurs et de destructeurs quand un objet a de nombreux sous-objets. L'exemple suivant montre exactement comment ça marche :

```
                        //: C14:Order.cpp
// Ordre des constructeurs/destructeurs
#include <fstream>
using namespace std;
ofstream out("order.out");

#define CLASS(ID) class ID { \
public: \
  ID(int) { out << #ID " : constructeur\n"; } \
  ~ID() { out << #ID " : destructeur\n"; } \
};

CLASS(Base1);
CLASS(Member1);
CLASS(Member2);
CLASS(Member3);
CLASS(Member4);

class Derived1 : public Base1 {
  Member1 m1;
  Member2 m2;
public:
  Derived1(int) : m2(1), m1(2), Base1(3) {
    out << "Derived1 : constructeur\n";
  }
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  ~Derived1() {
    out << "Derived1 : destructeur\n";
  }
};

class Derived2 : public Derived1 {
  Member3 m3;
  Member4 m4;
public:
  Derived2() : m3(1), Derived1(2), m4(3) {
    out << "Derived2 : constructeur\n";
  }
  ~Derived2() {
    out << "Derived2 : destructeur\n";
  }
};

int main() {
  Derived2 d2;
} ///:~
```

Tout d'abord, un objet **ofstream**est créé pour envoyer toute la sortie vers un fichier. Ensuite, pour éviter un peu de frappe et démontrer une technique de macro qui sera remplacée par une technique nettement améliorée dans le chapitre 16, une macro est créée pour construire certaines des classes, qui sont ensuite utilisées dans l'héritage et la composition. Chacun des constructeurs et des destructeurs se signale au fichier de traçage. Notez que les constructeurs ne sont pas les constructeurs par défaut ; chacun d'eux a un argument de type **int**. L'argument lui-même n'a pas d'identificateur ; sa seule raison d'être est de vous forcer à appeler explicitement les constructeurs dans la liste d'initialisation. (Eliminer l'identificateur a pour objet de supprimer les messages d'avertissement du compilateur.)

La sortie de ce programme est

```
                        Base1 : constructeur
Member1 : constructeur
Member2 : constructeur
Derived1 : constructeur
Member3 : constructeur
Member4 : constructeur
Derived2 : constructeur
Derived2 : destructeur
Member4 : destructeur
Member3 : destructeur
Derived1 : destructeur
Member2 : destructeur
Member1 : destructeur
Base1 : destructeur
```

Vous pouvez voir que la construction démarre à la racine de la hiérarchie de classe, et qu'à chaque niveau le constructeur de la classe de base est appelé d'abord, suivi par les constructeurs des objets membres. Les destructeurs sont appelés exactement en ordre inverse des constructeurs – c'est important à cause des dépendances potentielles (dans le constructeur ou le destructeur de la classe dérivée, vous devez pouvoir supposer que le sous-objet de la classe de base est toujours disponible, et a déjà été construit – ou bien n'est pas encore détruit).

Il est également intéressant que l'ordre des appels de constructeurs pour les objets membres n'est pas du tout affecté par l'ordre des appels dans la liste d'initialisation du constructeur. L'ordre est déterminé par l'ordre dans lequel les objets membres sont déclarés dans la classe. Si vous pouviez changer l'ordre des appels de constructeurs au moyen de la liste d'initialisation du constructeur, vous pourriez avoir deux suites d'appels différentes dans deux constructeurs différents, mais le pauvre destructeur ne saurait pas comment renverser correctement l'ordre des appels pour la destruction, et vous pourriez finir avec un problème de dépendance.

14.5 - Masquage de nom

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Si vous dérivez une classe et fournissez une nouvelle définition pour une de ses fonctions membres, il y a deux possibilités. La première est que vous fournissez exactement la même signature et le même type de retour dans la définition de la classe dérivée que dans la définition de la classe de base. On parle alors de *redéfinition*(*redefining*en anglais) d'une fonction membre dans le cas d'une fonction membre ordinaire, et de *supplantation*(*overriding*en anglais) d'une fonction membre quand celle de la classe de base est **virtual** En français, il est courant et généralement admis d'utiliser le terme *redéfinir*dans les deux cas, *supplanter*étant d'un usage nettement moins répandu. En conséquence, le terme *supplanter*n'a pas été retenu dans le cadre de la traduction de cet ouvrage, et le terme *redéfinir*est utilisé indifféremment dans les deux cas. (les fonctions **virtual**sont le cas normal, et seront traitées en détail dans le chapitre 15). Mais que se passe-t-il si vous changez la liste d'arguments ou le type de retour d'une fonction membre dans la classe dérivée ? Voici un exemple :

```
                    //: C14:NameHiding.cpp
// Masquage de noms par héritage
#include <iostream>
#include <string>
using namespace std;

class Base {
public:
  int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  int f(string) const { return 1; }
  void g() {}
};

class Derived1 : public Base {
public:
  void g() const {}
};

class Derived2 : public Base {
public:
  // Redéfinition :
  int f() const {
    cout << "Derived2::f()\n";
    return 2;
  }
};

class Derived3 : public Base {
public:
  // Change le type de retour:
  void f() const { cout << "Derived3::f()\n"; }
};

class Derived4 : public Base {
public:
  // Change la liste d'arguments :
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
};

int main() {
  string s("salut");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // version chaîne cachée
  Derived3 d3;
//!  x = d3.f(); // version à retour entier cachée
  Derived4 d4;
//!  x = d4.f(); // f() version cachée
  x = d4.f(1);
} ///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Dans **Base**vous voyez une fonction redéfinie **f( )**, et **Derived1**ne fait aucun changement à **f( )**mais redéfinit **g( )**. Dans **main( )**, vous pouvez voir que les deux versions surchargées de **f( )**sont disponibles dans **Derived1**. Toutefois, **Derived2**redéfinit une version surchargée de **f( )**mais pas l'autre, et le résultat est que la seconde forme surchargée n'est pas disponible. Dans **Derived3**, changer le type de retour masque les deux versions de la classe de base, et **Derived4**montre que changer la liste d'arguments masque également les deux versions de la classe de base. En général, nous pouvons dire que chaque fois que vous redéfinissez un nom de fonction de la classe de base, toutes les autres versions sont automatiquement masquées dans la nouvelle classe. Dans le chapitre 15, vous verrez que l'addition du mot clé **virtual**affecte un peu plus la surcharge des fonctions.

Si vous changez l'interface de la classe de base en modifiant la signature et/ou le type de retour d'une fonction membre de la classe de base, alors vous utilisez la classe d'une manière différente de celle supposée être normalement supportée par le processus d'héritage. Cela ne signifie pas nécessairement que vous ayez tort de procéder ainsi, c'est juste que le but fondamental de l'héritage est de permettre le *polymorphisme*, et si vous changez la signature ou le type de retour de la fonction, alors vous changez vraiment l'interface de la classe de base. Si c'est ce que vous aviez l'intention de faire alors vous utilisez l'héritage principalement pour réutiliser le code, et non pour maintenir l'interface commune de la classe de base (ce qui est un aspect essentiel du polymorphisme). En général, quand vous utilisez l'héritage de cette manière cela signifie que vous prenez une classe généraliste et que vous la spécialisez pour un besoin particulier – ce qui est considéré d'habitude, mais pas toujours, comme le royaume de la composition.

Par exemple, considérez la classe **Stack**du chapitre 9. Un des problèmes avec cette classe est qu'il vous fallait réaliser un transtypage à chaque fois que vous récupériez un pointeur du conteneur. C'est non seulement fastidieux, c'est aussi dangereux – vous pourriez transtyper en tout ce que vous voudriez.

Une approche qui semble meilleure à première vue consiste à spécialiser la classe générale **Stack**en utilisant l'héritage. Voici un exemple qui utilise la classe du chapitre 9 :

```
                    //: C14:InheritStack.cpp
// Spécialisation de la classe Stack
#include "../C09/Stack4.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class StringStack : public Stack {
public:
  void push(string* str) {
    Stack::push(str);
  }
  string* peek() const {
    return (string*)Stack::peek();
  }
  string* pop() {
    return (string*)Stack::pop();
  }
  ~StringStack() {
    string* top = pop();
    while(top) {
      delete top;
      top = pop();
    }
  }
};

int main() {
  ifstream in("InheritStack.cpp");
  assure(in, "InheritStack.cpp");
  string line;
  StringStack textlines;
  while(getline(in, line))
    textlines.push(new string(line));
  string* s;
  while((s = textlines.pop()) != 0) { // Pas de transtypage !
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
      cout << *s << endl;
      delete s;
    }
  } ///:~
```

Etant donné que toutes les fonctions membres dans **Stack4.h**sont 'inline', aucune édition de liens n'est nécessaire.

StringStackspécialise **Stack**de sorte que **push( )**n'acceptera que des pointeurs **String**. Avant, **Stack**acceptait des pointeurs **void**, aussi l'utilisateur n'avait pas de vérification de type à faire pour s'assurer que des pointeurs corrects étaient insérés. De plus, **peek( )**et **pop( )**retournent maintenant des pointeurs **String**au lieu de pointeurs **void**, de sorte qu'aucun transtypage n'est nécessaire pour utiliser le pointeur.

Aussi étonnant que cela puisse paraître, cette sécurité de contrôle de type supplémentaire est gratuite dans **push( )**, **peek( )**, et **pop( )**! Le compilateur reçoit une information de type supplémentaire qu'il utilise au moment de la compilation, mais les fonctions sont 'inline' et aucun code supplémentaire n'est généré.

Le masquage de nom entre en jeu ici parce que, en particulier, la fonction **push( )**a une signature différente : la liste d'arguments est différente. Si vous aviez deux versions de **push( )**dans la même classe, cela serait de la redéfinition, mais dans ce cas la redéfinition n'est *pas*ce que nous voulons parce que cela vous permettrait encore de passer n'importe quel type de pointeur dans **push( )**en tant que **void***. Heureusement, C++ cache la version de **push(void*)**dans la classe de base au profit de la nouvelle version qui est définie dans la classe dérivée, et pour cette raison, il nous permet seulement de **push(er)( )**des pointeurs **string**sur la **StringStack**.

Parce que nous pouvons maintenant garantir que nous savons exactement quels types d'objets sont dans le conteneur, le destructeur travaille correctement et le problème la propriété est résolu – ou au moins, un aspect de ce problème. Ici, si vous **push(ez)( )**un pointeur **string**sur la **StringStack**, alors (en accord avec la sémantique de la **StringStack**) vous passez également la propriété de ce pointeur à la **StringStack**. Si vous **pop(ez)( )**le pointeur, non seulement vous obtenez l'adresse, mais vous récupérez également la propriété de ce pointeur. Tous les pointeurs laissés sur la **StringStack**, lorsque le destructeur de celle-ci est appelé sont alors détruits par ce destructeur. Et comme ce sont toujours des pointeurs **string**et que l'instruction **delete**travaille sur des pointeurs **string**au lieu de pointeurs **void**, la destruction adéquate a lieu et tout fonctionne correctement.

Il y a un inconvénient : cette classe travaille *seulement*avec des pointeurs **string**. Si vous voulez une **Stack**qui travaille avec d'autres sortes d'objets, vous devez écrire une nouvelle version de la classe qui travaillera seulement avec votre nouvelle sorte d'objets. Cela devient rapidement fastidieux, et finit par se résoudre avec des templates, comme vous le verrez dans le chapitre 16.

Nous pouvons faire une remarque supplémentaire à propos de cet exemple : il change l'interface de la **Stack**dans le processus d'héritage. Si l'interface est différente, alors une **StringStack**n'est pas vraiment une **Stack**, et vous ne pourrez jamais utiliser correctement une **StringStack**en tant que **Stack**. Ce qui fait qu'ici, l'utilisation de l'héritage peut être mise en question ; si vous ne créez pas une **StringStack**qui *est-une*sorte de **Stack**, alors pourquoi héritez vous ? Une version plus adéquate de **StringStack**sera montrée plus loin dans ce chapitre.

14.6 - Fonctions qui ne s'héritent pas automatiquement

Toutes les fonctions de la classe de base ne sont pas automatiquement héritées par la classe dérivée. Les constructeurs et destructeurs traitent la création et à la destruction d'un objet, et ils ne peuvent savoir quoi faire qu'avec les aspects de l'objet spécifiques à leur classe particulière, si bien que tous les constructeurs et destructeurs dans la hiérarchie en dessous d'eux doivent être appelés. Ainsi, les constructeurs et destructeurs ne s'héritent pas et doivent être créés spécialement pour chaque classe dérivée.

En plus, le **operator=**ne s'hérite pas parce qu'il réalise une opération similaire à celle d'un constructeur. C'est-à-dire que le fait que vous sachiez comment affecter tous les membres d'un objet du côté gauche du **=**depuis

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

un objet situé à droite ne veut pas dire que l'affectation aura encore le même sens après l'héritage.

Au lieu d'être héritées, ces fonctions sont synthétisées par le compilateur si vous ne les créez pas vous-même. (Avec les constructeurs, vous ne pouvez créer *aucun*constructeur afin que le compilateur synthétise le constructeur par défaut et le constructeur par recopie.) Ceci a été brièvement décrit au Chapitre 6. Les constructeurs synthétisés utilisent l'initialisation membre à membre et l' **operator=**synthétisé utilise l'affectation membre à membre. Voici un exemple des fonctions qui sont synthétisées par le compilateur :

```
                    //: C14:SynthesizedFunctions.cpp
// Fonctions qui sont synthétisées par le compilateur
#include <iostream>
using namespace std;

class GameBoard {
public:
  GameBoard() { cout << "GameBoard()\n"; }
  GameBoard(const GameBoard&) {
    cout << "GameBoard(const GameBoard&)\n";
  }
  GameBoard& operator=(const GameBoard&) {
    cout << "GameBoard::operator=()\n";
    return *this;
  }
  ~GameBoard() { cout << "~GameBoard()\n"; }
};

class Game {
  GameBoard gb; // Composition
public:
  // Appel au contructeur de GameBoard par défaut :
  Game() { cout << "Game()\n"; }
  // Vous devez explicitement appeler le constructeur par recopie
  // de GameBoard sinon le constructeur par défaut
  // est appelé automatiquement à la place :
  Game(const Game& g) : gb(g.gb) {
    cout << "Game(const Game&)\n";
  }
  Game(int) { cout << "Game(int)\n"; }
  Game& operator=(const Game& g) {
    // Vous devez appeler explicitement l'opérateur
    // d'affectation de GameBoard ou bien aucune affectation
    // ne se produira pour gb !
    gb = g.gb;
    cout << "Game::operator=()\n";
    return *this;
  }
  class Other {}; // Classe imbriquée
  // Conversion de type automatique :
  operator Other() const {
    cout << "Game::operator Other()\n";
    return Other();
  }
  ~Game() { cout << "~Game()\n"; }
};

class Chess : public Game {};

void f(Game::Other) {}

class Checkers : public Game {
public:
  // Appel au constructeur par défaut de la classe de base :
  Checkers() { cout << "Checkers()\n"; }
  // Vous devez appeler explicitement le constructeur par recopie
  // de la classe de base ou le constructeur par défaut
  // sera appelé à sa place :
  Checkers(const Checkers& c) : Game(c) {
    cout << "Checkers(const Checkers& c)\n";
  }
  Checkers& operator=(const Checkers& c) {
    // Vous devez appeler explicitement la version de la
    // classe de base de l'opérateur=() ou aucune affectation
    // de la classe de base ne se produira :
    Game::operator=(c);
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    cout << "Checkers::operator=()\n";
    return *this;
  }
};

int main() {
  Chess d1;  // Constructeur par défaut
  Chess d2(d1); // Constructeur par recopie
//! Chess d3(1); // Erreur : pas de constructeur de int
  d1 = d2; // Operateur= synthétisé
  f(d1); // La conversion de type EST héritée
  Game::Other go;
//!  d1 = go; // Operateur= non synthétisé
           // pour les types qui diffèrent
  Checkers c1, c2(c1);
  c1 = c2;
} ///:~
```

Les constructeurs et l' **operator=**pour **GameBoard**et **Game**se signalent eux-mêmes si bien que vous pouvez voir quand ils sont utilisés par le compilateur. En plus, l' **operator Other( )**réalise une conversion de type automatique depuis un objet **Game**vers un objet de la classe imbriquée **Other**. La classe **Chess**hérite simplement de **Game**et ne crée aucune fonction (pour voir comment réagit le compilateur). La fonction **f( )**prend un objet **Other**pour tester la fonction de conversion de type automatique.

Dans **main( )**, le constructeur par défaut et le constructeur par recopie synthétisés pour la classe dérivée **Chess**sont appelés. La version **Game**de ces constructeurs est appelée comme élément de la hiérarchie des appels aux constructeurs. Même si cela ressemble à de l'héritage, de nouveaux constructeurs sont vraiment synthétisés par le compilateur. Comme vous pourriez le prévoir, aucun constructeur avec argument n'est automatiquement créé parce que cela demanderait trop d'intuition de la part du compilateur.

L' **operator=**est également synthétisé comme une nouvelle fonction dans **Chess**, utilisant l'affectation membre à membre (ainsi, la version de la classe de base est appelée) parce que cette fonction n'a pas été écrite explicitement dans la nouvelle classe. Et, bien sûr, le destructeur a été automatiquement synthétisé par le compilateur.

A cause de toutes ces règles concernant la réécriture des fonctions qui gèrent la création des objets, il peut paraître un peu étrange à première vue que l'opérateur de conversion de type automatique *soit*hérité. Mais ce n'est pas complètement déraisonnable – s'il y a suffisamment d'éléments dans **Game**pour constituer un objet **Other**, ces éléments sont toujours là dans toutes classe dérivée de **Game**et l'opérateur de conversion de type est toujours valide (même si vous pouvez en fait avoir intérêt à le redéfinir).

operator=est synthétisé *uniquement*pour affecter des objets de même type. Si vous voulez affecter des objets d'un type vers un autre, vous devez toujours écrire cet **operator=**vous-même.

Si vous regardez **Game**de plus près, vous verrez que le constructeur par recopie et les opérateurs d'affectation contiennent des appels explicites au constructeur par recopie et à l'opérateur d'affectation de l'objet membre. Vous voudrez normalement procéder ainsi parce que sinon, dans le cas du constructeur par recopie, le constructeur par défaut de l'objet membre sera utilisé à la place, et dans le cas de l'opérateur d'affectation, *aucune*affectation ne sera faite pour les objets membres !

Enfin, Regardez **Checkers**, qui rédige explicitement le constructeur par défaut, le constructeur par recopie et les opérateurs d'affectation. Dans le cas du constructeur par défaut, le constructeur par défaut de la classe de base est automatiquement appelé, et c'est typiquement ce que vous voulez. Mais, et c'est un point important, dès que vous décidez d'écrire votre propre constructeur par recopie et votre propre opérateur d'affectation, le compilateur suppose que vous savez ce que vous faites et n'appelle *pas*automatiquement les versions de la classe de base, comme il le fait dans les fonctions synthétisées. Si vous voulez que les versions de la classe de base soient appelées (et vous le faites typiquement) alors vous devez les appeler explicitement vous-même. Dans le constructeur par recopie de **Checkers**, cet appel apparaît dans la liste d'initialisation du constructeur :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
Checkers(const Checkers& c) : Game(c) {
```

Dans l'opérateur d'affectation de **Checkers**, l'appel à la classe de base est la première ligne du corps de la fonction :

```
Game::operator=(c);
```

Ces appels devraient être des parties de la forme canonique que vous utilisez à chaque fois que vous faites hériter une classe.

14.6.1 - Héritage et fonctions membres statiques

Les fonctions membres **static**agissent de la même façon que les fonctions membres non- **static**:

1 Elles sont héritées dans la classe dérivée.
2 Si vous redéfinissez un membre statique, toutes les autres fonctions surchargées dans la classe de base se retrouvent cachées.
3 Si vous modifiez la signature d'une fonction de la classe de base, toutes les versions de la classe de base avec ce nom de fonction se retrouvent cachées (c'est en fait une variation sur le point précédent).

Toutefois, les fonctions membres **static**ne peuvent pas être **virtual**(sujet traité en détail au Chapitre 15).

14.7 - Choisir entre composition et héritage

La composition et l'héritage placent tous deux des sous-objets dans votre nouvelle classe. Tous deux utilisent la liste d'initialisation du constructeur pour construire ces sous-objets. A présent, peut-être que vous vous demandez quel est la différence entre les deux, et quand choisir l'un ou l'autre.

La composition est généralement utilisée quand vous voulez trouver les fonctionnalités d'une classe existante au sein de votre nouvelle classe, mais pas son interface. C'est-à-dire que vous enrobez un objet pour implémenter des fonctionnalités de votre nouvelle classe, mais l'utilisateur de votre nouvelle classe voit l'interface que vous avez définie plutôt que l'interface de la classe d'origine. Pour ce faire, vous suivez la démarche type d'implantation d'objets de classes existantes **private**dans votre nouvelle classe.

Parfois, cependant, il est logique de permettre à l'utilisateur de la classe d'accéder directement à la composition de votre nouvelle classe, c'est-à-dire de rendre l'objet membre **public**. Les objets membres utilisent eux-mêmes le contrôle d'accès, et c'est donc une démarche sure et quand l'utilisateur sait que vous assemblez différents morceaux, cela rend l'interface plus compréhensible. Une classe **Car**(Voiture, NdT) est un bon exemple :

```
//: C14:Car.cpp
// Composition publique

class Engine {
public:
  void start() const {}
  void rev() const {}
  void stop() const {}
};

class Wheel {
public:
  void inflate(int psi) const {}
};
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
class Window {
public:
  void rollup() const {}
  void rolldown() const {}
};

class Door {
public:
  Window window;
  void open() const {}
  void close() const {}
};

class Car {
public:
  Engine engine;
  Wheel wheel[4];
  Door left, right; // 2-portes
};

int main() {
  Car car;
  car.left.window.rollup();
  car.wheel[0].inflate(72);
} ///:~
```

Comme la composition d'un **Car**fait partie de l'analyse du problème (et pas simplement de la conception sous-jacente), rendre les membres **public**aide le programmeur-client à comprendre comment utiliser la classe et requiert un code moins complexe de la part du créateur de la classe.

En y réfléchissant un peu, vous verrez aussi que cela n'aurait aucun sens de composer un **Car**en utilisant un objet “Véhicule” – une voiture ne contient pas un véhicule, c' *est*un véhicule. La relation *est un*est exprimée par l'héritage, et la relation *a un*est exprimée par la composition.

14.7.1 - Sous-typer

A présent, supposez que vous vouliez créer un objet de type **ifstream**qui non seulement ouvre un fichier mais en plus conserve le nom du fichier. Vous pouvez utiliser la composition et inclure un objet **ifstream**et un objet **string**dans la nouvelle classe :

```
                        //: C14:FName1.cpp
// Un fstream avec nom de fichier
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class FName1 {
  ifstream file;
  string fileName;
  bool named;
public:
  FName1() : named(false) {}
  FName1(const string& fname)
    : fileName(fname), file(fname.c_str()) {
    assure(file, fileName);
    named = true;
  }
  string name() const { return fileName; }
  void name(const string& newName) {
    if(named) return; // N'écrasez pas
    fileName = newName;
    named = true;
  }
  operator ifstream&() { return file; }
};
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
int main() {
  FName1 file("FName1.cpp");
  cout << file.name() << endl;
  // Erreur: close() n'est pas un membre :
//!  file.close();
} ///:~
```

Il y a un problème, cependant. On essaye de permettre l'utilisation de l'objet **FName1**partout où un objet **ifstream**est utilisé en incluant un opérateur de conversion de type automatique de **FName1**vers un **ifstream&**. Mais dans main, la ligne

```
                        file.close();
```

ne compilera pas car la conversion de type automatique n'a lieu que dans les appels de fonctions, pas pendant la sélection de membre. Cette approche ne fonctionnera donc pas.

Une deuxième approche consiste à ajouter la définition de **close( )**à **FName1**:

```
                        void close() { file.close(); }
```

Ceci ne fonctionnera que si il n'y a que quelques fonctions que vous voulez apporter depuis la classe **ifstream**. Dans ce cas, vous n'utilisez que des parties de la classe et la composition est appropriée.

Mais que se passe-t-il si vous voulez que tous les éléments de la classe soient transposés ? On appelle cela *sous-typage*parce que vous créez un nouveau type à partir d'un type existant, et vous voulez que votre nouveau type ait exactement la même interface que le type existant (plus toutes les autres fonctions membres que vous voulez ajouter), si bien que vous puissiez l'utiliser partout où vous utiliseriez le type existant. C'est là que l'héritage est essentiel. Vous pouvez voir que le sous-typage résoud parfaitement le problème de l'exemple précédent :

```
                        //: C14:FName2.cpp
// Le sous-typage résoud le problème
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class FName2 : public ifstream {
  string fileName;
  bool named;
public:
  FName2() : named(false) {}
  FName2(const string& fname)
    : ifstream(fname.c_str()), fileName(fname) {
    assure(*this, fileName);
    named = true;
  }
  string name() const { return fileName; }
  void name(const string& newName) {
    if(named) return; // N'écrasez pas
    fileName = newName;
    named = true;
  }
};

int main() {
  FName2 file("FName2.cpp");
  assure(file, "FName2.cpp");
  cout << "name: " << file.name() << endl;
  string s;
  getline(file, s); // Cela fonctionne aussi !
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  file.seekg(-200, ios::end);
  file.close();
} ///:~
```

A présent, toute fonction membre disponible pour un objet **ifstream**l'est également pour un objet **FName2**. Vous pouvez constater aussi que des fonctions non membres comme **getline( )**qui attendent un **ifstream**peuvent également fonctionner avec un **FName2**. C'est le cas parce qu'un **FName2** *est*un type d' **ifstream**; il n'en contient pas simplement un. C'est un problème très important qui sera abordé à la fin de ce chapitre et dans le suivant.

14.7.2 - héritage privé

Vous pouvez hériter d'une classe de base de manière privée en laissant de coté le **public**dans la liste des classes de base, ou en disant explicitement **private**(probablement une meilleure stratégie parce qu'alors c'est clair pour l'utilisateur que c'est ce que vous voulez). Quand vous héritez de façon privée, vous “implémentez en termes de” c'est-à-dire que vous créez une nouvelle classe qui a toutes les données et les fonctionnalités de la classe de base, mais ces fonctionnalités sont cachées, si bien qu'elles font seulement partie de l'implémentation sous-jacente. L'utilisateur de la classe n'a aucun accès à la fonctionnalité sous-jacente, et un objet ne peut pas être traité comme une instance de la classe de base (comme c'était le cas dans **FName2.cpp**).

Peut-être vous demandez-vous le but de l'héritage privée, parce que l'alternative d'utiliser la composition pour créer un objet privé dans la nouvelle classe semble plus appropriée. L'héritage privé est inclus dans le langage pour des raisons de complétude, mais ne serait-ce que pour réduire les sources de confusion, vous aurez généralement intérêt à utiliser la composition plutôt que l'héritage privé. Toutefois, il peut y avoir parfois des situations où vous voulez produire une partie de la même interface que la classe de base *et*ne pas autoriser le traitement de l'objet comme s'il était du type de la classe de base. L'héritage privé fournit cette possibilité.

Rendre publics les membres hérités de manière privée

Quand vous héritez de manière privée, toutes les fonctions membres **public**de la classe de base deviennent **private**. Si vous voulez que n'importe lesquelles d'entre elles soit visibles, dites simplement leur nom (sans argument ni valeur de retour) avec le mot-clef **using** dans la section **public**de la classe dérivée :

```
                        //: C14:PrivateInheritance.cpp
class Pet {
public:
  char eat() const { return 'a'; }
  int speak() const { return 2; }
  float sleep() const { return 3.0; }
  float sleep(int) const { return 4.0; }
};

class Goldfish : Pet { // Héritage privé
public:
  using Pet::eat; // Nommez les membres à rendre public
  using Pet::sleep; // Les deux fonctions surchargées sont exposées
};

int main() {
  Goldfish bob;
  bob.eat();
  bob.sleep();
  bob.sleep(1);
//! bob.speak();// Erreur : fonction membre privée
} ///:~
```

Ainsi, l'héritage **privé**est utile si vous voulez dissimuler une partie des fonctionnalités de la classe de base.

Notez qu' exposerle nom d'une fonction surchargée rend publiques toutes les versions de la fonction surchargée dans la classe de base.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Vous devriez bien réfléchir avant d'utiliser l'héritage **privé**au lieu de la composition ; l'héritage **privé**entraîne des complications particulières quand il est combiné avec l'identification de type à l'exécution (RTTI NdT) (c'est le sujet d'un chapitre du deuxième volume de ce livre, téléchargeable depuis *www.BruceEckel.com*).

14.8 - protected

A présent que vous avez été initiés à l'héritage, le mot-clef **protected**prend finalement un sens. Dans un monde idéal, les membres **private**seraient toujours strictement **private**, mais dans les projets réels vous voulez parfois dissimuler quelque chose au monde en général et en même temps autoriser l'accès pour les membres des classes dérivées. Le mot-clef **protected**est une concession au pragmatisme; il signifie “Ceci est **private**en ce qui concerne l'utilisateur de la classe, mais disponible à quiconque hérite de cette classe.”

La meilleure approche est de laisser les données membres **private**– vous devriez toujours préserver votre droit de modifier l'implémentation sous-jacente. Vous pouvez alors autoriser un accès controllé aux héritiers de votre classe grâce aux fonctions membres **protected**:

```
                    //: C14:Protected.cpp
// Le mot-clef protected
#include <fstream>
using namespace std;

class Base {
  int i;
protected:
  int read() const { return i; }
  void set(int ii) { i = ii; }
public:
  Base(int ii = 0) : i(ii) {}
  int value(int m) const { return m*i; }
};

class Derived : public Base {
  int j;
public:
  Derived(int jj = 0) : j(jj) {}
  void change(int x) { set(x); }
};

int main() {
  Derived d;
  d.change(10);
} ///:~
```

Vous trouverez des exemples de la nécessité de **protected**dans des exemples situés plus loin dans ce livre, et dans le Volume 2.

14.8.1 - héritage protégé

Quand vous héritez, la classe de base est **private**par défaut, ce qui signifie que toutes les fonctions membres publiques sont **private**pour l'utilisateur de la nouvelle classe. Normalement, vous rendrez l'héritage **public**afin que l'interface de la classe de base soit également celle de la classe dérivée. Toutefois, vous pouvez également utiliser le mot-clef **protected**pendant l'héritage.

La dérivation protégée signifie “ implémenté en termes de” pour les autres classes mais “est un” pour les classes dérivées et les amis. C'est quelque chose dont on ne se sert pas très souvent, mais cela se trouve dans le langage pour sa complétude.

14.9 - Surcharge d'opérateur & héritage

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

A part l'opérateur d'affectation, les opérateurs sont automatiquement hérités dans une classe dérivée. Ceci peut se démontrer en héritant de **C12:Byte.h**:

```
                    //: C14:OperatorInheritance.cpp
// Hériter des opérateurs surchargés
#include "../C12/Byte.h"
#include <fstream>
using namespace std;
ofstream out("ByteTest.out");

class Byte2 : public Byte {
public:
  // Les constructeurs ne s'héritent pas :
  Byte2(unsigned char bb = 0) : Byte(bb) {}
  // opérateur = ne s'hérite pas, mais
  // est fabriqué pour les assignations liées aux membres.
  // Toutefois, seul l'opérateur SameType = SameType
  // est fabriqué, et vous devez donc
  // fabriquer les autres explicitement :
  Byte2& operator=(const Byte& right) {
    Byte::operator=(right);
    return *this;
  }
  Byte2& operator=(int i) {
    Byte::operator=(i);
    return *this;
  }
};

// Fonction test similaire à celle contenue dans C12:ByteTest.cpp:
void k(Byte2& b1, Byte2& b2) {
  b1 = b1 * b2 + b2 % b1;

  #define TRY2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ";  b1 " #OP " b2 produces "; \
    (b1 OP b2).print(out); \
    out << endl;

  b1 = 9; b2 = 47;
  TRY2(+) TRY2(-) TRY2(*) TRY2(/)
  TRY2(%) TRY2(^) TRY2(&) TRY2(|)
  TRY2(<<) TRY2(>>) TRY2(+=) TRY2(-=)
  TRY2(*=) TRY2(/=) TRY2(%=) TRY2(^=)
  TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)
  TRY2(=) // Assignment operator

  // Instructions conditionnelles :
  #define TRYC2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ";  b1 " #OP " b2 produces "; \
    out << (b1 OP b2); \
    out << endl;

  b1 = 9; b2 = 47;
  TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)
  TRYC2(>=) TRYC2(&&) TRYC2(||)

  // Assignation en chaîne :
  Byte2 b3 = 92;
  b1 = b2 = b3;
}

int main() {
  out << "member functions:" << endl;
  Byte2 b1(47), b2(9);
  k(b1, b2);
} ///:~
```

Le code test est identique à celui utilisé dans **C12:ByteTest.cpp**sauf que **Byte2**est utilisé à la place de **Byte**. De cette façon, on vérifie que tous les opérateurs fonctionnent avec **Byte2**via l'héritage.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Quand vous examinez la classe **Byte2**, vous constatez que le constructeur doit être explicitement défini, et que, seul, **operator=**qui assigne un **Byte2**à un **Byte2**est fabriqué automatiquement; tout autre opérateur d'assignation dont vous avez besoin devra être fabriqué par vos soins.

14.10 - Héritage multiple

Vous pouvez hériter d'une classe, et il semblerait donc logique d'hériter de plus d'une seule classe à la fois. C'est, de fait, possible, mais savoir si cela est cohérent du point de vue de la conception est un sujet de débat perpétuel. On s'accorde généralement sur une chose : vous ne devriez pas essayer de le faire avant d'avoir programmé depuis quelques temps et de comprendre le langage en profondeur. A ce moment-là, vous réaliserez probablement que peu importe à quel point vous pensez avoir absolument besoin de l'héritage multiple, vous pouvez presque toujours vous en tirer avec l'héritage unique.

A première vue, l'héritage multiple semble assez simple : vous ajoutez plus de classes dans la liste des classes de base pendant l'héritage, en les séparant par des virgules. Toutefois, l'héritage multiple introduit nombre de possibilités d'ambigüités, et c'est pourquoi un chapitre est consacré à ce sujet dans le Volume 2.

14.11 - Développement incrémental

Un des avantages de l'héritage et de la composition est qu'ils soutiennent le *développement incrémental*en vous permettant d'introduire du nouveau code sans causer de bug dans le code existant. Si des bugs apparaissent, ils sont restreints au sein du nouveau code. En héritant de (ou composant avec) une classe existante fonctionnelle et en ajoutant des données et fonctions membres (et en redéfinissant des fonctions membres existantes pendant l'héritage) vous ne touchez pas au code existant – que quelqu'un d'autre peut encore être en train d'utiliser – et n'introduisez pas de bugs. Si un bug se produit, vous savez qu'il se trouve dans votre nouveau code, qui est beaucoup plus court et plus facile à lire que si vous aviez modifié le corps du code existant.

La façon dont les classes sont séparées est assez étonnante. Vous n'avez même pas besoin du code source des fonctions membres pour réutiliser le code, simplement du fichier d'en-tête décrivant la classe et le fichier objet ou le fichier de la bibliothèque avec les fonctions membres compilées. (Ceci est vrai pour l'héritage et la composition à la fois.)

Il est important de réaliser que le développement de programmes est un processus incrémental, exatement comme l'apprentissage chez l'homme. Vous pouvez faire toute l'analyse que vous voulez, mais vous ne connaîtrez toujours pas toutes les réponses quand vous démarrerez un projet. Vous aurez plus de succès – et plus de retour immédiat – si vous commencez à faire “croître” votre projet comme une créature organique, évolutionnaire plutôt qu'en le construisant entièrement d'un coup, comme un gratte-ciel Pour en apprendre davantage sur cette idée, reportez vous à *Extreme Programming Explained*, de Kent Beck (Addison-Wesley 2000). .

Bien que l'héritage pour l'expérimentation soit une technique très utile, après que les choses soient un peu stabilisées vous avez besoin d'examiner à nouveau la hiérarchie de votre classe pour la réduire à une structure rationnelle. Cf. *Refactoring: Improving the Design of Existing Code*de Martin Fowler (Addison-Wesley 1999). . Rappelez-vous que par dessus tout cela, l'héritage a pour fonction d'exprimer une relation qui dit, “Cette nouvelle classe est un *type de*cette ancienne classe”. Votre programme ne devrait pas être occupé à pousser des bits ici et là, mais plutôt à créer et manipuler des objets de types variés pour exprimer un modèle dans les termes qui vous sont fixés par l'espace du problème.

14.12 - Transtypage ascendant

Plus tôt dans ce chapitre, vous avez vu comment un objet d'une classe dérivée de **ifstream**possède toutes les caractéristiques et le comportement d'un objet **ifstream**. Dans **FName2.cpp**, toute fonction membre de

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

peut être appelée pour un objet **FName2**.

L'aspect le plus important de l'héritage n'est toutefois pas qu'il fournisse des fonctions membres à la nouvelle classe. C'est la relation exprimée entre la nouvelle classe et la classe de base. Cette relation peut être résumée en disant "La nouvelle classe *est un type de*la classe existante".

Cette description n'est pas simplement une jolie manière d'expliquer l'héritage. En guise d'exemple, considérez une classe de base appelée **Instrument**qui représente les instruments musicaux et une classe dérivée appelée **Wind**(vent, ndt). Comme l'héritage signifie que toutes les fonctions de la classe de base sont aussi disponibles dans la classe dérivée, tout message qui peut être envoyé à la classe de base peut également l'être à la classe dérivée. Donc si la classe **Instrument**a une fonction membre **play( )**(jouer, ndt), **Wind**en aura une également. Ceci signifie que nous pouvons dire avec justesse qu'un objet **Wind**est également un type d' **Instrument**. L'exemple suivant montre comment le compilateur supporte cette notion :

```
                    //: C14:Instrument.cpp
// Héritage & transtypage ascendant
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  void play(note) const {}
};

// Les objets Wind sont des Instruments
// parce qu'ils ont la même interface :
class Wind : public Instrument {};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Transtypage ascendant
} ///:~
```

Ce qui est intéressant dans cet exemple est la fonction **tune( )**, qui accepte une référence **Instrument**. Toutefois, dans **main( )**la fonction **tune( )**est appelée en lui passant une référence vers un objet **Wind**. Etant donné que le C++ est très exigeant avec la vérification de type, il semble surprenant qu'une fonction qui accepte un type en accepte facilement un autre, jusqu'à ce que l'on réalise qu'un objet **Wind**est également un objet **Instrument**, et il n'y a aucune fonction que **tune( )**pourrait appeler pour un **Instrument**qui ne soit pas également dans **Wind**(c'est ce que garantit l'héritage). Dans **tune( )**, le code fonctionne pour **Instrument**et tout ce qui en dérive, et l'action de convertir une référence ou un pointeur **Wind**en une référence ou un pointeur **Instrument**est appelée *transtypage ascendant*(*upcasting*an anglais).

14.12.1 - Pourquoi "ascendant" ?

L'origine du mot est historique et repose sur la façon dont les diagrammes d'héritage des classes sont traditionnellement dessinés : avec la racine en haut de la page, croissant vers le bas. (Bien sûr, vous pouvez dessiner vos diagrammes de la façon qui vous convient.) Le diagramme d'héritage pour **Instrument.cpp**est alors :

Transtyper (casting, ndt) depuis une classe dérivée vers une classe de base entraîne un déplacement vers le haut (up, ndt) sur le diagramme d'héritage, et on parle donc communément de transtypage ascendant. Le transtypage ascendant est toujours fiable parce que vous partez d'un type spécifique vers un type plus général : la seule chose qui peut arriver à l'interface de la classe est qu'elle peut perdre des fonctions membres, pas en gagner. C'est pourquoi le compilateur autorise le transtypage ascendant sans transtypage (cast, ndt) explicite ou tout autre notation spéciale.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

14.12.2 - Le transtypage ascendant et le constructeur de copie

Si vous autorisez le compilateur à synthétiser un constructeur de copie pour une classe dérivée, il appellera automatiquement le constructeur de copie de la classe de base, puis les constructeurs de copie de tous les objets membres (ou bien réalisera une copie de bits des types prédéfinis) si bien que vous obtiendrez le bon comportement :

```
                        //: C14:CopyConstructor.cpp
// Créer correctement le constructeur de copie
#include <iostream>
using namespace std;

class Parent {
  int i;
public:
  Parent(int ii) : i(ii) {
    cout << "Parent(int ii)\n";
  }
  Parent(const Parent& b) : i(b.i) {
    cout << "Parent(const Parent&)\n";
  }
  Parent() : i(0) { cout << "Parent()\n"; }
  friend ostream&
    operator<<(ostream& os, const Parent& b) {
    return os << "Parent: " << b.i << endl;
  }
};

class Member {
  int i;
public:
  Member(int ii) : i(ii) {
    cout << "Member(int ii)\n";
  }
  Member(const Member& m) : i(m.i) {
    cout << "Member(const Member&)\n";
  }
  friend ostream&
    operator<<(ostream& os, const Member& m) {
    return os << "Member: " << m.i << endl;
  }
};

class Child : public Parent {
  int i;
  Member m;
public:
  Child(int ii) : Parent(ii), i(ii), m(ii) {
    cout << "Child(int ii)\n";
  }
  friend ostream&
    operator<<(ostream& os, const Child& c){
    return os << (Parent&)c << c.m
              << "Child: " << c.i << endl;
  }
};

int main() {
  Child c(2);
  cout << "calling copy-constructor: " << endl;
  Child c2 = c; // Appelle le constructeur de copie
  cout << "values in c2:\n" << c2;
} ///:~
```

operator<<pour **Child**est intéressant à cause de la façon dont il appelle le **operator<<**pour la partie **Parent**qu'il contient : en transtypant l'objet **Child**en un **Parent&**(si vous transtypez vers un *objet*de la classe de base au lieu d'une référence vous obtiendrez généralement des résultats indésirables) :

```
                        return os &lt;&lt; (Parent&)c &lt;&lt; c.m
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Puisque le compilateur le voit ensuite comme un **Parent**, il appelle la version **Parent**de **operator<<**.

Vous pouvez constater que **Child**n'a pas de constructeur de copie défini explicitement. Le compilateur doit alors synthétiser le constructeur de copie (comme c'est une des quatre fonctions qu'il synthétise, ainsi que le constructeur par défaut – si vous ne créez aucun constructeur –, **operator=**et le destructeur) en appelant le constructeur de copie de **Parent**et le constructeur de copie de **Member**. Ceci est démontré dans la sortie du programme.

```
                        Parent(int ii)
Member(int ii)
Child(int ii)
calling copy-constructor:
Parent(const Parent&)
Member(const Member&)
values in c2:
Parent: 2
Member: 2
Child: 2
```

Toutefois, si vous essayez d'écrire votre propre constructeur de copie pour **Child**et que vous faites une erreur innocente et le faites mal :

```
                        Child(const Child& c) : i(c.i), m(c.m) {}
```

alors le constructeur par *défaut*sera automatiquement appelé pour la partie classe de base de **Child**, puisque c'est ce sur quoi le compilateur se rabat quand il n'a pas d'autre choix de constructeur à appeler (souvenez-vous qu'un constructeur doit toujours être appelé pour chaque objet, que ce soit un sous-objet ou une autre classe). La sortie sera alors :

```
                        Parent(int ii)
Member(int ii)
Child(int ii)
calling copy-constructor:
Parent()
Member(const Member&)
values in c2:
Parent: 0
Member: 2
Child: 2
```

Ce n'est probablement pas ce à quoi vous vous attendiez, puisqu'en général vous aurez intérêt à ce que la partie classe de base de l'objet soit copiée depuis l'objet existant vers le nouvel objet ; cette étape devrait être une partie de la construction de copie.

Pour pallier ce problème vous devez vous souvenir d'appeler correctement le constructeur de copie de la classe de base (comme le fait le compilateur) à chaque fois que vous écrivez votre propre constructeur de copie. Ceci peut paraître étrange à première vue, mais c'est un autre exemple de transtypage ascendant :

```
                          Child(const Child& c)
    : Parent(c), i(c.i), m(c.m) {
    cout << "Child(Child&)\n";
}
```

La partie bizarre est celle où le contructeur de copie de **Parent**est appelé : **Parent(c)**. Qu'est-ce que cela signifie

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

de passer un objet **Child**au constructeur d'un **Parent**? Mais **Child**hérite de **Parent**, si bien qu'une référence vers un **Child** *est*une référence vers un **Parent**. L'appel au constructeur de copie de la classe de base réalise l'upcast d'une référence vers **Child**en une référence vers **Parent**et l'utilise pour réaliser la construction de copie. Quand vous écrirez vos propres constructeurs de copie, vous aurez presque toujours intérêt à procéder ainsi.

14.12.3 - Composition vs. héritage (révisé)

Une des façons les plus claires pour déterminer si vous devriez utiliser la composition ou bien l'héritage est en vous demandant si vous aurez jamais besoin d'upcaster depuis votre nouvelle classe. Plus tôt dans ce chapitre, la classe **Stack**a été spécialisée en utilisant l'héritage. Toutefois, il y a des chances pour que les objets **StringStack**soient toujours utilisés comme des conteneurs de **string**et jamais upcastés, si bien qu'une alternative plus appropriée est la composition :

```
                        //: C14:InheritStack2.cpp
// Composition vs. héritage
#include "../C09/Stack4.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class StringStack {
  Stack stack; // Encapsulé au lieu d'être hérité
public:
  void push(string* str) {
    stack.push(str);
  }
  string* peek() const {
    return (string*)stack.peek();
  }
  string* pop() {
    return (string*)stack.pop();
  }
};

int main() {
  ifstream in("InheritStack2.cpp");
  assure(in, "InheritStack2.cpp");
  string line;
  StringStack textlines;
  while(getline(in, line))
    textlines.push(new string(line));
  string* s;
  while((s = textlines.pop()) != 0) // Pas de transtypage !
    cout << *s << endl;
} ///:~
```

Le fichier est identique à **InheritStack.cpp**, sauf qu'un objet **Stack**est encapsulé dans **StringStack**, et des fonctions membres sont appelées pour l'objet encapsulé. Il n'y a toujours aucun temps ou espace système excédentaire parce que le sous-objet occupe la même quantité d'espace, et toutes les vérifications de type additionnelles ont lieu à la compilation.

Bien que cela tende à être plus déroutant vous pourriez également utiliser l'héritage **private**pour exprimer l'idée "implémenté en termes de". Ceci résoudrait également le problème de manière adéquate. Une situation où cela devient important, toutefois, est lorsque l'héritage multiple doit être assuré. Dans ce cas, si vous voyez une forme pour laquelle la composition peut être utilisée plutôt que l'héritage, vous devriez pouvoir éliminer le besoin d'héritage multiple.

14.12.4 - Transtypage ascendant de pointeur & de reference

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Dans **Instrument.cpp**, le transtypage ascendant se produit lors de l'appel de la fonction – on prend la référence vers un objet **Wind**en-dehors de la fonction et elle devient une référence vers un **Instrument**à l'intérieur de la fonction. Le transtypage ascendant peut également se produire pendant une simple assignation à un pointeur ou une référence :

```
                        Wind w;
Instrument* ip = &w; // Upcast
Instrument& ir = w; // Upcast
```

Comme l'appel de fonction, aucun de ces deux cas ne requiert de transtypage explicite.

14.12.5 - Une crise

Bien sûr, tout upcast perd de l'information de type concernant un objet. Si vous écrivez :

```
                        Wind w;
Instrument* ip = &amp;w;
```

le compilateur ne peut traiter **ip**que comme un poiteur vers un **Instrument**et rien d'autre. C'est-à-dire qu'il ne peut savoir qu' **ip**est en fait un pointeur vers un objet **Wind**. Si bien que quand vous appelez la fonction membre **play( )**en disant :

```
                        ip->play(middleC);
```

le compilateur peut savoir uniquement qu'il est en train d'appeler **play( )**pour un pointeur vers un **Instrument**, et appelle la version de la classe de base de **Instrument::play( )**au lieu de ce qu'il devrait faire, à savoir appeler **Wind::play( )**. Ainsi, vous n'obtiendrez pas le comportement correct.

Ceci est un vrai problème ; il est résolu au Chapitre 15 en introduisant la troisième pierre angulaire de la programmetion orientée objet : le polymorphisme (implémenté en C++ par les fonctions **virtual**(virtuelles, ndt)).

14.13 - Résumé

L'héritage et la composition vous permettent tous deux de créer un nouveau type à partir de types existants, et tout deux enrobent des sous-objets des types existants dans le nouveau type. En général, toutefois, on utilise la composition pour réutiliser des types existants comme éléments de l'implémentation sous-jacente du nouveau type et l'héritage quand on veut forcer le nouveau type à être du même type que la classe de base (l'équivalence de type garantit l'équivalence d'interface). Comme la classe dérivée possède l'interface de la classe de base, elle peut être *transtypée*vers la classe de base, ce qui est critique pour le polymorphisme comme vous le verrez au Chapitre 15.

Bien que le code réutilisé via la composition et l'héritage est très pratique pour le développement rapide de projets, vous aurez souvent intérêt à re-concevoir votre hiérarchie de classe avant d'autoriser d'autres programmeurs à en dépendre. Votre but est une hiérarchie dans laquelle chaque classe a un usage spécifique et n'est ni trop grande (englobant tellement de fonctionnalités qu'elle n'est pas pratique à réutiliser) ni petite au point d'être gênante (vous ne pouvez vous en servir toute seule ou sans ajouter de fonctionnalité).

14.14 - Exercices

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Les solutions aux exercices choisis peuvent être trouvées dans le document électronique *The Thinking in C++ Annotated Solution Guide*, disponible sur www.BruceEckel.com pour une somme modique.

1 Modifiez **Car.cpp**de sorte qu'il hérite aussi d'une classe appelée **Vehicle**, en mettant les fonction membres appropriées dans **Vehicle**(c'est à dire construire quelques fonctions membres). Ajoutez un constructeur autre que celui par défaut à **Vehicle**, que vous devez appeler à l'intérieur du constructeur de **Car**
2 Créez deux classes, **A**et **B**, avec des constructeurs par défaut qui s'annoncent eux-même. Héritez une nouvelle classe **C**à partir de **A**, et créez un objet membre de type **B**dans **C**, mais ne créez pas de constructeur pour **C**. Créez un objet de classe **C**et observez les résultats.
3 Faites une hiérarchie de classes à trois niveaux avec des constructeurs par défaut, accompagnés de destructeurs, qui s'annoncent tous les deux sur **cout**. Vérifiez que pour un objet de type le plus dérivé, les trois constructeurs et destructeurs sont appelés automatiquement. Expliquez l'ordre dans lequel les appels sont effectués.
4 Modifiez **Combined.cpp**pour ajouter un autre niveau d'héritage et un nouvel objet membre. Ajoutez du code pour montrez quand les destructeurs et les constructeurs sont appelés.
5 Dans **Combined.cpp**, créez une classe **D**qui hérite de **B**et a un objet membre de classe **C**. Rajoutez du code pour montrer quand les constructeurs et les destructeurs sont appelés
6 Modifiez **Order.cpp**pour ajouter un autre niveau d'héritage **Derived3**avec des membres de classes **Member4**et **Member5**. Faites une trace de la sortie du programme.
7 Dans **NameHiding.cpp**, vérifiez que dans **Derived2**, **Derived3**, et **Derived4**, aucune des versions de **f( )**provenant de la classe de base ne sont disponibles.
8 Modifiez **NameHiding.cpp**en ajoutant trois fonctions surchargées **h( )**à **Base**, et montrez qu'en redéfinir une dans une classe dérivée masque les autres.
9 Héritez une classe **StringVector**de **vector<void*>**et redéfinissez les fonctions membres **push_back( )**et **operator[]**pour accepter et produire des **string***. Qu'est-ce qui arrive si vous essayez de **push_back( )**un **void***?
10 Ecrivez une classe contenant un **long**et utilisez la syntaxe d'appel du pseudo-constructeur dans le constructeur pour initialiser le **long**.
11 Créez une classe appelée **Asteroid**. Utilisez l'héritage pour spécialiser la classe **PStash**dans le Chapitre 13 (**PStash.h**& **PStash.cpp**) de sorte qu'elle accepte et retourne des pointeurs **Asteroid**. Egalement, modifiez **PStashTest.cpp**pour tester vos classes. Changez la classe pour que **PStash**soit un objet membre.
12 Refaites l'exercice 11 avec un **vector**au lieu d'un **PStash**.
13 Dans **SynthesizedFunctions.cpp**, modifiez **Chess**pour lui donner un constructeur par défaut, un constructeur par recopie, et un opérateur d'affectation. Prouvez que vous les avez écrits correctement.
14 Créez deux classes appelées **Traveler**et **Pager**sans constructeur par défaut, mais avec des constructeur qui prennent un argument de type **string**, qu'ils copient simplement dans une variable membre **string**. Pour chaque classe, écrivez le constructeur par copie et l'opérateur d'affectation corrects. Héritez maintenant une classe **BusinessTraveler**de **Traveler**et donnez lui un objet membre de type **Pager**. Ecrivez le constructeur par défaut correct, le constructeur qui prend un argument **string**, un constructeur par recopie, et un opérateur d'affectation.
15 Créez une classe avec deux fonctions membres **static**. Héritez de cette classe et redéfinissez l'une des deux fonctions membres. Montrez que l'autre est cachée dans la classe dérivée.
16 Examinez plus les fonctions de **ifstream**. Dans **FName2.cpp**, essayez les sur l'objet **file**.
17 Utilisez l'héritage **private**et **protected**pour faire deux nouvelles classes d'une classe de base. Essayez ensuite de transtyper les objets dérivés dans la classe de base. Expliquez ce qui arrive.
18 Dans **Protected.cpp**, rajoutez une fonction membre à **Derived**qui appelle le membre **protected read( )**de **Base**.
19 Changez **Protected.cpp**pour que **Derived**utilise l'héritage **protected**. Voyez si vous pouvez appeler **value( )**pour un objet **Derived**.
20 Créez une classe **SpaceShip**avec une méthode **fly( )**. Héritez **Shuttle**de **SpaceShip**et ajoutez une méthode **land( )**. Créez un nouvel objet **Shuttle**, transtypez le par pointeur ou référence en **SpaceShip**, et essayez d'appeler la méthode **land( )**. Expliquez les résultats.
21 Modifiez **Instrument.cpp**pour ajouter une méthode **prepare( )**à **Instrument**. Appelez **prepare( )**dans **tune( )**.
22 Modifiez **Instrument.cpp**pour que **play( )**affiche un message sur **cout**, et **Wind**redéfinisse **play( )**pour afficher un message différent sur **cout**. Exécutez le programme et expliquez pourquoi vous ne voudriez

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

probablement pas ce comportement. Maintenant mettez le mot-clé **virtual**(que vous apprendrez dans le chapitre 15) devant la déclaration de **play( )**dans **Instrument**et observez le changement de comportement.

23 Dans **CopyConstructor.cpp**, héritez une nouvelle classe de **Child**et donnez-lui un membre **m**. Ecrivez un constructeur, un constructeur par recopie, un opérateur **=**, et un opérateur **<<**pour ostream adaptés, et testez la classe dans **main( )**.

24 Prenez l'exemple de **CopyConstructor.cpp**et modifiez le en ajoutant votre propre constructeur par recopie à **Child** *sans*appeler le constructeur par recopie de la classe de base et voyez ce qui arrive. Corrigez le problème en faisant un appel explicite correct au constructeur par recopie de la classe de base dans liste d'initialisation du constructeur par recopie de **Child**

25 Modifiez **InheritStack2.cpp**pour utiliser un **vector<string>**au lieu d'un **Stack**.

26 Créez une classe **Rock**avec un constructeur par défaut, un constructeur par recopie, un opérateur d'affectation, et un destructeur qui annoncent tous sur **cout**qu'ils ont été appelés. Dans **main( )**, créez un **vector<Rock>**(c'est à dire enregistrer des objets **Rock**objects par valeur) et ajoutez quelques **Rock**s. Exécutez le programme et expliquez la sortie que vous obtenez. Notez si les destructeurs sont appelés pour les objets **Rock**dans le **vector**. Maintenant répétez l'exercice avec un **vector<Rock*>**. Est-il possible de créer un **vector<Rock&>**?

27 Cet exercice crée le modèle de conception appelé *proxy*. Commencez avec une classe de base **Subject**et donnez lui trois fonctions : **f( )**, **g( )**, et **h( )**. Maintenant héritez une classe **Proxy**et deux classes **Implementation1**et **Implementation2**de **Subject**. **Proxy**devrait contenir un pointeur vers un **Subject**, et toutes les méthodes pour **Proxy**devraient juste prendre le relais et faire les même appels à travers le pointeur de **Subject**. Le constructeur de **Proxy**prend un pointeur vers un **Subject**qui est installé dans **Proxy**(d'habitude par le constructeur). Dans **main( )**, créez deux objets **Proxy**différents qui utilisent deux implémentations différentes. Maintenant modifiez **Proxy**pour que vous puissiez changer dynamiquement d'implémentation

28 Modifiez **ArrayOperatorNew.cpp**du chapitre 13 pour montrer que, si vous héritez de **Widget**, l'allocation marche toujours correctement. Expliquez pourquoi l'héritage dans **Framis.cpp**du chapitre 13 *ne*marcherait *pas*correctement.

29 Modifiez **Framis.cpp**du chapitre 13 en héritant de **Framis**et créant des nouvelles versions de **new**et **delete**pour vos classes dérivées. Démontrez qu'elles fonctionnent correctement.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

15 - Polymorphisme & Fonctions Virtuelles

Le polymorphism (implémenté en C++ avec les fonctions **virtual**) est le troisième aspect essentiel d'un langage de programmation orienté objet, après l'abstraction des données et l'héritage.

Cela fournit une autre dimension de la séparation de l'interface et de l'implémentation, pour découpler *quoi*de *comment*. Le polymorphisme permet d'améliorer l'organisation et la lisibilité du code aussi bien que la création de programmes *extensibles*que l'on peut faire croître non seulement pendant la création originelle du projet, mais également quand de nouvelles caractéristiques sont souhaitées.

L'encapsulation crée de nouveaux types de données en combinant caractéristiques et comportements. Le contrôle d'accès sépare l'interface de l'implémentation en rendant les détails **private**. Ce genre d'organisation mécanique est rapidement claire pour quelqu'un qui a un passé en programmation procédurale. Mais les fonctions virtuelles traitent de découplage en terme de *types*. Au chapitre 14, vous avez vu comment l'héritage permettait le traitement d'un objet comme de son type propre *ou*comme son type de base. Cette capacité est critique parce qu'elle autorise à beaucoup de types (dérivés du même type de base) d'être traités comme s'ils étaient un seul type, et à un morceau de code unique de fonctionner indifféremment avec tous ces types. La fonction virtuelle permet à un type d'exprimer sa différence par rapport à un autre, similaire, pourvu qu'ils soient dérivés du même type de base. Cette distinction est exprimée par des différences de comportement des fonctions que vous pouvez appeler via la classe de base.

Dans ce chapitre vous étudierez les fonctions virtuelles, en partant des bases avec des exemples simples qui écartent tous les aspects sauf la “virtualité” au sein du programme.

15.1 - Evolution des programmeurs C++

Les programmeurs en C semblent acquérir le C++ en trois étapes. Premièrement, simplement comme un “C amélioré”, parce que le C++ vous force à déclarer toutes les fonctions avant de les utiliser et se montre plus pointilleux sur la façon d'utiliser des variables. Vous pouvez souvent trouver les erreurs dans un programme C simplement en le compilant avec un compilateur C++.

La deuxième étape est le C++ “basé sur les objets”. Cela signifie que vous voyez facilement les bénéfices en terme d'organisation de code qui résultent du regroupement de structures de données avec les fonctions qui agissent dessus, la valeur des constructeurs et des destructeurs, et peut-être un peu d'héritage simple. La plupart des programmeurs qui ont travaillé en C pendant quelques temps voit rapidement l'utilité de tout cela parce que, à chaque fois qu'ils créent une bibliothèque, c'est exactement ce qu'ils essayent de faire. Avec le C++, vous avez l'assistance du compilateur.

Vous pouvez rester coincé au niveau basé sur les objets parce qu'on y arrive vite et qu'on en retire beaucoup de bénéfices sans trop d'effort intellectuel. Vous pouvez également avoir l'impression de créer des types de données – vous fabriquez des classes et des objets, vous envoyez des messages à ces objets, et tout est beau et clair.

Mais ne vous laissez pas avoir. Si vous vous arrêtez ici, vous manquez la plus grande partie du langage, qui est le saut vers la vraie programmation orientée objet. Vous ne pouvez le faire qu'avec les fonctions virtuelles.

Les fonctions virtuelles renforcent le concept de type au lieu de simplement encapsuler du code dans des structures et derrière des murs, elles sont ainsi sans aucun doute le concept le plus difficile à comprendre pour le nouveau programmeur C++. Toutefois, elles sont également la charnière dans la compréhension de la programmation orientée objet. Si vous n'utilisez pas les fonctions virtuelles, vous ne comprenez pas encore la POO.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Parce que la fonction virtuelle est intimement liée au concept de type, et que le type est au coeur de la programmation orientée objet, il n'y a pas d'analogue aux fonctions virtuelles dans un langage procédural classique. Comme programmeur procédural, vous n'avez aucune référence à laquelle vous ramener pour penser aux fonctions virtuelles, contrairement à pratiquement toutes les autres caractéristiques du langage. Les caractéristiques d'un langage procédural peuvent être comprises à un niveau algorithmique, mais les fonctions virtuelles ne peuvent être comprises que du point de vue de la conception.

15.2 - Transtypage ascendant (upcasting)

Dans le chapitre 14 vous avez vu comment un objet peut être utilisé comme son propre type ou comme un objet de son type de base. En outre, il peut être manipulé via une adresse du type de base. Prendre l'adresse d'un objet (par un pointeur ou par une référence) et la traiter comme l'adresse du type de base est appelé *transtypage ascendant*(*upcasting*en anglais, ndt) à cause de la façon dont les arbres d'héritage sont dessinés, avec la classe de base en haut.

Vous avez également vu un problème se dessiner, qui est incarné dans le code ci-dessous :

```
                    //: C15:Instrument2.cpp
// Héritage & transtypage ascendant
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {
public:
  void play(note) const {
    cout << "Instrument::play" << endl;
  }
};

// Les objets Wind sont des Instruments
// parce qu'ils ont la même interface :
class Wind : public Instrument {
public:
  // Redéfinit la fonction interface :
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // transtypage ascendant
} ///:~
```

La fonction **tune( )**accepte (par référence) un **Instrument**, mais également, sans se plaindre, tout dérivé d' **Instrument**. Dans **main( )**, vous pouvez voir cela se produire quand un objet **Wind**est passé à **tune( )**, sans nécessiter de transtypage. C'est acceptable ; l'interface qui se trouve dans **Instrument**doit exister dans **Wind**, parce que **Wind**hérite publiquement d' **Instrument**. Faire un transtypage ascendant de **Wind**vers **Instrument**peut “rétrécir” cette interface, mais jamais la ramener à moins que l'interface complète d' **Instrument**.

Les mêmes arguments sont vrais quand on traite des pointeurs ; la seule différence est que l'utilisateur doit prendre explicitement les adresses des objets quand ils sont passés dans la fonction.

15.3 - Le problème

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Le problème avec **Instrument2.cpp**peut être vu en exécutant le programme. La sortie est **Instrument::play**. Ceci n'est clairement pas la sortie désirée, parce qu'il se trouve que vous savez que l'objet est en fait un **Wind**et pas juste un **Instrument**. L'appel devrait produire **Wind::play**. A ce sujet, n'importe quel objet d'une classe dérivée de **Instrument**devrait avoir sa version de **play( )**utilisé, indépendamment de la situation.

Le comportement de **Instrument2.cpp**n'est pas surprenant, étant donné l'approche du C pour les fonctions. Pour comprendre ces questions, vous devez être conscient du concept de *liaison*.

15.3.1 - Liaison d'appel de fonction

Connecter un appel de fonction à un corps de fonction est appelé *liaison*(*binding*en anglais ndt). Quand la liaison est effectuée avant que le programme ne soit exécuté (par le compilateur et le linker), elle est appelée *liaison précoce*(*early binding*ndt). Vous n'avez peut être jamais entendu ce terme auparavant parce qu'il n'a jamais été une option dans les langages procéduraux : les compilateurs C n'ont qu'un seul genre d'appels de fonctions, et il s'agit de la liaison précoce.

Le problème dans le programme précédent est causé par la liaison précoce parce que le compilateur ne peut pas savoir quelle est la fonction correcte qu'il doit appeler quand il ne dispose que de l'adresse d'un **Instrument**.

La solution est appelée *liaison tardive*(*late binding*ndt), ce qui signifie que la liaison se produit au moment de l'exécution, en fonction du type de l'objet. La liaison tardive est aussi appelée *liaison dynamique*(*dynamic binding*ndt) ou *liaison à l'exécution*(*runtime binding*ndt). Quand un langage implémente la liaison tardive, il doit posséder un mécanisme afin de déterminer le type de l'objet au moment de l'exécution et appeler la fonction membre appropriée. Dans le cas d'un langage compilé, le compilateur ne sait toujours pas le type réel de l'objet, mais il insert du code qui trouve et appelle le corps de la fonction qui convient. Le mécanisme de liaison tardive varie d'un langage à l'autre, mais vous vous doutez qu'une sorte d'information de type doit être installée dans les objets. Vous verrez comment cela fonctionne plus tard.

15.4 - Fonctions virtuelles

Pour provoquer une liaison tardive pour une fonction particulière, le C++ impose que vous utilisiez le mot-clef **virtual**quand vous déclarez la fonction dans la classe de base. La liaison tardive n'a lieu que pour les fonctions virtuelles, et seulement lorsque vous utilisez une adresse de la classe de base où ces fonctions virtuelles existent, bien qu'elles puissent être également définies dans une classe de base antérieure.

Pour créer une fonction membre **virtual**, vous n'avez qu'à faire précéder la déclaration de la fonction du mot-clef **virtual**. Seule, la déclaration nécessite ce mot-clef, pas la définition. Si une fonction est déclarée **virtual**dans la classe de base, elle est **virtual**dans toutes les classes dérivées. La redéfinition d'une fonction virtuelle dans une classe dérivée est généralement appelée *redéfinition*(overriding, ndt).

Remarquez que vous n'avez à déclarer une fonction **virtual**que dans la classe de base. Toutes les fonctions des classes dérivées dont signature correspond à celle de la déclaration dans la classe de base seront appelées en utilisant le mécanisme virtuel. Vous *pouvez*utiliser le mot-clef **virtual**dans les déclarations des classes dérivées (cela ne fait aucun mal), mais c'est redondant et cela peut prêter à confusion.

Pour obtenir de **Instrument2.cpp**le comportement désiré, ajoutez simplement le mot-clef **virtual**dans la classe de base avant **play( )**:

```
                    //: C15:Instrument3.cpp
// liaison tardive avec le mot-clef virtual
#include <iostream>
using namespace std;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
};

// Les objets Wind sont des Instruments
// parce qu'ils ont la même interface :
class Wind : public Instrument {
public:
  // Redéfinit la fonction d'interface :
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

Mise à part l'ajout du mot-clef **virtual**, ce fichier est identique à **Instrument2.cpp**, et pourtant le comportement est significativement différent : à présent la sortie est **Wind::play**.

15.4.1 - Extensibilité

play( )étant défini **virtual**dans la classe de base, vous pouvez ajouter autant de nouveaux types que vous le désirez sans changer la fonction **tune( )**. Dans un programme orienté objet bien conçu, la plupart voire toutes vos fonctions suivront le modèle de **tune( )**et communiqueront uniquement avec l'interface de la classe de base. Un tel programme est *extensible*parce que vous pouvez ajouter de nouvelles fonctionnalités en faisant hériter de nouveaux types à partir de la classe de base commune. Les fonctions qui manipulent l'interface de la classe de base n'auront pas besoin d'être modifiées pour s'adapter aux nouvelles classes.

Voici l'exemple d' **Instrument**avec davantage de fonctions virtuelles et quelques nouvelles classes, qui fonctionnent toutes correctement avec la vieille version, non modifiée, de la fonction **tune( )**:

```
                        //: C15:Instrument4.cpp
// Extensibilité dans la POO
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
  virtual char* what() const {
    return "Instrument";
  }
  // Supposez que ceci modifiera l'objet :
  virtual void adjust(int) {}
};

class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};

// Fonction identique à précédemment :
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// Nouvelle fonction :
void f(Instrument& i) { i.adjust(1); }

// Upcasting pendant l'initialisation du tableau :
Instrument* A[] = {
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
} ///:~
```

Vous pouvez voir qu'un niveau d'héritage nouveau a été ajouté en dessous de **Wind**, mais le mécanisme **virtual**fonctionne correctement indépendamment du nombre de niveaux. La fonction **adjust( )**n'est *pas*redéfinie pour **Brass**et **Woodwind**. Quand c'est le cas, la définition “la plus proche” dans la hiérarchie de l'héritage est utilisée automatiquement – le compilateur garantit qu'il y a toujours une définition pour une fonction virtuelle, afin que vous ne vous retrouviez jamais avec un appel qui ne soit pas associé à un corps de fonction. (Ce qui serait désastreux.)

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Le tableau **A[ ]**contient des pointeurs vers la classe de base **Instrument**, ainsi l'upcasting se produit pendant le processus d'initialisation du tableau. Ce tableau et la fonction **f( )**seront utilisés dans des discussions ultérieures.

Dans l'appel à **tune( )**, l'upcasting est réalisé sur chaque type d'objet différent, et pourtant, le comportement désiré a toujours lieu. On peut décrire cela comme “envoyer un message à un objet et laisser l'objet se préoccuper de ce qu'il doit faire avec”. La fonction **virtual**est la lentille à utiliser quand vous essayez d'analyser un projet : où les classes de base devraient-elle se trouver, et comment pourriez-vous vouloir étendre le programme ? Toutefois, même si vous ne trouvez pas les bonnes interfaces de classes de base et les fonctions virtuelles satisfaisantes lors de la création initiale du programme, vous les découvrirez souvent plus tard, même beaucoup plus tard, quand vous vous mettrez à étendre ou sinon à maintenir le programme. Ce n'est pas une erreur d'analyse ou de conception ; cela veut juste dire que vous ne connaissiez ou ne pouviez pas connaître toutes les informations au début. A cause de la modularité serrée des classes en C++, ce n'est pas un gros problème quand cela se produit, parce que les modifications que vous effectuez dans une partie d'un système n'ont pas tendance à se propager à d'autres comme elles le font en C.

15.5 - Comment le C++ implémente la liaison tardive

Comment la liaison tardive peut-elle se produire ? Tout le travail est effectué discrètement par le compilateur, qui installe le mécanisme de liaison tardive nécessaire quand vous le lui demandez (ce que vous réalisez en créant des fonctions virtuelles). Comme les programmeurs tirent souvent profit d'une bonne compréhension du mécanisme des fonctions virtuelles en C++, cette section va détailler la façon dont le compilateur implémente ce mécanisme.

Le mot-clef **virtual**dit au compilateur qu'il ne doit pas réaliser la liaison trop tôt. Au lieu de cela, il doit installer automatiquement tous les mécanismes nécessaires pour réaliser la liaison tardive. Ceci signifie que si vous appelez **play( )**pour un objet **Brass** *grâce à une adresse pour la classe de base* **Instrument**, vous obtiendrez la bonne fonction.

Pour ce faire, le compilateur typique Les compilateurs peuvent implémenter le comportement virtuel de la manière qu'ils veulent, mais la manière décrite ici est presque l'approche universelle.crée une table unique (appelée VTABLE)pour chaque classe qui contient des fonctions **virtual**. Le compilateur place les adresses des fonctions virtuelles pour cette classe particulière dans la VTABLE. Dans chaque classe dotée de fonctions virtuelles, il place secrètement un pointeur, appelé le *vpointeur*(abrégé en VPTR), qui pointe vers la VTABLE de cet objet. Quand vous faites un appel à une fonction virtuelle à travers un pointeur vers la classe de base (c'est-a-dire, quand vous faites un appel polymorphe), le compilateur insère discrètement du code pour aller chercher le VPTR et trouver l'adresse de la fonction dans la VTABLE, appelant ainsi la fonction correcte et réalisant ainsi la liaison tardive.

Tout cela – créer la VTABLE pour chaque classe, initialiser le VPTR, insérer le code pour l'appel à la fonction virtuelle – a lieu automatiquement, et vous n'avez donc pas à vous en inquiéter. Avec les fonctions virtuelles, la fonction qui convient est appelée pour un objet, même si le compilateur ne peut pas connaître le type de l'objet.

Les sections qui suivent analysent plus en détails ce processus.

15.5.1 - Stocker l'information de type

Vous pouvez constater qu'il n'y a aucune information explicite de type stockée dans aucune des classes. Mais les exemples précédents, et la simple logique, vous disent qu'il doit y avoir un genre d'information de type stockée dans les objets ; autrement le type ne pourrait être déterminé à l'exécution. Ceci est vrai, mais l'information de type est dissimulée. Pour la voir, voici un exemple permettant d'examiner la taille des classes qui utilisent les fonctions virtuelles comparée à celles des classes qui ne s'en servent pas :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                          //: C15:Sizes.cpp
// Taille des objets avec ou sans fonctions virtuelles
#include <iostream>
using namespace std;

class NoVirtual {
  int a;
public:
  void x() const {}
  int i() const { return 1; }
};

class OneVirtual {
  int a;
public:
  virtual void x() const {}
  int i() const { return 1; }
};

class TwoVirtuals {
  int a;
public:
  virtual void x() const {}
  virtual int i() const { return 1; }
};

int main() {
  cout << "int: " << sizeof(int) << endl;
  cout << "NoVirtual: "
       << sizeof(NoVirtual) << endl;
  cout << "void* : " << sizeof(void*) << endl;
  cout << "OneVirtual: "
       << sizeof(OneVirtual) << endl;
  cout << "TwoVirtuals: "
       << sizeof(TwoVirtuals) << endl;
} ///:~
```

Sans fonction virtuelle, la taille de l'objet est exactement ce à quoi vous pouvez vous attendre : la taille d'un **int**unique Certains compilateurs peuvent avoir des problèmes ici, mais c'est plutôt rare.. Avec une seule fonction virtuelle dans **OneVirtual**, la taille de l'objet est la taille de **NoVirtual**plus la taille d'un pointeur **void**. Il s'avère que le compilateur insère un unique pointeur (le VPTR) dans la structure si vous avez une *ou plusieurs*fonctions virtuelles. Il n'y a pas de différence de taille entre **OneVirtual**et **TwoVirtuals**. C'est parce que le VPTR pointe vers une table d'adresses de fonctions. Il n'y a besoin que d'une seule table parce que toutes les adresses des fonctions virtuelles sont contenues dans cette unique table.

Cet exemple nécessitait au moins une donnée membre. S'il n'y en avait eu aucune, le C++ aurait forcé les objets à avoir une taille non nulle parce que chaque objet doit avoir une adresse différente. Si vous imaginez l'indexation dans un tableau d'objets de taille nulle, vous comprendrez pourquoi. Un membre “factice” est inséré dans les objets qui autrement seraient de taille nulle. Quand l'information de type est insérée dans les objets via le mot-clef **virtual**, elle prend la place du membre “factice”. Essayez de passer le **int a**en commentaire dans les classes de l'exemple ci-dessus pour le voir.

15.5.2 - Représenter les fonntions virtuelles

Pour comprendre exactement ce qu'il se passe quand vous utilisez une fonction virtuelle, il est pratique de visualiser ce qu'il se passe derrière la scène. Voici un schéma du tableau des pointeurs **A[ ]**dans **Instrument4.cpp**:

Le tableau de pointeurs vers **Instrument**n'a pas d'information de type spécifique ; ils pointent tous vers un objet de type **Instrument**. **Wind**, **Percussion**, **Stringed**, et **Brass**rentrent tous dans cette catégorie parce qu'ils sont dérivés d' **Instrument**(et ont donc la même interface qu' **Instrument**, et peuvent répondre aux mêmes messages), et leur adresse peut donc également être placée dans le tableau. Toutefois, le compilateur ne sait pas qu'ils sont quelque chose de plus que des objets **Instrument**, et donc si on le laissait autonome il appellerait normalement

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

pour toutes les fonctions leur version de la classe de base. Mais dans ce cas, toutes ces fonctions ont été déclarées avec le mot-clef **virtual**, si bien que quelque chose de différent se produit.

A chaque fois que vous créez une classe qui contient des fonctions virtuelles, ou que vous dérivez d'une classe qui en contient, le compilateur crée une VTABLE unique pour cette classe, visible sur la droite du diagramme. Dans cette table il place les adresses de toutes les fonctions qui ont été déclarés virtuelles dans cette classe ou dans la classe de base. Si vous ne surchargez pas une fonction qui a été déclarée **virtual**dans la classe de base, le compilateur utilise l'adresse de la version de la classe de base dans la classe dérivée. (Vous pouvez le voir dans l'entrée **adjust**dans la VTABLE de **Brass**.) Puis il positionne le VPTR (découvert dans **Sizes.cpp**) au sein de la classe. Il n'y a qu'un seul VPTR pour chaque objet quand on utilise l'héritage simple comme ceci. Le VPTR doit être initialisé pour pointer vers l'adresse de début de la VTABLE appropriée. (Ceci se produit dans le constructeur, ce que vous verrez de manière plus détaillée ci-dessous.)

Une fois que le VPTR est initialisé à la bonne VTABLE, l'objet actuel “sait” de quel type il est. Mais cette connaissance de soi est sans valeur à moins d'être utilisée au moment de l'appel d'une fonction virtuelle.

Quand vous appelez une fonction virtuelle via l'adresse d'une classe de base (ce qui correspond à la situation où le compilateur ne dispose pas de toutes les informations nécessaires pour réaliser la liaison plus tôt), quelque chose de particulier se produit. Au lieu de réaliser un appel typique à une fonction, qui est simplement un **CALL**en langage assembleur vers une adresse particulière, le compilateur génère un code différent pour réaliser l'appel. Voici à quoi ressemble un appel à **adjust( )**pour un objet **Brass**, s'il est réalisé via un pointeur vers **Instrument**(Une référence vers **Instrument**produit le même résultat) :

Le compilateur commence avec le pointeur vers **Instrument**, qui pointe vers l'adresse de début de l'objet. Tous les objets **Instrument**ou dérivés d' **Instrument**ont leur VPTR au même endroit (souvent au début de l'objet), afin que le compilateur puisse le repérer dans l'objet. Le VPTR pointe vers l'adresse de début de la VTABLE. Toutes les adresses de fonction de la VTABLE sont disposées dans le même ordre, indépendamment du type spécifique de l'objet : en premier, **play( )**, puis **what( )**et enfin **adjust( )**. Le compilateur sait que, indépendamment du type spécifique d'objet, la fonction **adjust( )**est à l'emplacement VPTR+2. Ainsi, au lieu de dire : “Appelle la fonction à l'emplacement absolu **Instrument::adjust**” (liaison précoce; mauvaise démarche), il génère du code qui dit : “Appelle la fonction à VPTR+2”. Comme la recherche du VPTR et la détermination de la vraie adresse de la fonction a lieu à l'exécution, vous obtenez la liaison tardive désirée. Vous envoyer un message à l'objet, qui devine ce qu'il doit en faire.

15.5.3 - Sous le capot

Il peut être utile de voir le code assembleur généré par un appel à une fonction virtuelle, afin que vous voyiez que la liaison tardive a bien lieu. Voici la sortie d'un compilateur pour l'appel

```
                        i.adjust(1);
```

dans la fonction **f(Instrument& i)**:

```
                        push  1
push  si
mov   bx, word ptr [si]
call  word ptr [bx+4]
add   sp, 4
```

Les arguments d'un appel à une fonction en C++, comme en C, sont poussés sur la pile depuis la droite vers la gauche (cet ordre est requis pour supporter les listes d'arguments variables du C), si bien que l'argument **1**est poussé sur la pile en premier. A ce point de la fonction, le registre **si**(élément de l'architecture des processeurs

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Intel X86) contient l'adresse de **i**, qui est également poussé sur la pile parce que c'est l'adresse du début de l'objet qui nous intéresse. Rappelez vous que l'adresse du début correspond à la valeur de **this**, et **this**est discrètement poussé sur la pile comme un argument avant tout appel à fonction, afin que la fonction membre sache sur quel objet particulier elle travaille. Ainsi, vous verrez toujours un argument de plus que la liste des arguments de la fonction poussés sur la pile avant un appel à une fonction membre (sauf pour les fonctions membres **static**, qui n'ont pas de **this**).

A présent, le vrai appel à la fonction virtuelle doit être réalisé. Tout d'abord, le VPTR doit être produit, afin que la VTABLE puisse être trouvée. Pour ce compilateur, le VPTR est inséré au début de l'objet, si bien que le contenu de **this**correspond au VPTR. La ligne

```
mov bx, word ptr [si]
```

cherche le mot vers lequel pointe **si**(c'est-à-dire **this**), qui est le VPTR. Il place le VPTR dans le registre **bx**.

Le VPTR contenu dans **bx**pointe vers l'adresse du début de la VTABLE, mais le pointeur de la fonction à appeler n'est pas à l'meplacement zéro de la VTABLE, mais à l'emplacement deux (parce que c'est la troisième fonction dans la liste). Pour ce modèle de mémoire, chaque pointeur de fonction mesure deux octets, et donc le compilateur ajoute quatre au VPTR pour calculer où se trouve l'adresse de la fonction appropriée. Remarquez que c'est une valeur constante, établie à la compilation, et donc la seule chose qui importe est que le pointeur de fonction à l'emplacement numéro deux est celui d' **adjust( )**. Heureusement, le compilateur prend soin de toute la comptabilité pour vous et garantit que tous les pointeurs de fonction dans toutes les VTABLEs d'une hiérarchie de classe particulière apparaissent dans le même ordre, indépendamment de l'ordre dans lequel vous pouvez les surcharger dans les classes dérivées.

Une fois que le bon pointeur de fonction de la VTABLE est calculé, cette fonction est appelée. Ainsi, l'adresse est-elle cherchée et l'appel effectué simultanément dans l'instruction

```
call word ptr [bx+4]
```

Finalement, le pointeur vers la pile est repoussé vers le haut pour nettoyer les arguments qui ont été poussés avant l'appel. Dans le cide assembleur du C et du C++ vous verrez souvent l'appelant nettoyer les arguments mais cela peut varier selon les processeurs et les implémentations des compilateurs.

15.5.4 - Installer le vpointeur

Puisque le VPTR détermine le comportement de la fonction virtuelle de l'objet, vous voyez à quel point il est important que le VPTR pointe toujours vers la VTABLE appropriée. Vous n'avez jamais intérêt à être capable de réaliser un appel à une fonction virtuelle avant que le VPTR ne soit proprement initialisé. Bien sûr, l'endroit où l'initialisation peut être garantie est dans le constructeur, mais aucun des exemples d' **Instrument**n'en a.

C'est ici que la création d'un constructeur par défaut est essentielle. Dans les exemples d' **Instrument**, le compilateur crée un constructeur par défaut qui ne fait rien sauf initialiser le VPTR. Ce constructeur, bien sûr, est automatiquement appelé pour tous les objets **Instrument**avant que vous ne puissiez faire quoi que ce soit avec eux, et vous savez ainsi qu'il est toujours sûr d'appeler des fonctions virtuelles.

Les implications de l'initialisation automatique du VPTR dans le constructeur sont discutées dans une section ultérieure.

15.5.5 - Les objets sont différents

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Il est important de réaliser que l'upcasting traite seulement des adresses. Si le compilateur dispose d'un objet, il en connaît le type exact et ainsi (en C++) n'utilisera pas de liaison tardive pour les appels de fonction – ou tout au moins, le compilateur n'a pas *besoin*d'utiliser la liaison tardive. Par souci d'efficacité, la plupart des compilateurs réaliseront une liaison précoce quand ils réalisent un appel à une fonction virtuelle pour un objet parce qu'ils en connaissent le type exact. Voici un exemple :

```
                          //: C15:Early.cpp
// Liaison précoce & fonctions virtuelles
#include <iostream>
#include <string>
using namespace std;

class Pet {
public:
  virtual string speak() const { return ""; }
};

class Dog : public Pet {
public:
  string speak() const { return "Bark!"; }
};

int main() {
  Dog ralph;
  Pet* p1 = &ralph;
  Pet& p2 = ralph;
  Pet p3;
  // Liaison tardive pour les deux :
  cout << "p1->speak() = " << p1->speak() <<endl;
  cout << "p2.speak() = " << p2.speak() << endl;
  // Liaison précoce (probable):
  cout << "p3.speak() = " << p3.speak() << endl;
} ///:~
```

Dans **p1–>speak( )**et **p2.speak( )**, les adresses sont utilisées, ce qui signifie que l'information est incomplète : **p1**et **p2**peuvent représenter l'adresse d'un **Pet** *ou*de quelque chose qui en dérive, et le mécanisme virtuel doit donc être utilisé. L'appel de **p3.speak( )**est sans ambiguïté. Le compilateur connaît le type exact et sait que que c'est un objet, si bienqu'il ne peut en aucun cas être un objet dérivé de **Pet**– c'est *exactement*un **Pet**. Ainsi, la liaison précoce est ici probablement utilisée. Toutefois, si le compilateur ne veut pas travailler autant, il peut toujours utiliser la liaison tardive et le même comportement en découlera.

15.6 - Pourquoi les fonctions virtuelles ?

A ce point de la discussion vous pouvez avoir une question : “Si cette technique est si importante, et si elle réalise le ‘bon’ appel de fonction à chaque fois, pourquoi n'est-ce qu'une option ? Pourquoi ai-je même besoin de le connaître ?”

C'est une bonne question, et la réponse fait partie la philosophie fondamentale du C++ : “Parce que ce n'est pas aussi efficace”. Vous pouvez constater d'après les sorties en langage assembleur vues ci-dessus qu'au lieu d'un simple CALL à une adresse absolue, cela requiert deux instructions assembleur – plus sophistiquées – pour préparer l'appel à une fonction virtuelle. Cela demande à la fois du code et du temps d'exécution supplémentaire.

Certains langages orientés objet ont adopté l'approche qui considère que la liaison tardive est tellement intrinsèque à la programmation orientée objet qu'elle devrait toujours avoir lieu, que cela ne devrait pas être une option, et l'utilisateur ne devrait pas avoir à en entendre parler. C'est une décision de conception à prendre à la création d'un langage, et cette approche particulière est appropriée pour beaucoup de langages Smalltalk, Java et Python, par exemple, utilisent cette approche avec grand succès.. Toutefois, le C++ provient de l'héritage du C, où l'efficacité est critique. Après tout, le C a été créé pour remplacer le langage assembleur pour l'implémentation d'un système d'exploitation (rendant ainsi ce système d'exploitation – Unix – largement plus portable que ses prédécesseurs). Une des raisons principales de l'invention du C++ était de rendre les programmeurs C plus efficaces Aux

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

laboratoires Bell (Bell labs, ndt), où le C++ a été inventé, il y a *beaucoup*de programmeurs en C. Les rendre plus efficaces, même juste un peu, permettait à la compagnie d'économiser beaucoup de millions. . Et la première question posée quand les programmeurs C rencontrent le C++ est : “Quel type d'impact obtiendrais-je sur la taille et la vitesse ? “ Si la réponse était : ”Tout est parfait sauf pour les appels de fonctions où vous aurez toujours un petit temps de surcharge supplémentaire” beaucoup de monde serait resté au C plutôt que de passer au C++. En outre, les fonctions inline n'auraient pas été possibles, parce que les focntions virtuelles doivent disposer d'une adresse à mettre dans la VTABLE. Ainsi donc, la fonction virtuelle est une option, *et*le langage utilise par défaut les non virtuelles, ce qui est la configuration la plus rapide. Stroustrup affirmait que sa ligne de conduite était : “si vous ne vous en servez pas, vous n'en payez pas le prix.”

Ainsi, le mot-clef **virtual**est fourni pour moduler l'efficacité. Quand vous concevez vos classes, toutefois, vous ne devriez pas vous inquiéter de régler l'efficacité. Si vous allez utiliser le polymorphisme, utilisez des fonctions virtuelles partout. Vous devez seulement chercher des fonctions qui peuvent être rendues non virtuelles quand vous cherchez des manières d'accélérer votre code (et il y a le plus souvent de bien plus grands gains à réaliser dans d'autres domaines – un bon profileur réalisera un meilleur travail pour trouver les goulets d'étranglement que vous en faisant des conjectures).

Le retour d'expérience suggère que les impacts sur la taille et la vitesse quand on passe au C++ est inférieure à 10 pour cent de la taille et de la vitesse en C, et sont souvent plus prôches de l'égalité. La raison pour laquelle vous pourriez obtenir une plus petite taille et une plus grande vitesse est que vous pouvez concevoir un programme en C++ plus petit et plus rapide que vous ne le feriez en C.

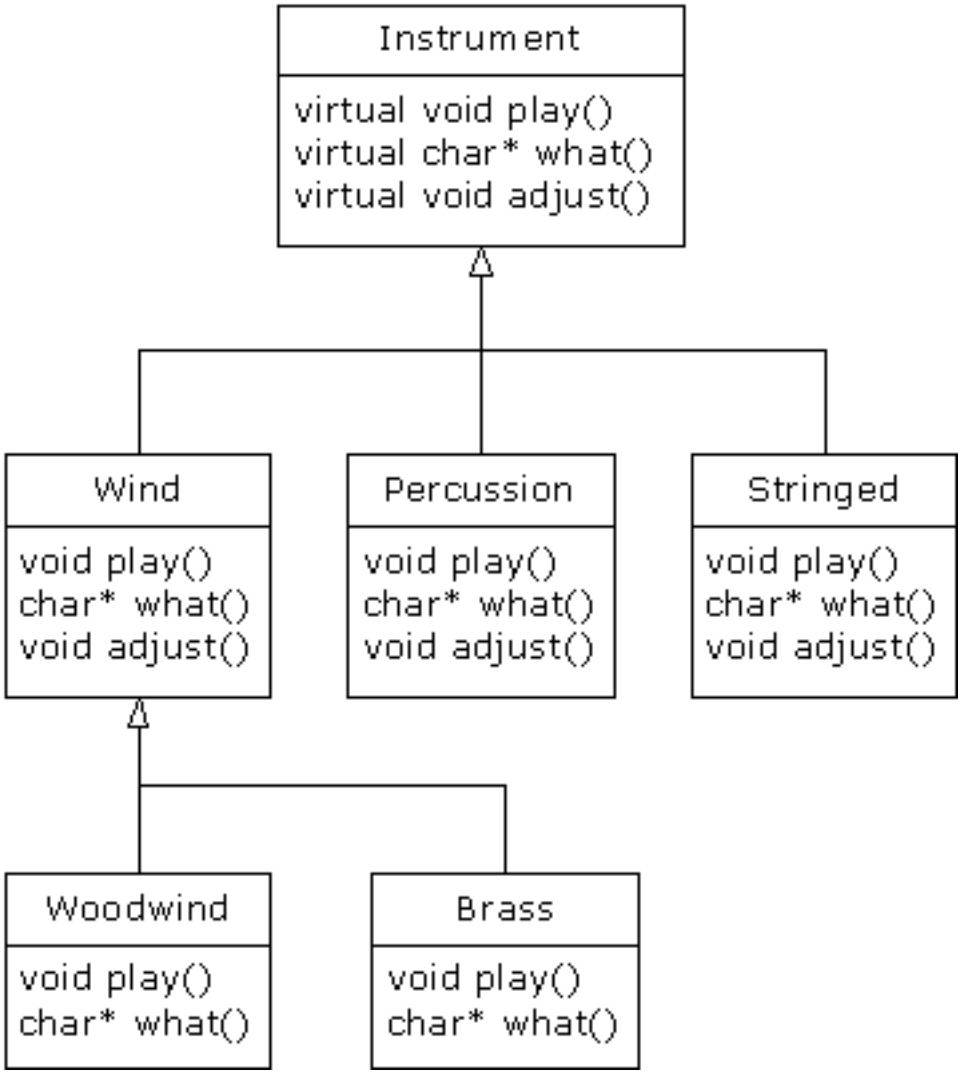
15.7 - Classes de base abstraites et fonctions virtuelles pures

Souvent dans une conception, vous désirez que la classe de base présente *uniquement*une interface pour ses classes dérivées. C'est-à-dire que vous ne voulez pas que quiconque crée un objet de la classe de base, mais seulement d'upcaster vers celle-ci afin que son interface puisse être utilisée. Ceci est réalisé en rendant cette classe *abstraite*, ce qui se produit si vous lui donnez au moins une fonction *virtuelle pure*. Vous pouvez reconnaître une fonction virtuelle pure parce qu'elle utilise le mot-clef **virtual**et est suivie par **= 0**. Si quelqu'un essaie de créer un objet d'une classe abstraite, le compilateur l'en empêche. C'est un outil qui vous permet d'imposer une conception donnée.

Quand une classe abstraite est héritée, toutes les fonctions virtuelles pures doivent être implémentées, sinon la classe qui hérite devient également abstraite. Créer une fonction virtuelle pure vous permet de mettre une fonction membre dans une interface sans avoir à fournir un code potentiellement dénué de sens pour cette fonction membre. En même temps, une fonction virtuelle pure force les classes qui en héritent à fournir pour elle une définition.

Dans tous les exemples d' **Instrument**, les fonctions de la classe de base **Instrument**étaient toujours des fonctions “factices”. Si jamais ces fonctions sont appelées, quelque chose cloche. C'est parce que le but de **Instrument**est de créer une interface commune pour toutes les classes qui en dérivent.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/



La seule raison d'établir une interface commune est afin qu'elle puisse être exprimée différemment pour chaque sous-type. Cela crée une forme élémentaire qui détermine ce qui est commun à toutes les classes dérivées – rien d'autre. Ainsi **Instrument**est un candidat approprié pour devenir une classe abstraite. Vous créez une classe abstraite quand vous voulez seulement manipuler un ensemble de classes à travers une interface commune, mais il n'est pas nécessaire que l'interface commune possède une implémentation (ou tout au moins, une implémentation complète).

Si vous disposez d'un concept comme **Instrument**qui fonctionne comme une classe abstraite, les objets de cette classe n'ont presque jamais de signification. C'est-à-dire que **Instrument**a pour but d'exprimer uniquement l'interface, et pas une implémentation particulière, et donc créer un objet qui soit uniquement **Instrument**n'a aucun sens, et vous voudrez sans doute empêcher l'utilisateur de le faire. Ceci peut être accompli en faisant imprimer un message d'erreur par toutes les fonctions virtuelles d' **Instrument**, mais cela repousse l'apparition de l'information d'erreur jusqu'au moment de l'exécution et requiert un test fiable et exhaustif de la part de l'utilisateur. Il vaut beaucoup mieux traiter le problème à la compilation.

Voici la syntaxe utilisée pour une déclaration virtuelle pure :

```
virtual void f() = 0;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Ce faisant, vous dites au compilateur de réserver un emplacement pour une fonction dans la VTABLE, mais pas de mettre une adresse dans cet emplacement particulier. Dans une classe, même si une seule fonction est déclarée virtuelle pure, la VTABLE est incomplète.

Si la VTABLE d'une classe est incomplète, qu'est-ce que le compilateur est censé faire quand quelqu'un essaie de créer un objet de cette classe ? Il ne peut pas créer sans risque un objet d'une classe abstraite, donc vous recevez un message d'erreur du compilateur. Ainsi, le compilateur garantit la pureté de la classe abstraite. En rendant une classe abstraite, vous vous assurez que le programmeur client ne peut pas en mésuser.

Voici **Instrument4.cpp**, modifié afin d'utiliser les fonctions virtuelles pures. Comme la classe n'a rien d'autre que des fonctions virtuelles pures, on l'appelle une *classe abstraite pure*:

```
                    //: C15:Instrument5.cpp
// Classes de base abstraites pures
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  // Fonctions virtuelles pures :
  virtual void play(note) const = 0;
  virtual char* what() const = 0;
  // Suppose que cela modifiera l'objet :
  virtual void adjust(int) = 0;
};
// Le reste du fichier est similaire...

class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};

// Fonctions identiques à précédemment :
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// Nouvelle fonction :
void f(Instrument& i) { i.adjust(1); }

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
} ///:~
```

Les fonctions virtuelles pures sont utiles parce qu'elles rendent explicite le caractère abstrait d'une classe et disent à la fois au compilateur et à l'utilisateur comment il était prévu de l'utiliser.

Remarquez que les fonctions virtuelles pures empêchent qu'une classe abstraite soit passée dans une fonction *par valeur*. Ainsi, c'est également un moyen d'éviter le *découpage d'objet*(qui sera bientôt décrit). En rendant abstraite une classe, vous pouvez garantir qu'un pointeur ou une référence est toujours utilisé pendant l'upcasting vers cette classe.

Le simple fait qu'une seule fonction virtuelle pure empêche la VTABLE d'être complètée ne signifie pas que vous ne voulez pas de corps pour certaines autres fonctions. Souvent, vous aurez envie d'appeler la version de la classe de base d'une fonction, même si elle est virtuelle. C'est toujours une bonne idée de placer le code commun aussi près que possible de la racine de votre hiérarchie. Non seulement cela économise du code, mais cela permet également une propagation facile des modifications.

15.7.1 - Définitions virtuelles pures

Il est possible de fournir une définition pour une fonction virtuelle pure dans la classe de base. Vous dites toujours au compilateur de ne pas permettre d'objets de cette classe de base abstraite, et les fonctions virtuelles pures doivent toujours être définies dans les classes dérivées afin de créer des objets. Toutefois, il pourrait y avoir une portion de code commune que vous désirez voir appelée par certaines ou toutes les définitions des classes dérivées plutôt que de dupliquer ce code dans chaque fonction.

Voici à quoi ressemble la définition d'une fonction virtuelle pure :

```
                        //: C15:PureVirtualDefinitions.cpp
// Définition de base virtuelles pures
#include <iostream>
using namespace std;

class Pet {
public:
  virtual void speak() const = 0;
  virtual void eat() const = 0;
  // Les définitions virtuelles pures inline sont prohibées :
  //!  virtual void sleep() const = 0 {}
};

// OK, pas définie inline
void Pet::eat() const {
  cout << "Pet::eat()" << endl;
}
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
void Pet::speak() const {
  cout << "Pet::speak()" << endl;
}

class Dog : public Pet {
public:
  // Utilise le code Pet commun :
  void speak() const { Pet::speak(); }
  void eat() const { Pet::eat(); }
};

int main() {
  Dog simba;  // Le chien (Dog, ndt) de Richard
  simba.speak();
  simba.eat();
} ///:~
```

L'emplacement dans la VTABLE de **Pet**est toujours vide, mais il se trouve qu'il y a une fonction de ce nom que vous pouvez appeler dans la classe dérivée.

L'autre bénéfice de cette fonctionnalité est qu'elle vous permet de passer de virtuel ordinaire à pur virtuel sans perturber le code existant. (C'est un moyen pour vous de détecter les classes qui ne surchargent pas cette fonction virtuelle.)

15.8 - L'héritage et la VTABLE

Vous pouvez imaginer ce qu'il se passe quand vous faites hériter et que vous surchargez certaines fonctions virtuelles. Le compilateur crée une nouvelle VTABLE pour votre nouvelle classe, et il insère les adresses de vos nouvelles fonctions en utilisant les adresses des fonctions de la classe de base pour les fonctions virtuelles que nous ne surchargez pas. D'une manière ou d'une autre, pour tout objet qui peut être créé (c'est-à-dire que sa classe ne contient pas de virtuelle pure) il y a toujours un ensemble complet d'adresses de fonctions dans la VTABLE, si bien que vous ne pourrez jamais faire un appel à une adresse qui n'existe pas (ce qui serait un désastre).

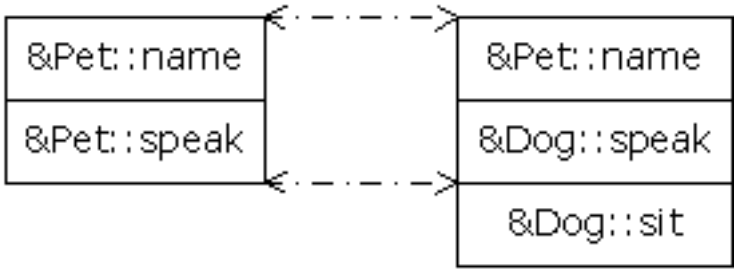
Mais que se passe-t-il quand vous faites hériter et ajoutez de nouvelles fonctions virtuelles dans la classe *dérivée*? Voici un exemple simple :

```
                    //: C15:AddingVirtuals.cpp
// Ajouter des virtuels dans la dérivation
#include <iostream>
#include <string>
using namespace std;

class Pet {
  string pname;
public:
  Pet(const string& petName) : pname(petName) {}
  virtual string name() const { return pname; }
  virtual string speak() const { return ""; }
};

class Dog : public Pet {
  string name;
public:
  Dog(const string& petName) : Pet(petName) {}
  // Nouvelle fonction virtuelle dans la clase Dog :
  virtual string sit() const {
    return Pet::name() + " sits";
  }
  string speak() const { // Surcharge
    return Pet::name() + " says 'Bark!'";
  }
};
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
int main() {
  Pet* p[] = {new Pet("generic"),new Dog("bob")};
  cout << "p[0]->speak() = "
       << p[0]->speak() << endl;
  cout << "p[1]->speak() = "
       << p[1]->speak() << endl;
//! cout << "p[1]->sit() = "
//!      << p[1]->sit() << endl; // Illegal
} ///:~
```

La classe **Pet**contient deux fonctions virtuelles : **speak( )**et **name( )**. **Dog**ajoute une troisième fonction virtuelle appelée **sit( )**, tout en surchargeant le sens de **speak( )**. Un diagramme vous aidera à visualiser ce qu'il se passe. Voici les VTABLE créées par le compilateur pour **Pet**et **Dog**:



Notez que le compilateur indexe la localisation de l'adresse de **speak( )**exactement au même endroit dans la VTABLE de **Dog**que dans celle de **Pet**. De même, si une classe **Pug**héritait de **Dog**, sa version de **sit( )**serait placée dans sa VTABLE exactement à la même position que dans **Dog**. C'est parce que (comme vous l'avez vu avec l'exemple en langage assembleur) le compilateur génère du code qui utilise un simple décalage numérique dans la VTABLE pour sélectionner la fonction virtuelle. Indépendamment du sous-type spécifique auquel appartient l'objet, sa VTABLE est organisée de la même façon, donc les appels aux fonctions virtuelles seront toujours réalisés de manière similaire.

Dans ce cas, toutefois, le compilateur travaille seulement avec un pointeur vers un objet de la classe de base. La classe de base contient seulement les fonctions **speak( )**et **name( )**, ainsi elles sont les seules fonctions que le compilateur vous autorisera à appeler. Comment pourrait-il bien savoir que vous travaillez avec un objet **Dog**, s'il a seulement un pointeur vers un objet de la classe de base ? Ce pointeur pourrait pointer vers quelqu'autre type, qui n'a pas de fonction **sit( )**. Il peut avoir ou ne pas avoir une autre adresse de fonction à cet endroit de la VTABLE, mais dans tous ces cas, faire un appel virtuel à cette adresse de la VTABLE n'est pas ce que vous voulez faire. Aussi, le compilateur fait son travail en vous empêchant de faire des appels virtuels à des fonctions qui n'existent que dans les classes dérivées.

Il y a des cas moins courants pour lesquels vous pouvez savoir que le pointeur pointe en fait vers un objet d'une sous-classe spécifique. Si vous voulez appeler une fonction qui n'existe que dans cette sous-classe, alors vous devez transtyper le pointeur. Vous pouvez supprimer le message d'erreur produit par le programme précédent de cette façon :

```
((Dog*)p[1])->sit()
```

Ici, il se trouve que vous savez que **p[1]**pointe vers un objet **Dog**, mais en général vous ne savez pas cela. Si votre problème est tel que vous devez savoir le type exact de tous les objets, vous devriez le repenser, parce vous n'utilisez probablement pas les fonctions virtuelles correctement. Toutefois, il y a certaines situations dans lesquelles la conception fonctionne mieux (ou bien où vous n'avez pas le choix) si vous connaissez le type exact de tous les objets conservés dans un conteneur générique. C'est le problème de l' *identification de type à l'exécution*(RTTI : run-time type identification, ndt).

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

RTTI est entièrement basé sur le transtypage de pointeurs de la classe de base *vers le bas*en des pointeurs vers la classe dérivée (“vers le haut” ou “vers le bas” sont relatives à un diagramme de classe typique, avec la classe de base au sommet). Transtyper *vers le haut*a lieu automatiquement, sans forcer, parce que c'est complètement sûr. Transtyper *vers le bas*n'est pas sûr parce qu'il n'y a pas d'information à la compilation sur les types réels, si bien que vous devez savoir exactement de quel type est l'objet. Si vous le transtypez dans le mauvais type, vous aurez des problèmes.

La RTTI est décrite plus tard dans ce chapitre, et le Volume 2 de cet ouvrage contient un chapitre consacré à ce sujet.

15.8.1 - Découpage d'objets en tranches

Il y a une différence réelle entre passer les adresses des objets et passer les objets par valeur quand on utilise le polymorphisme. Tous les exemples que vous avez vus ici, et à peu près tous les exemples que vous devriez voir, passent des adresses et non des valeurs. C'est parce que les adresses ont toutes la même taille En réalité, les pointeurs n'ont pas tous la même taille sur toutes les machines. Dans le contexte de cette discussion, toutefois, elles peuvent toutes être considérées identiques., si bien que passer l'adresse d'un objet d'un type dérivé (qui est généralement un objet plus gros) revient au même que passer l'adresse d'un objet du type de base (qui est généralement un objet plus petit). Comme on l'a expliqué auparavant, c'est le but quand on utilise le polymorphisme – du code qui manipule un type de base peut manipuler également de manière transparente des objets de type dérivé.

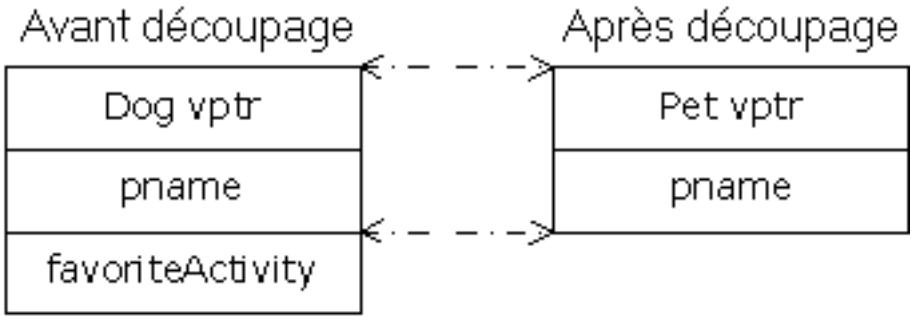
Si vous upcastez vers un objet au lieu de le faire vers un pointeur ou une référence, il va se produire quelque chose qui peut vous surprendre : l'objet est “tranché” jusqu'à que les restes soient le sous-objet qui correspond au type de destination de votre transtypage. Dans l'exemple suivant vous pouvez voir ce qui arrive quand un objet est tranché :

```
                          //: C15:ObjectSlicing.cpp
#include <iostream>
#include <string>
using namespace std;

class Pet {
  string pname;
public:
  Pet(const string& name) : pname(name) {}
  virtual string name() const { return pname; }
  virtual string description() const {
    return "This is " + pname;
  }
};

class Dog : public Pet {
  string favoriteActivity;
public:
  Dog(const string& name, const string& activity)
    : Pet(name), favoriteActivity(activity) {}
  string description() const {
    return Pet::name() + " likes to " +
      favoriteActivity;
  }
};

void describe(Pet p) { // Tranche l'objet
  cout << p.description() << endl;
}

int main() {
  Pet p("Alfred");
  Dog d("Fluffy", "sleep");
  describe(p);
  describe(d);
} ///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

La fonction **describe( )**reçoit un objet de type **Pet** *par valeur*. Elle appelle ensuite la fonction virtuelle **description( )**pour l'objet **Pet**. Dans **main( )**, vous pouvez vous attendre à ce que le premier appel produise “This is Alfred”, et le second “Fluffy likes to sleep”. En fait, les deux appels utilisent la version de **description( )**de la classe de base.

Deux choses se produisent dans ce programme. Premièrement, ârce que **describe( )**accepte un *objet* **Pet**(plutôt qu'un pointeur ou une référence), tous les appels à **describe( )**vont causer la poussée d'un objet de la taille de **Pet**sur la pile et son nettoyage après l'appel. Cela veut dire que si un objet d'une classe dérivée de **Pet**est passé à **describe( )**, le compilateur l'accepte, mais il copie uniquement la portion **Pet**de l'objet. Il *découpe*la portion dérivée de l'objet, comme ceci :



A présent, vous pouvez vous poser des questions concernant l'appel de la fonction virtuelle. **Dog::description( )**utilise des éléments à la fois de **Pet**(qui existe toujours) et de **Dog**, qui n'existe plus parce qu'il a été découpé ! Alors que se passe-t-il quand la fonction virtuelle est appelée ?

Vous êtes sauvés du désastre parce que l'objet est passé par valeur. A cause de cela, le compilateur connaît le type précis de l'objet puisque l'objet dérivé a été contraint de devenir un objet de base. Quand on passe par valeur, le constructeur par recopie pour un objet **Pet**est utilisé, et il initialise le VPTR avec la VTABLE de **Pet**et ne copie que la portion **Pet**de l'objet. Il n'y a pas de constructeur par recopie explicite ici, donc le compilateur en synthétise un. A tout point de vue, l'objet devient vraiment un **Pet**pendant la durée de son découpage.

Le découpage d'objet supprime vraiment des parties de l'objet existant alors qu'il le copie dans le nouvel objet, plutôt que de changer simplement le sens d'une adresse comme quand on utilise un pointeur ou une référence. A cause de cela, upcaster vers un objet n'est pas une opération fréquente ; en fait, c'est généralement quelque chose à surveiller et éviter. Remarquez que dans cet exemple, si **description( )**était une fonction virtuelle pure dans la classe de base (ce qui n'est pas absurde, puisqu'en fait elle ne fait rien dans la classe de base), alors le compilateur empêcherait le découpage de l'objet en tranches, parce que cela ne vous permettrait pas de “créer” un objet du type de base (ce qui se produit quand vous upcastez par valeur). C'est peut-être le plus grand intérêt des fonctions virtuelles pures : éviter le découpage d'objet en générant un message d'erreur à la compilation si quelqu'un essaie de le faire.

15.9 - Surcharge & redéfinition

Dans le chapitre 14, vous avez vu que redéfinir une fonction surchargée dans la classe de base cache toutes les autres versions de cette fonction dans la classe de base. Quand il s'agit de fonctions **virtual**ce comportement est légèrement différent. Considérez une version modifiée de l'exemple **NameHiding.cpp**du chapitre 14 :

```
                    //: C15:NameHiding2.cpp
// Les fonctions virtuelles restreignent la redéfinition
#include <iostream>
#include <string>
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
using namespace std;

class Base {
public:
  virtual int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  virtual void f(string) const {}
  virtual void g() const {}
};

class Derived1 : public Base {
public:
  void g() const {}
};

class Derived2 : public Base {
public:
  // Redéfinition d'une function virtuelle :
  int f() const {
    cout << "Derived2::f()\n";
    return 2;
  }
};

class Derived3 : public Base {
public:
  // Ne peut pas changer le type de retour:
  //! void f() const{ cout << "Derived3::f()\n";}
};

class Derived4 : public Base {
public:
  // Changer la liste d'arguments :
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
};

int main() {
  string s("hello");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // version string cachée
  Derived4 d4;
  x = d4.f(1);
//!  x = d4.f(); // version f() cachée
//!  d4.f(s); // version string cachée
  Base& br = d4; // Upcast
//!  br.f(1); // version dérivée non disponible
  br.f(); // version de base disponible
  br.f(s); // version de base disponible
} ///:~
```

La première chose à noter est que dans **Derived3**, le compilateur ne vous autorisera pas à changer le type de retour d'une fonction surchargée (il le ferait si **f( )**n'était pas virtuelle). Ceci est une restriction importante parce que le compilateur doit garantir que vous pouvez appeler polymorphiquement la fonction depuis la classe de base, car si la classe de base s'attend à ce que **f( )**retourne un **int**, alors la version de **f( )**dans la classe dérivée doit respecter ce contrat autrement il y aura des problèmes.

La règle présentée dans le chapitre 14 fonctionne toujours : si vous redéfinissez un des membres surchargés de la classe de base, les autres versions surchargées deviennent invisibles dans la classe dérivée. Dans **main( )**le code qui teste **Derived4**montre que ceci se produit même si la nouvelle version de **f( )**ne redéfinit pas vraiment l'interface d'une fonction virtuelle existante – les deux versions de **f( )**dans la classe de base sont masquées par **f(int)**. Cependant, si vous transtypez **d4**en **Base**, alors seulement les versions de la classe de base sont disponibles (parce que c'est ce que promet le contrat de la classe de base) et la version de la classe dérivée n'est pas disponible (parce qu'elle n'est pas spécifiée dans la classe de base).

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

15.9.1 - Type de retour covariant

La classe **Derived3**précédente suggère que vous ne pouvez pas modifier le type de retour d'une fonction virtuelle lors d'une redéfinition. Ceci est généralement vrai, mais il existe un cas particulier pour lequel vous pouvez légèrement modifier le type de retour. Si vous retournez un pointeur ou une référence sur une classe de base, alors la version redéfinie de cette fonction peut retourner un pointeur ou une référence sur une classe dérivée de celle retournée par la base. Par exemple :

```
                        //: C15:VariantReturn.cpp
// Reenvoyer un pointeur ou une référence vers un type derivé
// pendant la redéfiniton
#include <iostream>
#include <string>
using namespace std;

class PetFood {
public:
  virtual string foodType() const = 0;
};

class Pet {
public:
  virtual string type() const = 0;
  virtual PetFood* eats() = 0;
};

class Bird : public Pet {
public:
  string type() const { return "Bird"; }
  class BirdFood : public PetFood {
  public:
    string foodType() const {
      return "Bird food";
    }
  };
  // Upcast vers le type de base :
  PetFood* eats() { return &bf; }
private:
  BirdFood bf;
};

class Cat : public Pet {
public:
  string type() const { return "Cat"; }
  class CatFood : public PetFood {
  public:
    string foodType() const { return "Birds"; }
  };
  // Renvoie le type exact à la place :
  CatFood* eats() { return &cf; }
private:
  CatFood cf;
};

int main() {
  Bird b;
  Cat c;
  Pet* p[] = { &b, &c, };
  for(int i = 0; i < sizeof p / sizeof *p; i++)
    cout << p[i]->type() << " eats "
         << p[i]->eats()->foodType() << endl;
  // peut renvoyer le type exact :
  Cat::CatFood* cf = c.eats();
  Bird::BirdFood* bf;
  // ne peut pas renvoyer le type exact :
//!  bf = b.eats();
  // On doit transtyper :
  bf = dynamic_cast<Bird::BirdFood*>(b.eats());
} ///:~
```

La fonction membre **Pet::eats( )**retourne un pointeur sur un **PetFood**. Dans **Bird**, cette fonction membre est

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

surchargée exactement comme dans la classe de base, y compris au niveau du type de retour. Pour cette raison, **Bird::eats( )**effectue un transtypage de **BirdFood**en un **PetFood**.

Mais dans **Cat**, le type de retour de **eats( )**est un pointeur sur **CatFood**, un type dérivé de **PetFood**. Le fait que le type de retour soit hérité du type de retour de la fonction de la classe de base est l'unique raison qui permet à ce code de compiler. Car ce faisant, le contrat est toujours respecté; **eats( )**continue de retourner un pointeur sur **PetFood**.

Si vous pensez polymorphiquement, ceci ne semble pas nécessaire. Pourquoi ne pas simplement transtyper tous les types de retour en **PetFood***, comme l'a fait **Bird::eats( )**? C'est généralement une bonne solution, mais à la fin du **main( )**, vous pouvez voir la différence : **Cat::eats( )**peut retourner le type exact de **PetFood**, alors que la valeur de retour de **Bird::eats( )**doit être transtypée vers son type exact.

Ainsi, être capable de retourner le type exact est un petit peu plus général, et évite de perdre l'information sur le type exact à cause de transtypage automatique. Cependant, retourner le type de base va en général résoudre vos problèmes donc ceci est un dispositif plutôt spécialisé.

15.10 - Fonctions virtuelles & constructeurs

Quand un objet contenant des fonctions virtuelles est créé, son VPTR doit être initialisé pour pointer vers la VTABLE adaptée. Ceci doit être fait avant qu'il n'y ait la moindre possibilité d'appeler une fonction virtuelle. Comme vous pouvez l'imaginer, comme le constructeur a la charge d'amener un objet à l'existence, c'est également le travail du constructeur d'initialiser le VPTR. Le compilateur insère secrètement du code au début du constructeur qui initialise le VPTR. Et, comme décrit au Chapitre 14, si vous ne créez pas explicitement un constructeur pour une classe, le compilateur en synthétisera un pour vous. Si la classe contient des fonctions virtuelles, le constructeur synthétisé incluera le code approprié d'initialisation du VPTR. Ceci a plusieurs implications.

La première concerne l'efficacité. La raison pour l'existence des fonctions **inline**est de réduire le surcoût de temps système de l'appel pour les petites fonctions. Si le C++ ne fournissait pas les fonctions **inline**, le préprocesseur pourrait être utilisé pour créer ces “macros”. Toutefois, le préprocesseur ne dispose pas des concepts d'accès ou de classes, et ainsi ne pourrait pas être utilisé pour créer des macros de fonctions membres. En outre, avec des constructeurs qui doivent avoir du code caché inséré par le compilateur, une macro du préprocesseur ne fonctionnerait pas du tout.

Vous devez être conscient, quand vous chassez des trous d'efficacité, que le compilateur insère du code caché dans votre fonction constructeur. Non seulement il doit initialiser le VPTR, mais il doit aussi vérifier la valeur de **this**(au cas où **operator new**retourne zéro) et appeler les constructeurs des classes de base. Le tout peut avoir un impact sur ce que vous pensiez être un petit appel à une fonction inline. En particulier, la taille du constructeur peut annihiler les économies que vous faisiez en réduisant le surcoût de temps de l'appel de la fonction. Si vous faites beaucoup d'appels inline au constructeur, la taille de votre code peut s'accroître sans aucun bénéfice de vitesse.

Bien sûr, vous ne rendrez probablement tous les petits constructeurs non-inline immédiatement, parce qu'il sont beaucoup plus faciles à écrire quand ils sont inline. Mais quand vous réglez votre code, souvenez-vous d'envisager à supprimer les constructeurs inline.

15.10.1 - Ordre des appels au constructeur

La deuxième facette intéressante des constructeurs et fonctions virtuelles concerne l'ordre des appels au constructeur et la façon dont les appels virtuels sont réalisés au sein des constructeurs.

Tous les constructeurs des classes de base sont toujours appelés dans le constructeur pour une classe héritée.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

C'est logique parce que le constructeur a un travail spécial : s'assurer que l'objet est construit correctement. Une classe dérivée a uniquement accès à ses propres membres, et pas à ceux de la classe de base. Seul, le constructeur de la classe de base peut initialiser proprement ses propres éléments. De ce fait, il est essentiel que tous les constructeurs soient appelés ; autrement, l'objet dans son ensemble ne serait pas correctement construit. C'est pourquoi le compilateur impose un appel au constructeur pour chaque portion d'une classe dérivée. Il appellera le constructeur par défaut si vous n'appelez pas explicitement un constructeur de la classe de base dans la liste d'initialisation du constructeur. S'il n'y a pas de constructeur par défaut, le compilateur se plaindra.

L'ordre des appels au constructeur est important. Quand vous héritez, vous savez tout ce qui concerne la classe de base et vous pouvez accéder à tout membre **public**et **protected**de la classe de base. Ceci signifie que vous devez pouvoir émettre l'hypothèse que tous les membres de la classe de base sont valides quand vous vous trouvez dans la classe dérivée. Dans une fonction membre normale, la construction a déjà eu lieu, si bien que tous les membres de toutes les parties de l'objet ont été construits. Dans le constructeur, toutefois, vous devez pouvoir supposer que tous les membres que vous utilisez ont été construits. La seule façon de le garantir est que le constructeur de la classe de base soit appelé en premier. Ainsi, quand vous vous trouvez dans le constructeur de la classe dérivée, tous les membres de classe de base auxquels vous pouvez accéder ont été initialisés. “Savoir que tous les membres sont valides” au sein du constructeur est également la raison pour laquelle, à chaque fois que c'est possible, vous devriez initialiser tous les objets membres (c'est-à-dire, les objets placés dans la classe en utilisant la composition) dans la liste d'initialisation du constructeur. Si vous suivez cette pratique, vous pouvez faire l'hypothèse que tous les membres des classes de base *et*les objets membres de l'objet courant ont été initialisés.

15.10.2 - Comportement des fonctions virtuelles dans les constructeurs

La hiérarchie des appels de constructeurs soulève un dilemme intéressant. Que se passe-t-il si vous vous trouvez dans un constructeur et que vous appelez une fonction virtuelle ? Dans une fonction membre ordinaire vous pouvez imaginer ce qui va se produire – l'appel virtuel est résolu à l'exécution parce que l'objet ne peut pas savoir s'il appartient à la classe dans laquelle se trouve la fonction membre, ou bien à une classe dérivée de celle-ci. Pour des raisons de cohérence, vous pourriez penser que c'est ce qui devrait se produire dans les constructeurs.

Ce n'est pas le cas. Si vous appelez une fonction virtuelle dans un constructeur, seule la version locale de la fonction est utilisée. C'est-à-dire que le mécanisme virtuel ne fonctionne pas au sein du constructeur.

Ce comportement est logique pour deux raisons. Conceptuellement, le travail du constructeur est d'amener l'objet à l'existence (ce qui est à peine un exploit ordinaire). Dans tout constructeur, l'objet peut être seulement partiellement formé – tout ce que vous pouvez savoir c'est que les objets de la classe de base ont été initialisés, mais vous ne pouvez pas savoir quelles classes héritent de vous. Un appel à une fonction virtuelle, toutefois, porte “au-delà” ou “en dehors” dans la hiérarchie d'héritage. Il appelle une fonction située dans une classe dérivée. Si pouviez faire cela dans un constructeur, vous pourriez appeler une fonction qui risquerait de manipuler des membres qui n'ont pas encore été initialisés, recette qui garantit le désastre.

La deuxième raison est mécanique. Quand un constructeur est appelé, une des premières choses qu'il fait est d'initialiser son VPTR. Toutefois, tout ce qu'il peut savoir c'est qu'il est du type “courant” – le type pour lequel le constructeur a été écrit. Le code du constructeur est incapable de savoir si l'objet est à la base d'une autre classe ou pas. Quand le compilateur génère le code pour ce constructeur, il génère le code pour un constructeur de cette classe, pas une classe de base ni une classe dérivée de celle-ci (parce qu'une classe ne peut pas savoir qui hérite d'elle). Ainsi, le VPTR qu'il utilise doit être pour la VTABLE de *cette*classe. Le VPTR demeure initialisé à cette VTABLE pour le reste de la vie de cet objet *à moins que*ce ne soit pas le dernier appel au constructeur. Si un constructeur plus dérivé est appelé par la suite, ce constructeur initialise le VPTR à *sa*VTABLE, et ainsi de suite, jusqu'à ce que le dernier constructeur se termine. L'état du VPTR est déterminé par le constructeur qui est appelé en dernier. C'est une autre raison pour laquelle les constructeurs sont appelés dans l'ordre de la base vers le plus dérivé.

Mais pendant que cette série d'appels aux constructeurs a lieu, chaque constructeur a initialisé le VPTR à sa

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

propre VTABLE. S'il utilise le mécanisme virtuel pour les appels de fonctions, il ne produira qu'un appel à travers sa propre VTABLE, pas la VTABLE la plus dérivée (comme ce serait le cas après que tous les constructeurs aient été appelés). En outre, beaucoup de compilateurs reconnaissent quand une fonction virtuelle est appelée dans un constructeur, et réalisent une liaison précoce parce qu'ils savent qu'une liaison tardive produira seulement un appel vers la fonction locale. Dans un cas comme dans l'autre, vous n'obtiendrez pas le résultat que vous pouviez attendre initialement d'un appel à une fonction virtuelle dans un constructeur.

15.11 - Destructeurs et destructeurs virtuels

Vous ne pouvez pas utiliser le mot-clef **virtual**avec les constructeurs, mais les destructeurs, eux, peuvent et souvent doivent être virtuels.

Au constructeur incombe la tâche particulière d'assembler un objet morceau par morceau, d'abord en appelant le constructeur de base, puis les constructeurs dérivés dans l'ordre d'héritage (il doit également appeler les constructeurs des objets membres dans le même temps). De la même façon, le destructeur a un travail spécial : il doit désassembler un objet qui peut appartenir à une hiérarchie de classes. Pour ce faire, le compilateur génère du code qui appelle les destructeurs, mais dans l'ordre inverse de l'appel des constructeurs lors de la création. C'est-à-dire que le destructeur démarre par la classe la plus dérivée et trace sa route vers le bas de la hiérarchie, jusqu'à la classe de base. C'est la démarche fiable et donc désirable parce que le constructeur courant peut toujours savoir que les membres de la classe de base sont vivants et actifs. Si vous avez besoin d'appeler une fonction membre de la classe de base dans votre destructeur, cette façon de procéder est sûre. Ainsi, le destructeur peut réaliser son propre nettoyage, puis appeler le destructeur sous-jacent, qui réalisera *son*propre nettoyage, etc. Chaque destructeur sait *de quoi*dérive sa classe, mais pas *ce qui*dérive *d'elle*.

Vous devriez toujours garder à l'esprit que les constructeurs et les destructeurs sont les seuls endroits où cette hiérarchie d'appels doit se produire (et de ce fait, la hiérarchie appropriée est générée par le compilateur). Dans toutes les autres fonctions, seule *cette*fonction sera appelée (et pas les versions de la classe de base), qu'elle soit virtuelle ou non. La seule façon de provoquer un appel à la version de la classe de base d'une fonction (virtuelle ou non) est que vous appeliez *explicitement*cette fonction.

Normalement, l'action du destructeur est assez adéquate. Mais que se passe-t-il si vous voulez manipuler un objet via un pointeur vers sa classe de base (c'est-à-dire que vous manipulez l'objet via son interface générique) ? Cette activité est un objetctif majeur en programmation orientée objet. Le problème apparaît quand vous voulez détruire (**delete**, ndt) un pointeur de ce type pour un objet qui a été créé sur le tas avec **new**. Si le pointeur pointe vers la classe de base, le compilateur ne peut avoir conscience que de la nécessité d'appeler la version de la classe de base du destructeur pendant la destruction par **delete**. Cela a l'air familier ? C'est le même problème que les fonctions virtuelles ont permis de résoudre dans le cas général. Heureusement, les fonctions virtuelles fonctionnent pour les destructeurs comme elles le font pour toutes les autres fonctions, excepté les constructeurs.

```
                    //: C15:VirtualDestructors.cpp
// Comportement du destructeur virtuel vs. non-virtuel
#include <iostream>
using namespace std;

class Base1 {
public:
  ~Base1() { cout << "~Base1()\n"; }
};

class Derived1 : public Base1 {
public:
  ~Derived1() { cout << "~Derived1()\n"; }
};

class Base2 {
public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
class Derived2 : public Base2 {
public:
  ~Derived2() { cout << "~Derived2()\n"; }
};

int main() {
  Base1* bp = new Derived1; // Upcast
  delete bp;
  Base2* b2p = new Derived2; // Upcast
  delete b2p;
} ///:~
```

Quand vous exécuterez le programme, vous verrez que **delete bp**appelle seulement le destructeur de la classe de base, alors que **delete b2p**appelle le destructeur de la classe dérivée suivi par le destucteur de la classe de base, qui est le comportement que nous recherchons. Oublier de rendre un destructeur **virtual**constitue un bug insidieux parce que souvent il n'affecte pas directement le comportement de votre programme, mais peut introduire discrètement une fuite de mémoire. En outre, le fait *qu'un peu*de destruction ait lieu peut encore plus masquer le problème.

Même si le destructeur, comme le constructeur, est une fonction “exceptionnelle”, il est possible que le destructeur soit virtuel parce que l'objet sait déjà de quel type il est (alors qu'il ne le sait pas durant la construction). Une fois qu'un objet a été construit, sont VPTR est initialisé, et les appels aux fonctions virtuelles peuvent avoir lieu.

15.11.1 - Destructeurs virtuels purs

Alors que les destructeurs virtuels purs sont licites en C++ standard, il y a une contrainte supplémentaire quand on les utilise : il faut fournir un corps de fonction pour le destructeur virtuel pur. Ceci a l'air contre intuitif ; comment une fonction virtuelle peut-elle être “pure” si elle a besoin d'un corps de fonction ? Mais si vous gardez à l'esprit que les destructeurs et les constructeurs sont des fonctions spéciales cela paraît plus logique, spécialement si vous vous souvenez que tous les destructeurs dans une hiérarchie de classe sont toujours appelés. Si vous *pouviez*ne pas donner une définition pour un destructeur virtuel pur, quel corps de fonction serait appelé pendant la destruction ? Ainsi, il est absolument nécessaire que le compilateur et l'éditeur de liens imposent l'existence d'un corps de fonction pour un destructeur virtuel pur.

S'il est pur, mais doit avoir un corps de fonction, quel est l'avantage ? La seule différence que vous verrez entre un destructeur virtuel pur ou non pur est que le destructeur virtuel pur rend la classe de base abstraite pure, si bien que vous ne pouvez pas créer un objet de la classe de base (ceci-dit, ce serait également vrai si n'importe quelle autre fonction membre de la classe de base était virtuelle pure).

Les choses deviennent un peu confuses, toutefois, quand une classe hérite d'une classe qui contient un destructeur virtuel pur. Contrairement à toute autre fonction virtuelle pure, il ne vous est *pas*imposé de fournir la définition d'un destructeur virtuel pur dans la classe dérivée. Le fait que ce qui suit peut être compilé et lié le démontre :

```
                        //: C15:UnAbstract.cpp
// Destructeurs virtuels purs
// semble se comporter étrangement

class AbstractBase {
public:
  virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};
// Pas de redéfinition nécessaire du destructeur ?

int main() { Derived d; } ///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Normalement, une fonction virtuelle pure dans une classe de base rendrait la classe dérivée abstraite à moins qu'elle (et toutes les autres fonctions virtuelles pures) reçoive une définition. Mais ici, cela ne semble pas être le cas. Toutefois, rappelez-vous que le compilateur crée *automatiquement*une définition du destructeur pour chaque classe si vous n'en créez pas un vous-même. C'est ce qu'il se produit ici – le destructeur de la classe de base est discrètement redéfini, et ainsi la définition se trouve fournie par le compilateur et **Derived**n'est de fait pas abstraite.

Ceci soulève une question intéressante : quel est l'intérêt d'un destructeur virtuel pur ? Contrairement à une fonction virtuelle pure ordinaire, vous *devez*lui donner un corps de fonction. Dans une classe dérivée, vous n'êtes pas obligés de fournir une définition puisque le compilateur synthétise le destructeur pour vous. Quelle est donc la différence entre un destructeur virtuel ordinaire et un destructeur virtuel pur ?

La seule différence se manifeste quand vous avez une classe qui n'a qu'une seule fonction virtuelle pure : le destructeur. Dans ce cas, le seul effet de la pureté du destructeur est d'éviter l'instanciation de la classe de base. Si n'importe quelle autre fonction virtuelle était pure, elle éviterait l'instanciation de la classe de base, mais s'il n'y en a pas d'autre, alors le destructeur virtuel pur le fera. Ainsi, alors que l'addition d'un destructeur virtuel est essentielle, qu'il soit pur ou non n'est pas tellement important.

Quand vous exécutez l'exemple suivant, vous pouvez constater que le corps de la fonction virtuelle pure est appelé après la version de la classe dérivée, exactement comme avec tout autre destructeur :

```
                        //: C15:PureVirtualDestructors.cpp
// Destructeurs virtuels purs
// requiert un corps de fonction
#include <iostream>
using namespace std;

class Pet {
public:
  virtual ~Pet() = 0;
};

Pet::~Pet() {
  cout << "~Pet()" << endl;
}

class Dog : public Pet {
public:
  ~Dog() {
    cout << "~Dog()" << endl;
  }
};

int main() {
  Pet* p = new Dog; // Transtypage ascendant
  delete p; // Appel au destructeur virtuel
} ///:~
```

Comme principe général, à chaque fois que vous avez une fonction virtuelle dans une classe, vous devriez ajouter immédiatement un destructeur virtuel (même s'il ne fait rien). De cette façon, vous vous immunisez contre des surprises ultérieures.

15.11.2 - Les virtuels dans les destructeurs

Il y a quelque chose qui se produit pendant la destruction à laquelle vous pouvez ne pas vous attendre immédiatement. Si vous êtes dans une fonction membre ordinaire et que vous appelez une fonction virtuelle, cette fonction est appelée en utilisant le mécanisme de liaison tardive. Ce n'est pas vrai avec les destructeurs, virtuels ou non. Dans un destructeur, seule, la version “locale” de la fonction membre est appelée ; le mécanisme virtuel est ignoré.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                        //: C15:VirtualsInDestructors.cpp
// Appels virtuels dans les destructeurs
#include <iostream>
using namespace std;

class Base {
public:
  virtual ~Base() {
    cout << "Base1()\n";
    f();
  }
  virtual void f() { cout << "Base::f()\n"; }
};

class Derived : public Base {
public:
  ~Derived() { cout << "~Derived()\n"; }
  void f() { cout << "Derived::f()\n"; }
};

int main() {
  Base* bp = new Derived; // Transtypage ascendant
  delete bp;
} ///:~
```

Pendant l'appel au destructeur, **Derived::f( )** *n'*est *pas*appelée, même si **f( )**est virtuelle.

Pourquoi cela ? Supposez que le mécanisme virtuel soit utilisé dans le destructeur. Il serait alors possible pour l'appel virtuel de résoudre vers une fonction qui se trouvait “plus bas” (plus dérivée) dans la hiérarchie de l'héritage que le destructeur courant. Mais les destructeurs sont appelés depuis “plus haut” (depuis le destructeur le plus dérivé vers le destructeur de base), si bien que la fonction réellement appelée reposerait sur des portions d'objet qui ont *déjà été détruites*! Au lieu de cela, le compilateur résoud les appels à la compilation et n'appelle que la version “locale” de la fonction. Notez que c'est la même chose pour le constructeur (comme cela a été décrit ci-dessus), mais dans le cas du constructeur l'information de type n'était pas disponible, alors que dans le destructeur l'information (c'est-à-dire le VPTR) est présente, mais n'est pas fiable.

15.11.3 - Créer une hiérarchie basée sur objet

Un problème récurrent dans ce livre pendant la démonstration des classes de conteneurs **Stack**et **Stash**est celui de la “propriété”. Le “propriétaire” fait référence ce qui est responsable de l'appel de **delete**pour les objets qui ont été créés dynamiquement (en utilisant **new**). Le problème quand on utilise des conteneurs est qu'ils doivent être suffisamment flexibles pour contenir différents types d'objets. Pour ce faire, les conteneurs contiennent des pointeurs **void**et ne connaissent donc pas le type de l'objet qu'ils ont contenu. Effacer un pointeur **void**n'appelle pas le destructeur, et le conteneur ne peut donc pas être responsable du nettoyage de ses objets.

Une solution a été présentée dans l'exemple **C14:InheritStack.cpp**, dans lequel la **Stack**était héritée dans une nouvelle classe qui acceptait et produisait uniquement des pointeurs **string**. Comme elle savait qu'elle ne pouvait contenir que des pointeurs vers des objets **string**, elle pouvait les effacer proprement. C'était une bonne solution, mais qui vous imposait d'hériter une nouvelle classe de conteneur pour chaque type que vous voulez contenir dans le conteneur. (Bien que cela semble fastidieux pour le moment, cela fonctionnera plutôt bien au Chapitre 16, quand les templates seront introduits.)

Le problème est que vous voulez que le conteneur contienne plus d'un type, mais vous ne voulez pas utiliser des pointeurs **void**. Une autre solution consiste à utiliser le polymorphisme en forçant tous les objets inclus dans le conteneur à être hérités de la même classe de base. C'est-à-dire que le conteneur contient les objets de la classe de base, et qu'ensuite vous pouvez appeler les fonctions virtuelles – notamment, vous pouvez appeler les destructeurs virtuels pour résoudre le problème de la propriété.

Cette solution utilise ce que l'on appelle une *hiérarchie à racine unique*ou une *hiérarchie basée sur objet*(parce que

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

la classe racine de la hiérarchie est généralement appelée “Object”). Il s'avère qu'il y a beaucoup d'autres avantages à utiliser une hiérarchie à racine unique ; en fait, tous les autres langages orientés objet sauf le C++ imposent l'utilisation d'une telle hiérarchie – quand vous créez une classe, vous l'héritez automatiquement soit directement soit indirectement d'une classe commune, classe de base qui a été établie par le créateur du langage. En C++, on a considéré que l'usage imposé de cette classe de base commune coûterait trop de temps système, et elle a été abandonnée. Toutefois, vous pouvez choisir d'utiliser une classe de base comune dans vos propres projets, et ce sujet sera examiné plus en détails dans le Volume 2 de ce livre.

Pour résoudre le problème de la propriété, nous pouvons créer un **Object**extrêmement simple pour la classe de base, qui contient uniquement un destructeur virtuel. La **Stack**peut alors contenir des classes héritées d' **Object**:

```
                          //: C15:OStack.h
// Utiliser une hiérarchie à racine unique
#ifndef OSTACK_H
#define OSTACK_H

class Object {
public:
  virtual ~Object() = 0;
};

// Définition requise :
inline Object::~Object() {}

class Stack {
  struct Link {
    Object* data;
    Link* next;
    Link(Object* dat, Link* nxt) :
      data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack(){
    while(head)
      delete pop();
  }
  void push(Object* dat) {
    head = new Link(dat, head);
  }
  Object* peek() const {
    return head ? head->data : 0;
  }
  Object* pop() {
    if(head == 0) return 0;
    Object* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};
#endif // OSTACK_H ///:~
```

Pour simplifier les choses en conservant tout dans le fichier d'en-tête, la définition (requise)du destructeur virtuel pur est inlinée dans le fichier d'en-tête, et **pop( )**(qui peut être considérée trop grande pour être inline) est également inlinée.

Les objets **Link**contiennent maintenant des pointeurs vers **Objet**plutôt que des pointeurs **void**, et la **Stack**acceptera et retournera uniquement des pointeurs **Object**. Maintenant, **Stack**est bien plus flexible, puisqu'elle contiendra beaucoup de types différents mais détruira également tout objet qui sont laissés dans la **Stack**. La nouvelle limitation (qui sera finalement dépassée quand les templates seront appliqués au problème dans le Chapitre 16) est que tout ce qui est placé dans **Stack**doit hériter d' **Object**. C'est valable si vous créer votre classe à partir de zéro, mais que faire si vous disposez déjà d'une classe telle que **string**que vous voulez être capable de mettre dans **Stack**? Dans ce cas, la nouvelle classe doit être les deux : **string**et **Object**, ce qui signifie qu'elle doit hériter des deux classes. On appelle cela l' *héritage multiple*et c'est le sujet d'un chapitre entier dans le

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Volume 2 de ce livre (téléchargeable depuis *www.BruceEckel.com*). Quand vous lirez ce chapitre, vous verrez que l'héritage multiple peut être très complexe, et c'est une fonctionnalité que vous devriez utiliser avec parcimonie. Dans cette situation, toutefois, tout est suffisamment simple pour que nous ne tombions dans aucun piège de l'héritage multiple :

```
                        //: C15:OStackTest.cpp
//{T} OStackTest.cpp
#include "OStack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

// Utilise l'héritage multiple. Nous voulons à la fois
// un string et un Object :
class MyString: public string, public Object {
public:
  ~MyString() {
    cout << "deleting string: " << *this << endl;
  }
  MyString(string s) : string(s) {}
};

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // Le nom de fichier (File Name) est un argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  string line;
  // Lit le fichier et stocke les lignes dans la stack :
  while(getline(in, line))
    textlines.push(new MyString(line));
  // Dépile des lignes de la stack :
  MyString* s;
  for(int i = 0; i < 10; i++) {
    if((s=(MyString*)textlines.pop())==0) break;
    cout << *s << endl;
    delete s;
  }
  cout << "Letting the destructor do the rest:"
    << endl;
} ///:~
```

Bien que ce soit similaire à la version précédente du programme test pour **Stack**, vous remarquerez que seuls 10 éléments sont dépilés de la pile (stack, ndt), ce qui signifie qu'il y a probablement des objets qui restent dedans. Comme la **Stack**sait qu'elle contient des **Object**, le destructeur peut nettoyer les choses proprement, et vous le verrez dans la sortie du programme, puisque les objets **MyString**impriment des messages lorsqu'ils sont détruits.

Créer des conteneurs qui contiennt des **Object**n'est pas une approche déraisonnable – *si*vous avez une hiérarchie à racine unique (imposée soit par le langage ou par la condition que toute classe hérite d' **Object**). Dans ce cas, il est garantit que tout soit un **Object**et il n'est donc pas très compliqué d'utiliser les conteneurs. En C++, toutefois, vous ne pouvez pas vous attendre à ce que ce soit le cas pour toutes les classes, et vous êtes donc obligés d'affronter l'héritage multiple si vous prenez cette approche. Vous verrez au Chapitre 16 que les templates résolvent ce problème d'une manière beaucoup plus simple et plus élégante.

15.12 - Surcharge d'opérateur

Vous pouvez rendre les opérateurs **virtuels**exactement comme les autres fonctions membres. Implémenter des opérateurs **virtuels**devient souvent déroutant, toutefois, parce que vous pouvez manipuler deux objets, chacun d'un type inconnu. C'est généralement le cas avec les composants mathématiques (pour lesquels vous surchargez souvent les opérateurs). Par exemple, considérez un système qui traite de matrices, de vecteurs et de valeurs scalaires, toutes les trois étant dérivées de la classe **Math**:

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                    //: C15:OperatorPolymorphism.cpp
// Polymorphisme avec des opérateurs surchargés
#include <iostream>
using namespace std;

class Matrix;
class Scalar;
class Vector;

class Math {
public:
  virtual Math& operator*(Math& rv) = 0;
  virtual Math& multiply(Matrix*) = 0;
  virtual Math& multiply(Scalar*) = 0;
  virtual Math& multiply(Vector*) = 0;
  virtual ~Math() {}
};

class Matrix : public Math {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2ème répartition
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Matrix" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Matrix" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Matrix" << endl;
    return *this;
  }
};

class Scalar : public Math  {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2ème répartition
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Scalar" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Scalar" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Scalar" << endl;
    return *this;
  }
};

class Vector : public Math  {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2ème répartition
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Vector" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Vector" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Vector" << endl;
    return *this;
  }
};

int main() {
  Matrix m; Vector v; Scalar s;
  Math* math[] = { &m, &v, &s };
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    for(int i = 0; i < 3; i++)
      for(int j = 0; j < 3; j++) {
        Math& m1 = *math[i];
        Math& m2 = *math[j];
        m1 * m2;
      }
  } ///:~
```

Pour simplifier, seul **operator***a été surchargé. Le but est d'être capable de multiplier n'importe quel couple d'objets **Math**et de produire le résultat désiré – et notez que multiplier un vecteur par une matrice est une opération très différente de la multiplication d'une matrice par un vecteur.

Le problème est que, dans **main( )**, l'expression **m1 * m2**contient deux références vers **Math**upcastées, et donc deux objets de type inconnu. Une fonction virtuelle n'est capable de réaliser qu'une seule répartition – c'est-à-dire déterminer le type d'un seul objet inconnu. Pour déterminer le type des deux objets une technique baptisée *répartition multiple*est utilisée dans cet exemple, par laquelle ce qui semble être un appel unique à une fonction virtuelle résulte en un deuxième appel à une fonction virtuelle. Lorsque ce deuxième appel est effectué, vous avez déterminé le type des deux objets, et pouvez accomplir l'action appropriée. Ce n'est pas transparent à première vue, mais si vous examinez cet exemple quelque temps cela devrait commencer à prendre du sens. Ce sujet est exploré plus en profondeur dans le chapitre sur les Design Pattern dans le Volume 2, que vous pouvez télécharger sur *www.BruceEckel.com*(et bientôt *www.developpez.com*, ndt).

15.13 - Transtypage descendant

Vous pouvez le deviner, comme il existe une chose telle que le transtypage ascendant – monter d'un degré dans la hiérarchie de l'héritage – il devait également y avoir un *transtypage descendant*pour descendre cette même hiérarchie. Mais le transtypage ascendant est facile puisque comme vous remontez la hiérarchie d'héritage les classes convergent toujours vers des classes plus générales. C'est-à-dire que quand vous réalisez un transtypage ascendant vous êtes toujours clairement dérivé d'une classe ancestrale (typiquement une seule, sauf dans le cas de l'héritage multiple)mais quand vous transtypez de manière descendante il y a généralement plusieurs possibilités vers lesquelles vous pouvez transtyper. Plus précisément, un **Circle**(cercle, ndt) est un type de **Shape**(forme, ndt) (transtypage ascendant), mais si vous essayez de transtyper un **Shape**en descendant cela pourrait être un **Circle**, un **Square**(carré, ndt), un **Triangle**, etc. Le dilemme est alors de trouver une manière sûre de réaliser le transtypage descendant. (Mais un problème encore plus important est de vous demander avant tout pourquoi vous avez besoin de le faire au lieu d'utiliser simplement le polymorphisme pour deviner automatiquement le type correct. La manière d'éviter le transtypage descendant est abordé dans le Volume 2 de ce livre.)

Le C++ fournit un *transtypage explicite*spécial (introduit au Chapitre 3) appelé **dynamic_cast**qui est une opération de *transtypage descendant fiable*. Quand vous utilisez le **dynamic_cast**pour essayer de transtyper de manière descendante vers un type particulier, la valeur de retour sera un pointeur vers le type désiré uniquement si le transtypage est propre et a réussi, autrement cela retournera un zéro pour indiquer que ce n'était pas le type correct. Voici un petit exemple :

```
                    //: C15:DynamicCast.cpp
#include <iostream>
using namespace std;

class Pet { public: virtual ~Pet(){}};
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
  Pet* b = new Cat; // Transtypage ascendant
  // Essaye de le transtyper en Dog* :
  Dog* d1 = dynamic_cast<Dog*>(b);
  // Essaye de le transtyper en Cat* :
  Cat* d2 = dynamic_cast<Cat*>(b);
  cout << "d1 = " << (long)d1 << endl;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  cout << "d2 = " << (long)d2 << endl;
} ///:~
```

Quand vous utilisez **dynamic_cast**, vous devez travailler ave une hiérarchie polymorphique vraie – qui contient des fonctions virtuelles – parce que le **dynamic_cast**utilise des informations stockées dans la VTABLE pour déterminer le type réel. Ici, la classe de base contient un destructeur virtuel et cela suffit. Dans **main( )**, un pointeur **Cat**est transtypé en un **Pet**(transtypage ascendant), puis on tente un transtypage descendant vers un pointeur **Dog**et un pointeur **Cat**. Les deux pointeurs sont imprimés, et vous verrez quand vous exécuterez le programme que le transtypage descendant incorrect produit un résultat nul. Bien sûr, à chaque fois que vous réalisez un transtypage descendant vous avez la charge de vérifier que le résultat de l'opération n'est pas nul. Vous ne deriez également pas supposer que le pointeur sera exactement identique, parce qu'il arrive que des ajustements de pointeur se réalisent pendant les transtypages ascendant et descendant (en particulier avec l'héritage multiple).

Un **dynamic_cast**requiert un peu de temps système supplémentaire pour s'exécuter ; pas beaucoup, mais si vous réalisez de nombreux **dynamic_cast**(auquel cas vous devriez sérieusement remettre en cause votre conception) cela peut devenir un probème en terme de performance. Dans certains cas, vous pouvez savoir quelque chose de précis pendant le transtypage descendant qui vous permette de dire à coup sûr à quel type vous avez à faire, et dans ce cas le temps supplémentaire du **dynamic_cast**n'est plus nécessaire et vous pouvez utiliser un **static_cast**à sa place. Voici un exemple montrant comment cela peut marcher :

```
                    //: C15:StaticHierarchyNavigation.cpp
// Naviguer dans la hiérarchie de classe avec static_cast
#include <iostream>
#include <typeinfo>
using namespace std;

class Shape { public: virtual ~Shape() {}; };
class Circle : public Shape {};
class Square : public Shape {};
class Other {};

int main() {
  Circle c;
  Shape* s = &c; // Transtypage ascendant : normal et OK
  // Plus explicite mais pas nécessaire :
  s = static_cast<Shape*>(&c);
  // (Comme le transtypage ascendant est un opération sûre et
  // courante, le transtypage devient encombrant)
  Circle* cp = 0;
  Square* sp = 0;
  // Navgation statique dans les hiérarchies de classes
  // requiert un type d'information supplémentaire :
  if(typeid(s) == typeid(cp)) // RTTI C++
    cp = static_cast<Circle*>(s);
  if(typeid(s) == typeid(sp))
    sp = static_cast<Square*>(s);
  if(cp != 0)
    cout << "It's a circle!" << endl;
  if(sp != 0)
    cout << "It's a square!" << endl;
  // La navigation statique est SEULEMENT un hack d'efficacité ;
  // le dynamic_cast est toujours plus sûr. Toutefois :
  // Other* op = static_cast<Other*>(s);
  // Donne de manière appropriée un message d'erreur, alors que
  Other* op2 = (Other*)s;
  // ne le fait pas
} ///:~
```

Dans ce programme, une nouvelle fonctionnalité est utilisée que l'on ne décrira pas complètement avant le Volume 2 de ce livre, où un chapitre est consacré au sujet du mécanisme d' *information de type à l'exécution*(*run-time type information*, ndt, ou RTTI) du C++. Le RTTI vous permet de découvrir une information de type qui a été perdu par le transtypage ascendant. Le **dynamic_cast**est en fait une forme de RTTI. Ici, le mot-clef **typeid**(déclaré dans le fichier d'en-tête **<typeinfo>**) est utilisé pour détecter les types des pointeurs. Vous pouvez constater que le type du pointeur **Shape**transtypé vers le haut est successivement comparé à un pointeur **Circle**et à un pointeur **Square**pour voir s'il y a correspondance. Le RTTI contient plus de choses que seulement **typeid**, et vous pouvez

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

aussi imaginer qu'il serait relativement facile d'implémenter votre propre système d'information de type en utilisant une fonction virtuelle.

Un objet **Circle**est créé et une opération de transtypage ascendant est réalisée vers un pointeur **Shape**; la deuxième version de l'expression montre comment utiliser **static_cast**pour être plus explicite en ce qui concerne le transtypage ascendant. Toutefois, comme un transtypage ascendant est toujours fiable et que c'est une opération courante, je considère qu'un transtypage explicite n'est dans ce cas pas nécessaire et se révèle encombrant.

Le RTTI est utilisé pour déterminer le type, puis **static_cast**est utilisé pour réaliser le transtypage descendant. Remarquez toutefois quand dans cette approche le processus est en fait le même que quand on utilise **dynamic_cast**, et le programmeur client doit faire des tests pour découvrir le transtypage qui a vraiment réussi. Typiquement, vous aurez intérêt à vous trouver dans une situation plus déterministe que l'exemple ci-dessus avant d'utiliser **static_cast**de préférence à **dynamic_cast**(et, encore une fois, vous avez intérêt à examiner soigneusement votre conception avant d'utiliser **dynamic_cast**).

Si une hiérarchie de classes n'a pas de fonctions virtuelles (ce qui est une conception discutable) ou si vous avez une autre information qui vous permet de réaliser un transtypage descendant fiable, il est un peu plus rapide de réaliser le transtypage avec **static_cast**plutôt qu'avec **dynamic_cast**. En outre, **static_cast**ne vous permettra pas de transtyper hors de la hiérarchie, comme le transtypage traditionnel laisserait faire, et est donc plus sûr. Toutefois, naviguer dans la hiérarchie des classes en statique est toujours risqué et vous devriez utiliser **dynamic_cast**à moins que vous ne vous trouviez dans une situation spéciale.

15.14 - Résumé

Le polymorphisme – implémenté en C++ à l'aide des fonctions virtuelles – signifie “formes variées”. Dans la programmation orientée objet, vous avez le même visage (l'interface commune dans la classe de base) et plusieurs formes utilisant ce visage : les différentes versions des fonctions virtuelles.

Vous avez vu dans ce chapitre qu'il est impossible de comprendre, ou même de créer, un exemple de polymorphisme sans utiliser l'abstraction des données et l'héritage. Le polymorphisme est une fonctionnalité qui ne peut être vue isolément (comme les instructions **const**ou **switch**, par exemple), mais au lieu de cela ne fonctionne que de concert, comme un élément d'un “grand tableau” des relations entre classes. Les gens sont souvent embrouillés par d'autres aspects du C++, qui ne sont pas orientés objet, comme la surcharge et les arguments par défaut, qui sont parfois présentés comme orientés objet. Ne vous laissez pas avoir : si ce n'est pas de la liaison tardive, ce n'est pas du polymorphisme.

Pour utiliser le polymorphisme – et donc les techniques orientées objet – efficacement dans vos programmes, vous devez élargir votre vision de la programmation pour inclure non seulement les membres et les messages d'une classe individuelle, mais également ce qu'il y a de commun entre des classes et leurs relations les unes avec les autres. Bien que cela demande des efforts significatifs, c'est un jeu qui vaut la chandelle, parce que les résultats sont un développement plus rapide, une meilleure organisation du code, des programmes extensibles et une maintenance plus facile du code.

Le polymorphisme complète les fonctionnalités orientées objet du langage, mais il y a deux autres fonctionnalités majeures en C++ : les templates (qui sont introduits au Chapitre 16 et couverts plus en détails dans le Volume 2), et la gestion des exceptions (qui est couverte dans le Volume 2). Ces fonctionnalités vous apportent autant d'augmentation de la puissance de programmation que chacune des fonctionnalités orientées objet : typage abstrait des données , héritage et polymorphisme.

15.15 - Exercices

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Les solutions à certains exercices peuvent être trouvées dans le document électronique *The Thinking in C++ Annotated Solution Guide*, disponible pour une somme modique sur www.BruceEckel.com.

1 Créez une hiérarchie simple de “shape” (formes ndt) : une classe de base appelée **Shape**et des classes dérivées appelées **Circle**, **Square**, et **Triangle**. Dans la classe de base, créez une fonction virtuelle appelée **draw( )**et redéfinissez-là dans les classes dérivées. Faites un tableau de pointeurs vers les objets **Shape**que vous créez sur le tas (et ainsi réalisez un transtypage ascendant des pointeurs), et appelez **draw( )**via les pointeurs de la classe de base, pour vérifier le comportement de la fonction virtuelle. Si votre debogueur le supporte, suivez le code pas à pas.
2 Modifiez l'exercice 1 de telle sorte que **draw( )**soit une fonction virtuelle pure. Essayez de créer un objet de type **Shape**. Essayez d'appeler la fonction virtuelle pure dans le constructeur et voyez ce qui se produit. En laissant **draw( )**virtuelle pure, donnez-lui une définition.
3 En développant l'exercice 2, créez une fonction qui prenne un objet **Shape** *par valeur*et essayez de réaliser le transtypage ascendant d'un objet dérivé passé comme argument. Voyez ce qui se produit. Corrigez la fonction en prenant une référence vers l'objet **Shape**.
4 Modifiez **C14:Combined.cpp**pour que **f( )**soit **virtual**dans la classe de base. Modifiez **main( )**pour réaliser un transtypage ascendant et un appel virtuel.
5 Modifez **Instrument3.cpp**en ajoutant une fonction **virtual prepare( )**. Appelez **prepare( )**depuis **tune( )**.
6 Créez une hiérarchie d'héritage de **Rodent**: **Mouse**, **Gerbil**, **Hamster**, etc. Dans la classe de base, fournissez des méthodes qui sont communes à tous les **Rodent**, et redéfinissez-les dans les classes dérivées pour générer des comportements différents selon le type spécifique de **Rodent**. Créez un tableau de pointeurs vers **Rodent**, remplissez-le avec différents types spécifiques de **Rodent**, et appelez vos méthodes de la classe de base pour voir ce qui se produit.
7 Modifiez l'exercice 6 pour utiliser un **vector<Rodent*>**au lieu d'un tableau de pointeurs. Assurez-vous que la mémoire est nettoyée proprement.
8 En partant de la hiérarchie **Rodent**précédente, faites hériter **BlueHamster**de **Hamster**(oui, un tel animal existe ; j'en ai eu un quand j'étais enfant), redéfinissez les méthodes de la classe de base, et montrez que le code qui appelle les méthodes de la classe de base n'a pas besoin d'être modifié pour s''asapter au nouveau type.
9 En partant de la hiérarchie **Rodent**précédente, ajoutez un destructeur non virtuel, créez un objet de classe **Hamster**en utilisant **new**, réalisez un transtypage ascendant du pointeur vers un **Rodent***, et **detruisez**le pointeur pour montrer qu'il n'appelle pas tous les destructeurs dans la hiérarchie. Rendez le destructeur virtuel et démontrez que le comportement est à présent correct.
10 En partant de la hiérarchie **Rodent**précédente, modifiez **Rodent**pour en faire une classe de base abstraite pure.
11 Créez un système de contrôle aérien avec la classe de base **Aircraft**et différents types dérivés. Créez une classe **Tower**avecc un **vector<Aircraft*>**qui envoie les messages appropriés aux différents aircraft sous son contrôle.
12 Créez un modèle de serre en héritant différents types de **Plantes**et en construisant des mécanismes dans votre serre qui prennnent soin des plantes.
13 Dans **Early.cpp**, faites de **Pet**une classe abstraite pure.
14 Dans **AddingVirtuals.cpp**, rendez toutes les fonctions membre de **Pet**virtuelles pures, mais donnez une définition pour **name( )**. Corrigez **Dog**comme il faut, en utilisant la définition de la classe de base de **name( )**.
15 Ecrivez un petit programme pour montrer la différencee entre appeler une fonction virtuelle dans une fonction membre normale et appeler une fonction virtuelle dans un constructeur. Le programme devrait prouver que les deux appels produisent des résultats différents.
16 Modifiez **VirtualsInDestructors.cpp**en faisant hériter une classe de **Derived**et en redéfinissant **f( )**et le destructeur. Dans **main( )**, créez une objet de votre nouveau type puis réalisez un transtypage ascendant et détruisez-le avec **delete**.
17 Prenez l'exercice 16 et ajoutez des appels à **f( )**dans chaque destructeur. Expliquez ce qui se passe.
18 Créez une classe qui contient une donnée membre et une classe dérivée qui ajoute une autre donnée membre. Ecrivez une fonction non-membre qui prend un objet de la classe de base *par valeur*et affiche la taille de cet objet en utilisant **sizeof**. Dans **main( )**, créez un objet de la classe dérivée, affichez sa taille, puis appelez votre fonction. Expliquez ce qui se passe.
19 Créez un exemple simple d'un appel à une fonction virtuelle et générez la sortie assembleur. Localisez le

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

code assembleur pour l'appel virtuel et tracez et expliquez le code.

20 Ecrivez une classe avec une fonction virtuelle et une non virtuelle. Faites-en hériter une nouvelle classe, créez un objet de cette classe, et faites un transtypage ascendant vers un pointeur du type de la classe de base. Utilisez la fonction **clock( )**qui se trouve dans **<ctime>**(vous aurez besoin de chercher cela dans votre guide local de la bibliothèque C) pour mesurer la différence entre un appel virtuel et non virtuel. Vous aurez besoin de faire plusieurs appels à chaque fonction dans votre boucle de chronométrage afin de voir la différence.

21 Modifiez **C14:Order.cpp**en ajoutant une fonction virtuelle dans la classe de base de la macro **CLASS**(faites-la afficher quelque chose) et en rendant le destructeur virtuel. Créez des objets des différentes sous-classes et transtypez les vers la classe de base. Vérifiez que le comportement virtuel fonctionne et que la construction et la destruction appropriées ont lieu.

22 Créez une classe avec trois focntions virtuelles suchargées. Héritez une nouvelle classe de celle-ci et redéfinissez une des fonctions. Créez un objet de votre classe dérivée. Pouvez-vous appeler toutes les fonctions de la classe de base via l'objet de la classe dérivée ? Transtypez l'adresse de l'objet vers la base. Pouvez-vous appeler les trois fonctions via la base ? Supprimez la redéfinition dans la classe dérivée. A présent, pouvez-vous appeler toutes les fonctions de la classe de base via l'objet de la classe dérivée ?

23 Modifiez **VariantReturn.cpp**pour montrer que son comportement fonctionne avec les références comme avec les pointeurs.

24 Dans **Early.cpp**, comment pouvez-vous dire si le compilateur fait l'appel en utilisant la liaison précoce ou retardée ? Déterminez ce qu'il en est pour votre propre compilateur.

25 Créez une classe de base contenant une fonction **clone( )**qui retourne un pointeur vers une *copie*de l'objet courant. Dérivez deux sous-classes qui redéfinissent **clone( )**pour retourner des copies de leur type spécifique. Dans **main( )**, créez puis transtypez des objets de vos deux types dérivés, puis appelez **clone( )**pour chacun d'eux et vérifiez que les copies clonées sont du sous-type correct. Testez votre fonction **clone( )**pour que vous retourniez le type de base, puis essayez de retourner le type dérivé exact. Pouvez-vous imaginer des situations pour lesquelles cette dernière approche soit nécessaire ?

26 Modifiez **OStackTest.cpp**en créant votre propre classe, puis en faisant une dérivée multiple avec **Object**pour créer quelque chose qui peut être placé dans **Stack**. Testez votre classe dans **main( )**.

27 Ajoutez un type appelé **Tensor**à **OperatorPolymorphism.cpp**.

28 (Intermédiaire) Créez une **classe X**de base sans donnée membre ni constructeur, mais avec une fonction virtuelle. Créez une **classe Y**qui hérite de **X**, mais sans constructeur explicite. Générer le code assembleur et examinez-le pour déterminer si un constructeur est créé et appelé pour **X**, et si c'est le cas, ce que fait le code. Expliquez ce que vous découvrez. **X**n'a pas de constructeur par défaut, alors pourquoi le compilateur ne se plaint-il pas ?

29 (Intermédiaire) Modifiez l'exercise 28 en écrivant des constructeurs pour les deux classes afin que chaque constructeur appelle une fonction virtuelle. Générez le code assembleur. Déterminez où le VPTR est affecté dans chaque constructeur. Est-ce que le mécanisme virtuel est utilisé par votre compilateur dans le constructeur ? Etablissez pourquoi la version locale de la fonction est toujours appelée.

30 (Avancé) Si les appels aux fonctions contenant un objet passé par valeur *n'étaient pas*lié précocément, un appel virtuel pourrait accéder à des parties qui n'existent pas. Est-ce possible ? Ecrivez du code qui force un appel virtuel, et voyez si cela provoque un plantage. Pour expliquer le comportement, examinez ce qui se passe quand vous passez un objet par valeur.

31 (Avancé) Trouvez exactement combien de temps supplémentaire est requis pour un appel à une fonction virtuelle en allant consulter l'information concernant le langage assembleur de votre processeur ou un autre manuel technique et trouvez le nombre de cycles d'horloge requis pour un simple appel contre celui requis pour les instructions de l'appel de la fonction virtuelle.

32 Déterminez la taille du VPTR avec **sizeof**pour vorte implémentation. A présent, héritez multiple de deux classes qui contiennent des fonctions virtuelles. Avez-vous obtenu un ou deux VPTR dans la classe dérivée ?

33 Créez une classe avec des données membres et des fonctions virtuelles. Ecrivez une fonction qui regarde le contenu mémoire dans un objet de votre classe et en affiche les différentes parties de celui-ci. Pour ce faire, vous aurez besoin d'expérimenter et de découvrir par itérations où est localisé le VPTR dans l'objet.

34 Imaginez que les fonctions virtuelles n'existent pas, et modifiez **Instrument4.cpp**afin qu'elle utilise **dynamic_cast**pour faire l'équivalent des appels virtuels. Expliquez pourquoi c'est une mauvaise idée.

35 Modifiez **StaticHierarchyNavigation.cpp**afin qu'au lieu d'utiliser la RTTI du C++ vous créiez votre propre RTTI via une fonction virtuelle dans la classe de base appelée **whatAmI( )**et une **énumération { Circles,**

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Squares };.

36 Partez de **PointerToMemberOperator.cpp**du chapitre 12 et montrez que le polymorphisme fonctionne toujours avec des pointeurs vers membres, même si **operator->***est surchargé.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

16 - Introduction aux Templates

La composition et l'héritage fournissent un moyen de réutiliser le code objet. La fonctionnalité *template*fournit, en C++, un moyen de réutiliser du code *source*.

Bien que les templates du C++ soient un outil de programmation à but générique, quand ils furent introduits dans le langage, ils semblaient décourager l'utilisation de hiérarchies de conteneurs de classes basées sur les objets (démontré à la fin du chapitre 15). Par exemple, les conteneurs et algorithmes du C++ standard (expliqués dans deux chapitre dans le Volume 2 de ce livre, téléchargeable sur *www.BruceEckel.com*) sont bâtis exclusivement avec des templates et sont relativement faciles à utiliser pour le programmeur.

Ce chapitre montre non seulement les bases des templates, mais c'est également une introduction aux conteneurs, qui sont des composants fondamentaux de la programmation orientée objet et presque complètement réalisés par les conteneurs de la librairie standard du C++. Vous verrez que tout au long de ce livre on a utilisé des exemples de conteneurs – **Stash**et **Stack**–, précisément afin de vous rendre à l'aise avec les conteneurs ; dans ce chapitre le concept d' *itérateur*sera également introduit. Bien que les conteneurs soient des exemples idéaux pour l'utilisation des templates, dans le Volume 2 (qui a un chapitre avancé sur les templates) vous apprendrez qu'il y a également beaucoup d'autres manières d'utiliser les templates.

16.1 - Les conteneurs

Imaginez que vous vouliez créer une pile (stack, ndt), comme nous l'avons tout au long du livre. Cette classe stack contiendra des **int**, pour faire simple :

```
                    //: C16:IntStack.cpp
// Pile d'entiers simple
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
using namespace std;

class IntStack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
  IntStack() : top(0) {}
  void push(int i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  int pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
};

int main() {
  IntStack is;
  // Ajoute des nombres des Fibonacci, par curiosité :
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Les dépile & les affiche :
  for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;
} ///:~
```

La classs **IntStack**est un exemple trivial d'une pile refoulée. Par souci de simplicité, elle a été créée ici avec une taille constante, mais vous pouvez aussi la modifier pour s'accroître automatiquement en allouant de la mémoire sur le tas, comme dans la classe **Stack**que l'on a examinée tout au long de ce livre.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

main( )ajoute des entiers à la pile, puis les dépile. Pour rendre l'exemple plus intéressant, les entiers sont créés à l'aide de la fonction **fibonacci( )**, qui génère les traditionnels nombres de la reproduction des lapins (ndt : la suite de Fibonacci trouve son origine dans l'étude de la vitesse de reproduction des lapins dans un milieu idéal par Fibonacci (XIIIème siècle)). Voici le fichier d'en-tête qui déclare la fonction :

```
//: C16:fibonacci.h
// Générateur de nombres de Fibonacci
int fibonacci(int n); ///:~
```

Voici l'implémentation :

```
//: C16:fibonacci.cpp {O}
#include "../require.h"

int fibonacci(int n) {
  const int sz = 100;
  require(n < sz);
  static int f[sz]; // Initialisé à zéro
  f[0] = f[1] = 1;
  // recherche d'éléments non remplis du tableau :
  int i;
  for(i = 0; i < sz; i++)
    if(f[i] == 0) break;
  while(i <= n) {
    f[i] = f[i-1] + f[i-2];
    i++;
  }
  return f[n];
} ///:~
```

C'est une implémentation relativement efficace, parce qu'elle ne génère jamais les nombres plus d'une fois. Elle utilise un tableau **static**de **int**, et repose sur le fait que le compilateur initialisera un tableau de **static**à zéro. La première boucle **for**déplace l'index **i**au niveau du premier élément à zéro du tableau, puis une boucle **while**ajoute des nombres de Fibonacci au tableau jusqu'à ce que l'élément voulu soit atteint. Mais remarquez que si les nombres de Fibonacci jusqu'à l'élément **n**sont déjà initialisés, la boucle **while**est entièrement évitée.

16.1.1 - Le besoin de conteneurs

De toute évidence, une pile d'entiers n'est pas un outil de première importance. Le vrai besoin de conteneurs se fait sentir quand vous commencez à créer des objets sur le tas en utilisant **new**et que vous les détruisez avec **delete**. Dans le cas général de programmation, vous ne savez pas de combien d'objets vous allez avoir besoin lorsque vous écrivez le programme. Par exemple, dans un système de contrôle du traffic aérien vous ne voulez pas limiter le nombre d'avions que votre système peut gérer. Vous ne voulez pas que le programme plante juste parce que vous avez dépassé un certain nombre. Dans un système de conception assisté par ordinateur, vous traitez beaucoup de formes, mais seul l'utilisateur détermine (à l'exécution) de combien de formes exactement vous allez avoir besoin. Une fois que vous avez remarqué cette tendance, vous découvrirez beaucoup d'exemples de ce type dans vos propres cas de programmation.

Les programmeurs C qui s'appuient sur la mémoire virtuelle pour traiter leur “gestion mémoire” trouvent souvent perturbante l'idée de **new**, **delete**et des classes de conteneurs. Apparemment, une pratique en C est de créer un énorme tableau global, plus grand que tout ce dont pourrait avoir besoin le programme. Ceci ne nécessite peut-être pas beaucoup de réflexion (ou de connaissance de **malloc( )**et **free( )**), mais cela produit des programmes qui ne sont pas très portables et qui cachent des bugs subtils.

En outre, si vous créez un énorme tableau global d'objets en C++, le surcout de temps d'appel du constructeur et du destructeur peut ralentir les choses significativement. L'approche C++ marche beaucoup mieux : quand vous avez besoin d'un objet, créez-le avec **new**, et placez son pointeur dans un conteneur. Plus tard, repêchez-le et

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

faites quelque chose avec. Ainsi, vous ne créez que les objets dont vous avez absolument besoin. Et vous ne disposez habituellement pas de toutes les conditions d'initialisation au démarrage du programme. **new**vous permet d'attendre jusqu'à ce que quelque chose se produise dans l'environnement qui vous donne les moyens de vraiment créer l'objet.

Ainsi dans la situation la plus courante, vous créerez un conteneur qui contient des pointeurs vers les objets d'intéret. Vous créerez ces objets en utilisant **new**et placerez le pointeur résultant dans le conteneur (potentiellement en upcastant au cours de ce processus), le retirant plus tard, quand vous voudrez faire quelque chose avec l'objet. Cette technique produit les programmes les plus flexibles et les plus généraux.

16.2 - Survol des templates

Un problème apparaît, à présent. Vous disposez d'un **IntStack**, qui contient des entiers. Mais vous voulez une pile qui contiennent des formes ou des avions ou des plantes ou autre chose. Réinventer votre code source à chaque fois ne semble pas être une approche très intelligente avec un langage qui promeut la ré-utilisabilité. Il doit exister une meilleure technique.

Il y a trois techniques pour la ré-utilisation du code source dans cette situation : la manière du C, présentée ici pour comparer ; l'approche de Smalltalk, qui a significativement influencé le C++ ; et l'approche du C++ : les templates.

La solution du C. Bien sûr, vous essayez de vous affranchir de l'approche du C parce qu'elle est sale, source d'erreurs et complètement inélégante. Dans cette approche, vous copiez le code source pour un **Stack**et faites les modifictions à la main, introduisant de cette façon de nouvelles erreurs. Ce n'est certainement pas une technique très productive.

La solution Smalltalk. Le Smalltalk (et Java, qui suit son exemple) a adopté une approche simple et directe : vous voulez réutiliser le code, utilisez donc l'héritage. Pour implémenter cela, chaque classe de conteneur contient des items de la classe de base générique **Object**(similaire à l'exemple de la fin du Chapite 15). Mais puisque la bibliothèque en Smalltalk est d'une importance si fondamentale, vous ne créez jamais une classe à partir de rien. Au lieu de cela, vous devez toujours la faire hériter d'une classe existante. Vous trouvez une classe aussi proche que possible de celle que vous voulez, vous en dérivez, et faites quelques changements. De toute évidence, c'est un progrès parce que cela minimise vos efforts (et explique pourquoi vous devez passer beaucoup de temps à apprendre la biliothèque de classes avant de devenir un programmeur efficace en Smalltalk).

Mais cela signifie également que toutes les classes en Smalltalk finissent par appartenir à un arbre d'héritage unique. Vous devez hériter d'une branche de cet arbre quand vous créez une classe. L'essentiel de l'arbre est déjà là (c'est la bibliothèque de classes de Smalltalk), et à la racine de l'arbre se trouve une classe appelée **Object**–la même classe que contient chaque conteneur de Smalltalk.

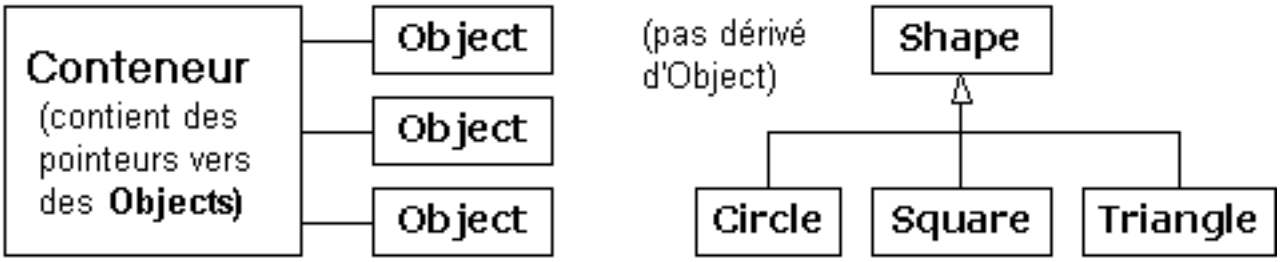
C'est une belle astuce parce que cela signifie que toute classe dans la hiérarchie de classe de Smalltalk (et de Java à l'exception, en Java, des types de données primitifs. Ceux-ci n'ont pas été rendus **Object**pour des questions d'efficacité.) est dérivée d' **Object**, si bien que toute classe peut être contenue dans chaque conteneur (y compris le conteneur lui-même). Ce type de hiérarchie à arbre unique basée sur un type générique fondamental (souvent appelé **Object**, ce qui est aussi le cas en Java) est dénommée “hiérarchie basée sur une classe object”. Vous avez peut-être entendu ce terme et imaginé que c'était un nouveau concept fondamental en POO, comme le polymorphisme. Cela fait simplement référence à une hiérarchie de classe avec **Object**(ou un nom similaire) à sa racine et des classes conteneurs qui contiennent **Object**.

Comme il y a beaucoup plus d'histoire et d'expérience derrière la bibliothèque de classes de Smalltalk que celle du C++, et puisque les compilateurs originaux du C++ n'avaient *aucune*bibliothèque de classes de conteneurs, cela semblait une bonne idée de dupliquer la bibliothèque de Smalltalk en C++. Ceci a été fait à titre expérimental avec une ancienne implémentation du C++ La librairie OOPS, de Keith Gorlen pendant qu'il était au NIH., et comme cela

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

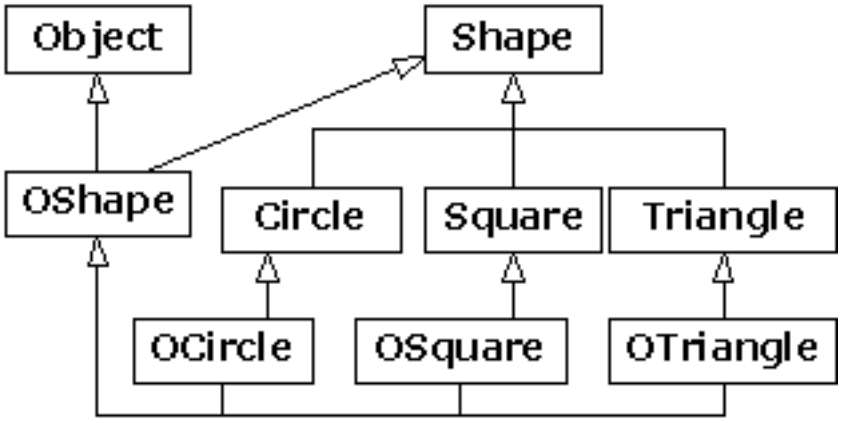
représentait une quantité de code importante, beaucoup de gens ont commencé à l'utiliser. En essayant d'utiliser les classes de conteneurs, ils ont découvert un problème.

Le problème était qu'en Smalltalk (et dans la plupart des autres langages orientés objet que je connais), toutes les classes sont automatiquement dérivées d'une hiérarchie unique, mais ce n'est pas vrai en C++. Vous pourriez avoir votre sympathique hiérarchie basée sur une classe object, avec ses classes de conteneurs, mais vous pouvez alors acheter un ensemble de classes de formes ou d'avions d'un autre vendeur qui n'utilise pas cette hiérarchie. (Pour une raison, utiliser cette hiérarchie coûte du temps sytème, ce que les programmeurs C évitent). Comment insérez-vous un arbre de classe différent dans la classe de conteneurs de votre hiérarchie basée sur une classe object ? Voici à quoi ressemble le problème :



Parce que le C++ supporte plusieurs hiérarchies indépendantes, la hiérarchie basée sur une classe object de Smalltalk ne fonctionne pas si bien.

La solution parut évidente. Si vous pouvez avoir beaucoup de hiérarchies d'héritage, vous devriez alors pouvoir hériter de plus d'une classe : l'héritage multiple résoudra le problème. Vous procédez alors comme suit (un exemple similaire a été donné à la fin du Chapitre 15) :



A présent **OShape**a les caractéristiques et le comportement de **Shape**, mais comme il dérive aussi de **Object**il peut être placé dans un **Container**. L'héritage supplémentaire dans **OCircle**, **OSquare**, etc. est nécessaire afin que ces classes puissent être transtypées en **OShape**et conserver ainsi le comportement correct. Vous vous rendez compte que les choses deviennent rapidement embrouillées.

Les vendeurs de compilateurs ont inventé et inclus leurs propres hiérarchies de classes de conteneurs basées sur une classe object, la plupart desquelles ont depuis été remplacées par des versions templates. Vous pouvez souligner que l'héritage multiple est nécessaire pour résoudre des problèmes de programmation générale, mais vous verrez dans le volume 2 de ce livre qu'il vaut mieux éviter sa complexité sauf dans certains cas particuliers.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

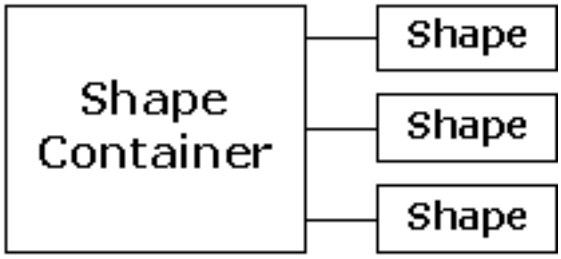
16.2.1 - La solution template

Bien qu'une hiérarchie basée sur une classe object avec héritage multiple soit conceptuellement claire, son utilisation s'avère douloureuse. Dans son livre original *The C++ Programming Language*par Bjarne Stroustrup (1ère édition, Addison-Wesley, 1986). Stroustrup a démontré ce qu'il considérait comme une alternative préférable à la hiérarchie basée sur une classe object. Les classes de conteneurs étaient créées sous forme de grandes macros du préprocesseur avec des arguments qui pouvaient être substitués par le type que vous désirez. Quand vous vouliez créer un conteneur pour stocker un type particulier, vous faisiez une paire d'appels de macros.

Malheureusement, cette approche a été embrouillée par toute la littérature et l'expérience de programmation existantes en Smalltalk, et elle était un peu difficile à manier. En gros, personne ne la comprenait.

Entre temps, Stroustrup et l'équipe C++ des Bell Labs avaient modifié leur approche macro originale, en la simplifiant et en la déplaçant du domaine du préprocesseur vers celui du compilateur. Ce nouvell outil de substitution de code est appelée un **template** Il semble que ce soient les generics en ADA qui aient inspiré les templates., et cela représente une façon complètement différente de réutiliser le code. Au lieu de réutiliser le code objet, comme avec l'héritage et la composition, un template réutilise le *code source*. Le conteneur ne stocke plus une classe générique de base appelée **Object**, mais stocke à la place un paramètre non spécifié. Quand vous utilisez un template, le paramètre est substitué *par le compilateur*, de manière très semblable à l'ancienne approche par les macros, mais de façon plus propre et plus facile à utiliser.

Maintenant, au lieu d'voir à s'inquiéter d'héritage ou de composition quand vous voulez utiliser une classe de conteneur, vous prenez la version template du conteneur et fabriquez une version spécifique pour votre problème particulier, comme ceci :



Le compilateur fait le travail pour vous, et vous vous retrouvez avec exactement le conteneur dont vous avez besoin pour faire le travail, plutôt qu'une hiérarchie d'héritage indémêlable. En C++, le template implémente le concept de *type paramétré*. Un autre bénéfice de l'approche template est que le programmeur novice qui n'est pas nécessairement familier ou à l'aise avec l'héritage peut tout de même utiliser les classes de conteneurs immédiatement (comme nous l'avons fait avec **vector**tout au long du livre).

16.3 - Syntaxe des templates

Le mot-clef **template**dit au compilateur que la définition de classe qui suit manipulera un ou plusieurs types non spécifiés. Au moment où le code réel de la classe est généré à partir du template, ces types doivent être spécifiés afin que le compilateur puisse les substituer.

Pour démontrer la syntaxe, voici un petit exemple qui produit un tableau dont les bornes sont vérifiées :

```
                    //: C16:Array.cpp
#include "../require.h"
#include <iostream>
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
using namespace std;

template<class T>
class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index < size,
      "Index out of range");
    return A[index];
  }
};

int main() {
  Array<int> ia;
  Array<float> fa;
  for(int i = 0; i < 20; i++) {
    ia[i] = i * i;
    fa[i] = float(i) * 1.414;
  }
  for(int j = 0; j < 20; j++)
    cout << j << ": " << ia[j]
         << ", " << fa[j] << endl;
} ///:~
```

Vous pouvez voir que cela ressemble à une classe normale, sauf en ce qui concerne la ligne

```
                    template<class T&gt;
```

qui dit que **T**est le paramètre de substitution, et qu'il représente un nom de type. Vous voyez également que **T**est utilisé dans la classe partout où vous verriez normalement le type spécifique stocké par le conteneur.

Dans **Array**, les éléments sont insérés *et*extraits avec la même fonction : l'opérateur surchargé **operator [ ]**. Il renvoie une référence, afin qu'il puisse être utilisé des deux côtés d'un signe égal (c'est-à-dire àla fois comme *lvalue*et comme *rvalue*). Notez que si l'index est hors limite, la fonction **require( )**est utilisée pour afficher un message. Comme **operator[]**est **inline**, vous pourriez utiliser cette approche pour garantir qu'aucune violation des limites du tableau ne se produit, puis retirer **require( )**pour le code livré.

Dans **main( )**, vous pouvez voir comme il est facile de créer des **Array**qui contiennent différents types. Quand vous dites

```
                    Array<int&gt; ia;
Array<float&gt; fa;
```

le compilateur développe le template **Array**(c'est appelé *instanciation*) deux fois, pour créer deux nouvelles *classes générées*, que vous pouvez considérer comme **Array_int**et **Array_float**. (Différents compilateurs peuvent décorer les noms de façons différentes.) Ce sont des classes exactement comme celles que vous auriez produites si vous aviez réalisé la substitution à la main, sauf que le compilateur les crée pour vous quand vous définissez les objets **ia**et **fa**. Notez également les définitions de classes dupliquées sont soit évitées par le compilateur, soit funsionnées par l'éditeur de liens.

16.3.1 - Définitions de fonctions non inline

Bien sûr, vous voudrez parfois définir des fonctions non inline. Dans ce cas, le compilateur a besoin de voir la déclaration **template**avant la définition de la fonction membre. Voici l'exempleci-dessus, modifié pour montrer la définition de fonctions non inline :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                          //: C16:Array2.cpp
// Définition de template non-inline
#include "../require.h"

template<class T>
class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index);
};

template<class T>
T& Array<T>::operator[](int index) {
  require(index >= 0 && index < size,
    "Index out of range");
  return A[index];
}

int main() {
  Array<float> fa;
  fa[0] = 1.414;
} ///:~
```

Toute référence au nom de la classe d'un template doit être accompagnée par sa liste d'arguments de template, comme dans **Array<T>::operator[]**. Vous pouvez imaginer qu'au niveau interne, le nom de classe est décoré avec les arguments de la liste des arguments du template pour produire un unique identifiant de nom de classe pour chaque instanciation du template.

Fichiers d'en-tête

Même si vous créez des définitions de fonctions non inline, vous voudrez généralement mettre toutes les déclarations *et* les définitions d'un template dans un fichier d'en-tête. Ceci paraît violer la règle normale des fichiers d'en-tête qui est “Ne mettez dedans rien qui alloue de l'espace de stockage”, (ce qui évite les erreurs de définitions multiples à l'édition de liens), mais les définitions de template sont spéciales. Tout ce qui est précédé par **template<...>**signifie que le compilateur n'allouera pas d'espace de stockage pour cela à ce moment, mais, à la place, attendra qu'il lui soit dit de le faire (par l'instanciation du template). En outre, quelque part dans le compilateur ou le linker il y a un mécanisme pour supprimer les définitions multiples d'un template identique. Donc vous placerez presque toujours la déclaration *et* la définition complètes dans le fichier d'en-tête, par commodité.

Vous pourrez avoir parfois besoin de placer la définition du template dans un fichier **cpp**séparé pour satisfaire à des besoins spéciaux (par exemple, forcer les instanciations de templates à n'exister que dans un seul fichier **dll**de Windows). La plupart des compilateurs ont des mécanismes pour permettre cela ; vous aurez besoin de chercher dans la notice propre à votre compilateur pour ce faire.

Certaines personnes pensent que placer tout le code source de votre implémentation dans un fichier d'en-tête permet à des gens de voler et modifier votre code si ils vous achètent une bibliothèque. Cela peut être un problème, mais cela dépend probablement de la façon dont vous regardez la question : achètent-ils un produit ou un service ? Si c'est un produit, alos vous devez faire tout ce que vous pouvez pour le protéger, et vous ne voulez probablement pas donner votre code source, mais seulement du code compilé. Mais beaucoup de gens considèrent les logiciels comme des services, et même plus que cela, un abonnement à un service. Le client veut votre expertise, il veut que vous continuiez à maintenir cet élément de code réutilisable afin qu'ils n'aient pas à le faire et qu'ils puissent se concentrer sur le travail qu' *ils*ont à faire. Je pense personnellement que la plupart des clients vous traiteront comme une ressource de valeur et ne voudront pas mettre leur relations avec vous en péril. Pour le peu de personnes qui veulent voler plutôt qu'acheter ou produire un travail original, ils ne peuvent probablement pas continuer avec vous de toute façon.

16.3.2 - IntStack comme template

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Voici le conteneur et l'itérateur de **IntStack.cpp**, implémentés comme une classe de conteneur générique en utilisant les templates :

```
                        //: C16:StackTemplate.h
// Template stack simple
#ifndef STACKTEMPLATE_H
#define STACKTEMPLATE_H
#include "../require.h"

template<class T>
class StackTemplate {
  enum { ssize = 100 };
  T stack[ssize];
  int top;
public:
  StackTemplate() : top(0) {}
  void push(const T& i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  T pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  int size() { return top; }
};
#endif // STACKTEMPLATE_H ///:~
```

Remarquez qu'un template fait quelques suppositions sur les objets qu'il contient. Par exemple, **StackTemplate**suppose qu'il existe une sorte d'opération d'assignation pour **T**dans la fonction **push( )**. Vous pourriez dire qu'un template “implique une interface” pour les types qu'il est capable de contenir.

Une autre façon de le dire est que les templates fournissent une sorte de mécanisme de *typage faible*en C++, qui est ordinairement un langage fortement typé. Au lieu d'insister sur le fait qu'un objet soit d'un certain type pour être acceptable, le typage faible requiert seulement que les fonctions membres qu'ils veut appeler soient *disponibles*pour un objet particulier. Ainsi, le code faiblement typé peut être appliqué à n'importe quel objet qui peut accepter ces appels de fonctions membres, et est ainsi beaucoup plus flexible Toutes les méthodes en Smalltalk et Python sont faiblement typées, et ces langages n'ont donc pas besoin de mécanismes de template. De fait, vous obtenez des templates sans templates..

Voici l'exemple révisé pour tester le template :

```
                        //: C16:StackTemplateTest.cpp
// Teste le template stack simple
//{L} fibonacci
#include "fibonacci.h"
#include "StackTemplate.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  StackTemplate<int> is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;
  ifstream in("StackTemplateTest.cpp");
  assure(in, "StackTemplateTest.cpp");
  string line;
  StackTemplate<string> strings;
  while(getline(in, line))
    strings.push(line);
  while(strings.size() > 0)
    cout << strings.pop() << endl;
} ///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

La seule différence se trouve dans la création de **is**. Dans la liste d'arguments du template vous spécifiez le type d'objet que la pile et l'itérateur devraient contenir. Pour démontrer la généricité du template, un **StackTemplate**est également créé pour contenir des **string**. On le teste en lisant des lignes depuis le fichier du code source.

16.3.3 - Constantes dans les templates

Les arguments des templates ne sont pas restreints à des types classes ; vous pouvez également utiliser des type prédéfinis. La valeur de ces arguments devient alors des constantes à la compilation pour cette instanciation particulière du template. Vous pouvez même utiliser des valeurs par défaut pour ces arguments. L'exemple suivant vous permet de fixer la taille de la classe **Array**pendant l'instanciation, mais fournit également une valeur par défaut :

```
                        //: C16:Array3.cpp
// Types prédéfinis comme arguments de template
#include "../require.h"
#include <iostream>
using namespace std;

template<class T, int size = 100>
class Array {
  T array[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index < size,
      "Index out of range");
    return array[index];
  }
  int length() const { return size; }
};

class Number {
  float f;
public:
  Number(float ff = 0.0f) : f(ff) {}
  Number& operator=(const Number& n) {
    f = n.f;
    return *this;
  }
  operator float() const { return f; }
  friend ostream&
    operator<<(ostream& os, const Number& x) {
      return os << x.f;
  }
};

template<class T, int size = 20>
class Holder {
  Array<T, size>* np;
public:
  Holder() : np(0) {}
  T& operator[](int i) {
    require(0 <= i && i < size);
    if(!np) np = new Array<T, size>;
    return np->operator[](i);
  }
  int length() const { return size; }
  ~Holder() { delete np; }
};

int main() {
  Holder<Number> h;
  for(int i = 0; i < 20; i++)
    h[i] = i;
  for(int j = 0; j < 20; j++)
    cout << h[j] << endl;
} ///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Comme précédemment, **Array**est un tableau vérifié d'objets et vous empêche d'indexer en dehors des limites. La classe **Holder**ressemble beaucoup à **Array**sauf qu'elle contient un pointeur vers un **Array**au lieu d'inclure objet de type **Array**. Ce pointeur n'est pas initialisé dans le constructeur ; l'initialisation est repoussée jusqu'au premier accès. On appelle cela l' *initialisation paresseuse*; vous pourriez utiliser une technique comme celle-ci si vous créez beaucoup d'objets, mais n'accédez pas à tous, et que vous voulez économiser de l'espace de stockage.

Vous remarquerez que la valeur **size**dans les deux templates n'est jamais stockée au sein de la classe, mais est utilisée comme si elle était une donnée membre dans les fonctions membres.

16.4 - Stack et Stash comme templates

Les problèmes récurrents de “propriété” avec les classes de conteneurs **Stash**et **Stack**qui ont été revisitées tout au long de ce livre viennent du fait que ces conteneurs n'ont pas été capables de savoir exactement quels types ils contiennent. Le plus proche qu'ils ont trouvé est le conteneur **Stack**“conteneur d' **Object**”qui a été vu à la fin du Chapitre 15 dans **OStackTest.cpp**.

Si le programmeur client ne supprime pas explicitement les pointeurs sur objets qui sont contenus dans le conteneur, alors le conteneur devrait être capable d'effacer correctement ces pointeurs. Il faut le dire, le conteneur “s'approprie” tous les objets qui n'ont pas été supprimés, et est ainsi chargé de leur effacement. Le hic est que le nettoyage nécessite de connaître le type de l'objet, et que la création d'une classe de conteneur générique nécessite de ne *pas*connaître le type de l'objet. Avec les templates, cependant, nous pouvons écrire du code qui ne connaît pas le type de l'objet, et modéliser facilement une nouvelle version du conteneur pour chaque type que nous voulons contenir. Les conteneurs modélisés individuels *connaissent*le type des objets qu'ils contiennent et peuvent ainsi appeler le destructeur correct (en supposant, dans le cas typique où le polymorphisme est mis en jeu, qu'un destructeur virtuel a été fourni).

Pour **Stack**cela se révèle être assez simple puisque toutes les fonctions membres peuvent être raisonnablement "inlined":

```
                    //: C16:TStack.h
// Stack comme template
#ifndef TSTACK_H
#define TSTACK_H

template<class T>
class Stack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt):
      data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack(){
    while(head)
      delete pop();
  }
  void push(T* dat) {
    head = new Link(dat, head);
  }
  T* peek() const {
    return head ? head->data : 0;
  }
  T* pop(){
    if(head == 0) return 0;
    T* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
#endif // TSTACK_H ///:~
```

Si vous comparez cela à l'exemple **OStack.h**à la fin du Chapitre 15, vous verrez que **Stack**est virtuellement identique, excepté le fait que **Object**a été remplacé par **T**. Le programme de test est aussi presque identique, excepté le fait que la nécessité pour le multi-héritage, de **string**et de **Object**(et même la nécessité pour **Object**lui-même) a été éliminée. Maintenant il n'y a pas de classe **MyString**pour annoncer sa destruction, ainsi une nouvelle petite classe est ajoutée pour montrer qu'un conteneur **Stack**efface ses objets:

```
                    //: C16:TStackTest.cpp
//{T} TStackTest.cpp
#include "TStack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

class X {
public:
  virtual ~X() { cout << "~X " << endl; }
};

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // File name is argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack<string> textlines;
  string line;
  // Lit le fichier et stocke les lignes dans Stack:
  while(getline(in, line))
    textlines.push(new string(line));
  // Saute quelques lignes de Stack:
  string* s;
  for(int i = 0; i < 10; i++) {
    if((s = (string*)textlines.pop())==0) break;
    cout << *s << endl;
    delete s;
  } // Le destructeur efface les autres chaînes de caractères.
  // Montre que la destruction correcte a lieu:
  Stack<X> xx;
  for(int j = 0; j < 10; j++)
    xx.push(new X);
} ///:~
```

Le destructeur pour **X**est virtuel, non pas parce que cela est nécessaire ici, mais parce que **xx**pourrait être utilisé plus tard pour contenir des objets derivés de **X**.

Notez comme il est facile de créer différentes sortes de **Stacks**pour **strings**et pour **X**. Grâce au template, vous obtenez le meilleur des deux mondes: la facilité d'utilisation de la classe **Stack**avec un nettoyage approprié.

16.4.1 - Pointeur Stash modélisé

Réorganiser le code de **PStash**dans un template n'est pas si simple parce qu'un certain nombre de fonctions membres ne devraient pas être "inlined". Néanmoins, en tant que templates ces définitions de fonctions sont toujours présentes dans le fichier d'en-tête (le compilateur et l'éditeur de liens s'occupent de tout problème de définition multiple). Le code semble assez similaire à l'ordinaire **PStash**excepté que vous noterez que la taille de l'incrémentation (utilisée par **inflate( )**) a été modélisée comme un paramètre non-classe avec une valeur par défaut, si bien que la taille d'incrémentation peut être modifiée au point de modélisation (notez que cela signifie que la taille d'incrémentation est fixée ; vous pourriez aussi objecter que la taille d'incrémentation devrait être changeable tout au long de la durée de vie de l'objet):

```
                    //: C16:TPStash.h
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
#ifndef TPSTASH_H
#define TPSTASH_H

template<class T, int incr = 10>
class PStash {
  int quantity; // Nombre d'espaces de stockage
  int next; // Prochain espace vide
  T** storage;
  void inflate(int increase = incr);
public:
  PStash() : quantity(0), next(0), storage(0) {}
  ~PStash();
  int add(T* element);
  T* operator[](int index) const; // Fetch
  // Efface la référence de ce PStash:
  T* remove(int index);
  // Nombre d'éléments dans Stash:
  int count() const { return next; }
};

template<class T, int incr>
int PStash<T, incr>::add(T* element) {
  if(next >= quantity)
    inflate(incr);
  storage[next++] = element;
  return(next - 1); // Index number
}

// Propriété des pointeurs restants:
template<class T, int incr>
PStash<T, incr>::~PStash() {
  for(int i = 0; i < next; i++) {
    delete storage[i]; // Null pointers OK
    storage[i] = 0; // Just to be safe
  }
  delete []storage;
}

template<class T, int incr>
T* PStash<T, incr>::operator[](int index) const {
  require(index >= 0,
    "PStash::operator[] index negative");
  if(index >= next)
    return 0; // To indicate the end
  require(storage[index] != 0,
    "PStash::operator[] returned null pointer");
  // Produit un pointeur pour l'élément désiré:
  return storage[index];
}

template<class T, int incr>
T* PStash<T, incr>::remove(int index) {
  // operator[] exécute des contrôles de validité:
  T* v = operator[](index);
  // "Supprime" le pointeur:
  if(v != 0) storage[index] = 0;
  return v;
}

template<class T, int incr>
void PStash<T, incr>::inflate(int increase) {
  const int psz = sizeof(T*);
  T** st = new T*[quantity + increase];
  memset(st, 0, (quantity + increase) * psz);
  memcpy(st, storage, quantity * psz);
  quantity += increase;
  delete []storage; // Ancien stockage
  storage = st; // Pointe vers la nouvelle mémoire
}
#endif // TPSTASH_H ///:~
```

La taille d'incrémentation par défaut utilisée ici est petite pour garantir que les appels à **inflate( )**ont lieu. De cette manière nous pouvons nous assurer que cela fonctionne correctement.

Pour tester le contrôle de propriété du **PStash**modélisé, la classe suivante va afficher les créations et destructions

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

d'elle-même, et ainsi garantir que tous les objets qui ont été créés ont aussi été détruits. **AutoCounter**va seulement autoriser les objets de son type à être créés sur le Stack:

```
                        //: C16:AutoCounter.h
#ifndef AUTOCOUNTER_H
#define AUTOCOUNTER_H
#include "../require.h"
#include <iostream>
#include <set> // conteneur de la Bibliothèque Standard C++
#include <string>

class AutoCounter {
  static int count;
  int id;
  class CleanupCheck {
    std::set<AutoCounter*> trace;
  public:
    void add(AutoCounter* ap) {
      trace.insert(ap);
    }
    void remove(AutoCounter* ap) {
      require(trace.erase(ap) == 1,
        "Tentative de double suppression de AutoCounter");
    }
    ~CleanupCheck() {
      std::cout << "~CleanupCheck()"<< std::endl;
      require(trace.size() == 0,
       "Tous les objets AutoCounter ne sont pas effacés");
    }
  };
  static CleanupCheck verifier;
  AutoCounter() : id(count++) {
    verifier.add(this); // Register itself
    std::cout << "created[" << id << "]"
              << std::endl;
  }
  // Empêche l'affectation et la construction de copie:
  AutoCounter(const AutoCounter&);
  void operator=(const AutoCounter&);
public:
  // Vous pouvez seulement créer des objets avec cela:
  static AutoCounter* create() {
    return new AutoCounter();
  }
  ~AutoCounter() {
    std::cout << "destroying[" << id
              << "]" << std::endl;
    verifier.remove(this);
  }
  // Affiche les objets et les pointeurs:
  friend std::ostream& operator<<(
    std::ostream& os, const AutoCounter& ac){
    return os << "AutoCounter " << ac.id;
  }
  friend std::ostream& operator<<(
    std::ostream& os, const AutoCounter* ac){
    return os << "AutoCounter " << ac->id;
  }
};
#endif // AUTOCOUNTER_H ///:~
```

La classe **AutoCounter**effectue deux choses. Premièrement, elle dénombre séquentielleent chaque instance de l' **AutoCounter**: la valeur de ce nombre est gardée dans **id**, et le nombre est généré en utilisant la donnée membre statique **count**.

Deuxièmement, plus complexe encore, une instance **static**(appelée **verifier**) de la classe imbriquée **CleanupCheck**garde la trace de tous les objets **AutoCounter**qui sont créés et détruits, et vous l'affiche en retour si vous ne les nettoyez pas tous (i.e. s'il y a une fuite de mémoire). Ce comportement est accompli en utilisant une classe **set**de la Bibliothèque Standard C++, ce qui est un merveilleux exemple montrant comment les bibliothèques de templates optimisées peuvent rendre la vie facile (vous pouvez prendre connaissance de tous les conteneurs de la Bibliothèque Standard C++ dans le Volume 2 de ce livre , disponible en ligne).

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

La classe **set**est modélisée sur le type qu'elle contient; ici, elle est représentée par une instance pour contenir les pointeurs **AutoCounter**. Un **set**permet seulement d'ajouter une instance distincte de chaque objet; dans **add( )**vous pouvez voir cela intervenir avec la fonction **set::insert( )**. **insert( )**vous informe en réalité par la valeur qu'elle retourne, si vous essayez d'ajouter quelque chose qui a déjà été ajouté; néanmoins, puisque les adresses d'objets sont ajoutées, nous pouvons compter sur la garantie du C++ que tous les objets ont des adresses uniques.

Dans **remove( )**, **set::erase( )**est utilisé pour effacer du **set**le pointeur de l' **AutoCounter**. La valeur retournée vous indique combien d'instances de l'élément ont été effacées; dans notre cas nous attendons seulement zéro ou une. Si la valeur est zéro, toutefois, cela signifie que cet objet a déjà été effacé du **set**et que vous essayez de l'effacer une seconde fois, ce qui est une erreur de programation qui va être affichée par **require( )**.

Le destructeur pour **CleanupCheck**effectue une vérification finale en s'assûrant que la taille du **set**est zéro – cela signifie que tous les objets ont été proprement effacés. Si elle n'est pas de zéro, vous avez une fuite de mémoire, ce qui est affiché par **require( )**.

Le constructeur et le destructeur pour **AutoCounter**s'inscrivent et se désinscrivent avec l'objet **verifier**. Notez que le constructeur, le constructeur de copie, et l'opérateur d'affectation sont **private**, ainsi le seul moyen pour vous de créer un objet est avec la fonction membre **static create( )**– ceci est un exemple simple de *factory*, et cela garantit que tous les objets sont créés sur le tas, ainsi le **verifier**ne va pas s'y perdre entre les affectations et les constructeurs de copies.

Puisque toutes les fonctions membres ont été "inlined", le seul intérêt du fichier d'implémentation est de contenir les définitions des données membres statiques:

```
                        //: C16:AutoCounter.cpp {O}
// Definition des classes membres statiques
#include "AutoCounter.h"
AutoCounter::CleanupCheck AutoCounter::verifier;
int AutoCounter::count = 0;
///:~
```

Avec l' **AutoCounter**en main, nous pouvons maintenant tester les possibilités du **PStash**. L'exemple suivant ne montre pas seulement que le destructeur **PStash**nettoie tous les objets qu'il possède, mais il démontre aussi comment la classe d' **AutoCounter**détecte les objets qui n'ont pas été nettoyés:

```
                        //: C16:TPStashTest.cpp
//{L} AutoCounter
#include "AutoCounter.h"
#include "TPStash.h"
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  PStash<AutoCounter> acStash;
  for(int i = 0; i < 10; i++)
    acStash.add(AutoCounter::create());
  cout << "Enlève 5 manuellement:" << endl;
  for(int j = 0; j < 5; j++)
    delete acStash.remove(j);
  cout << "En enlève deux sans les effacer:"
       << endl;
  // ... pour générer le message d'erreur de nettoyage.
  cout << acStash.remove(5) << endl;
  cout << acStash.remove(6) << endl;
  cout << "Le destructeur nettoie le reste:"
       << endl;
  // Répete le test des chapitres précédents:
  ifstream in("TPStashTest.cpp");
  assure(in, "TPStashTest.cpp");
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    PStash<string> stringStash;
    string line;
    while(getline(in, line))
      stringStash.add(new string(line));
    // Affiche les chaînes de caractères:
    for(int u = 0; stringStash[u]; u++)
      cout << "stringStash[" << u << "] = "
           << *stringStash[u] << endl;
  } ///:~
```

Quand les éléments de l' **AutoCounter**5 et 6 sont éffacés du **PStash**, ils passent sous la responsabilité de l'appelant, mais tant que l'appelant ne les nettoie pas, ils causent des fuites de mémoire, qui sont ensuites détectées par l' **AutoCounter**au moment de l'excution.

Quand vous démarrez le programme, vous voyez que le message d'erreur n'est pas aussi spécifique qu'il pourrait l'être. Si vous utilisez le schéma présenté dans l' **AutoCounter**pour découvrir les fuites de mémoire dans votre propre système vous voudrez probablement voir s'afficher des informations plus détaillées sur les objets qui n'ont pas été nettoyés. Le Volume 2 de ce livre montre des manières plus sophitiquées de le faire.

16.5 - Activer et désactiver la possession

Revenons sur le cas de la possession. Les conteneurs qui stockent les objets par valeur ne tiennent pas compte de la possession parce qu'ils possèdent clairement l'objet qu'ils contiennent. Mais si votre conteneur stocke des pointeurs (ce qui est plus courant en C++, spécialement avec le polymorphisme), il est alors très probable que ces pointeurs soient aussi utilisés autre part dans le programme, et vous ne voulez pas nécessairement supprimer l'objet parce qu'alors les autres pointeurs du programme référenceraient un objet détruit. Pour éviter ça, vous devez penser à la possession quand vous concevez et utilisez un conteneur.

Beaucoup de programmes sont plus simples que cela et ne rencontrent pas le problème de possession : un conteneur stocke des pointeurs sur des objets qui sont utilisés uniquement par ce conteneur. Dans ce cas la possession est vraiment simple : le conteneur possède ses objets.

La meilleure approche pour gérer le problème de possession est de donner un choix au programmeur client. C'est souvent accompli par un argument du constructeur qui indique par défaut la possession (cas le plus simple). En outre il peut y avoir des fonctions “get” et “set” pour lire et modifier la possession du conteneur. Si le conteneur a des fonctions pour supprimer un objet, l'état de possession affecte généralement cette suppression, et vous pouvez donc trouver des options pour contrôler la destruction dans la fonction de suppression. On peut concevoir que vous puissiez ajouter une donnée de possession pour chaque élément dans le conteneur, de telle sorte que chaque position saurait si elle a besoin d'être détruite ; c'est une variante du compteur de référence, excepté que le conteneur et non l'objet connaît le nombre de références pointant sur un objet.

```
                    //: C16:OwnerStack.h
// Stack avec contrôle de possession à l'exécution
#ifndef OWNERSTACK_H
#define OWNERSTACK_H

template<class T> class Stack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt)
      : data(dat), next(nxt) {}
  }* head;
  bool own;
public:
  Stack(bool own = true) : head(0), own(own) {}
  ~Stack();
  void push(T* dat) {
    head = new Link(dat,head);
  }
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  T* peek() const {
    return head ? head->data : 0;
  }
  T* pop();
  bool owns() const { return own; }
  void owns(bool newownership) {
    own = newownership;
  }
  // Auto-conversion de type : vrai si pas vide :
  operator bool() const { return head != 0; }
};

template<class T> T* Stack<T>::pop() {
  if(head == 0) return 0;
  T* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

template<class T> Stack<T>::~Stack() {
  if(!own) return;
  while(head)
    delete pop();
}
#endif // OWNERSTACK_H ///:~
```

Le comportement par défaut pour le conteneur est de détruire ses objets mais vos pouvez le changer soit en modifiant l'argument du constructeur soit en utilisant les fonctions membres **owns( )**de lecture/écriture.

Comme avec le plus grand nombre de templates que vous êtes susceptibles de voir, toute l'implémentation est contenue dans le fichier d'entête. Voici un petit test qui exerce les facultés de possession :

```
                    //: C16:OwnerStackTest.cpp
//{L} AutoCounter
#include "AutoCounter.h"
#include "OwnerStack.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  Stack<AutoCounter> ac; // possession activée
  Stack<AutoCounter> ac2(false); // Désactive la possession
  AutoCounter* ap;
  for(int i = 0; i < 10; i++) {
    ap = AutoCounter::create();
    ac.push(ap);
    if(i % 2 == 0)
      ac2.push(ap);
  }
  while(ac2)
    cout << ac2.pop() << endl;
  // Pas de destruction nécessaire puisque
  // ac "possède" tous les objets
} ///:~
```

L'objet **ac2**ne possède pas les objets que vous mettez dedans, ainsi **ac**est le conteneur “maître” qui prend la responsabilité de la possession. Si, à un moment quelconque de la durée de vie du conteneur, vous voulez changer le fait que le conteneur possède ses objets, vous pouvez le faire en utilisant **owns( )**.

Il serait aussi possible aussi de changer la granularité de la possession afin qu'elle soit définie objet par objet, mais cela rendra probablement la solution au problème de possession plus complexe que le problème lui-même.

16.6 - Stocker des objets par valeur

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

En fait, créer une copie des objets dans un conteneur générique est un problème complexe si vous n'avez pas les templates. Avec les templates, ces choses sont relativement simple – vous dites juste que vous stockez des objets plutôt que des pointeurs :

```
                    //: C16:ValueStack.h
// Stockage d'objets par valeur dans une pile
#ifndef VALUESTACK_H
#define VALUESTACK_H
#include "../require.h"

template<class T, int ssize = 100>
class Stack {
  // Le constructeur paar défaut effectue
  // l'initialisation de chaque élément dans le tableau :
  T stack[ssize];
  int top;
public:
  Stack() : top(0) {}
  // Le constructeur pa recopie copie les objets dans le tableau :
  void push(const T& x) {
    require(top < ssize, "Too many push()es");
    stack[top++] = x;
  }
  T peek() const { return stack[top]; }
  // L'Objet existe encore quand vous le dépilez ;
  // il est seulement plus disponible :
  T pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
};
#endif // VALUESTACK_H ///:~
```

Le constructeur par recopie pour l'objet contenu effectue la majorité du travail en passant et retournant les objets par valeur. A l'intérieur de **push( )**, le stockage de l'objet dans la table de la **Pile**est accomplie avec **T::operator=**. Pour garantir que ceci fonctionne, une classe appelé **SelfCounter**garde la trace des créations d'objets et des constructeurs par recopie:

```
                    //: C16:SelfCounter.h
#ifndef SELFCOUNTER_H
#define SELFCOUNTER_H
#include "ValueStack.h"
#include <iostream>

class SelfCounter {
  static int counter;
  int id;
public:
  SelfCounter() : id(counter++) {
    std::cout << "Created: " << id << std::endl;
  }
  SelfCounter(const SelfCounter& rv) : id(rv.id){
    std::cout << "Copied: " << id << std::endl;
  }
  SelfCounter operator=(const SelfCounter& rv) {
    std::cout << "Assigned " << rv.id << " to "
              << id << std::endl;
    return *this;
  }
  ~SelfCounter() {
    std::cout << "Destroyed: "<< id << std::endl;
  }
  friend std::ostream& operator<<(
    std::ostream& os, const SelfCounter& sc){
    return os << "SelfCounter: " << sc.id;
  }
};
#endif // SELFCOUNTER_H ///:~

//: C16:SelfCounter.cpp {O}
#include "SelfCounter.h"
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
int SelfCounter::counter = 0; ///:~

//: C16:ValueStackTest.cpp
//{L} SelfCounter
#include "ValueStack.h"
#include "SelfCounter.h"
#include <iostream>
using namespace std;

int main() {
  Stack<SelfCounter> sc;
  for(int i = 0; i < 10; i++)
    sc.push(SelfCounter());
  // peek() possible, le résultat est un temporaire :
  cout << sc.peek() << endl;
  for(int k = 0; k < 10; k++)
    cout << sc.pop() << endl;
} ///:~
```

Quand un conteneur **Stack**est créé, le constructeur par défaut de l'objet contenu est appelé pour chaque objet dans le tableau. Vous verrez initialement 100 objets **SelfCounter**créés pour raison apparente, mais c'est juste l'initialisation du tableau. Cela peut être un peu onéreux, mais il n'y a pas d'autre façon de faire dans une conception simple comme celle-ci. Une situation bien plus complexe surgit si vous rendez **Stack**plus générale en permettant à la taille d'augmenter dynamiquement, parce que dans l'implémentation montrée précédemment ceci impliquerait de créer un nouveau tableau (plus large), en copiant l'ancien tableau dans le nouveau, et en détruisant l'ancien tableau (ceci est, en fait, ce que fait la classe **vector**de la librairie standard C++).

16.7 - Présentation des itérateurs

Un *itérateur*est un objet qui parcourt un conteneur d'autres objets et en sélectionne un à la fois, sans fournir d'accès direct à l'implémentation de ce conteneur. Les itérateurs fournissent une manière standardisée d'accéder aux éléments, qu'un conteneur fournisse ou non un moyen d'accéder directement aux éléments. Vous verrez les itérateurs utilisés le plus souvent en association avec des classes conteneurs, et les itérateurs sont un concept fondamental dans la conception et l'utilisation des conteneurs standards du C++, qui sont complètement décrits dans le Volume 2 de ce livre (téléchargeable à *www.BruceEckel.com*). Un itérateur est également une sorte de *modèle de conception*, qui est le sujet d'un chapitre du Volume 2.

De bien des façons, un itérateur est un “pointeur futé”, et vous noterez en fait que les itérateurs imitent généralement la plupart des opérations des pointeurs. Contrairement à un pointeur, toutefois, l'itérateur est conçu pour être sûr, et il est donc moins probable que vous fassiez l'équivalent d'excéder les limites d'un tableau (ou si vous le faites, vous le découvrez plus facilement).

Considérez le premier exemple dans ce chapitre. Le voilà, avec un simple itérateur ajouté :

```
                    //: C16:IterIntStack.cpp
// Pile simple d'entiers avec des itérateurs
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
using namespace std;

class IntStack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
  IntStack() : top(0) {}
  void push(int i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  int pop() {
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  friend class IntStackIter;
};

// Un itérateur est comme un ponteur "futé" :
class IntStackIter {
  IntStack& s;
  int index;
public:
  IntStackIter(IntStack& is) : s(is), index(0) {}
  int operator++() { // Prefix
    require(index < s.top,
      "iterator moved out of range");
    return s.stack[++index];
  }
  int operator++(int) { // Postfix
    require(index < s.top,
      "iterator moved out of range");
    return s.stack[index++];
  }
};

int main() {
  IntStack is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Traverse avec un itérateur:
  IntStackIter it(is);
  for(int j = 0; j < 20; j++)
    cout << it++ << endl;
} ///:~
```

Le **IntStackIter**a été créé pour fonctionner seulement avec un **IntStack**. Notez que **IntStackIter**est un **friend**de **IntStack**, ce qui lui donne accès à tous les éléments **private**de **IntStack**.

Comme un pointeur, le travail de **IntStackIter**et de se déplacer dans un **IntStack**et de récupérer des valeurs. Dans cet exemple simple, le **IntStackIter**ne peut se déplacer que vers l'avant (utilisant les deux formes, pre- et postfixée, de **operator++**). Toutefois, il n'y a pas de limite à la façon dont un itérateur peut être défini, autre que celles imposées par les contraintes du conteneur avec lequel il travaille. Il est tout à fait acceptable pour un itérateur (dans les limites du conteneur sous-jacent) de se déplacer de n'importe quelle façon au sein de son conteneur associé et d'être à l'origine de modifications des valeurs contenues.

Il est habituel qu'un itérateur soit créé avec un constructeur qui l'attache à un unique conteneur d'objets, et que l'itérateur ne soit pas attaché à un autre conteneur pendant sa durée de vie. (Les itérateurs sont souvent petits et peu coûteux, et vous pouvez donc facilement en faire un autre.)

Avec l'itérateur, vous pouvez traverser les éléments de la pile sans les dépiler, exactement de la même façon qu'un pointeur qui peut se déplacer à travers les éléments d'un tableau. Toutefois, l'itérateur connaît la structure sous-jacente de la pile et comment traverser les éléments, et donc bien que vous vous déplaciez comme si vous “incrémentiez un pointeur”, ce qui se déroule en dessous est plus compliqué. C'est la clef des itérateurs : ils abstraient le processus complexe du déplacement d'un élément d'un conteneur au suivant en quelque chose qui ressemble à un pointeur. Le but est que *chaque*itérateur dans votre programme ait la même interface si bien que n'importe quel code qui utilise l'itérateur n'ait pas à se soucier de ce vers quoi il pointe – il sait juste qu'il peut repositionner tous les itérateurs de la même façon, et le conteneur vers lequel pointe l'itérateur n'a pas d'importance. De cette façon, vous pouvez écrire du code plus générique. Tous les conteneurs et algorithmes dans la bibliothèque standard du C++ sont basés sur ce principe d'itérateurs.

Pour aider à faire des choses plus génériques, il serait pratique d'être capable de dire “chaque conteneur a une classe associée appelée **iterator**, “ mais cela causera généralement des problèmes de noms. La solution est d'ajouter une classe **iterator**imbriquée à chaque conteneur (remarquez que dans ce cas, “ **iterator**” commence par une minuscule afin de se conformer au style de la bibliothèque standard du C++). Voici **IterIntStack.cpp**avec un **iterator**imbriqué :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                    //: C16:NestedIterator.cpp
// Imbriquer un itérateur dans le conteneur
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
#include <string>
using namespace std;

class IntStack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
  IntStack() : top(0) {}
  void push(int i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  int pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  class iterator;
  friend class iterator;
  class iterator {
    IntStack& s;
    int index;
  public:
    iterator(IntStack& is) : s(is), index(0) {}
    // Pour créer l'itérateur surveillant la fin :
    iterator(IntStack& is, bool)
      : s(is), index(s.top) {}
    int current() const { return s.stack[index]; }
    int operator++() { // Prefix
      require(index < s.top,
        "iterator moved out of range");
      return s.stack[++index];
    }
    int operator++(int) { // Postfix
      require(index < s.top,
        "iterator moved out of range");
      return s.stack[index++];
    }
    // Déplace un itérateur vers l'avant
    iterator& operator+=(int amount) {
      require(index + amount < s.top,
        "IntStack::iterator::operator+=() "
        "tried to move out of bounds");
      index += amount;
      return *this;
    }
    // Pour voir sivous êtes au bout :
    bool operator==(const iterator& rv) const {
      return index == rv.index;
    }
    bool operator!=(const iterator& rv) const {
      return index != rv.index;
    }
    friend ostream&
    operator<<(ostream& os, const iterator& it) {
      return os << it.current();
    }
  };
  iterator begin() { return iterator(*this); }
  // Créer la "sentinelle de fin" :
  iterator end() { return iterator(*this, true);}
};

int main() {
  IntStack is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  cout << "Traverse the whole IntStack\n";
  IntStack::iterator it = is.begin();
  while(it != is.end())
    cout << it++ << endl;
  cout << "Traverse a portion of the IntStack\n";
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    IntStack::iterator
      start = is.begin(), end = is.begin();
    start += 5, end += 15;
    cout << "start = " << start << endl;
    cout << "end = " << end << endl;
    while(start != end)
      cout << start++ << endl;
  } ///:~
```

Quand vous créez une classe **friend**imbriquée, vous devez suivre la procédure qui consiste à déclarer d'abord le nom de la classe, puis de la déclarer comme une **friend**, enfin de définir la classe. Autrement, le compilateur sera perdu.

De nouveaux trucs ont été ajoutés à l'itérateurs. La fonction membre **current( )**produit l'élément du conteneur que sélectionne actuellement l'itérateur. Vous pouvez faire “sauter” un itérateur vers l'avant d'un nombre arbitraire d'éléments en utilisant **operator+=**. Vous verrez également deux opérateurs surchargés : **==**et **!=**qui compareront un itérateur avec un autre. Ils peuvent comparer n'importe quel couple de **IntStack::iterator**, mais ils sont avant tout destinés à vérifer si un itérateur est à la fin d'une séquence, de la même façon dont les itérateurs de la “vraie” bibliothèque standard du C++ le font. L'idée est que deux itérateurs définissent un intervalle, incluant le premier élément pointé par le premier itérateur jusqu'au dernier élément pointé par le deuxième itérateur, *sans*l'inclure. Donc si vous voulez vous déplacez dans l'intervalle défini par les deux itérateurs, vous dites quelque chose comme :

```
                      while(start != end)
  cout << start++ << endl;
```

où **start**et **end**sont les deux itérateurs dans l'intervalle. Notez que l'itérateur **end**, auquel nous nous référons souvent comme à la *sentinelle de la fin*, n'est pas déréférencé et est ici pour vous dire que vous êtes à la fin de la séquence. Ainsi il représente “un pas après la fin”.

La plupart du temps, vous voudrez vous déplacer à travers toute la séquence d'un conteneur, et le conteneur a donc besoin d'un moyen de produire les itérateurs indiquant le début de la séquence et la sentinelle de la fin. Ici, comme dans la bibliothèque standard du C++, ces itérateurs sont produits par les fonctions membres du conteneur **begin( )**et **end( )**. **begin( )**utilise le premier contructeur de **iterator**qui pointe par défaut au début du conteneur (c'est le premier élément poussé sur la pile). Toutefois un second constructeur, utilisé par **end( )**, est nécessaire pour créer l' **iterator**sentinelle de la fin. Etre “à la fin” signifie pointer vers le haut de la pile, parce que **top**indique toujours le prochain espace disponible – mais non utilisé – sur la pile. Ce constructeur d' **iterator**prend un second argument du type **bool**, qui est un argument factice pour distinguer les deux constructeurs.

Les nombres Fibonacci sont utilisés à nouveau pour remplir le **IntStack**dans **main( )**, et les **iterator**sont utilisés pour se déplacer à travers tout le **IntStack**ainsi que dans un intervalle étroit de la séquence.

L'étape suivante, bien sûr, est de rendre le code général en le transformant en template sur le type qu'il contient, afin qu'au lieu d'être forcé à contenir uniquement des **int**vous puissiez stocker n'importe quel type :

```
                    //: C16:IterStackTemplate.h
// Template de pile simple avec itérateur imbriqué
#ifndef ITERSTACKTEMPLATE_H
#define ITERSTACKTEMPLATE_H
#include "../require.h"
#include <iostream>

template<class T, int ssize = 100>
class StackTemplate {
  T stack[ssize];
  int top;
public:
  StackTemplate() : top(0) {}
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
    void push(const T& i) {
      require(top < ssize, "Too many push()es");
      stack[top++] = i;
    }
    T pop() {
      require(top > 0, "Too many pop()s");
      return stack[--top];
    }
    class iterator; // Déclaration requise
    friend class iterator; // Faites-en un ami
    class iterator { // A présent définissez-le
      StackTemplate& s;
      int index;
    public:
      iterator(StackTemplate& st): s(st),index(0){}
      // pour créer l'itérateur "sentinelle de fin" :
      iterator(StackTemplate& st, bool)
        : s(st), index(s.top) {}
      T operator*() const { return s.stack[index];}
      T operator++() { // Forme préfixe
        require(index < s.top,
          "iterator moved out of range");
        return s.stack[++index];
      }
      T operator++(int) { // Forme postfix
        require(index < s.top,
          "iterator moved out of range");
        return s.stack[index++];
      }
      // Fait avancer un itérateur
      iterator& operator+=(int amount) {
        require(index + amount < s.top,
          " StackTemplate::iterator::operator+=() "
          "tried to move out of bounds");
        index += amount;
        return *this;
      }
      // Pour voir si vous êtes à la fin :
      bool operator==(const iterator& rv) const {
        return index == rv.index;
      }
      bool operator!=(const iterator& rv) const {
        return index != rv.index;
      }
      friend std::ostream& operator<<(
        std::ostream& os, const iterator& it) {
        return os << *it;
      }
    };
    iterator begin() { return iterator(*this); }
    // Crée la "sentinelle de la fin" :
    iterator end() { return iterator(*this, true);}
  };
  #endif // ITERSTACKTEMPLATE_H ///:~
```

Vous pouvez voir que la transformation d'une classe normale en un **template**est relativement claire. Cette approche qui consiste à créer et débugger d'abord une classe ordinaire, puis d'en faire un template, est généralement considérée plus simple que de créer le template à partir de rien.

Notez qu'au lieu de dire simplement :

```
                    friend iterator; // Faites en un ami
```

Ce code a :

```
                    friend class iterator; // Faites en un ami
```

C'est important parce que le nom “iterator” est déjà dans la portée, depuis un fichier inclu.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Au lieu de la fonction membre **current( )**, l' **iterator**a un **operator***pour sélectionner le membre courant, qui fait plus ressembler l'itérateur à un pointeur. C'est une pratique commune.

Voici l'exemple révisé pour tester le template :

```
                    //: C16:IterStackTemplateTest.cpp
//{L} fibonacci
#include "fibonacci.h"
#include "IterStackTemplate.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  StackTemplate<int> is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Traverse avec un iterateur :
  cout << "Traverse the whole StackTemplate\n";
  StackTemplate<int>::iterator it = is.begin();
  while(it != is.end())
    cout << it++ << endl;
  cout << "Traverse a portion\n";
  StackTemplate<int>::iterator
    start = is.begin(), end = is.begin();
  start += 5, end += 15;
  cout << "start = " << start << endl;
  cout << "end = " << end << endl;
  while(start != end)
    cout << start++ << endl;
  ifstream in("IterStackTemplateTest.cpp");
  assure(in, "IterStackTemplateTest.cpp");
  string line;
  StackTemplate<string> strings;
  while(getline(in, line))
    strings.push(line);
  StackTemplate<string>::iterator
    sb = strings.begin(), se = strings.end();
  while(sb != se)
    cout << sb++ << endl;
} ///:~
```

La première utilisation de l'itérateur se déplace simplement du début à la fin (et montre que la sentinelle de fin fonctionne correctement). Dans la deuxième utilisation, vous pouvez voir comment les itérateurs vous permettent de spécifier facilement un intervalle d'éléments (les conteneurs et les itérateurs dans la bibliothèque standard du C++ utilisent ce concept d'intervalle presque partout). Le **operator+=**surchargé déplace les itérateurs **start**et **end**vers des positions situées au milieu de l'intervalle des éléments dans **is**, et ces éléments sont affichés. Notez dans la sortie que la sentinelle n'est *pas*incluse dans l'intervalle, ainsi elle peut se trouver un cran au-delà de la fin pour vous faire savoir que vous avez passé la fin – mais vous ne déréférencez pas la sentinelle de fin, ou sinon vous puvez finir par déréférencer un pointeur nul. (J'ai mis une garde dans le **StackTemplate::iterator**, mais il n'y a pas de code de ce genre dans les conteneurs et les itérateurs de la bibliothèque standard du C++ – pour des raisons d'efficacité – et vous devez donc faire attention.)

Finalement, pour vérifier que le **StackTemplate**fonctionne avec des objets de classes, on en instancie un pour **string**et on le remplit de lignes du fichier de code source, qui sont ensuite affichées.

16.7.1 - Stack avec itérateurs

Nous pouvons répéter le procédé avec la classe **Stack**dimensionnée dynamiquement qui a été utilisée comme exemple tout au long du livre. Voici la classe **Stack**dans laquelle on a glissé un itérateur imbriqué :

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
                          //: C16:TStack2.h
// Stack rendue template avec itérateur imbriqué
#ifndef TSTACK2_H
#define TSTACK2_H

template<class T> class Stack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt)
      : data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack();
  void push(T* dat) {
    head = new Link(dat, head);
  }
  T* peek() const {
    return head ? head->data : 0;
  }
  T* pop();
  // Classe iterator imbriquée :
  class iterator; // Déclaration requise
  friend class iterator; // En fait un friend
  class iterator { // Le définit maintenant
    Stack::Link* p;
  public:
    iterator(const Stack<T>& tl) : p(tl.head) {}
    // Constructeur par recopie :
    iterator(const iterator& tl) : p(tl.p) {}
    // Itérateur sentinelle de fin :
    iterator() : p(0) {}
    // operator++ retourne un booléen indiquant la fin :
    bool operator++() {
      if(p->next)
        p = p->next;
      else p = 0; // Indique la fin de la liste
      return bool(p);
    }
    bool operator++(int) { return operator++(); }
    T* current() const {
      if(!p) return 0;
      return p->data;
    }
    // opérateur de déréférencement de pointeur :
    T* operator->() const {
      require(p != 0,
        "PStack::iterator::operator->returns 0");
      return current();
    }
    T* operator*() const { return current(); }
    // Conversion en bool pour test conditionnel :
    operator bool() const { return bool(p); }
    // Comparaison pour tester la fin :
    bool operator==(const iterator&) const {
      return p == 0;
    }
    bool operator!=(const iterator&) const {
      return p != 0;
    }
  };
  iterator begin() const {
    return iterator(*this);
  }
  iterator end() const { return iterator(); }
};

template<class T> Stack<T>::~Stack() {
  while(head)
    delete pop();
}

template<class T> T* Stack<T>::pop() {
  if(head == 0) return 0;
  T* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  return result;
}
#endif // TSTACK2_H ///:~
```

Vous remarquerez également que la classe a été modifiée pour supporter la propriété, qui maintenant fonctionne parce que la classe connaît le type exact (ou au moins le type de base, ce qui fonctionnera en supposant que les destructeurs virtuels sont utilisés). Par défaut le conteneur détruit ses objets mais vous êtes responsable de chaque pointeur que vous dépilez avec **pop( )**.

L'itérateur est simple, et physiquement très petit – de la taille d'un simple pointeur. Quand vous créez un **iterator**, il est initialisé à la tête de la liste chaînée, et vous ne pouvez l'incrémenter que vers l'avant de la liste. Si vous voulez recommencer au début, vous créez un nouvel itérateur, et si vous voulez vos rappeler d'un point dans la liste, vous créez un nouvel itérateur à partir de l'itérateur existant qui pointe vers ce point (en utilisant le constructeur par recopie de l'itérateur).

Pour appeler des fonctions pour l'objet référencé par l'itérateur, vous pouvez utiliser la fonction **current( )**, le **operator***, ou le **operator->**déréférenceur (commun dans les itérateurs). Ce dernier a une implémentation qui *paraît*identique à **current( )**parce qu'il retourne un pointeur vers l'objet courant, mais il est différent parce que l'opérateur déréfenceur réalise des niveaux suplémentaires de déréférencement (cf. Chapitre 12).

La classe **iterator**suit la forme que vous avez vue dans l'exemple précédent. La **classe iterator**est imbriquée dans la classe conteneur, il contient des constructeurs pour créer un itérateur pointant vers un élément dans le conteneur et pour créer un itérateur “sentinelle de fin”, et la classe conteneur possède les méthodes **begin( )**et **end( )**pour produire ces itérateurs. (Quand vous ne apprendrez plus sur la bibliothèque standard du C++, vous verrez que les noms **iterator**, **begin( )**, et **end( )**utilisés ici étaient clairement inspirés des classes conteneur standards. A la fin de ce chapitre, vous verrez que cela permet à ces classes de conteneur d'être utilisées comme si elles étaient des classes conteneurs de la bibliothèque standard du C++.)

L'implémentation entière est contenue dans le fichier d'en-tête, et il n'y a donc pas de fichier **cpp**distinct. Voici un petit test qui expérimente l'itérateur :

```
                        //: C16:TStack2Test.cpp
#include "TStack2.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  ifstream file("TStack2Test.cpp");
  assure(file, "TStack2Test.cpp");
  Stack<string> textlines;
  // Lit le fichier et stocke les lignes dans la Stack :
  string line;
  while(getline(file, line))
    textlines.push(new string(line));
  int i = 0;
  // Utilise l'iterateur pour afficher les lignes de la liste :
  Stack<string>::iterator it = textlines.begin();
  Stack<string>::iterator* it2 = 0;
  while(it != textlines.end()) {
    cout << it->c_str() << endl;
    it++;
    if(++i == 10) // se souvenir de la 10 ème ligne
      it2 = new Stack<string>::iterator(it);
  }
  cout << (*it2)->c_str() << endl;
  delete it2;
} ///:~
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Un **Stack**est instancié pour contenir des objets **string**et rempli par les lignes d'un fichier. Puis un itérateur est créé et utilisé pour se déplacer dans la séquence. La dixième ligne est mémorisée par copie-construction d'un deuxième itérateur à partir du premier ; plus tard, cette ligne est affichée et l'itérateur – créé dynamiquement – est détruit. Ici, la création dynamique des objets est utilisée pour contrôler la durée de vie de l'objet.

16.7.2 - PStash avec les iterateurs

Pour la plupart des classes de conteneur, il est logique d'avoir un itérateur. Voici un itérateur ajouté à la classe **PStash**:

```
                          //: C16:TPStash2.h
// PStash rendu template avec itérateur imbriqué
#ifndef TPSTASH2_H
#define TPSTASH2_H
#include "../require.h"
#include <cstdlib>

template<class T, int incr = 20>
class PStash {
  int quantity;
  int next;
  T** storage;
  void inflate(int increase = incr);
public:
  PStash() : quantity(0), storage(0), next(0) {}
  ~PStash();
  int add(T* element);
  T* operator[](int index) const;
  T* remove(int index);
  int count() const { return next; }
  // Classe itérateur imbriquée :
  class iterator; // Declaration requise
  friend class iterator; // En fait un ami
  class iterator { // A présent le définit
    PStash& ps;
    int index;
  public:
    iterator(PStash& pStash)
      : ps(pStash), index(0) {}
    // Pour créer la sentinelle de fin :
    iterator(PStash& pStash, bool)
      : ps(pStash), index(ps.next) {}
    // Constructeur par recopie :
    iterator(const iterator& rv)
      : ps(rv.ps), index(rv.index) {}
    iterator& operator=(const iterator& rv) {
      ps = rv.ps;
      index = rv.index;
      return *this;
    }
    iterator& operator++() {
      require(++index <= ps.next,
        "PStash::iterator::operator++ "
        "moves index out of bounds");
      return *this;
    }
    iterator& operator++(int) {
      return operator++();
    }
    iterator& operator--() {
      require(--index >= 0,
        "PStash::iterator::operator-- "
        "moves index out of bounds");
      return *this;
    }
    iterator& operator--(int) {
      return operator--();
    }
    // Déplace l'itérateur vers l'avant ou l'arrière :
    iterator& operator+=(int amount) {
      require(index + amount < ps.next &&
        index + amount >= 0,
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
          "PStash::iterator::operator+= "
          "attempt to index out of bounds");
        index += amount;
        return *this;
      }
      iterator& operator-=(int amount) {
        require(index - amount < ps.next &&
          index - amount >= 0,
          "PStash::iterator::operator-= "
          "attempt to index out of bounds");
        index -= amount;
        return *this;
      }
      // Crée un nouvel itérateur qui est déplacé vers l'avant
      iterator operator+(int amount) const {
        iterator ret(*this);
        ret += amount; // op+= does bounds check
        return ret;
      }
      T* current() const {
        return ps.storage[index];
      }
      T* operator*() const { return current(); }
      T* operator->() const {
        require(ps.storage[index] != 0,
          "PStash::iterator::operator->returns 0");
        return current();
      }
      // Supprime l'élément courant :
      T* remove(){
        return ps.remove(index);
      }
      // Test de comparaison pour la fin :
      bool operator==(const iterator& rv) const {
        return index == rv.index;
      }
      bool operator!=(const iterator& rv) const {
        return index != rv.index;
      }
    };
    iterator begin() { return iterator(*this); }
    iterator end() { return iterator(*this, true);}
  };

  // Destruction des objets contenus :
  template<class T, int incr>
  PStash<T, incr>::~PStash() {
    for(int i = 0; i < next; i++) {
      delete storage[i]; // Pointeurs nuls OK
      storage[i] = 0; // Juste pour être sûr
    }
    delete []storage;
  }

  template<class T, int incr>
  int PStash<T, incr>::add(T* element) {
    if(next >= quantity)
      inflate();
    storage[next++] = element;
    return(next - 1); // Numéro de l'index
  }

  template<class T, int incr> inline
  T* PStash<T, incr>::operator[](int index) const {
    require(index >= 0,
      "PStash::operator[] index negative");
    if(index >= next)
      return 0; // Pour indiquer la fin
    require(storage[index] != 0,
      "PStash::operator[] returned null pointer");
    return storage[index];
  }

  template<class T, int incr>
  T* PStash<T, incr>::remove(int index) {
    // operator[] vérifie la validité :
    T* v = operator[](index);
    // "Supprime" le pointeur:
    storage[index] = 0;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  return v;
}

template<class T, int incr>
void PStash<T, incr>::inflate(int increase) {
  const int tsz = sizeof(T*);
  T** st = new T*[quantity + increase];
  memset(st, 0, (quantity + increase) * tsz);
  memcpy(st, storage, quantity * tsz);
  quantity += increase;
  delete []storage; // Vieux stockage
  storage = st; // Pointe vers le nouvel emplacement
}
#endif // TPSTASH2_H ///:~
```

L'essentiel de ce fichier est une traduction relativement directe du précédent **PStash**et de l' **iterator**imbriqué en un template. Cette fois-ci, cependant, les opérateurs retournent des références vers l'itérateur courant, ce qui est l'approche la plus typique et la plus flexible qu'il convient d'adopter.

Le destructeur appelle **delete**pour tous les pointeurs contenus, et comme le type est capturé par le template, une destruction appropriée aura lieu. Vous devriez être conscient du fait que si le conteneur contient des pointeurs vers un type de classe de base, ce type devra avoir un destructeur virtuel pour garantir le nettoyage correct des objets dérivés dont les adresses ont été transtypée lorsqu'elles ont été placées dans le conteneur.

Le **PStash::iterator**suit le modèle d'itérateur attaché à un objet conteneur unique pour toute sa durée de vie. En outre, le constructeur par recopie vous permet de créer un nouvel itérateur pointant sur le même point que l'itérateur depuis lequel vous le créez, créant de fait un marque-page dans le conteneur. Les fonctions membres **operator+=**et **operator-=**vous permettent de déplacer un itérateur d'un certain nombre d'emplacements, tout en respectant les limites du conteneur. Les opérateurs d'incrémentation et décrémentation surchargés déplacent l'itérateur d'un emplacement. **operator+**produit un nouvel itérateur qui est déplacé vers l'avant de la quantité de l'ajout. Comme dans l'exemple précédent, les opérateurs de déréférencement de pointeurs sont utilisés pour agir sur l'élément auquel renvoie l'itérateur, et **remove( )**détruit l'objet courant en appelant le **remove( )**du conteneur.

Le même genre de code qu'avant (dans le style des *conteneurs de la bibliothèque standard du C++*) est utilisée pour créer la sentinelle de fin : un deuxième constructeur, la fonction membre **end( )**du conteneur, et **operator==**et **operator!=**pour comparaison.

L'exemple suivant crée et teste deux types différents d'objets **Stash**, un pour une nouvelle classe appelée **Int**qui annonce sa contruction et sa destruction et une qui contient des objets de la classe **string**de la bibliothèque standard.

```
                        //: C16:TPStash2Test.cpp
#include "TPStash2.h"
#include "../require.h"
#include <iostream>
#include <vector>
#include <string>
using namespace std;

class Int {
  int i;
public:
  Int(int ii = 0) : i(ii) {
    cout << ">" << i << ' ';
  }
  ~Int() { cout << "~" << i << ' '; }
  operator int() const { return i; }
  friend ostream&
    operator<<(ostream& os, const Int& x) {
      return os << "Int: " << x.i;
  }
  friend ostream&
    operator<<(ostream& os, const Int* x) {
      return os << "Int: " << x->i;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
  }
};

int main() {
  { // Pour forcer l'appel au destructeur
    PStash<Int> ints;
    for(int i = 0; i < 30; i++)
      ints.add(new Int(i));
    cout << endl;
    PStash<Int>::iterator it = ints.begin();
    it += 5;
    PStash<Int>::iterator it2 = it + 10;
    for(; it != it2; it++)
      delete it.remove(); // suppression par défaut
    cout << endl;
    for(it = ints.begin();it != ints.end();it++)
      if(*it) // Remove() provoque des "trous"
        cout << *it << endl;
  } // Destructeur de "ints" appelé ici
  cout << "\n-------------------\n";
  ifstream in("TPStash2Test.cpp");
  assure(in, "TPStash2Test.cpp");
  // Instancié pour String:
  PStash<string> strings;
  string line;
  while(getline(in, line))
    strings.add(new string(line));
  PStash<string>::iterator sit = strings.begin();
  for(; sit != strings.end(); sit++)
    cout << **sit << endl;
  sit = strings.begin();
  int n = 26;
  sit += n;
  for(; sit != strings.end(); sit++)
    cout << n++ << ": " << **sit << endl;
} ///:~
```

Par commodité, **Int**a un **ostream operator<<**associé pour un **Int&**comme pour un **Int***.

Le premier block de code dans **main( )**est entouré d'accolades pour forcer la destruction du **PStash<Int>**et ainsi le nettoyage automatique par ce destructeur. Un ensemble d'éléments est enlevé et effacé à la main pour montrer que le **PStash**nettoie le reste.

Pour les deux instances de **PStash**, un intérateur est créé et utilisé pour se déplacer dans le conteneur. Notez l'élégance qui résulte de l'utilisation de ces constructions ; vous n'êtes pas assaillis par les détails de l'implémentation de l'utilisation d'un tableau. Vous dites aux objets conteneur et itérateur *que*faire, pas comment. Ceci permet de conceptualiser, construire et modifier plus facilement la solution.

16.8 - Pourquoi les itérateurs ?

Jusqu'à maintenant vous avez vu la mécanique des itérateurs, mais comprendre pourquoi ils sont aussi importants nécessite un exemple plus complexe.

Il est courant de voir le polymorphisme, la création d'objets dynamique, et les conteneurs utilisés ensembles dans un vrai programme orienté objet. Les conteneurs et la création dynamique des objets résolvent le problème de ne pas savoir combien ou de quel type d'objets vous aurez besoin. Et si le conteneur est configuré pour contenir des pointeurs vers des objets d'une classe de base, un transtypage a lieu à chaque fois que vous mettez un pointeur vers une classe dérivée dans le conteneur (avec les avantages associés de l'organisation du code et de l'extensibilité). Comme le code final du Volume 1 de ce livre, cet exemple assemblera également les différents aspects de tout ce que vous avez appris jusqu'ici – si vous pouvez suivre cet exemple, alors vous êtes prêt pour le volume 2.

Supposez que vous créez un programme qui permette à l'utilisateur d'éditer et de produire différents types de

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

dessins. Chaque dessin est un objet qui contient une collection d'objets **Shape**:

```
//: C16:Shape.h
#ifndef SHAPE_H
#define SHAPE_H
#include <iostream>
#include <string>

class Shape {
public:
  virtual void draw() = 0;
  virtual void erase() = 0;
  virtual ~Shape() {}
};

class Circle : public Shape {
public:
  Circle() {}
  ~Circle() { std::cout << "Circle::~Circle\n"; }
  void draw() { std::cout << "Circle::draw\n";}
  void erase() { std::cout << "Circle::erase\n";}
};

class Square : public Shape {
public:
  Square() {}
  ~Square() { std::cout << "Square::~Square\n"; }
  void draw() { std::cout << "Square::draw\n";}
  void erase() { std::cout << "Square::erase\n";}
};

class Line : public Shape {
public:
  Line() {}
  ~Line() { std::cout << "Line::~Line\n"; }
  void draw() { std::cout << "Line::draw\n";}
  void erase() { std::cout << "Line::erase\n";}
};
#endif // SHAPE_H ///:~
```

Cela utilise la structure classique des fonctions virtuelles dans la classe de base, qui sont redéfinies dans la classe dérivée. Remarquez que la classe **Shape**inclut un destructeur virtuel, ce que vous devriez automatiquement ajouter à chaque classe dotée de fonctions virtuelles. Si un conteneur stocke des pointeurs ou des références vers des objets **Shape**, alors quand les destructeurs virtuels sont appelés pour ces objets tout sera proprement nettoyé.

Chaque type de dessin dans l'exemple suivant fait usage d'un type différent de classe conteneur via les templates : **PStash**et **Stack**qui ont été définies dans ce chapitre, et la classe **vector**de la bibliothèque standard du C++. L'utilisation des conteneurs est extrêmement simple, et en général l'héritage ne constitue pas la meilleure approche (la composition paraît plus logique), mais dans le cas présent l'héritage est une approche simple et n'amoindrit pas l'aspect abordé dans l'exemple.

```
//: C16:Drawing.cpp
#include <vector> // Utilise les vecteurs standard également !
#include "TPStash2.h"
#include "TStack2.h"
#include "Shape.h"
using namespace std;

// Un dessin est fondamentalement un conteneur de Shapes :
class Drawing : public PStash<Shape> {
public:
  ~Drawing() { cout << "~Drawing" << endl; }
};

// Un Plan est un autre conteneur de Shapes :
class Plan : public Stack<Shape> {
public:
  ~Plan() { cout << "~Plan" << endl; }
};
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
// Un Schematic est un autre conteneur de Shapes :
class Schematic : public vector<Shape*> {
public:
  ~Schematic() { cout << "~Schematic" << endl; }
};

// Un template de fonction :
template<class Iter>
void drawAll(Iter start, Iter end) {
  while(start != end) {
    (*start)->draw();
    start++;
  }
}

int main() {
  // Chaque type de conteneur a
  // une interface différente :
  Drawing d;
  d.add(new Circle);
  d.add(new Square);
  d.add(new Line);
  Plan p;
  p.push(new Line);
  p.push(new Square);
  p.push(new Circle);
  Schematic s;
  s.push_back(new Square);
  s.push_back(new Circle);
  s.push_back(new Line);
  Shape* sarray[] = {
    new Circle, new Square, new Line
  };
  // Les itérateurs et le template de fonction
  // leur permet d'être traité de manière générique :
  cout << "Drawing d:" << endl;
  drawAll(d.begin(), d.end());
  cout << "Plan p:" << endl;
  drawAll(p.begin(), p.end());
  cout << "Schematic s:" << endl;
  drawAll(s.begin(), s.end());
  cout << "Array sarray:" << endl;
  // Fonctionne même avec les tableaux de pointeurs :
  drawAll(sarray,
    sarray + sizeof(sarray)/sizeof(*sarray));
  cout << "End of main" << endl;
} ///:~
```

Les différents types de conteneurs contiennent tous des pointeurs vers **Shape**et des pointeurs pour transtyper des objets des classes dérivées de **Shape**. Toutefois, grâce au polymorphisme, le comportement correct se produit toujours qund les fonctions virtuelles sont appelées.

Remarquez que **sarray**, le tableau de **Shape***peut aussi être considéré comme un conteneur.

16.8.1 - Les templates de fonction

Dans **drawAll( )**vous voyez quelque chose de nouveau. Jusqu'ici, dans ce chapitre, nous n'avions utilisé que des *templates de classes*, qui instancient des nouvelles classes basées sur un ou plusieurs types paramètrés. Toutefois, vous pouvez facilement créer des *templates de fonction*, qui créent de nouvelles fonctions basées sur les types paramètrés. La raison pour laquelle vous créez un template de fonction est la même que celle qui motive les templates de classes : vous essayez de créer du code générique, et vous le faites en retardant la spécification d'un ou de plusieurs types. Vous voulez juste dire que ces types paramètrés supportent certaines opérations, mais pas quels types ils sont exactement.

Le template de fonction **drawAll( )**peut être considéré comme un *algorithme*(et c'est comme cela que sont appelés la plupart des templates de fonctions dans la bibliothèque standard du C++). Il dit juste comment faire quelque chose avec des itérateurs décrivant un ensemble d'éléments, tant que ces itérateurs peuvent être déréférencés,

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

incrémentés, et comparés. C'est exactement le type d'itérateurs que nous avons développés dans ce chapitre, et aussi – pas par coïncidence – le type d'itérateurs qui sont produits par les conteneurs dans la bibliothèque standard du C++, mise en évidence par l'utilisation de **vector**dans cet exemple.

Nous aimerions également que **drawAll( )**fût un *algorithme générique*, afin que les conteneurs puissent être de n'importe quel type et que nous n'ayons pas à écrire une nouvelle version de l'algorithme pour chaque type de conteneur différent. C'est là que les templates de fonctions sont essentiels, parce qu'ils génèrent automatiquement le code spécifique pour chaque type de conteneur. Mais sans l'indirection supplémentaire fournie par les itérateurs, cette généricité ne serait pas possible. C'est pourquoi les itérateurs sont importants ; ils vous permettent d'écrire du code à visée générale qui implique les conteneurs sans connaître la structure sous-jacente du conteneur. (Notez qu'en C++ les itérateurs et les algorithmes génériques requièrent les templates de fonction pour fonctionner.)

Vous pouvez en voir la preuve dans **main( )**, puisque **drawAll( )**fonctionne sans modification avec chaque type de conteneur différent. Et, encore plus intéressant, **drawAll( )**fonctionne également avec des pointeurs vers le début et la fin du tableau **sarray**. Cette capacité à traiter les tableaux comme des conteneurs est intégrée à la conception de la bibliothèque standard du C++, dont les algorithmes ressemblent beaucoup à **drawAll( )**.

Parce que les templates de classes de conteneurs sont rarement soumis à l'héritage et au transtypage ascendant que vous voyez avec les classes “ordinaires”, vous ne verrez presque jamais de fonctions virtuelles dans les classes conteneurs. La ré-utilisation des classes conteneurs est implémentée avec des templates, pas avec l'héritage.

16.9 - Résumé

Les classes conteneurs sont une partie essentielle de la programmation orientée objet. Ils sont une autre manière de simplifier et de dissimuler les détails d'un programme et d'en accélérer le processus de développement. En outre, ils fournissent beaucoup plus de sécurité et de flexibilité en remplaçant les tableaux primitifs et les techniques relativement grossières de structures de données trouvées en C.

Comme le programmeur client a besoin de conteneurs, il est essentiel qu'ils soient faciles à utiliser. C'est là que les **template**s interviennent. Avec les templates la syntaxe pour la ré-utilisation du code source (par opposition à la ré-utilisation du code objet fournie par l'héritage et la composition) devient suffisamment simple pour l'utilisateur novice. De fait, ré-utiliser du code avec les templates est nettement plus facile que l'héritage et la composition.

Bien que vous ayez appris comment créer des classes de conteneurs et d'itérateurs dans ce livre, il est en pratique beaucoup plus opportun d'apprendre les conteneurs et itérateurs de la bibliothèque standard du C++, puisque vous pouvez vous attendre à ce qu'ils soient disponibles sur chaque compilateur. Comme vous le verrez dans le Volume 2 de ce livre (téléchargeable depuis *www.BruceEckel.com*), les conteneurs et les algorithmes de la bibliothèque standard du C++ satisferont presque toujours vos besoins et vous n'avez donc pas besoin d'en créer vous-mêmes de nouveaux.

Certaines questions liées à la conception des classes de conteneurs ont été abordés dans ce chapitre, mais vous avez pu deviner qu'ils peuvent aller beaucoup plus loin. Une bibliothèque compliquée de classes de conteneurs peut couvrir toutes sortes d'autres aspects, en incluant le multithreading, la persistance et le ramasse-miettes (garbage collector, ndt).

16.10 - Exercices

Les solutions à certains exercices peuvent être trouvées dans le document électronique *The Thinking in C++ Annotated Solution Guide*, disponible pour une somme modique sur www.BruceEckel.com.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

1 Implémentez la hiérarchie du diagramme **OShape**de ce chapitre.
2 Modifiez le résultat de l'exercice 1 du Chapitre 15 de façon à utiliser **Stack**et **iterator**dans **TStack2.h**au lieu d'un tableau de pointeurs vers **Shape**. Ajoutez des destructeurs à la hiérarchie de classe afin que vous puissiez voir que les objets **Shape**sont détruits quand le **Stack**sort de la portée.
3 Modifiez **TPStash.h**de telle sorte que la valeur d'incrément utilisée par **inflate( )**puisse être changée durant la durée de vie d'un objet conteneur particulier.
4 Modifiez **TPStash.h**de façon à ce que la valeur d'incrément utilisée par **inflate( )**se réajuste automatiquement pour réduire le nombre de fois nécessaires à son appel. Par exemple, à chaque fois qu'elle est appelée, elle pourrait doubler la valeur de l'incrément à utiliser dans l'appel suivant. Démontrez cette fonctionnalité en en rapportant chaque fois qu' **inflate( )**est appelée, et écrivez du code test dans **main( )**.
5 Transformez en template la fonction **fibonacci( )**sur le type de valeur qu'elle produit (de façon qu'elle puisse produire des **long**, des **float**, etc. au lieu d'uniquement des **int**).
6 En utilisant la bibliothèque standard du C++ **vector**comme implémentation sous-jacente, créez une classe template **Set**qui accepte uniquement un seul de chaque type d'objet que vous placez dedans. Faites une classe **iterator**imbriquée qui supporte le concept de “sentinelle de fin” développé dans ce chapitre. Ecrivez un code de test pour votre **Set**dans **main( )**, puis substituez le template **set**de la bibliothèque standard du C++ pour vérifer que son comportement est correct.
7 Modifiez **AutoCounter.h**afin qu'il puisse être utilisé comme objet membre au sein de n'importe quelle classe dont vous voulez suivre la création et la destruction. Ajoutez un membre **string**pour retenir le nom de la classe. Testez cet outils dans une classe de votre cru.
8 Créez une version de **OwnerStack.h**qui utilise un **vector**de la bibliothèque standard du C++ comme implémentation sous-jacente. Vous pourriez avoir besoin de chercher certaines des fonctions membres de **vector**pour ce faire (ou bien regardez simplement le fichier d'en-tête **<vector>**).
9 Modifiez **ValueStack.h**de façon à ce qu'elle croisse dynamiquement lorsque vous empilez des objets avec **push( )**et qu'elle se trouve à court d'espace. Modifiez **ValueStackTest.cpp**pour tester la nouvelle fonctionnalité.
10 Répétez l'exercice 9 mais utilisez un **vector**de la bibliothèque standard du C++ comme implémentation interne de **ValueStack**. Notez comme s'est plus facile.
11 Modifiez **ValueStackTest.cpp**de façon à utiliser un **vector**decla bibliothèque standard du C++ au lieu d'un **Stack**dans **main( )**. Notez le comportement à l'exécution : est-ce que le **vector**crée automatiquement un paquet d'objets par défaut lorsqu'il est créé ?
12 Modifiez **TStack2.h**de telle sorte qu'elle utilise un **vector**de la bibliothèque standard du C++ comme implémentation sous-jacente. Assurez-vous que vous ne modifiez pas l'interface, pour que **TStack2Test.cpp**fonctionne sans modification.
13 Répétez l'exercice 12 en utilisant une **stack**de la bibiothèque standard du C++ au lieu d'un **vector**(vous pouvez avoir besoin de chercher de l'information sur **stack**ou bien d'examiner le fichier d'en-tête **<stack>**).
14 Modifiez **TPStash2.h**de telle sorte qu'elle utilise un **vector**de la bibliothèque standard du C++ comme implémentation sous-jacente. Assurez-vous que vous ne modifiez pas l'interface, pour que **TPStash2Test.cpp**fonctionne sans modification.
15 Dans **IterIntStack.cpp**, modifiez **IntStackIter**pour lui donner un constructeur “sentinelle de fin”, et ajoutez **operator==**et **operator!=**. Dans **main( )**, utilisez un itérateur pour vous déplacer à travers des éléments du conteneur jusqu'à ce que vous ayez atteint la sentinelle de fin.
16 En utilisant **TStack2.h**, **TPStash2.h**, et **Shape.h**, instanciez les conteneurs **Stack**et **PStash**pour des **Shape***, remplissez-les avec un assortiment de pointeurs **Shape**upcastés, puis utilisez les itérateurs pour vous déplacer à travers chaque conteneur et appelez **draw( )**pour chaque objet.
17 Transformez en template la classe **Int**dans **TPStash2Test.cpp**de façon à ce qu'elle puisse contenir n'importe quel type d'objet (sentez vous libre de modifier le nom de la classe en quelque chose de plus approprié).
18 Transformez en template la classe **IntArray**dans **IostreamOperatorOverloading.cpp**du chapitre 12, en paramétrant à la fois le type d'objet qu'elle contient et la taille du tableau interne.
19 Changez **ObjContainer**(dans **NestedSmartPointer.cpp**du chapitre 12) en template. Testez avec deux classes différentes.
20 Modifiez **C15:OStack.h**et **C15:OStackTest.cpp**en transformant en template la **class Stack**de telle sorte qu'elle hérite automatiquement de la classe contenue et d' **Object**(héritage multiple). La **Stack**générée devrait accepter et produire uniquement des pointeurs du type contenu.
21 Répétez l'exercice 20 en utilisant **vector**au lieu de **Stack**.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

22 Dérivez une classe **StringVector**de **vector<void*>**et redéfinissez les fonctions membres **push_back( )**et **operator[]**pour n'accepter et ne produire que des **string***(et réalisez le transtypage approprié). A présent créez un template qui créera automatiquement une classe conteneur pour faire la même chose pour des pointeurs de n'importe quel type. Cette technique est souvent utilisée pour réduire le gonflement de code résultant de nombreuses instanciations de template.
23 Dans **TPStash2.h**, ajoutez et testez un **operator-**à **PStash::iterator**, suivant la logique de **operator+**.
24 Dans **Drawing.cpp**, ajoutez et testez une fonction template pour appeler les fonctions membres **erase( )**.
25 (Avancé) Modifiez la classe **Stack**dans **TStack2.h**pour permettre une granularité complète de la propriété : ajoutez un signal à chaque lien pour indiquer si ce lien possède l'objet vers lequel il pointe, et supportez cette information dans la fonction **push( )**et dans le destructeur. Ajoutez des fonctions membres pour lire et changer la propriété de chaque lien.
26 (Avancé) Modifiez **PointerToMemberOperator.cpp**du Chapitre 12 de telle sorte que **FunctionObject**et **operator->***soient des templates qui puissent fonctionner avec n'importe quel type de retour (pour **operator->***vous aurez à utilisez des *templates membres*décrits au volume 2). Ajoutez et tester un support pour zéro, un et deux arguments dans les fonctions membres de **Dog**.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

17 - A: Le style de codage

Cet appendice ne traîte pas de l'indentation ni du placement des parenthèses ou des accolades, ce sera cependant mentionné. Il traîte des lignes de conduites générales utilisées dans l'organisation des codes de ce livre.

Bien que beaucoup de ces questions ont été introduites tout au long de ce livre, cet apendice apparaît à la fin de manière à ce que l'on puisse considérer qu'aucun sujet ne vous mettra en difficulté, et vous pouvez retourner à la section appropriée si vous ne comprenez pas quelque chose.

Toutes les décisions concernant le style de codage dans ce livre ont été considérées et prises délibérément, parfois sur une période de plusieurs années. Evidemment, chacun a ses propres raisons pour organiser le code de la manière dont il le fait, et j'essaye juste de vous expliquer comment j'en suis arrivé aux miennes ainsi que les contraintes et les facteurs environnementaux qui m'ont conduit à ces décisions.

Général

Dans le texte de ce livre, les identifiants (de fonction, de variable, et les noms de classe) sont écrits en **gras**. La plupart des mots clés sont aussi écrits en gras, sauf les mots clé tellement utilisés que leur affichage en gras deviendrait pénible, comme “class” et “virtual.”

J'utlise un style de codage particulier pour les exemples de ce livre. Il a été développé sur un certain nombre années, et est partiellement inspiré par le style de Bjarne Stroutrup dans son livre original *The C++ Programming Language*. Ibid.Le style de formatage est matière à de nombreuses heures de débat animé, je vais donc juste dire que je n'essaye pas de dicter un style correct par mes exemples; j'ai mes propres raisons pour utiliser ce style. Comme le C++ est un langage de programmation libre de forme, vous pouvez continuer à utiliser le style que vous préférez.

Ceci dit, j'insisterai sur le fait qu'il est important d'avoir un style de formatage cohérent à l'intérieur d'un projet. Si vous effectuez une recherche sur Internet, vous trouverez un certain nombre d'outils qui peuvent être utilisés pour restructurer l'ensemble du code de votre projet pour atteindre cette cohérence essentielle.

Les programmes dans ce livre sont des fichiers qui sont extraîts automatiquement du texte, ce qui permet de les tester afin de s'assurer qu'ils fonctionnent correctement. De ce fait, les fichiers de code imprimés dans ce livre vont tous fonctionner sans erreur de compilation si vous utilisez un compilateur conforme au standard du C++ (notez que tous les compilateurs ne supportent pas toutes les fonctionnalités du langage). Les erreurs qui *devraient*causer des erreurs de compilation sont commentées avec ***//!***afin qu'elles puissent être facilement trouvées et testées en utilisant des méthodes automatiques. Les erreurs trouvées et rapportées à l'auteur apparaîtront dans un premier temps dans la version électronique du livre (sur *www.BruceEckel.com*) et plus tard dans les mises à jour du livre.

L'un des standards de ce livre est que tous les programmes sont compilables et peuvent subir l'édition de liens sans erreur (même s'il vont parfois occasionner des avertissements). A cette fin, certains de ces programmes, qui démontrent seulement un exemple de code et ne représentent pas un programme complet, vont disposer de fonction **main( )**vide, comme ceci

```
int main() {}
```

Cela permet à l'éditeur de liens de travailler sans erreur.

La norme pour **main( )**est de renvoyer un **int**, mais la norme déclare que s'il n'y a pas d'instruction **return**dans

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

main( ), le compilateur doit générer automatiquement du code pour effectuer **return 0**. Cette possibilité (pas d'instruction **return**dans **main( )**est utilisée dans ce livre (certains compilateurs peuvent encore générer un avertissement pour cela, mais ils ne sont alors pas compatible avec le Standrard C++).

Les noms de fichiers

En C, la tradition est de nommer les fichiers d'en-tête (qui contiennent les déclarations) avec l'extension **.h**et les fichier d'implémentation (qui occasionnent l'allocation pour le stockage et la génération du code) avec l'extension **.c**. C++ est passé par une évolution. Il a été développé au départ sur Unix, qui est un système d'exploitation faisant une différence entre les noms de fichiers en majuscule et ceux en minuscule. Les noms de fichiers étaient à l'origine de simples versions mises en majuscule des extensions du C: **.H**et **.C**. Cela ne fonctionnait évidemment pas sur les systèmes d'exploitation qui ne font pas la distinction entre majuscules et minuscules, comme DOS. Les fournisseurs C++ pour DOS utilisaient des extensions **hxx**et **cxx**, respectivement pour les fichiers d'en-têtes et pour les fichiers d'implémentation, ou **hpp**et **cpp**. Et puis, certaines personnes ont remarqué que la seule raison d'avoir une extension différente était de permettre au compilateur de déterminer s'il devait le compiler comme un fichier C ou un fichier C++. Comme le compilateur ne compile jamais les fichiers d'en-tête directement, seule l'extension du fichier d'implémentation devait être changée. La coutume est donc, sur presque tous les sytèmes, d'utiliser **cpp**pour les fichiers d'implémentation et **h**pour les fichiers d'en-tête. Notez que lorsque vous incluez les fichiers d'en-tête du Standard C++, on utilise l'option de ne pas avoir d'extension, par exemple: **#include <iostream>**.

Les balises de commentaires de début et de fin

Un point très important de ce livre est que tous les codes que vous y trouvez doivent être vérifiés pour s'assurer qu'ils sont corrects (avec au minimum un compilateur). Ceci est accompli par extraction automatique des fichier du livre. Pour facliter les choses, tous les codes sources qui sont sensés être compilé (par opposition aux fragments de code, qui sont peu nombreux) disposent de balises commentées à leur début et à leur fin. Ces balises sont utilisées par l'outil d'extraction de code **ExtractCode.cpp**présenté dans le volume 2 du livre (que vous pouvez vous procurer sur le site *www.BruceEckel.com*) pour extraîre les codes source de la version en texte brut du livre.

La balise de fin de code indique simplement à **ExtractCode.cpp**qu'il est à la fin du code source, mais la balise de début de code est suivie par certaines informations concernant le répertoire dans lequel devra se trouver le fichier (généralement organisé par chapitre, ainsi un fichier qui se trouve dans le chapitre 8 aura la balise **C08**), suivie par deux points et le nom du fichier.

Du fait que **ExtractCode.cpp**crée aussi un **makefile**pour chaque répertoire, des informations concernant la manière dont le programme est créé et la ligne de commande utilisée pour le tester sont aussi incorporées dans le code source. Si un programme est destiné à être utilisé seul (qu'il ne doit pas être lié avec quoi que ce soit d'autre), il n'a aucune information supplémentaire. C'est aussi vrai pour les fichiers d'en-tête. Par contre, s'il ne contient pas de **main( )**et qu'il doit être lié avec quelque chose d'autre, le code source contiendra un **{O}**après le nom de fichier. Si le programme est censé être le programme principal mais qu'il doit être lié avec d'autres composants, une ligne séparée commence par **//{L}**et continue avec les fichiers qui doivent y être liés (sans extension, du fait qu'elles peuvent varier en fonction des plateformes).

Vous trouverez des exemples tout au long du livre.

Si un fichier doit être extrait mais que les balises de début et de fin ne doivent pas être incluses dans le fichier extrait (par exemple, si c'est un fichier de données de test) la balise de début est directement suivie par un ‘ **!**’.

Les parenthèses, les accolades et l'indentation

Vous pourrez noter que le style de formatage de ce livre est différent de bien des styles traditionnels en C. Bien sûr, chacun pense que son style personnel est le plus rationnel. Cependant le style utilisé ici suit une logique

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

simple, qui sera présentée couplée avec des idées sur les raisons pour lesquelles d'autres styles se sont développés.

Le style de formatage est motivé par une chose: la présentation, aussi bien à l'impression que lors de séminaires. Vous pouvez penser que vos besoins sont différents parce que vous ne faites pas beaucoup de présentations. Cependant, un code fonctionnel est plus souvent lu qu'il n'est écrit, et il devrait donc être facile à comprendre pour le lecteur. Mes deux critères les plus importants sont la “lisibilité linéaire” (la facilité avec laquelle un lecteur peut saisir le sens d'une seule ligne) et le nombre de lignes qui peuvent tenir sur une page. Ce dernier peut sembler futile, mais quand vous faites une présentation publique, cela distrait énormément l'assistance si le présentateur doit changer sans arrêt les transparents qu'il montre, et quelques lignes gâchées peuvent en être responsables.

Tout le monde semble d'accord avec le fait qu'un code entre deux accolades doit être indenté. Ce sur quoi les gens ne sont pas d'accord ’ et lepoint sur lequel il y a leplus d'incohérence entre les styles de formatages différents – est ceci : où mettre les accolades ouvrantes ? Cette seule question est, je pense, ce qui cause de telles variations dans les styles de formatage de code (pour une liste des différents styles de formatages de code, voyez *C++ Programming Guidlines*de Tom Plum et Dan Sals, Plum Hall 1991.) Je vais essayer de vous convaincre que beaucoup de styles de codages courants nous viennent d'avant les contraintes émanant du C Standard (avant les prototypes de fonctions) et sont donc inappropriés maintenant.

D'abord, ma réponse à cette question clé : les accolades ouvrante devraient toujours aller sur la même ligne que leur “précurseur” (par cela, j'entends “ tout ce dont traite le corps : une classe, une fonction, une définition d'objet, une instruction if, etc.”). C'est une règle unique, cohérente que j'applique à tous les code que j'écris, et qui rend le formattage du code beaucoup plus simple. Cela rend la “lisibilité linéaire” plus grande, par exemple, quand vous regardez cette ligne:

```
int func(int a);
```

Vous savez, par le point virgule à la fin de cette ligne, qu'il s'agit d'une déclaration, et que cela ne va pas plus loin, mais quand vous voyez cette ligne:

```
int func(int a) {
```

vous savez immédiatement que vous avez affaire à une définition parce que la ligne finit par une accolade, et non par un point virgule. En utilisant cette approche, il n'y a pas de différence entre l'endroit où vous placez une parenthèse dans une définition multi lignes:

```
int func(int a) {
  int b = a + 1;
  return b * 2;
}
```

et pour une définition sur une ligne, qui est parfois utilisée pour les fonctions *inline*:

```
int func(int a) { return (a + 1) * 2; }
```

De manière similaire, pour une classe:

```
class Thing;
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

est la déclaration d'un nom de classe, et

```
class Thing {
```

est la définition d'une classe. Vous pouvez dire, rien qu'en regardant cette seule ligne, si c'est une déclaration ou si c'est une définiton. Et, bien sûr, le fait de mettre l'accolade ouvrante sur la même ligne, au lieu de la mettre sur une ligne séparée, vous permet de faire tenir plus de lignes sur une page.

Alors, pourquoi avons nous tellement d'autres styles? En particulier, vous aurez noté que beaucoup de gens créent les classes en suivant le style ci-dessus (qui est celui que Stroutrup utilise dans toutes les éditions de son livre *The C++ Programming Language*aux éditions Addison-Wesley) mais créent les définitions de fonctions en plaçant l'ouverture des accolades sur une ligne unique (qui engendre aussi beaucoup de styles d'indentation différents). Stroutrup fait celasauf pour les courtes fonctions *inline*. Avec l'approche que je décris ici, tout est cohérent – Vous nommez ce que c'est (**class**, fonction, **enum**, etc.) et vous placez l'accolade sur la même ligne, pour indiquer que le corps de cet élément va suivre. En outre, c'est la même chose pour les fonctions *inline*et pour les définitions de fonctions ordinaires.

J'affirme que le style des définitions de fonctions utilisé par beaucoup de gens vient d'avant le prototypage de fonction du C, dans lequel on ne déclare pas les arguments à l'intérieur des parenthèses, mais bien entre la parenthèse fermante et l'ouverture de l'accolade (ainsi que le montrent les racines du C en assembleur):

```
void bar()
 int x;
 float y;
{
 /* corps ici */
}
```

Ici, ce serait assez maladroit de placer l'accolade sur la même ligne, et personne ne le fait. Cependant, cela a occasionné des décisions variées sur la manière dont les accolades devraient être indentées par rapport au corps du code ou à quel niveau elles devraient se trouver par rapport au “précurseur”. Ainsi, nous avons beaucoup de styles de formatage différents.

Il y a d'autres arguments en faveur du fait de placer l'accolade ouvrante sur la ligne qui suit celle de la déclaration (d'une classe, d'une structure, d'une fonction, etc.). Ce qui suit vient d'un lecteur et vous est présenté afin que vous sachiez quels sont les problèmes :

Les utilisateurs expérimentés de ‘vi’ (vim) savent qu'enfoncer deux fois la touche ‘]’ va emmener l'utilisateur à l'occurence suivante du ‘{‘ (or ^L) à la colonne 0. Cette fonctionnalité est extrêmement utile lors de la navigation dans le code (saut à la définition de fonction ou de classe suivante). [Mon commentaire: quand j'ai travaillé au départ sous Unix, GNU Emacs venait juste d'apparaître et je m'y suis habitué. De ce fait ‘vi’ n'a jamais eu de sens pour moi, et ne ne pense donc pas en terme de “position colonne 0.” Cependant il existe un certain nombre d'utilisateurs de ‘vi’, et ils sont donc affectés par cette question.]

Placer le ‘{‘ sur la ligne suivante élimine la confusion dans le code de conditions complexes, ce qui aide à la lisibilité. Par exemple:

```
if(cond1
   && cond2
   && cond3) {
   statement;
}
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Ce qui précède [affirme le lecteur] n'est pas très lisible. Cependant,

```
                if (cond1
&& cond2
&& cond3)
{
statement;
}
```

sépare l'instruction ‘if’ du corps, ce qui résulte en une meilleure lisibilité. [Votre avis sur la véracité va varier en fonction de ce à quoi vous êtes habitués.]

Finalement, il est bien plus facile d'aligner visuellement les accolades quand elles sont alignées sur la même colonne. Elles "collent" visuellement beaucoup mieux. [Fin du commentaire du lecteur]

La question de savoir où placer l'accolade ouvrante est probablement la question la plus discordante. J'ai appris à scanner les deux formes, et en fin de compte on en vient à la méthode avec laquelle vous êtes à l'aise. Cependant, je note que le standard officiel Java (trouvé sur le site web de Sun) est, de fait, le même que celui que je vous présente ici – puisque de plus en plus de gens commencent à programmer dans les deux langages, la cohérence entre les modèles de langages peut être utile.

L'approche que j'utilise supprime toutes les exceptions et les cas spéciaux, et produit logiquement un style unique d'indentation. Même à l'intérieur d'un corps de fonction, la cohérence persiste, comme dans:

```
                for(int i = 0; i < 100; i++) {
  cout << i << endl;
  cout << x * i << endl;
}
```

Le style est facile à apprendre et à retenir– vous employez une règle simple et cohérente pour tout votre formattage, pas une pour les classes, deux pour les fonctions (monoligne *inline*scontre multiligne), et probablement d'autres pour pour les boucles **for**, instructions **if**, etc. La cohérence seule la rend, je pense, digne de considération. Par dessus tout, C++ est un langage plus récent que le C, et bien que nous devions faire beaucoup de concessions au C, nous ne devrions pas y importer des causes de problèmes supplémentaires. De petits problèmes multipliés par de nombreuses lignes de code deviennent de gros problèmes. Pour un examen complet du sujet, quoi que traité pour le C, voyez *C Style: Standards and Guidelines*, de David Straker (Prentice-Hall 1992).

L'autre contrainte sur laquelle je dois travailler est la largeur, puisque le livre a une limitation à 50 caractères. Que se produit-il quand quelque chose est trop long pour tenir sur une ligne? Eh bien, j'essaye encore d'avoir une approche cohérente sur la manière dont les lignes sont brisées, de manière à ce qu'elle puissent être facilement visualisées. Aussi longtemps que quelque chose fait partie d'une définition unique, liste d'arguments, etc. les lignes de continuation devraient être indentées d'un niveau par rapport au début de la définition ou de la liste d'arguments.

Les noms identifiants

Ceux qui sont familiers avec Java auront noté que j'ai utilisé le standard Java pour tous les identifiants de noms. Cependant, je ne peux pas être parfaîtement cohérent là dessus parce que les identifiants provenant des bibliothèques Standards C et C++ ne suivent pas ce style.

Le style est assez simple. La première lettre d'un identifiant est en majuscule uniquement s'il s'agit d'une classe. Si c'est une fonction ou une variable, la première lettre est en minuscule. Le reste de l'identifiant consiste en un ou plusieurs mots, l'un suivant l'autre, mais distingués en mettant la première lettre du mot en majuscule. Ainsi, une

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

classe ressemble à ceci:

```
class FrenchVanilla : public IceCream {
```

L'identifiant d'un objet ressemblant à:

```
FrenchVanilla myIceCreamCone(3);
```

et une fonction ressemble à:

```
void eatIceCreamCone();
```

(qu'il s'agisse d'une fonction membre ou d'une fonction régulière).

La seule exception concerne les constantes de compilation (**const**ou **#define**), pour lesquelles toutes les lettres de l'identifiant sont en majuscules.

La valeur de ce style est que la mise en majuscule a du sens – vous êtes en mesure de voir dès la première lettre si vous parlez d'une classe ou d'un objet/méthode. C'est particulièrement utile quand on accède à des membres de classes **static**.

Ordre d'inclusion des fichers d'en-tête

Les en-têtes sont inclus dans l'ordre “du plus spécifique vers le plus général.” Cela signifie que les fichiers d'en-tête du dossier local sont inclus en premier, puis ceux de mes propres en-tête qui me servent d'"outils", tels que **require.h**, puis les en-têtes des bibliothèques tierces, puis celles des bibliothèques du Standard C++, et finalement les en-têtes des bibliothèques C.

La justification de cet ordre vient de John Lakos dans *Large-Scale C++ Software Design*(Addison-Wesley, 1996):

Des erreurs latentes d'utilisation peuvent être évitées en s'assurant que le fichier .h d'un composant est analysé par lui-même – sans déclarations ou définitions extérieures... L'inclusion dufichier .h à la première ligne du fichier .c vous assure qu'aucun information critique intrinsèque à l'interface physique du composant ne manque dans le fichier .h (ou, s'il y en a, que vous le remarquerez dès que vous essayerez de compiler le fichier .c).

Si l'ordre d'inclusion des fichiers d'en-tête va “du plus spécifique au plus général”, alors il est plus probable que si votre fichier d'en-tête ne s'analyse pas de lui-même, vous découvrirez plus tôt la raison et cela vous évitera des ennuis en cours de route.

Ajouter des gardes dans les fichier d'en-tête

*Les gardes d'inclusion*sont toujours utilisés dans un fichier d'en-tête durant la compilation d'un seul fichier **.cpp**pour éviter l'inclusion multiple d'un fichier d'en-tête. Les gardes d'inclusion sont implémentés en utilisant la directive préprocesseur **#define**et en vérifiant que le nom n'a pas encore été défini. Le nom utilisé comme garde est basé sur le nom du fichier d'en-tête, avec toutes les lettres du nom de fichier en majuscule et en remplaçant le ‘ **.**’ par un underscore (ND: _). Par exemple:

```
// IncludeGuard.h
#ifndef INCLUDEGUARD_H
```


Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

```
#define INCLUDEGUARD_H
// Corps du fichier d'en-tête ici...
#endif // INCLUDEGUARD_H
```

L'identifiant sur la dernière ligne est ajouté par soucis de clarté. Cependant, certains préprocesseurs ignorent tout caractère suivant un **#endif**, ce n'est pas un comportement standard et l'identifiant est donc commenté.

Utilisation des espaces de nommages

Dans les fichier d'en-têtes, toute “pollution” de l'espace de nommage dans lequel le fichier d'en-tête est inclus doit être scrupuleusement évitée. La raison en est que si vous changez l'espace de nommage en dehors d'une foncton ou d'une classe, vous provoquerez le changement dans tout fichier qui inclus votre fichier d'en-tête, ce qui peut se traduire par toutes sortes de problèmes. Aucune déclaration **using**d'aucun type n'est autorisée en dehors des déclarations de fonctions, et aucune directive globale **using**n'est autorisée dans les fichiers d'en-tête.

Dans les fichiers **cpp**, toute directive globale **using**n'affectera que ce fichier, et elles sont donc généralement utilisées dans ce livre pour produire un code plus facilement lisible, spécialement dans les petits programmes.

Utilisation de require( ) et de assure( )

Les fonctions **require( )**et **assure( )**définies dans **require.h**sont utilisées de manière cohérente tout au long de ce livre, de manière à ce qu'elles puissent reporter correctement les problèmes. Si vous êtes familiers avec les conceptes de *préconditions*et *postconditions*(introduites par Bertrand Meyer) vous reconnaîtrez que l'utilisation de **require( )**et de **assure( )**fournissent plus ou moins la précondition (c'est le cas en général) et la postcondition (occasionnellement). Ainsi, au début d'une fonction, avant que tout élément du “coeur” de la fonction ne soit effectué, les préconditions sont testées pour être sûr que tout est en ordre et que toutes les conditions nécessaires sont remplies. Après que le “coeur” de la fonction ait été exécuté, certaines postconditions sont testées pour s'assurer que le nouvel état des données se situe dans un périmètre prédéfinis. Vous noterez que les tests de postconditions sont rares dans ce livre, et que **assure( );**est principalement utilisé pour s'assurer que les fichiers ont été ouverts avec succès.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

18 - B: Directives de programmation

Cet appendice est une collection de suggestions pour la programmation en C++. Elles ont été rassemblé au cours de mon enseignement, de mon expérience et sont

également le fruit de la perspicacité d'amis y compris Dan Saks (co-auteur avec Tom Plum de *C++ Programming Guidelines*, Plum Hall, 1991), Scott Meyers (auteur de *Effective C++*, 2nd edition, Addison-Wesley, 1998), et Rob Murray (auteur de *C++ Strategies & Tactics*, Addison-Wesley, 1993). Enfin, beaucoups de conseils sont des résumés des pages de *Thinking in C++*.

1 Faites le d'abord fonctionner, puis rendez le rapide. C'est vrai même si vous êtes certain qu'un morceau de code est réellement important et que ce sera un goulot d'étranglement majeur de votre système. Ne le faites pas. Faites un système qui fonctionne avec un design aussi simple que possible. Ensuite, si cela ne va pas assez vite, profilez le. Vous découvrirez presque toujours que votre goulot d'étranglement n'est pas le problème. Economisez votre temps pour les choses réellement importantes.

2 L'élégance paye toujours. Ce n'est pas une recherche frivolle. Non seulement cela vous donnera un programme plus facile à construire et à débugger, mais il sera aussi plus facile à comprendre et à maintenir, et c'est là que se trouve la valeur financière. Ce point peut nécessiter un peu d'expérience pour être cru, car il peut sembler que quand vous écrivez un morceau de code élégant, vous n'êtes pas productif. La productivité vient quand le code s'intègre sans heurts dans votre système, et plus encore, quand le code ou le système sont modifiés.

3 Souvenez vous du principe “diviser pour mieux régner”. Si le problème que vous essayez de résoudre est trop confus, essayez d'imaginer l'opération de base du programme, en donnant naissance à un “morceau magique ” qui va gérer les parties difficiles. Ce “morceau” est un objet – écrivez le code qui utilise cet objet, puis regardez l'objet et encapsulez *ses*parties difficiles dans d'autres objets, etc.

4 Ne réécrivez pas automatiquement tout le code C existant en C++ à moins que vous n'ayez besoin de changer significativement ses fonctionnalités (autrement dit: ne réparez pas ce qui n'est pas cassé). *Recompiler*du C dans du C++ est une activité utile car elle permet de révéler des bugs cachés. Cependant, prendre du code C qui fonctionne bien et le réécrire en C++ peut ne pas être le meilleur usage de votre temps, à moins que la version C++ ne fournisse beaucoup d'opportunités de réemplois, comme une classe.

5 Si vous avez affaire à une grande portion de code C qui doit être changée, isolez d'abord les parties de code qui ne doivent pas être modifiées, placez les dans une “classe API” en tant que fonctions membres statiques. Ensuite, concentrez vous sur le code qui doit être changé, en le refactorisant dans des classes pour facilier les processus de modifications et de maintenance.

6 Séparez le créateur de la classe de l'utilisateur de la classe (*programmeur client*). L'utlisateur de la classe est le “client” et n'a aucun besoin ou ne veut pas savoir ce qui se passe derrière la scène de la classe. Le créateur de la classe doit etre l'expert en design de classe et doit écrire la classe de manière à ce qu'elle puisse être utilisée par un programmeur le plus novice possible, et fonctionner de manière robuste dans l'application. L'usage d'une bibliothèque ne sera facile que si elle est transparente.

7 Quand vous créez une classe, rendez les noms aussi clairs que possible. Votre but est de rendre l'interface du client programmeur conceptuellement simple. Essayez de rendre vos noms suffisemment clairs pour que les commentaires deviennent inutiles. A cette fin, utilisez la surcharge de fonction et les arguments par défaut pour créer une interface intuitive et facile d'usage.

8 Le contrôle d'accès vous permet (en tant que créateur de la classe) de faire autant de changements que possible dans le futur sans causer de dommages au code du client dans lequel la classe est utilisée. De ce point de vue, gardez tout ce que vous pouvez aussi **private**que possible, et rendez seulement l'interface **public**, utilisez toujours des fonctions plutôt que des données. Ne rendez des données **public**que quand vous y êtes forcés. Si l'utilisateur de la classe n'a pas besoin d'avoir accès à une fonction, rendez la **private**Si une partie de votre classe doit être exposée à des héritiers en tant que **protected**, fournissez une fonction d'interface plutôt que d'exposer la donnée elle-même. De cette manière les changements d'implémentation auront un impact minime sur les classes dérivées.

9 Ne sombrez pas dans la paralysie de l'analyse. Il y a plusieurs choses qui ne s'apprennent qu'en commençant à coder et en faisant fonctionner un type ou l'autre de système. Le C++ a ses propres pares-feux ; laissez les travailler pour vous. Vos erreurs dans une classe ou dans un jeu de classes ne

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

détruiront pas l'intégrité de tout le système.

10 Votre analyse et votre conception doivent produire, au minimum, les classes de votre système, leurs interfaces publiques, et leurs relations aux autres classes. Spécialement les classes de base. Si votre méthodologie de design produit plus que cela, demandez vous si toutes les pièces produites par cette méthodologie ont de la valeur sur la durée de vie de votre programme. Si elles n'en ont pas, les maintenir aura un prix. Les membres des équipes de développement ne tendent pas à maintenir autre chose que ce qui contribue à leur productivité ; c'est un fait de la vie que beaucoup de méthodes de conception ne prennent pas en compte.

11 Ecrivez le test du code en premier (avant d'écrire la classe), et gardez-le avec la classe. Automatisez l'exécution de vos tests au travers d'un makefile ou d'un outil similaire. De cette manière, chaque changement peut être automatiquement vérifié en exécutant le code de test, et vous découvrirez immédiatement les erreurs. Comme vous savez que vous diposez du filet de sécurité de votre cadre de tests, vous serez plus enclins à effectuer des changements quand vous en découvrirez le besoin. Retenez que les plus grandes améliorations dans le langage viennent de la vérification intégrée, que procurent la vérification de type, la gestion des exceptions,... mais cela ne fera pas tout. Vous devez parcourir le reste du chemin en créant un sytème robuste par des tests qui vérifieront les aspects spécifiques à votre classe ou à votre programme.

12 Ecrivez le code de test en premier (avant d'écrire la classe) de manière à vérifier que le design de votre classe est complet. Si vous n'êtes pas en mesure d'écrire le code de test, vous ne savez pas de quoi votre classe a l'air. De plus, le fait d'écrire du code de test permettra parfois de faire ressortir des dispositions supplémentaires ou des contraintes dont vous avez besoin dans la classe – ces contraintes ou dispositions n'apparaissent pas forcément durant la phase d'analyse et de conception.

13 Retenez une règle fondamentale du génie logiciel Qui me fut expliquée par Andrew Koenig.: *Tout problème de conception de logiciel peut être simplifié en introduisant un niveau supplémentaire d'indirection conceptuelle.*Cette idée est la base de l'abstraction, le premier dispositif de la programmation orientée-objet.

14 Rendez les classes aussi atomiques que possible ; autrement dit, donnez à chaque classe un but simple et clair. Si vos classes ou votre système deviennent trop compliqués, séparez les classes les plus complexes en d'autres classes plus simples. L'indicateur le plus évident est sa petite taille : si une classe est grosse, il y a des chances pour qu'elle en fasse trop et qu'elle doivent être éclatée.

15 Cherchez les longues définitions de fonctions membres. Une fonction qui est longue et compliquée est difficile et honnéreuse à maintenir, et essaye peut-être d'en faire trop par elle-même. Si vous voyez une telle fonction, elle indique, au minimum, qu'elle devrait être divisée en plusieurs fonctions. Elle peut aussi suggérer la création d'une nouvelle classe.

16 Cherchez les longues listes d'arguments. Les appels de fonctions deviennent alors difficiles à écrire, à lire et à maintenir. A l'opposé, essayez de déplacer les fonctions membres vers une classe où ce serait plus approprié, et/ou passez des objets comme arguments.

17 Ne vous répétez pas. Si une partie de code est récurrente dans de nombreuses fonctions dans des classes dérivées, placez en le code dans une fonction simple dans la classe de base et appelez-la depuis les classes dérivées. Non seulement, vous épargnerez de l'espace de code, mais vous facilitez aussi la propagation des changements. Vous pouvez utiliser une fonction *inline*par souci d'efficacité. Parfois, la découverte de ce code commun permettra d'ajouter une fonctionnalité utile à votre interface.

18 Cherchez les instructions **switch**ou les suites de clauses **if-else**. C'est typiquement l'indicateur du *codage de vérification de type*, qui signifie que vous choisissez quel code exécuter sur un type d'information (le type exact peut ne pas être évident au début). Vous pouvez habituellement remplacer ce type de code par l'héritage et le polymorphisme ; l'appel d'une fonction polymorphique va effectuer le test du type pour vous, et facilite l'extension du code.

19 Du point de vue de la conception, cherchez et séparez les éléments qui changent de ceux qui restent identiques. C'est-à-dire, cherchez les éléments dans le système que vous pourriez vouloir modifier sans imposer une reconception, puis encapsulez ces éléments dans des classes. Vous trouverez beaucoup plus d'information sur ce concept dans le chapitre traîtant des *design patterns*dans le deuxième volume de ce livre, disponible sur le site *www.BruceEckel.com*.

20 Cherchez la *variance*. Deux objets sémantiquement différents peuvent avoir des actions, ou des responsabilités identiques, et il y a une tentation naturelle de créer une sous classe de l'autre juste pour bénéficier de l'héritage. Cela s'appelle la variance, mais il n'y a pas de réelle justification à forcer une relation de superclasse à sous classe là où elle n'existe pas. Une meilleure solution est de créer une classe de base générale qui produit une interface pour toutes les dérivées ” cela requiert un peu plus d'espace, mais vous

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

bénéficiez toujours de l'héritage et vous ferez peut-être une découverte importante au niveau du design.

21 Recherchez les *limtations*durant l'héritage. Les designs les plus clairs ajoutent de nouvelles capacités à celles héritées. Un design suspect retire les anciennes capacités durant l'héritage sans en ajouter de nouvelles. Mais les règles sont faites pour être contournées, et si vous travaillez sur une ancienne bibliothèque de classes, il peut être plus efficace de restreindre une classe existante dans son héritière que de restructurer la hiérarchie de manière à ce que votre nouvelle classe puisse trouver sa place au-dessus de l'ancienne.

22 N'étendez pas une fonctionnalité fondamentale dans une classe héritée. Si un élément d'interface est essentiel à une classe, il doit être dans la classe de base, et non ajouté en cours de dérivation. Si vous ajoutez des fonctions membres par héritage, vous devriez peut être repenser le design.

23 Le moins est plus. Commencez avec une interface de la classe aussi petite et aussi simple que ce qu'il vous faut pour résoudre le problème en cours, mais n'essayez pas d'anticiper toutes les façons dont votre classe *peut*être utilisée. Quand la classe sera utilisée, vous découvrirez les façons dont vous devrez étendre l'interface. Cependant, une fois qu'une classe est en usage, vous ne pouvez pas diminuer l'interface sans déranger le code client. Si vous devez ajouter plus de fonctions, c'est bon ; cela ne va pas déranger le code, autrement qu'en forçant une recompilation. Mais même si de nouvelles fonctions membres remplacent la fonctionnalié d'anciennes, laissez l'interface existante tranquille (vous pouvez combiner les fonctionnalités dans l'implémentation sous-jacente si vous le voulez). Si vous devez étendre l'interface d'une fonciton existante en ajoutant des arguments, laissez les arguments existants dans l'ordre actuelle, et fournissez des valeurs par défaut pour tous les nouveaux arguments ; de cette manière, vous ne dérangerez pas les appels existants à cette fonction.

24 Lisez vos classe à voix haute pour vous assurez qu'elles sont logiques, en vous référant à une relation d'héritage entre une classe de base et une classe dérivée comme “est un(e)” et aux objets membres comme “a un(e)”.

25 Quand vous décidez entre l'héritage et la composition, demandez vous si vous devez transtyper vers le type de base. Sinon, préférez la composition (objets membres) à l'héritage. Ceci peut éliminer la nécessitée ressentie de recours à l'héritage multiple. Si vous faites hériter une classe, les utilisateurs vont penser qu'il sont supposés transtyper.

26 Parfois, vous devez faire hériter une classe afin de permettre l'accès aux membres **protected**de la classe de base. Cela peut mener à l'impression d'avoir besoin de l'héritage multiple. Si vous n'avez pas besoin de transtyper l'objet, commencez par dériver une nouvelle classe pour effectuer l'accès **protected**. Puis faites en un objet membre à l'intérieur de toute classe qui en a besoin, plutôt que de la faire hériter.

27 Typiquement, une classe de base sera utilisée pour en premier lieu pour créer une interface vers les classes qui en dérivent. Ainsi, quand vous créez une classe de base, créez les fonctions membres en tant que virtuelles pures par défaut. Le destructeur peut aussi être virtuel pur (pour forcer les classes héritières à le surcharger explicitement), mais rappelez-vous de donner un corps de fonction au destructeur, car tous les destructeurs de la hiérarchie de classe sont toujours appelés.

28 Quand vous décidez de créer une fonction **virtual**dans une classe, rendez toutes les fonctions dans cette classe **virtual**, et rendez le destructeur **virtual**. Cette approche évite les surprises dans le comportement de l'interface. Ne retirez le mot clé **virtual**que lorsque vous en êtes à l'optimisation et que votre *profiler*vous a dirigé par là.

29 Utilisez des données membres pour les variations de valeurs et des fonctions **virtual**pour des variation de comportement. C'est-à-dire que si vous trouvez une classe qui utilise des variables d'état tout au long de ses fonctions et qui modifie son comportement en fonction de ces variables, vous devriez probablement refaire le design pour exprimer ces différences de comportement à l'intérieur de classes héritées et de fonctions virtuelles surchargées.

30 Si vous devez faire quelque chose de non portable, faites une abstraction de ce service et localisez le à l'intérieur d'une classe. Ce niveau supplémentaire d'indirection évite que la non portabilité ne se répande au travers de tout votre porgramme.

31 Evitez les héritages multiples. Cela vous épargnera de vous trouver en mauvaises situations, principalement le fait de devoir réparer une interface dont vous ne contrôlez pas la classe endommagée (voir volume 2). Vous devriez être un programmeur expérimenté avant de vouloir implémenter le héritage multiple dans votre système.

32 N'utilisez pas l'héritage **private**. Bien que ce soit admis par le langage et que cela semble avoir certaines fonctionnalités, cela introduit des ambiguités importantes lorsque c'est combiné avec l'identification de type à l'exécution. Créez un objet membre privé plutôt que d'utiliser l'héritage privé.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

33 Si deux classes sont fonctionnellement associées (comme les conteneurs et les itérateurs), essayez de rendre l'une des classes **friend**imbriqué **public**de l'autre, à la manière de ce que fait la bibliothèques standard du C++ avec les itérateurs dans les conteneurs (des exemples en sont montrés dans la dernière partie du chapitre 16). Cela permet non seulement l'association entre les classes, mais cela permet aussi de réutiliser le nom en le nichant au sein une autre classe. La bibliothèque standard du C++ le fait en définissant un **iterator**niché au sein de chaque classe conteneur, fournissant ainsi une interface commune aux conteneurs. L'autre raison pour laquelle vous voudriez nicher une classe au sein d'une autre est en tant que partie de l'implémentation **private**. Ici, ce sera bénéfique pour cacher l'implémentation au lieu d'utiliser une association de classe et pour éviter la pollution des espaces de nommages dont il a été question plus haut.

34 La surcharge d'opérateur n'est qu'une “douceur syntaxique“ : une manière différente d'effectuer un appel à une fonciton. Si la surcharge d'un opérateur ne rend pas l'interface plus claire et plus facile à utiliser, ne le surchargez pas. Ne créez qu'un seul opérateur de conversion de type automatique pour une classe. En général, suivez les règles et formats donnés dans le chapitre 12 lorsque vous surchargez des opérateurs.

35 Ne vous laissez pas tenter par l'optimisation précoce. Ce chemin est une folie. En particulier, ne vous souciez pas d'écrire (ou d'éviter) des fonctions **inline**, de rendre certaines fonctions non **virtual**, ou de tordre le système pour être efficace quand vous construisez le système. Votre but premier est de prouver le design, à moins que le design ne requiert une certaine efficacité.

36 Normalement, ne laissez pas le compilateur créer les constructeurs et les destructeurs, ou l' **operator=**pour vous. Les concepteurs de classes devraient toujours dire ce que la classe devrait faire et garder la classe entièrement sous contrôle. Si vous ne voulez pas d'un constructeur de copie ou d'un **operator=**, déclarez le **private**. Rappelez-vous que si vous créez un constructeur, il empêche la création automatique du constructeur par déffaut.

37 Si votre classe contient des pointeurs, vous devez créer un constructeur par copie, un **operator=**et un destructeur pour que la classe fonctionne correctement.

38 Quand vous écrivez un constructeur de copie pour une classe dérivée, souvenez vous d'appeler le constructeur de copie de la classe de base de manière explicite (ainsi que ceux des objets membres) (cf. chapitre 14.) Si vous ne le faites pas, le constructeur par défaut sera appelé pour la classe de base (ou pour les objets membres) et ce n'est probablement pas ce que vous voulez. Pour appeler le constructeur de copie de la classe de base, passez lui l'objet dérivé que vous êtes en train de copier: **Derived(const Derived& d) : Base(d) { // ...**

39 Quand vous écrivez un opérateur d'assignation pour une classe dérivée, pensez à appeler explicitement l'opérateur d'assignation de la classe de base (cf. chapitre 14.) Si vous ne le faites pas, rien ne se passera (la même chose est vraie pour les objets membres). Pour appeler l'opérateur d'assignation de la classe de base, utilisez le nom de la classe de base et la résolution de portée: **Derived& operator=(const Derived& d) { Base::operator=(d);**

40 Si vous avez besoin de minimiser les temps de compilation en phase de développement d'un gros projet, utilisez la technique des classes chats de Cheshire expliquée au chapitre 5 et retirez la uniquement si l'efficacité à l'exécution est un problème.

41 Evitez le préprocesseur. Utilisez toujours **const**pour la substitution de valeur et **inline**pour les macros.

42 Gardez les portées aussi petites que possible, pour que la visibilité et la durée de vie de vos objets soient eux aussi aussi courtes que possible. Cela réduit les risques d'utilisation d'un objet dans un mauvais contexte et de cacher un bug difficile à trouver. Par exemple, supposons que vous ayiez un conteneur et une partie de code qui le parcourt de manière itérative. Si votre copiez ce code pour l'utiliser avec un nouveau conteneur, vous pouvez accidentellement terminer en utilisant la taille de l'ancien conteneur en tant que limite supérieure du nouveau. Par contre, si l'ancien conteneur est hors de portée, l'erreur sera affichée lors de la compilation.

43 Evitez les variables globales. Tâchez de placer les données à l'intérieur de classes. Les fonctions globales sont plus susceptibles d'apparaître naturellement que les variables globales, cependant vous découvrirez sans doute plus tard qu'une fonction globale devrait être une membre **static**d'une classe.

44 Si vous devez déclarer une classe ou une fonction issue d'une bibliothèque, faites le toujours en incluant un fichier d'en-tête. Par exemple, si vous voulez créer une fonction pour écrire dans un **ostream**, ne déclarez jamais **ostream**vous-même en utilisant une spécification de type incomplète telle que **class ostream;**Cette approche rend votre code vulnérable aux éventuels changements de représentation. (Par exemple, **ostream**pourrait être en fait un **typedef**.) Au lieu de cela, utilisez le fichier d'en-tête : **#include <iostream>**. Lorsque vous créez vos propres classes, si la bibliothèque est importante fournissez une forme abrégée à vos utilisateurs en leur fournissant des spécifications incomplètes (déclarations de nom de classes) pour le

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

cas où elles ne doivent utiliser que des pointeurs. (Cela peut accélérer la compilation.)

45 Lorsque vous choisissez le type de retour d'un opérateur surchargé, considérez ce qui peut arriver si les expressions sont écrites à la suite. Renvoyez une copie ou une référence vers une *lvalue*(**return *this**) de manière à permettre le chaînage d'expression (**A = B = C**). Quand vous définissez l' **operator=**rappelez vous du cas **x=x**.

46 Quand vous écrivez une fonction, préférez le passage des arguments par référence **const**. Tant que vous ne devez pas modifier l'objet passé, cette pratique est la meilleure car elle a la simplicité d'un passage par valeur mais ne nécessite ni construction ni destruction pour créer un objet local, ce qui arrive lorsque l'on passe un objet par valeur. Normalement, vous ne devriez pas vous inquiéter de l'efficacité en concevant et en construisant votre système mais, cette habitude est gagnante à tous les coups.

47 Soyez conscients des temporaires **.**Lorsque vous cherchez à améliorer les perfromances, cherchez la création de temporaires, principalement avec surcharge d'opérateurs. Si vos constructeurs et destructeurs sont compliqués, le coût de la création et de la destruction d'objets temporaires peut être élevé. Quand vous retourner une valeur depuis une fonction, essayez toujours de construire un objet “sur place” avec l'appel à un constructeur lors du retour: **return MyType(i, j);**plutôt que **MyType x(i, j); return x;**Le premier retour (appelée *optimisation de la valeur de retour*) élimine l'appel à un constructeur par copie et à un destructeur.

48 Lorsque vous créez des constructeurs, envisagez les exceptions. Dans le meilleur des cas, le constructeur ne fera rien qui lance une exception. Dans le meilleur scénario suivant, la classe sera composée par des classes héritées robustes, et elles pourront alors se nettoyer correctement d'elles-mêmes si une exception est lancée. Si vous devez avoir devez avoir des pointeurs nus, vous êtes responsables de la récupération de vos propres exceptions et de veiller à la déallocation de chaque ressource pointée avant de lancer une exception au sein de votre constructeur. Si un constructeur doit échouer, l'action appropriée est de lancer une exception.

49 Faites juste le minimum dans vos constructeurs. Non seulement cela produit des constructeurs moins coûteux lors de leur appel (dont certains ne sont peut être pas sous votre contrôle) mais alors il sera moins probable que vos constructeurs lancent une exception ou pausent problème.

50 La responsabilité du destructeur est de libérer les ressources allouée durant la vie de l'objet, pas seulement durant la construction.

51 Utilisez une hiérarchie d'exceptions, de préférence dérivée de la hiérarchie des exceptions du Standard C++, et imbriquée comme une classe publique dans la classe qui lance l'exception. La personne qui les récupérere peut ainsi récupérer les types spécifiques d'exceptions, suivis par le type de base. Si vous ajoutez de nouvelles exceptions dérivées, le code client existant pourra toujours récupérer les exceptions au travers de leur type de base.

52 Lancez les exceptions par valeur, et récupérez les par référence. Laissez la gestion de la mémoire au mécanisme de gestion des exceptions. Si vous lancez des pointeurs vers des objets exceptions qui ont été créés sur tas, le responsable de la récupération de l'exception devra être en mesure de détruire l'exception, ce qui est un mauvais couplage. Si vous récupérez les exceptions par valeur, vous occasionnez des constructions et destructions supplémentaires ; pire encore, les portions dérivées de votre exception peuvent être perdues en cas de *transtypage*par valeur.

53 N'écrivez vos propres templates de classes que si vous en avez besoin. Regardez d'abord dans la bibliothèque Standard C++, puis vers les fournisseurs qui créent des outils dans un but particulier. Devenez efficace dans leur usage et vous augmenterez considérablement votre productivité.

54 Lorsque vous créez des templates, cherchez le code qui ne dépend pas d'un type particulier et placez le dans une classe de base non-template pour éviter le gonflement de code. En utilisant l'héritage ou la composition, vous pouvez créer des templates dans lesquels les partie de codes qu'ils contiennent dépendent du type et sont donc essentielles.

55 N'utilisez pas les fonctions de **<cstdio>**telles que **printf( )**. Apprenez à utiliser les **iostream**à la place (flux d'entrée/sortie, NdC) ; Ils sont sécurisés et malléables au niveau du type, et significativement plus efficaces. Votre investissement sera régulièrement récompensé. En général, préférez utiliser une bibliothèque C++ plutôt qu'une bibliothèque C.

56 Evitez d'avoir recours au types intégrés du C. Ils sont supportés pour des raisons de rétrocompatibilité, mais sont bien moins robustes que les classes C++, ce qui risque d'augmenter le temps que vous passerez à la chasse aux bugs.

57 Chaque fois que vous utilisez des types intégrés comme variables globales ou comme variables automatiques, ne les définissez pas jusqu'à ce que vous puissiez également les initialiser. Définissez une variable par ligne, en même temps que leur initialisation. Quand vous définissez des pointeurs, placez l'étoile

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

‘ ***** ’ à coté du nom du type. Vous pouvez le faire sans risque quand vous définissez une variable par ligne. Cette manière de faire tend à être moins confuse pour le lecteur.

58 Garantissez l'apparition de l'initialisaion dans tous les aspect de votre code. Effectuez l'initialisation de tous les membres dans la liste d'initialisation du constructeur, même pour les types intégrés (en utilisant l'appel à de pseudo constructeurs). L'utilisation de la liste d'iniitalisation du constructeur est souvent plus efficace lors de l'initialisation de sous-objets; autrement, le constructeur par défaut est appelé, et vous vous retrouvez en plus de cela à appeler d'autres fonctions membres (probablement **operator=**) de manière à avoir l'initialisation que vous désirez.

59 N'utilisez pas la forme **MyType a= b;**pour définir un objet. Ce dispositif est une source importante de confusion parce qu'elle appelle un constructeur au lieu de l' **operator=**. Pour la clarté du code, utilisez la forme **MyType a(b);**à la place. Le résultat est identique, mais cela évitera d'égarer les programmeurs.

60 Utilisez les transtypages explicites décrit au chapitre 3. Un transtypage supplante le système de typage normal et est un lieu d'erreur potentiel. Comme les transtypages explicites divisent le transtypage unique-qui-fait-tout du C en classes bien délimitées de transtypage, toute personne débuggant et maintenant le code peut facilement trouver tous les endroits où les erreurs logiques ont le plus de chance de se produire.

61 Pour qu'un programme soit robuste, chaque composant doit être robuste. Utilisez tous les outils fournis par le C++ : contrôle d'accès, exceptions, type **const**ou non correct, vérification de type, etc. dans chaque classe que vous créez. De cette manière vous pouvez passer au niveau d'abstraction suivant lorsque vous créez votre système.

62 Veillez à la l'exactitude de la **const**ance. Cela permet au compilateur de pointer certains bugs subtils qui seraient autrement difficiles à trouver. Cette pratique demande un peu de discipline et doit être utilisée de manière consistante à travers toutes vos classes, mais c'est payant.

63 Utilisez les vérifications d'erreur du compilateur à votre avantage. Effectuez toutes les compilations avec tous les avertissement, et corriger votre code pour supprimer tous les avertissements. Ecrivez du code qui utilise les erreurs de compilation plutôt qu'un code qui cause des erreurs d'exécution (par exemple, n'utilisez pas de liste d'arguments variables, qui supprime le test de type). Utilisez **assert( )**pour le débuggage, mais utilisez les exception pour les erreurs à l'exécution.

64 Préférez les erreurs à la compilation aux erreur à l'exécution. Essayez de gérer une erreur aussi près que possible de l'endroit où elle se produit. Préférez la gestion d'une erreur en ce point plutôt que de lancer une exception. Capturez une exception dans le premier gestionnaire (handler, ndc) qui dispose de suffisamment d'information pour la gérer. Faites ce que vous pouvez avec l'exception au niveau actuel ; si cela ne résoud pas le problème, relancez l'exception (voir Volume 2 pour plus de détails.)

65 Si vous employez les caractéristiques d'exception (voir le volume 2 de ce livre, téléchargeable sur *www.BruceEckel.com*, pour en apprendre davantage sur la manipulation des exceptions), installez votre propre fonction **unexpected( )**en utilisant **set_unexpected( )**. Votre **unexpected( )**devrait reporter l'erreur dans un fichier avant de relancer l'exception courante. De cette manière si une fonction existante se trouve supplantée et se met à lancer une exception, vous aurez une sauvegarde du coupable et pourrez modifier votre code pour manipuler l'exception.

66 Créez une fonction **terminate( )**définie par l'utilisateur (indiquant une erreur du programmeur) pour reporter l'erreur qui a causé l'exception, puis libérer les ressources système, et quittez le programme

67 Si un destructeur appelle des fonctions, ces fonctions peuvent lancer des exceptions. Un destructeur ne peut pas lancer une exception (qui pourrait résulter en un appel à **terminate( )**, qui indique une erreur de programmation), si bien que les destructeurs qui appellent des fonctions doivent récupérer et gérer leurs propres exceptions.

68 Ne créez pas vos propres noms de membre décorés (par ajout d'underscores, par notation hongroise, etc.) à moins que vous n'ayez un grand nombre de valeurs globales pré existantes ; autrement, laissez les classes et les espaces de nommage gérer la portée des noms pour vous.

69 Vérifiez les surcharges. Une fonction ne devrait pas exécuter conditionnellement du code selon la valeur d'un argument, qu'il soit fournis par défaut ou non. Dans ce cas, vous devriez créer deux fonctions surchargées ou plus.

70 Cachez vos pointeurs à l'intérieur de classes conteneurs. Ne les sortez que quand vous devez faire des opérations directement dessus. Les pointeurs ont toujours été une source majeure de bugs. Quand vous utililsez **new**, essayez de placer le pointeur résultant dans un conteneur. Préférez qu'un conteneur "possède" ses pointeurs, de manière à ce qu'il soit responsable de leur nettoyage. Mieux encore, placez le pointeur à l'intérieur d'une classe ; si vous voulez toujours qu'elle ressemble à un pointeur, surchargez **operator->**et

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

operator*. Si vous devez avoir un pointeur libre, initialisez le toujours, de préférence à l'adresse d'un objet, mais à zéro si nécessaire. Mettez le à zéro quand vous l'effacer pour éviter les suppressions multiples accidentelles.

71 Ne surchargez pas les **new**et **delete**globaux ; surchargez les toujours classe par classe. Surcharger les versions globales affectent l'ensemble du projet du client, chose que seul le créateur du projet doit contrôler. Quand vous surchargez **new**et **delete**pour des classes, ne présumez pas que vous connaissez la taille de l'objet ; quelqu'un peu hériter de vos classes. Utilisez les arguments fournis. Si vous faites quelque chose de spécial, considérez l'effet que cela peut avoir sur les héritiers.

72 Prévenez le saucissonnage des objets. Il n'y a pratiquement aucun sens à effectuer un transtypage ascendant d'un objet par valeur. Pour éviter le transtypage ascendant par valeur, placez des fonctions virtuelles pures dans votre classe de base.

73 Parfois, une simple agrégation fait le travail. Un “sytème confortable pour les passagers u sur une ligne aérienne est fait d'éléments disjoints : sièges, air conditionnée, vidéo... et pourtant vous devez en créer beaucoup dans un avion. Allez vous faire des membres privés et créer une toute nouvelle interface ? Non – dans ce cas, les composant font aussi parti de l'interface publique, et vous devriez donc créer des objet membres publiques. Ces objets ont leur propre implémentation privée, qui est toujours sécurisée. Soyez conscient que l'aggrégation simple n'est pas une solution à utiliser souvent, mais cela peut se produire.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

19 - C: Lectures recommandées

Ressources pour approfondir.

19.1 - C

Thinking in C: Foundations for Java & C++, de Chuck Allison (un séminaire sur CD-ROM de MindView, Inc., ©2000, attaché au dos de ce livre et également disponible sur *www.BruceEckel.com*). C'est un cours qui inclut leçons et transparents sur les bases du langage C pour vous préparer à apprendre le Java ou le C++. Ce n'est pas un cours de C exhaustif ; il ne contient que le strict nécessaire pour aller vers les autres langages. Des sections supplémentaires spécifiques à d'autres langages introduisent des caractéristiques pour le futur programmeur C++ ou Java. Pré-requis recommandés : de l'expérience dans un langage de programmation de haut niveau, comme Pascal, Basic, Fortran ou LISP (il est possible de tracer son chemin à travers les CD sans cette expérience, mais le cours n'est pas conçu comme une introduction aux bases de la programmation).

19.2 - C++ en général

The C++ Programming Language, 3rd edition, de Bjarne Stroustrup (Addison-Wesley 1997). D'un certain point de vue, le livre que vous tenez en ce moment a pour but de vous permettre d'utiliser le livre de Bjarne comme référence. Comme ce livre contient la description du langage par l'auteur de ce langage, c'est typiquement l'endroit où vous irez pour résoudre n'importe quelle incertitude concernant ce que le C++ est supposé faire ou ne pas faire. Quand vous vous serez fait la main sur le langage et serez prêt pour passer aux choses sérieuses, vous en aurez besoin.

C++ Primer, 3rd Edition, de Stanley Lippman et Josee Lajoie (Addison-Wesley 1998). Plus vraiment un livre d'introduction ; il s'est tranformé en un livre épais rempli de beaucoup de détails, et c'est celui que j'utilise avec celui de Stroustrup quand j'essaye de résoudre un problème. *Penser en C++*devrait fournir une base pour comprendre *C++ Primer*ainsi que le livre de Stroustrup.

C & C++ Code Capsules, de Chuck Allison (Prentice-Hall, 1998). Ce livre suppose que vous connaissiez déjà le C et le C++, et couvre certains problèmes sur lesquels vos pouvez être rouillés, ou que vous pouvez ne pas avoir bien compris au début. Ce livre bouche des trous en C comme en C++.

The C++ Standard. C'est le livre sur lequel le comité a travaillé si dur pendant toutes ces années. Il n'est *pas*gratuit, malheureusement. Au moins, vous pouvez acheter la version électronique en PDF pour seulement 18 dollars à *www.cssinfo.com*.

19.2.1 - Ma propre liste de livres

Listés par ordre de publication. Tous ne sont pas disponible actuellement.

Computer Interfacing with Pascal & C(Auto-publié via Eisys impression, 1988. Uniquement disponible via *www.BruceEckel.com*). Une introduction à l'électronique qui remonte à l'époque où CP/M était toujours le roi et DOS un arriviste. J'utilisais des langages de haut niveau et souvent le port parallèle de l'ordinateur pour conduire divers projets électroniques. Adapté d'après mes chroniques pour le premier et le meilleur magazine pour lequel j'ai écrit, *Micro Cornucopia*(pour paraphraser Larry O'Brien, longtemps éditeur de *Software Development Magazine*: le meilleur magazine informatique jamais publié – ils avaient même des plans pour construire un robot dans un pot de fleurs !). Hélas, Micro C s'est perdu longtemps avant qu'internet n'apparaisse. Créer ce livre fut une expérience de publication extrêmement satisfaisante.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Using C++(Osborne/McGraw-Hill 1989). Un des premiers livres sur le C++. Il est épuisé et remplacé par sa deuxième édition, renommée **C++ Inside & Out**.

C++ Inside & Out(Osborne/McGraw-Hill 1993). Comme mentionné ci-dessus, c'est en fait la deuxième édition de **Using C++**. Le C++ dans ce livre est assez exact, mais il remonte à 1992 et *Penser en C*++est fait pour le remplacer. Vous pouvez en trouver davantage sur ce livre et télécharger les codes sources à *www.BruceEckel.com*.

Thinking in C++, 1st edition(Prentice-Hall 1995).

Black Belt C++, the Master's Collection, Bruce Eckel, éditeur (M&T Books 1994). Epuisé. Un ensemble de chapitres par différentes lumières du C++ basé sur leurs présentations dans le circuit C++ à la *Software Development Conference*, que j'ai présidée. La couverture de ce livre m'a poussé à gagner le contrôle de la conception de toutes les couvertures futures.

Thinking in Java, 2nd edition (Prentice-Hall, 2000). La première édition de ce livre a gagné le prix de la Productivité de *Software Development Magazine*, et le prix de l'Editeur du *Java Developer's Journal*en 1999. Il est téléchargeable à *www.BruceEckel.com*, (et sa traduction en français, *Penser en Java*, peut se télécharger ici : http://penserenjava.free.fr/, ndt).

19.3 - Profondeurs et recoins

Ces livres s'enfoncent plus profondément dans le langage, et vous aident à éviter les embûches habituelles inhérentes au développement de programmes C++.

Effective C++(2nd Edition, Addison-Wesley 1998) et **More Effective C++**(Addison-Wesley 1996), de Scott Meyers. Le classique à posséder absolument pour la résolution sérieuse de problèmes et la conception de code en C++. Je me suis efforcé de capturer et exprimer beaucoup des concepts de ces livres dans *Penser en C*++, mais je ne m'abuse pas à penser que j'y suis parvenu. Si vous passez un temps significatif avec le C++ vous finirez avec ces livres. Disponibles également sur CD-ROM.

Ruminations on C++, de Andrew Koenig et Barbara Moo (Addison-Wesley, 1996). Andrew a travaillé directement avec Stroustrup sur beaucoup d'aspects du C++ et est une autorité fiable en la matière. J'ai également trouvé rafraichissante sa perspicacité incisive, et j'ai beaucoup appris de lui, par écrit et en personne, au long des années.

Large-Scale C++ Software Design, de John Lakos (Addison-Wesley, 1996). Couvre des problèmes et répond à des questions que vous rencontrerez pendant la création de grands projets, mais souvent de plus petits aussi.

C++ Gems, Stan Lippman, editeur (SIGS publications, 1996). Une sélection d'articles de *The C++ Report*.

The Design & Evolution of C++, de Bjarne Stroustrup (Addison-Wesley 1994). Des explications de l'inventeur du C++ sur les raisons pour lesquelles il a pris certaines décisions. Pas essentiel, mais intéressant.

19.4 - Analyse & conception

Extreme Programming Explainedpar Kent Beck (Addison-Wesley 2000). J' *adore*ce livre. Oui, j'ai tendance à avoir une approche des choses radicale, mais j'ai toujours senti qu'il pourrait y avoir un procédé de développement de programme très différent et bien meilleur, et je pense que XP s'en approche diablement. Le seul livre qui a eu un impact similaire sur moi fut *PeopleWare*(décrit ci-dessous), qui parle avant tout de l'environnement et traite de la culture d'entreprise. *Extreme Programming Explained*parle de programmation et bouleverse la plupart des choses, même les récentes "découvertes". Ils vont même jusqu'à dire que les les images sont acceptables tant que vous ne

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

passez pas trop de temps dessus et êtes prêts à vous en débarrasser. (Vous remarquerez que ce livre ne comporte *pas*“de logo de certification UML” sur sa couverture.) Je pourrais imaginer que l'on décide de travailler pour une société en fonction du seul fait qu'ils utilisent XP. Petit livre, petits chapitres, se lit sans effort, excitant d'y penser. Vous commencez à vous imaginer travailler dans une telle ambiance et cela fait naître les visions d'un monde complètement nouveau.

UML Distilledpar Martin Fowler (2éme édition, Addison-Wesley, 2000). Quand vous rencontrez UML pour la première fois c'est d'abord intimidant à cause du nombre de diagrammes et de détails. D'après Fowler, la plupart de ces choses ne sont pas nécessaires donc il réduit à l'essentiel. Pour la plupart des projets, vous n'avez besoin de conaître que quelques outils de diagramme, et le but de Fowler est d'arriver à une bonne conception plutôt que de s'inquiéter de toutes les façons d'y arriver. Un petit livre agréable, fin et lisible ; le premier à vous procurer si vous avez besoin de comprendre UML.

The Unified Software Development Processpar Ivar Jacobsen, Grady Booch, et James Rumbaugh (Addison-Wesley 1999). Je suis entré dans ce live en étant tout prêt à ne pas l'aimer. Il semblait avoir toutes les qualités d'un ennuyeux texte d'université. J'ai été heureusement surpris – seul l'emballage du livre contient des explications qui donnent l'impression que ces concepts ne sont pas clairs pour les auteurs. Le coeur du livre est non seulement clair, mais également agréable. Et le meilleur de tout, le procédé contient beaucoup de sens pratique. Ce n'est pas de l'"Extreme Programming" (et n'a pas leur clarté sur les tests) mais c'est aussi un élément de la force de frappe écrasante d'UML – même si vous ne pouvez pas faire adopter XP, la plupart des gens ont grimpé dans le train du mouvement “UML est bon” (indépendamment de leur *réel*niveau d'expérience en la matière) et vous pouvez donc probablement obtenir son adoption. Je pense que ce livre devrait être le livre phare d'UML, celui que vous pouvez lire après *UML Distilled*de Fowler, quand vous voulez plus de détails.

Avant que vous ne choisissiez une méthode, il est utile d'avoir le point de vue de ceux qui n'essayent pas d'en vendre une. Il est facile d'adopter une méthode sans vraiment comprendre ce que vous voulez en tirer ou ce qu'elle fera pour vous. D'autres l'utilisent, ce qui semble une raison suffisante. Toutefois, les humains ont un étrange petit caprice psychologique : s'ils veulent croire que quelque chose résoudra leurs problèmes, ils l'essaieront. (C'est l'expérimentation, ce qui est bon.) Mais si cela ne résout pas leurs problèmes, ils peuvent redoubler leurs efforts et commencer à annoncer haut et fort quelle grande chose ils ont découvert. (C'est du déni, ce qui n'est pas une bonne chose.) L'hypothèse ici pourrait être que si vous pouvez faire monter d'autres personnes dans le même navire, vous ne vous retrouverez pas seul, même si ça ne va nulle part (ou si ça coule).

Ce n'est pas pour insinuer que les méthodologies ne mènent nulle part, mais que vous devriez être armés jusqu'aux dents avec des outils mentaux qui vous aident à rester en mode expérimentation (“Ca ne fonctionne pas ; essayons autre chose”) et hors du mode déni (“Ce n'est pas vraiment un problème. Tout est merveilleux, nous n'avons pas besoin de changer”). Je pense que les livres suivants, lus *avant*de choisir une méthode, vous fourniront ces outils.

Software Creativity, par Robert Glass (Prentice-Hall, 1995). C'est le meilleur livre que j'ai vu qui discute du *point de vue*de toute la question de la méthodologie. C'est une collection de courts essais et de papiers que Glass a écrit et parfois acquis (P.J. Plauger est un contributeur), reflétant ses nombreuses années d'étude et de réflexion sur le sujet. Ils sont divertissants et juste assez longs pour dire ce qui est nécessaire ; il ne radote ni ne vous ennuie. Il ne fait pas simplement des effets de manches, non plus ; il y a des centaines de références à d'autres papiers et d'autres études. Tous les programmeurs et managers devraient lire ce livre avant de patauger dans le marais de la méthodologie.

Software Runaways: Monumental Software Disasters, par Robert Glass (Prentice-Hall 1997). La grande chose avec ce livre est qu'il met en avant ce dont on ne parle pas : combien de projets non seulement échouent, mais échouent spectaculairement. Il me semble que la plupart d'entre nous pense encore “Ca ne peut pas m'arriver” (ou “Ca ne peut *encore*se produire !”) et je pense que cela nous désavantage. En gardant à l'esprit que les choses peuvent toujours aller mal, vous êtes en bien meilleure position pour qu'elles aillent bien.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

Object Lessonspar Tom Love (SIGS Books, 1993). Un autre bon livre de *point de vue*.

Peopleware, par Tom Demarco et Timothy Lister (Dorset House, 2ème edition 1999). Bien qu'ils aient une expérience dans le développement de logiciels, ce livre traite des projets et des équipes en général. Mais l'accent est mis sur les *personnes*et leurs besoins plutôt que sur la technologie et ses besoins. Ils décrivent la création d'un environnement où les gens sont heureux et productifs, plutôt que de décider des règles que ces personnes devront suivre pour être les composants adéquats d'une machine. Je pense que cette dernière attitude est la grande responsable du comportement de certains programmeurs qui sourient et approuvent quand on adopte une méthode ou l'autre et continuent tranquillement à faire ce qu'ils ont toujours fait.

Complexity, par M. Mitchell Waldrop (Simon & Schuster, 1992). Ce sont les chroniques d'un groupe de scientifiques de différentes disciplines à Santa Fe, New Mexico, qui discutent de réels problèmes que les disciplines individuelles ne pouvaient pas résoudre (le marché boursier en économie, la formation initiale de la vie en biologie, pourquoi les gens font ce qu'ils font en sociologie, etc.). En croisant la physique, l'économie, la chimie, les mathématiques, l'informatique, la sociologie et autres, une approche multidisciplinaire de ces problèmes se développe. Mais plus important, une nouvelle manière d' *aborder*ces problèmes ultra-complexes émerge : loin du déterminisme mathématique et de l'illusion que vous pouvez écrire une équation qui prédit tous les comportements, il s'agit plutôt d' *observer*et de chercher un modèle et d'essayer d'émuler ce modèle par tous les moyens. (Le livre raconte, par exemple, la naissance des algorithmes génétiques.) Je crois que ce mode de pensée est utile quand nous observons des manières de gérer des projets de logiciels de plus en plus complexes.

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

20 - Copyright et traduction

20.1 - Pour la version anglaise :

Prentice Hall Upper Saddle River, New Jersey 07458 http://www.prenhall.com

Publisher: Alan Apt Production Editor: Scott Disanno Executive Managing Editor: Vince O'Brien Vice President and Editorial Director: Marcia Horton Vice President of Production and Manufacturing: David W. Riccardi Project Manager: Ana Terry Book Design, Cover Design and Cover Line Art: Daniel Will-Harris, daniel@will-harris.com Cover Watercolor: Bruce Eckel Copy Editor: Stephanie English Production Coordinator: Lori Bulwin Editorial Assistant: Toni Holm Marketing Managers: Jennie Burger, Bryan Gambrel

©2000 by Bruce Eckel, MindView, Inc. Published by Prentice Hall Inc. Pearson Higher Education Upper Saddle River, New Jersey 07632

Les informations contenues dans ce livre sont distribuées "telles quelles", sans garantie. Toutes les précautions ont été prises dans la préparation de ce livre, ni l'auteur ni l'éditeur ne peuvent être tenus responsables pour la perte ou les dommages causés ou supposés avoir été causés directement ou indirectement par les instructions contenues dans ce livre ou par les programmes ou éléments informatiques décrits dedans.

Tous droits réservés. Aucune partie de ce livre ne peut être reproduite sous quelle que forme que ce soit ou par n'importe quel moyen électronique ou mécanique y compris les systèmes de stockage et de recherche sans la permission écrite de l'éditeur ou de l'auteur, sauf par les critiques qui sont autorisés à en citer de courts passages dans un article. Tous les noms utilisés dans les exemple de ce livre sont fictifs ; toute relation avec des personnes vivantes ou décédée ou des personnages de fiction d'autres ouvrages est purement accidentelle.

Printed in the United States of America 10 9 8 7 6 5 4 3 2 1 ISBN 0-13-979809-9 Prentice-Hall International (UK) Limited, London Prentice-Hall of Australia Pty. Limited, Sydney Prentice-Hall Canada, Inc., Toronto Prentice-Hall Hispanoamericana, S.A., Mexico Prentice-Hall of India Private Limited, New Delhi Prentice-Hall of Japan, Inc., Tokyo Pearson Education Asia Ltd., Singapore Editora Prentice-Hall do Brasil, Ltda., Rio de Janeiro

20.2 - Pour la version française :

20.2.1 - Equipe de traduction :

1 Aurélien Regat-Barrel
2 Nourdine Falola
3 bricerive
4 chessperso
5 coyotte507
6 deeal
7 ebgérard
8 elrond64
9 Sylvain Jorge Do Marco
10 Jean Christophe Beyler
11 Higestromm
12 Marc Blétry
13 jolatouf
14 koala01
15 mont5piques
16 mujigka

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/

17 Pascal Barbier
18 roulious
19 Traroth2
20 toxcct
21 Zavonen

Des extraits de chapitres de Thinking In Java, 2nd Edition traduits par des membres du groupe de traduction tij2ndenfrancais on été récupérés et intégrés à cette traduction de Thinking In C++, 2nd Edition, Volume1 par Aurélien Regat-Barrel (le contacter pour plus d'informations). Les personnes concernées sont les suivants :

1 Jean-Pierre Vidal (chapitres 0.4, 0.6 et 0.7)
2 Jérome Quelin (chapitres 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.8, 1.9, 1.10, 1.11 et 1.13)

20.2.2 - Relecteurs

- Aitone le chien
- amaury pouly
- Aurélien Regat-Barrel
- Beldom
- bricerive
- coyotte507
- disturbedID
- ebgérard
- elrond64
- HanLee
- Marc Blétry
- Hylvenir
- Kerwando
- koala01
- LaurentN
- maxim_um
- mujigka
- Pascal Barbier
- r0d
- Zavonen

20.2.3 - Mise en place du projet

- Aurélien Regat-Barrel : responsable du projet
- Gnux : motivation des troupes :-)
- lalystar : découpage du document principal en sous-chapitres
- Laurent Gomila : mise au gabarit à partir du document HTML original
- Mathieu : développement de l'infrastructure du site en Php
- Marc Blétry : coup de main PHP, mise en forme du document final

Les sources présentés sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2005 - Bruce Eckel. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://bruce-eckel.developpez.com/livres/cpp/traduction/ticpp2vol1/