Herb Sutter

Mieux programmer en C++

47 problèmes pratiques résolus

Eyrolles

Mieux programmer en C++

47 problèmes pratiques résolus

Chez le même éditeur

Langages C/C++

J.-B. BOICHAT. – **Apprendre Java et C++ en parallèle**.
N°9158, 2000, 620 pages.

C. JAMSA. – **C/C++ : la bible du programmeur**.
N°9058, 1999, 1 010 pages.

DEITEL. – **Comment programmer en C++**. *Cours et exercices*
N°13000, 1999, 1 100 pages.

C. DELANNOY. – **Programmer en langage C++**.
N°9019, 4e édition, 1998, 624 pages.

C. DELANNOY. – **La référence du C norme ANSI/ISO**.
N°9036, 1998, 950 pages.

C. DELANNOY. – **Exercices en langage C**.
N°8984, 1992, 272 pages.

Environnements de programmation

G. LEBLANC. – **Borland C++ Builder 3**.
N°9184, 2000, 780 pages.

C. DELANNOY. – **Apprendre le C++ avec Visual C++ 6**.
N°9088, 1999, 496 pages.

I. HORTON. – **Visual C++ 6**.
Avec un CD-Rom contenant le produit Microsoft Visual C++ 6 Introductory Edition.
N°9043, 1999, 1 250 pages.

Programmation Internet-intranet

D. K. FIELDS, M. A. KOLB. – **JSP – JavaServer Pages**.
N°9228, 2000, 550 pages.

L. LACROIX, N. LEPRINCE, C. BOGGERO, C. LAUER. – **Programmation Web avec PHP**.
N°9113, 2000, 382 pages.

A. HOMER, D. SUSSMAN, B. FRANCIS. – **ASP 3.0 professionnel**.
N°9151, 2000, 1 150 pages.

N. MCFARLANE. – **JavaScript professionnel**.
N°9141, 2000, 950 pages.

A. PATZER *et al*. – **Programmation Java côté serveur**.
Servlets, JSP et EJB. N°9109, 2000, 984 pages.

J.-C. BERNADAC, F. KNAB. – **Construire une application XML**.
N°9081, 1999, 500 pages.

P. CHALÉAT, D. CHARNAY. – **Programmation HTML et JavaScript**.
N°9182, 1998, 480 pages.

Mieux programmer en C++

47 problèmes pratiques résolus

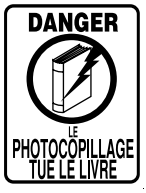
Traduit de l'anglais par Thomas Pétillon

ÉDITIONS EYROLLES
61, Bld Saint-Germain

75240 Paris cedex 05
www.editions-eyrolles.com

Traduction autorisée de l'ouvrage en langue anglaise intitulé
Exceptional C++, 47 Engineering Puzzles, Programming Problems and Exception-Safety Solutions
The Addison-Wesley Bjarne Stroustrup Series
Addison-Wesley Longman, a Pearson Education Company,
Tous droits réservés
ISBN 0-201-615622

Traduit de l'anglais par Thomas Pétillon



 Version eBook (ISBN) de l'ouvrage : 2-212-28116-1.

À ma famille, pour sa patience et son soutien sans faille,
À Eric Wilson, Jeff Summer, Juana Chang, Larry Palmer,
ainsi qu'à l'ensemble de l'équipe de développement de PeerDirect,
qui a été en quelque sorte ma « seconde famille »
au cours de ces quatre dernières années :

Declan West
Doug Shelley
Duk Loi
Ian Long
Justin Karabegovic
Kim Nguyen
Margot Fulcher
Mark Cheers
Morgan Jones
Violetta Lukow
À tous, merci pour tout.

Sommaire

Préface

Vous avez entre les mains un ouvrage remarquable, probablement le premier dans son genre. Consacré aux développeurs C++ déjà expérimentés, il apporte un éclairage nouveau sur le langage : de la robustesse aux exceptions aux subtilités des espaces de nommage, des ressources cachées de la bibliothèque standard à l'emploi judicieux de l'héritage, tous les sujets sont abordés dans le contexte de leur utilisation professionnelle. Des surprises que peuvent réserver les itérateurs, les algorithmes de résolution de noms ou les fonctions virtuelles aux techniques permettant de concevoir efficacement des classes, de minimiser les dépendances au sein d'un programme ou d'utiliser au mieux les modèles génériques, la majorité des techniques et des pièges du C++ sont passées en revue sous la forme de cas pratiques très pertinents.

Au fur et à mesure de ma lecture, en comparant mes solutions à celles proposées par Sutter, je me suis vu tomber dans des pièges bien plus souvent que je n'aimerais l'admettre. Ceci pourrait, certes, s'expliquer par une insuffisante maîtrise du langage. Mon opinion est plutôt que tout développeur, si expérimenté qu'il soit, doit être très prudent face à la puissance du C++, arme à double tranchant. Des problèmes complexes peuvent être résolus avec le C++, à condition de parfaitement connaître les avantages mais aussi les risques des nombreuses techniques disponibles. C'est justement l'objet de ce livre, qui sous la forme originale de problèmes résolus, inspirés d'articles initialement parus sur l'Internet dans « *Guru Of The Week* », fait un tour complet du langage et de ses fonctionnalités.

Les habitués des groupes de discussion Internet savent à quel point il est difficile d'être élu « gourou » de la semaine. Grâce à ce livre, vous pourrez désormais prétendre écrire du code de « gourou » chaque fois que vous développerez.

Scott Meyers
Juin 1999

Avant-propos

Ce livre a pour but de vous aider à écrire des programmes plus robustes et plus performants en C++. La majorité des techniques de programmation y sont abordées sous la forme de cas pratiques, notamment inspirés des 30 premiers problèmes initialement parus sur le groupe de discussion Internet « *Guru of The Week*[1] ».

Le résultat n'est pas un assemblage disparate d'exemples : cet ouvrage est, au contraire, spécialement conçu pour être le meilleur des guides dans la réalisation de vos programmes professionnels. Si la forme problème/solution a été choisie, c'est parce qu'elle permet de situer les techniques abordées dans le contexte de leur utilisation réelle, rendant d'autant plus profitable la solution détaillée, les recommandations et discussions complémentaires proposées au lecteur à l'occasion de chaque étude de cas. Parmi les nombreux sujets abordés, un accent particulier est mis sur les thèmes cruciaux dans le cadre du développement en entreprise : robustesse aux exceptions, conception de classes et de modules faciles à maintenir, utilisation raisonnée des optimisations, écriture de code portable et conforme à la norme C++.

Mon souhait est que cet ouvrage vous accompagne efficacement dans votre travail quotidien, et, pourquoi pas, vous fasse découvrir quelques unes des techniques élégantes et puissantes que nous offre le C++.

Comment lire ce livre ?

Avant tout, ce livre s'adresse aux lecteurs ayant déjà une bonne connaissance du langage C++. Si ce n'est pas encore votre cas, je vous recommande de commencer par la lecture d'une bonne introduction au langage (*The C++ Programming Language*[2]

1. Littéralement : le « gourou » de la semaine.
2. Stroustrup B. *The C++ Programming Language, Third Edition* (Addison Wesley Longman, 1997)

de *Bjarne Stroustrup* ou *C++ Primer*[1], de *Stan Lippman* et *Josée Lajoie)*, et l'incontournable classique de Scott Meyer : *Effective C++* (la version CD-Rom est très facile d'emploi)[2].

Chacun des problèmes est présenté de la manière suivante :

PB N°##. TITRE DU PROBLÈME **DIFFICULTÉ : X**

Le chiffre indiquant la difficulté varie en pratique entre 3 et $9^{1/2,}$ sur une échelle de 10. Cette valeur subjective indique plus les difficultés relatives des différents problèmes que leur difficulté dans l'absolu : tous les problèmes sont techniquement abordables pour un développeur C++ expérimenté.

Les problèmes n'ont pas nécessairement à être lus dans l'ordre, sauf dans le cas de certaines séries de problèmes (indiqués " 1re partie », « 2e partie »...) qu'il est profitable d'aborder d'un bloc.

Ce livre contient un certain nombre de recommandations, au sein desquelles les termes sont employés avec un sens bien précis :

- (employez) **systématiquement** : il est absolument nécessaire, indispensable, d'employer telle ou telle technique.
- **préférez** (l'emploi de): l'emploi de telle ou telle technique est l'option la plus usuelle et la plus souhaitable. N'en déviez que dans des cas bien particuliers lorsque le contexte le justifie.
- **prenez en considération** : l'emploi de telle ou telle technique ne s'impose pas, mais mérite de faire l'objet d'une réflexion.
- **évitez** (l'emploi) : telle ou telle technique n'est certainement pas la meilleure à employer ici, et peut même s'avérer dangereuse dans certains cas. Ne l'utilisez que dans des cas bien particuliers, lorsque le contexte le justifie.
- (n'employez) **jamais** : il est absolument nécessaire, crucial, de ne pas employer telle ou telle technique.

Comment est née l'idée de ce livre ?

L'origine de ce livre est la série « *Guru of the Week* », initialement créée dans le but de faire progresser les équipes de développements internes de PeerDirect en leur soumettant des problèmes pratiques pointus et riches en enseignements, abordant tant

1. Lippman S. and Lajoie J. *C++ Primer, Third Edition* (Addison Wesley Longman, 1998)
2. Meyer S. *Effective C++ CD : 85 Specific Ways to Improve Your Programs and Designs* (Addison Wesley Longman 1999). Voir aussi la démonstration en-ligne sur http://www.meyerscd.awl.com

l'utilisation de techniques C++ (emploi de l'héritage, robustesse aux exceptions), que les évolutions progressives de la norme C++. Forte de son succès, la série a été par la suite publiée sur le groupe de discussion Internet `comp.lang.c++.moderated`, sur lequel de nouveaux problèmes sont désormais régulièrement soumis.

Tirer parti au maximum du langage C++ est un souci permanent chez nous, à PeerDirect. Nous réalisons des systèmes de gestion de bases de données distribuées et de réplication, pour lesquels la fiabilité, la robustesse, la portabilité et la performance sont des contraintes majeures. Nos logiciels sont amenés à être portés sur divers compilateurs et systèmes d'exploitation, ils se doivent d'être parfaitement robustes en cas de défaillance de la base de données, d'interruption des communications réseau ou d'exceptions logicielles. De la petite base de données sur PalmOS ou Windows CE jusqu'aux gros serveurs de données utilisant Oracle, en passant par les serveurs de taille moyenne sous Windows NT, Linux et Solaris, tous ces systèmes doivent pouvoir être gérés à partir du même code source, près d'un demi-million de lignes de code, à maintenir et à faire évoluer. Un redoutable exercice de portabilité et de fiabilité. Ces contraintes, ce sont peut être les vôtres.

Cette préface est pour moi l'occasion de remercier les fidèles lecteurs de *Guru of the Week* pour tous les messages, commentaires, critiques et requêtes au sujet des problèmes publiés, qui me sont parvenus ces dernières années. Une de ces requêtes était la parution de ces problèmes sous forme d'un livre ; la voici exaucée, avec, au passage, de nombreuses améliorations et remaniements : tous les problèmes ont été révisés, mis en conformité avec la norme C++ désormais définitivement établie, et, pour certains d'entre eux, largement développés – le problème unique consacré à la gestion des exceptions est, par exemple, devenu ici une série de dix problèmes. En conclusion, les anciens lecteurs de *Guru of the Week*, trouverons ici un grand nombre de nouveautés.

J'espère sincèrement que ce livre vous permettra de parfaire votre connaissance des mécanismes du C++, pour le plus grand bénéfice de vos développements logiciels.

Remerciements

Je voudrais ici exprimer toute ma reconnaissance aux nombreux lecteurs enthousiastes de *Guru of the Week*, notamment pour leur participation active à la recherche du titre de ce livre. Je souhaiterais remercier tout particulièrement Marco Dalla Gasperina, pour avoir proposé *Enlightened C++* et Rob Steward pour avoir proposé *Practical C++ Problems et Solutions*. L'assemblage de ces deux suggestions ont conduit au titre final[1], à une petite modification près, en référence à l'accent particulier mis, dans ce livre, sur la gestion des exceptions.

Merci à Bjarne Stroustrup, responsable de la collection *C++ In-Depth Series*, ainsi qu'à Marina Lang, Debbie Lafferty, et à toute l'équipe éditoriale de Addison

1. NdT : Le titre original de ce livre est « *Exceptional C++ : 47 Engineering Puzzles , Programming problems, and Solutions* ».

Wesley Longman[1] pour leur enthousiasme, leur disponibilité permanente, et pour la gentillesse avec laquelle ils ont organisé la réception lors de la réunion du comité de normalisation à Santa Cruz.

Merci également à tous les relecteurs, parmi lesquels se trouvent d'éminents membres du comité de normalisation C++, pour leurs remarques pertinentes et les améliorations qu'ils m'ont permis d'apporter au texte : Bjarne Stroustrup, Scott Meyers, Andrei Alexandrescu, Steve Clamage, Steve Dewhurst, Cay Horstmann, Jim Hyslop, Brendan Kehoe et Dennis Mancl.

Merci, pour finir, à ma famille et à tous mes amis, pour leur soutien sans faille.

Herb Sutter

Juin 1999

1. NdT : Éditeur de l'édition originale.

Programmation générique avec la bibliothèque standard C++

Pour commencer, nous étudierons quelques sujets relatifs à la programmation générique en nous focalisant en particulier sur les divers éléments réutilisables (conteneurs, itérateurs, algorithmes) fournis par la bibliothèque standard C++.

Pb n° 1. Itérateurs — Difficulté : 7

Les itérateurs sont indissociables des conteneurs, dont ils permettent de manipuler les éléments. Leur fonctionnement peut néanmoins réserver quelques surprises.

Examinez le programme suivant. Il comporte un certain nombre d'erreurs dues à une mauvaise utilisation des itérateurs (au moins quatre). Pourrez-vous les identifier ?

```
int main()
{
  vector<Date> e;
  copy( istream_iterator<Date>( cin ),
        istream_iterator<Date>(),
        back_inserter( e ) );
  vector<Date>::iterator first =
        find( e.begin(), e.end(), "01/01/95" );
  vector<Date>::iterator last =
        find( e.begin(), e.end(), "31/12/95" );
  *last = "30/12/95";
```

```
  copy( first,
        last,
        ostream_iterator<Date>( cout, "\n" ) );
  e.insert( --e.end(), DateDuJour() );
  copy( first,
        last,
        ostream_iterator<Date>( cout, "\n" ) );
}
```

Solution

Examinons ligne par ligne le code proposé :

```
int main()
{
  vector<Date> e;
  copy( istream_iterator<Dathe>( cin ),
        istream_iterator<Date>(),
        back_inserter( e ) );
```

Pour l'instant, pas d'erreur. La fonction `copy()` effectue simplement la copie de dates dans le tableau `e`. On fait néanmoins l'hypothèse que l'auteur de la classe `Date` a fourni une fonction `operator>>(istream&,Date&)` pour assurer le bon fonctionnement de l'instruction `istream_iterator<Date>(cin)`, qui lit une date à partir du flux d'entrée `cin`.

```
  vector<Date>::iterator first =
        find( e.begin(), e.end(), "01/01/95" );
  vector<Date>::iterator last =
        find( e.begin(), e.end(), "31/12/95" );
  *last = "30/12/95";
```

Cette fois, il y a une erreur ! Si la fonction `find()` ne trouve pas la valeur demandée, elle renverra la valeur `e.end()`, l'itérateur `last` sera alors situé au-delà de la fin de tableau et l'instruction « `*last= "30/12/95"` » échouera.

```
  copy( first,
        last,
        ostream_iterator<Date>( cout, "\n" ) );
```

Nouvelle erreur ! Au vu des lignes précédentes, rien n'indique que `first` pointe vers une position située avant `last`, ce qui est pourtant une condition requise pour le bon fonctionnement de la fonction `copy()`. Pis encore, `first` et `last` peuvent tout à fait pointer vers des positions situées au-delà de la fin de tableau, dans le cas où les valeurs cherchées n'auraient pas été trouvées.

Certaines implémentations de la bibliothèque standard peuvent détecter ce genre de problèmes, mais dans la majorité des cas, il faut plutôt s'attendre à une erreur brutale lors de l'exécution de la fonction `copy()`.

```
  e.insert( --e.end(), DateDuJour() );
```

Cette ligne introduit deux erreurs supplémentaires.

La première est que si `vector<Date>::iterator` est de type `Date*` (ce qui est le cas dans de nombreuses implémentations de la bibliothèque standard), l'instruction `--e.end()` n'est pas autorisée, car elle tente d'effectuer la modification d'une variable temporaire de type prédéfini, ce qui est interdit en C++. Par exemple, le code suivant est illégal en C++ :

```
Date* f();              // f() est une fonction renvoyant une Date*
Date* p = --f();    // Erreur !
```

Cette première erreur peut être résolue si on écrit :

```
e.insert( e.end()-1, DateDuJour() );
```

La seconde erreur est que si le tableau `e` est vide, l'appel à `e.end()-1` échouera.

```
copy( first,
      last,
      ostream_iterator<Date>( cout, "\n" ) );
```

Erreur ! Les itérateurs `first` et `last` peuvent très bien ne plus être valides après l'opération d'insertion. En effet, les conteneurs sont réalloués « par à-coups » en fonction des besoins, lors de chaque opération d'insertion. Lors d'une réallocation, l'emplacement mémoire où sont stockés les objets contenus peut varier, ce qui a pour conséquence d'invalider tous les itérateurs faisant référence à la localisation précédente de ces objets. L'instruction `insert()` précédant cette instruction `copy()` peut donc potentiellement invalider les itérateurs `last` et `first`.

Recommandation

Assurez-vous de n'utiliser que des itérateurs valides.

En résumé, faites attention aux points suivants lorsque vous manipulez des itérateurs :

- *N'utilisez un itérateur que s'il pointe vers une position valide*. Par exemple, l'instruction « `*e.end()` » provoquera une erreur.
- *Assurez-vous que les itérateurs que vous utilisez n'ont pas été invalidés par une opération survenue auparavant*. Les opérations d'insertion, notamment, peuvent invalider des itérateurs existants si elles effectuent une réallocation du tableau contenant les objets.
- *Assurez-vous de transmettre des intervalles d'itérateurs valides aux fonctions qui le requièrent*. Par exemple, la fonction `find()` nécessite que le premier itérateur pointe vers une position située avant le second ; assurez-vous également, que les deux itérateurs pointent vers le même conteneur !
- *Ne tentez pas de modifier une valeur temporaire de type prédéfini*. En particulier, l'instruction « `--e.end()` » vue ci-dessus provoque une erreur si `vector<Date>::iterator` est de type pointeur (dans certaines implémentations de la bibliothèque standard, il se peut que l'itérateur soit de type objet et que la fonction `operator()--` ait été redéfi-

nie pour la classe correspondante, rendant ainsi la syntaxe « `--e.end()` » autorisée ; cependant, il vaut mieux l'éviter dans un souci de portabilité du code).

Pb n° 2. Chaînes insensibles à la casse (1re partie) Difficulté : 7

L'objet de ce problème est d'implémenter une classe chaîne « insensible à la casse », après avoir précisé ce que cela signifie.

1. Que signifie être « insensible à la casse » pour une chaîne de caractères ?
2. Implémentez une classe `ci_string` se comportant de manière analogue à la classe standard `std::string` mais qui soit insensible à la casse, comme l'est la fonction `stricmp()`[1]. Un objet `ci_string` doit pouvoir être utilisé de la manière suivante :

```
ci_string s( "AbCdE" );
// Chaîne insensible à la casse
//
assert( s == "abcde" );
assert( s == "ABCDE" );
//
// La casse originale doit être conservée en interne
assert( strcmp( s.c_str(), "AbCdE" ) == 0 );
assert( strcmp( s.c_str(), "abcde" ) != 0 );
```

3. Une classe de ce genre est-elle utile ?

Solution

1. *Que signifie être « insensible à la casse » pour une chaîne de caractères ?*

Être « insensible à la casse » signifie ne pas faire la différence entre majuscules et minuscules. Cette définition générale peut être modulée en fonction de la langue utilisée : par exemple, pour une langue utilisant des accents, être « insensible à la casse » peut éventuellement signifier ne pas faire de différence entre lettres accentuées ou non. Dans ce problème, nous nous cantonnerons à la première définition.

2. *Implémentez une classe `ci_string` se comportant de manière analogue à la classe standard `std::string` mais qui soit insensible à la casse, comme l'est la fonction `stricmp()`.*

1. La fonction `stricmp()` ne fait pas partie de la bibliothèque standard à proprement parler mais est fournie avec de nombreux compilateurs C et C++.

Commençons par un petit rappel sur la classe `string` de la bibliothèque standard. Un coup d'œil dans le fichier en-tête `<string>` nous permet de voir que `string` est en réalité un modèle de classe :

```
typedef basic_string<char> string;
```

Le modèle de classe `basic_string` est, pour sa part, déclaré de la manière suivante :

```
template<class charT,
         class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_string;
```

En conclusion, `string` signifie en fait :

```
basic_string<char, char_traits<char>, allocator<char>>
```

Autrement dit, `string` est une instanciation de `basic_string` avec le type `char` pour lequel on utilise les paramètres par défaut du modèle de classe. Parmi ces paramètres, le deuxième peut nous intéresser tout particulièrement : il permet en effet de spécifier la manière dont les caractères seront comparés.

Rentrons un peu dans le détail. Les capacités de comparaison fournies par `basic_string` sont en réalité fondées sur les fonctions correspondantes de `char_traits`, à savoir `eq()` et `lt()` pour les tests d'égalité et les comparaisons de type « est inférieur à » entre deux caractères ; `compare()` et `find()` pour la comparaison et la recherche de séquences de caractères.

Pour implémenter une chaîne insensible à la casse, le plus simple est donc d'implémenter notre propre classe dérivée de `char_traits` :

```
struct ci_char_traits : public char_traits<char>
               // On redéfinit les fonctions dont on souhaite
               // spécialiser le comportement
  {
   static bool eq( char c1, char c2 )
     { return toupper(c1) == toupper(c2); }

   static bool lt( char c1, char c2 )
     { return toupper(c1) <  toupper(c2); }

   static int compare( const char* s1,
                               const char* s2,
                               size_t n )
     { return memicmp( s1, s2, n ); }
           // memicmp n'est pas fournie pas tous les compilateurs
          // En cas d'absence, elle peut facilement être implémentée.

   static const char* find( const char* s, int n, char a )
   {
     while( n-- > 0 && toupper(*s) != toupper(a) )
     {
          ++s;
     }
     return n > 0 ? s : 0;
   }
  };
```

Ce qui nous amène finalement à la définition de notre classe insensible à la casse :

```
typedef basic_string<char, ci_char_traits> ci_string;
```

Il est intéressant de noter l'élégance de cette solution : nous avons réutilisé toutes les fonctionnalités du type `string` standard en remplaçant uniquement celles qui ne convenaient pas. C'est une bonne démonstration de la réelle *extensibilité* de la bibliothèque standard.

3. *Une classe de ce genre est-elle utile ?*

A priori, non. Il est en général plus clair de disposer d'une fonction de comparaison à laquelle on peut spécifier de prendre en compte ou non la casse, plutôt que deux types de chaîne distincts.

Considérons par exemple le code suivant :

```
string    a = "aaa";
ci_string b = "aAa";
if( a == b ) /* ... */
```

Quel doit être le résultat de la comparaison entre `a` et `b` ? Faut-il prendre en compte le caractère sensible à la casse de l'opérande de gauche et effectuer une comparaison sensible à la casse ? Faut-il au contraire faire prédominer l'opérande de droite et effectuer une comparaison insensible à la casse ? On pourrait certes, par convention, adopter une option ou l'autre. Mais cela ne ferait qu'éluder le problème sans le résoudre. Pour vous en convaincre, imaginez un troisième mode de comparaison implémenté par les chaînes de type `yz_string` :

```
typedef basic_string<char, yz_char_traits> yz_string;

ci_string b = "aAa";
yz_string c = "AAa";
if( b == c ) /* ... */
```

Quel doit être le résultat de la comparaison entre `b` et `c` ? Il apparaît ici clairement qu'il n'y aucune raison valable de préférer un mode de comparaison à l'autre.

Si en revanche, le mode de comparaison utilisé est spécifié par l'emploi d'une fonction particulière au lieu d'être lié au type des objets comparés, tout devient plus clair :

```
string a = "aaa";
string b = "aAa";
if( stricmp( a.c_str(), b.c_str() ) == 0 ) /* ... */
string c = "AAa";
if( EstEgalAuSensDeLaComparaisonYZ( b, c ) ) /* ... */
```

Dans la majorité des cas, il est donc préférable que le mode de comparaison dépende de la fonction utilisée plutôt que du type des objets comparés. Néanmoins, dans certains cas, il peut être utile d'implémenter une classe de manière à ce qu'elle puisse être comparée à des objets ou variables d'un autre type : par exemple, il est intéressant que les objets de type `string` puissent être comparés naturellement à des `char*`.

En résumé, ce problème a permis de mettre en lumière le fonctionnement interne de `basic_string` et d'indiquer une manière élégante et efficace de réutiliser ce modèle de classe, dans le cas où on souhaite personnaliser les fonctions des comparaison utilisées par une chaîne de caractère.

Pb n° 3. Chaînes insensibles à la casse (2e partie) Difficulté : 5

Indépendamment de son utilité pratique qui pourrait être discutée ailleurs, peut-on dire que la classe `ci_string` implémentée dans le problème précédent est fiable techniquement ?

Reprenons la déclaration de la classe `ci_string` vue dans le problème précédent :

```
struct ci_char_traits : public char_traits<char>
  {
    static bool eq( char c1, char c2 )   { /*...*/ }
    static bool lt( char c1, char c2 )   { /*...*/ }
    static int compare( const char* s1,
                        const char* s2,
                        size_t n )       { /*...*/ }
    static const char*
    find( const char* s, int n, char a ) { /*...*/ }
  };
```

1. Est-il techniquement fiable de faire dériver ainsi `ci_char_traits` de `char_traits <char>` ?
2. Le code suivant se compilera-t-il correctement ?

```
ci_string s = "abc";
cout << s << endl;
```

3. Est-il possible d'utiliser les opérateurs d'addition et d'affectation (`+`,`+=` et `=`) en mélangeant les types des opérandes (`string` et `ci_string`), comme le montre l'exemple ci-dessous?

```
string    a = "aaa";
ci_string b = "bbb";
string    c = a + b;
```

Solution

1. *Est-il techniquement fiable de faire dériver ainsi `ci_char_traits` de `char_traits<char>` ?*

Comme le précise le principe de substitution de Liskov[1], l'utilisation de l'héritage public doit normalement être réservée à l'implémentation d'une relation de type « EST-UN » ou « FONCTIONNE-COMME-UN » entre la classe de base et la classe dérivée. On peut donc considérer qu'une dérivation publique ne se justifie pas dans notre exemple, étant donné que les objets `ci_char_traits<char>` ne seront pas amenés à être utilisés de manière polymorphique à partir d'un pointeur de type `char_traits<char>`. La dérivation est donc utilisée ici plutôt par paresse et confort d'utilisation que par réel besoin.

Nathan Myers[2] précise avec raison que lorsque l'on instancie un modèle de classe avec un type donné, il faut s'assurer que ce type est compatible avec les exigences requises par le modèle : cette règle est plus connue sous le nom du « principe générique de substitution de Liskov[3] ».

Dans notre cas, nous devons donc nous assurer que la classe `ci_char_traits` répond bien aux exigences du modèle de classe `basic_string`, à savoir que le type passé en deuxième paramètre doit avoir la même interface publique que le modèle de classe `char_traits`.

Le fait que `ci_char_traits` dérive de `char_traits` nous assure que ce dernier point est vérifié et confirme donc la validité technique cette classe.

En résumé, l'utilisation de la dérivation est suffisante car elle nous assure le respect du « Principe Générique de Substitution de Liskov » ; elle n'est en revanche pas nécessaire car elle assure, en plus, le respect du « Principe de Substitution de Liskov » proprement dit, ce qui n'est en l'occurrence pas requis.

L'héritage a donc été utilisé ici par commodité. C'est d'autant plus flagrant que la classe `ci_char_traits` ne comporte que des membres statiques et que la classe `char_traits` ne peut pas être utilisée de manière polymorphique. Nous aurons l'occasion de revenir plus tard sur les raisons qui justifient l'emploi de l'héritage (problème n° 24).

2. *Le code suivant se compilera-t-il correctement ?*

```
ci_string s = "abc";
cout << s << endl;
```

Petite indication : il est spécifié dans la norme C++ [21.3.7.9, lib.string.io] que la déclaration de l'opérateur << pour la classe `basic_string` doit être la suivante :

```
template<class charT, class traits, class Allocator>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os,
           const basic_string<charT,traits,Allocator>& str);
```

Par ailleurs, rappelons que l'objet `cout` est de type `basic_ostream<char, char_traits<char>`.

1. Liskov Substitution Principle (LSP). Voir à ce sujet les problèmes n° 22 et 28.
2. Nathan Myers est membre de longue date de comité de normalisation C++. Il est, en particulier, l'auteur de la classe `locale` de la bibliothèque standard.
3. Generic Liskov Substitution Principle (GLSP).

À partir de ces éléments, il apparaît clairement que le code ci-dessus pose un problème : en effet, l'opérateur << est un modèle de fonction faisant lui même appel à deux autres modèles `basic_ostream` et `basic_string` auxquels il passe des paramètres. L'implémentation de `operator<<()`est telle que le deuxième paramètre (`traits`) est nécessairement le même pour chacun de ces deux modèles. Autrement dit, l'opérateur << qui permettra d'afficher un `ci_string` devra accepter une première opérande de type `basic_ostream<char,ci_char_traits>`, ce qui n'est malheureusement pas le type de `cout`. Le fait que `ci_char_traits` dérive de `char_traits` n'améliore en rien le problème.

Il y a deux moyens de résoudre le problème : définir votre propre opérateur << pour la classe `ci_string` (il faudrait alors, pour être cohérent, définir également l'opérateur >>) ou bien utiliser la fonction membre `c_str()` pour se ramener au cas classique d'`operator<<(const char*)` :

```
cout << s.c_str() << endl;
```

3. *Est-il possible d'utiliser les opérateurs d'addition et d'affectation (+,+= et =) en mélangeant les types des opérandes (`string` et `ci_string`), comme le montre l'exemple ci-dessous?*

```
string    a = "aaa";
ci_string b = "bbb";
string    c = a + b;
```

Là encore, la réponse est non. Pour s'en sortir, il faut redéfinir une fonction `operator+()` spécifique ou utiliser la fonction `c_str()` pour se ramener à l'utilisation de `operator+(const char*)`.

Pb n° 4. Conteneurs génériques réutilisables (1re partie) Difficulté : 8

Comment rendre un conteneur le plus générique possible ? Dans quelle mesure le fait d'avoir des membres de type paramétrable a t-il un impact sur le spectre des utilisations possibles d'un modèle de classe ?

Voici une déclaration possible pour une classe `fixed_vector`, similaire à la classe `vector` de la bibliothèque standard, permettant d'implémenter un tableau de taille fixe contenant des éléments de type paramétrable :

```
class fixed_vector
{
public:
  typedef T*       iterator;
  typedef const T* const_iterator;
  iterator       begin()       { return v_; }
  iterator       end()         { return v_+size; }
```

```
  const_iterator begin() const { return v_; }
  const_iterator end()   const { return v_+size; }

private:
  T v_[size];
};
```

Proposez une implémentation pour le constructeur de copie et l'opérateur d'affectation de cette classe (leurs déclarations ne sont pas mentionnées ici). Efforcez-vous de le rendre le plus générique possible : en particulier, tentez de réduire au minimum les contraintes imposées au type `T` contenu.

Concentrez-vous sur ces deux fonctions et ne tenez pas compte des autres imperfections de cette classe, qui n'est de toute façon pas parfaitement compatible avec les exigences de la bibliothèque standard.

Solution

Nous allons, pour une fois, adopter une technique un peu différente : nous proposons une implémentation possible pour le constructeur de copie et l'opérateur d'affectation de la classe `fixed_vector` dans l'énoncé du prochain problème. Vôtre rôle sera de la critiquer.

Pb n° 5. Conteneurs génériques réutilisables (2e partie) Difficulté : 6

Nous présentons donc ici la solution du problème précédent. Précisons que l'exemple de `fixed_vector` est inspiré d'une publication originale de Kevlin Henney, complétée plus tard par les analyses de Jon Jagger parues dans les numéros 12 à 20 du magazine « `Overload` ». Précisons aux lecteurs ayant eu connaissance du problème sous sa forme originale que nous présentons ici une solution légèrement différente (en particulier, les optimisations présentées dans le n° 20 de « Overload » ne fonctionneront pas avec notre solution).

Voici une implémentation possible pour le constructeur de copie et l'opérateur d'affectation de `fixed_vector` :

```
template<typename T, size_t size>
class fixed_vector
{
public:
  typedef T*       iterator;
  typedef const T* const_iterator;
  fixed_vector() { }
   template<typename O, size_t osize>
```

```
  fixed_vector( const fixed_vector<O,osize>& other )
  {
    copy( other.begin(),
          other.begin()+min(size,osize),
          begin() );
  }

  template<typename O, size_t osize>
  fixed_vector<T,size>&
  operator=( const fixed_vector<O,osize>& other )
  {
    copy( other.begin(),
          other.begin()+min(size,osize),
          begin() );
    return *this;
  }

  iterator       begin()       { return v_; }
  iterator       end()         { return v_+size; }
  const_iterator begin() const { return v_; }
  const_iterator end()   const { return v_+size; }

private:
  T v_[size];
};
```

Commentez ces implémentations. Présentent-elles des défauts ?

Solution

Nous allons analyser les fonctions présentées ci-dessus notamment du point de vue de leur réutilisabilité et des contraintes qu'elles imposent au type `T` contenu.

Constructeur de copie et opérateur d'affectation

Une remarque préliminaire s'impose : la classe `fixed_vector` n'a en réalité pas besoin d'un constructeur de copie et d'un opérateur d'affectation spécifique ; les fonctions fournies par défaut par le compilateur fonctionnent parfaitement !

La question posée était en quelque sorte un piège...

Néanmoins, il est vrai que ces fonctions par défaut limitent la réutilisabilité de la classe `fixed_vector` , notamment en présence de diverses instances contenant des types différents. L'objet de cette solution est donc de proposer l'implémentation d'un constructeur de copie et d'un opérateur d'affectation paramétrables, afin de rendre la classe `fixed_vector` plus souple d'utilisation.

```
fixed_vector( const fixed_vector<O,osize>& other )
{
  copy( other.begin(),
        other.begin()+min(size,osize),
        begin() );
}

template<typename O, size_t osize>
fixed_vector<T,size>&
operator=( const fixed_vector<O,osize>& other )
{
  copy( other.begin(),
        other.begin()+min(size,osize),
        begin() );
  return *this;
}
```

Notons tout de suite que les fonctions ci-dessus ne constituent pas à proprement parler un constructeur de copie et un opérateur d'affectation, car ce sont des modèles de fonctions membres pour lesquels les types passés en paramètre ne sont pas nécessairement du type de la classe :

```
struct X
{
  template<typename T>
  X( const T& );   // N'est PAS un constructeur de copie
                   // Il se peut que T ne soit pas X
  template<typename T>
  operator=( const T& );
                   // N'est PAS un opérateur d'affectation
                   // Il se peut que T ne soit pas X
};
```

Si le type `T` est remplacé par `X`, nous obtenons des fonctions ayant la *même signature* qu'un constructeur de copie et un opérateur d'affectation. Néanmoins, la présence de ces modèles de fonction ne dispense pas le compilateur d'implémenter un constructeur de copie et un opérateur d'affectation par défaut, comme le précise la section 12.8/2 (note 4) de la norme C++ :

> La présence d'un modèle de fonction membre ayant la même signature qu'un constructeur de copie ne supprime pas la déclaration implicite du constructeur de copie par défaut. Les modèles de constructeurs sont pris en compte par l'algorithme de résolution des noms pour les appels de fonctions ; on peut tout à fait les préférer à un constructeur normal si leur signature correspond mieux à la syntaxe de l'appel.

Un paragraphe similaire est consacré, un peu plus loin, à l'opérateur d'affectation (12.8/9 note 7). En pratique, les fonctions par défaut, déjà présentes dans l'implémentation originale, existent toujours dans notre nouvelle version de `fixed_vector` : nous ne les avons pas remplacées ; nous avons au contraire ajouté deux nouvelles fonctions.

Pour bien illustrer la différence entre ces quatre fonctions, considérons l'exemple suivant :

```
fixed_vector<char,4> v;
fixed_vector<int,4>  w;

fixed_vector<int,4>  w2(w);
// Appelle le constructeur de copie par défaut

fixed_vector<int,4>  w3(v);
// Appelle le modèle de constructeur de copie

w = w2;
// Appelle l'opérateur d'affectation par défaut

w = v;
// Appelle le modèle d'opérateur d'affectation
```

En résumé, nous avons implémenté ici des fonctions supplémentaires permettant de construire ou de réaliser une affectation vers un `fixed_vector` depuis un `fixed_vector` d'un autre type.

Spectre d'utilisation de la classe `fixed_vector`

Dans quelle mesure notre classe `fixed_vector` est-elle utilisable ?

Pour être utilisée dans de nombreux contextes, `fixed_vector` doit être performante sur deux points :

- Capacité à gérer des types hétérogènes

Il doit être possible de construire une instance de `fixed_vector` à partir d'un autre `fixed_vector` contenant des objets de type différent dans la mesure où le type de l'objet source est convertible dans le type de l'objet cible (idem pour l'affectation).

Autrement dit, il faut qu'il soit possible d'écrire :

```
fixed_vector<char,4> v;

fixed_vector<int,4>  w(v);
// Appel au modèle de constructeur de copie

w = v;
// Appel au modèle d'opérateur d'affectation

class B            { /*...*/ };
class D : public B { /*...*/ };

fixed_vector<D*,4> x;

fixed_vector<B*,4> y(x);
// Appel au modèle de constructeur de copie

 y = x;
// Appel au modèle d'opérateur d'affectation
```

Ceci fonctionne car une variable de type `D*` peut être affectée à une variable de type `B*`.

- Capacité à gérer des tableaux de taille différente.

Il doit être possible d'affecter la valeur d'un tableau à un autre tableau de taille différente, par exemple :

```
fixed_vector<char,6> v;
fixed_vector<int,4>  w(v);
// Initialise w en utilisant les 4 premières valeurs de v

w = v;
// Affecte à w les 4 valeurs de v

class B            { /*...*/ };
class D : public B { /*...*/ };

fixed_vector<D*,16> x;
fixed_vector<B*,42> y(x);
// Initialise y en utilisant le 16 premières valeurs de x

y = x;
// Affecte à y les 16 valeurs de x
```

Solution adoptée par la bibliothèque standard

De nombreux conteneurs de la bibliothèque standard fournissent des fonctions de copie et d'affectation similaires. Celles-ci sont néanmoins implémentées sous une forme légèrement différente, que nous exposons ici.

1. Constructeur de copie

```
template<class Iter>
fixed_vector( Iter first, Iter last )
{
  copy( first,
        first+min(size,(size_t)last-first),
        begin() );
}
```

`Iter` désigne un type d'itérateur.

Avec notre implémentation, nous écrivions :

```
fixed_vector<char,6> v;
fixed_vector<int,4>  w(v);
// Initialise w à partir des 4 premières valeurs de v
```

Avec la version de la bibliothèque standard, cela donne :

```
fixed_vector<char,6> v;
fixed_vector<int,4>  w(v.begin(), v.end());
// Initialise w à partir des 4 premières valeurs de v
```

Aucune de ces deux versions ne s'impose véritablement par rapport à l'autre : notre implémentation originale est plus simple d'emploi ; la deuxième version présente l'avantage d'être plus flexible (l'utilisateur peut choisir la plage des objets à copier).

2. Opérateur d'affectation

Il n'est pas possible de fournir une implémentation de l'opérateur d'affectation prenant en paramètre deux valeurs d'itérateurs : la fonction `operator()=` ne prend obligatoirement qu'un seul paramètre. La bibliothèque standard fournit ici une fonction nommée `assign` (affecter) :

```
template<class Iter>
fixed_vector<T,size>&
assign( Iter first, Iter last )
{
  copy( first,
        first+min(size,(size_t)last-first),
        begin() );
  return *this;
}
```

Avec notre implémentation, nous écrivions :

```
w = v;
// Copie les 4 premières valeurs de v dans w
```

La version de la bibliothèque standard ressemblerait à ceci :

```
w.assign(v.begin(),v.end()) ;
// Copie les 4 premières valeurs de v dans w
```

Remarquons qu'on pourrait techniquement parlant se passer de la fonction `assign()`. Ce ne serait qu'au prix d'une lourdeur d'écriture supplémentaire et d'une efficacité moindre :

```
w = fixed_vector<int,4>(v.begin(), v.end());
// Initialise v et copie les 4 premières valeurs de v dans w
```

Quelle implémentation préférer ? Celle de notre solution ou celle proposée par la bibliothèque standard ? L'argument de plus grande flexibilité invoqué pour le constructeur de copie ne tient plus ici. En effet, au lieu d'écrire :

```
w.assign( v.begin(), v.end() );
```

l'utilisateur peut tout à fait atteindre le même niveau de flexibilité en écrivant :

```
copy( v.begin(), v.begin()+4, w.begin() );
```

Il est donc préférable d'utiliser la solution proposée initialement plutôt que la fonction `assign()`, en se réservant la possibilité de recourir à `copy()` lorsque l'on souhaite affecter à un tableau cible un sous-ensemble d'un tableau source.

Pourquoi un constructeur par défaut explicite ?

La solution proposée fournit un constructeur par défaut explicite, qui fait a priori la même chose que le constructeur par défaut implicite du compilateur. Est-il nécessaire ?

La réponse est claire et nette : à partir du moment où nous déclarons un constructeur dans une classe (même s'il s'agit d'un modèle de constructeur), le compilateur cesse de générer un constructeur par défaut. Or, nous avons clairement besoin d'un constructeur par défaut pour `fixed_vector`, d'où l'implémentation explicite.

Un problème persiste...

La deuxième question posée de l'énoncé était : « ce code présente-il des défauts ? »

Malheureusement, oui : il risque de ne pas se comporter correctement en présence d'exceptions. Nous verrons plus loin en détail (problèmes n° 8 à 11) les différents niveaux de robustesse aux exceptions. Il y en a principalement deux : en présence d'une exception, une fonction doit correctement libérer toutes les ressources qu'elle se serait allouées (en zone mémoire dynamique) et elle doit se comporter d'une manière atomique (exécution totale ou exécution sans effet, les exécutions partielles étant exclues).

Notre opérateur d'affectation garantit le premier niveau de robustesse, mais pas le second. Reprenons le détail de l'implémentation :

```
template<typename O, size_t osize>
fixed_vector<T,size>&
operator=( const fixed_vector<O,osize>& other )
{
  copy( other.begin(),
        other.begin()+min(size,osize),
        begin() );
  return *this;
}
```

Si, au cours l'exécution de la fonction `copy()`, une opération d'affectation d'un des objets `T` échoue sur une exception, la fonction `operator=()` se terminera prématurément laissant l'objet `fixed_vector` dans un état incohérent, une partie seulement des objets contenus ayant été remplacée. En d'autres termes, la fonction ne se comporte pas d'une manière atomique.

Il n'y a malheureusement aucun moyen de remédier à ce problème avec l'implémentation actuelle de `fixed_vector`. En effet :

- Pour obtenir une fonction `operator=()` atomique, il faudrait normalement implémenter une fonction `Swap()` capable d'échanger les valeurs de deux objets `fixed_vector` sans générer d'exception, puis implémenter l'opérateur `=` de manière à ce qu'il réalise l'affectation sur un objet temporaire puis, en cas de réussite de l'opération, échange les valeurs de cet objet temporaire et de l'objet

principal. Cette technique du « valider ou annuler » sera étudiée en détail à l'occasion du problème n° 13.

- Or, il n'y a pas de moyen d'implémenter une fonction `Swap()` ne générant pas d'exception dans le cas de `fixed_vector`, cette classe comportant une variable membre de type tableau qu'il n'est pas possible de copier de manière atomique. Nous retrouvons, au passage, le problème original de notre fonction `operator()=`.

Il y a une solution pour s'en sortir : elle consiste à modifier l'implémentation interne de `fixed_vector` de manière à stocker les objets contenus dans un tableau alloué dynamiquement. Nous obtenons ainsi une robustesse forte aux exceptions, au prix, il est vrai, d'une légère perte d'efficacité due aux opérations d'allocation et de désallocation.

```
// Une version robuste aux exceptions
//
template<typename T, size_t size>
class fixed_vector
{
public:
  typedef T*          iterator;
  typedef const T* const_iterator;

  fixed_vector() : v_( new T[size] ) { }

  ~fixed_vector() { delete[] v_; }

  // Modèle de constructeur
  template<typename O, size_t osize>
  fixed_vector( const fixed_vector<O,osize>& other )
  : v (new_T[size])
  {
     try
     {
    copy( other.begin(),
          other.begin()+min(size,osize),
          begin() );
     }
     catch(...)
     {
        delete [] v_;
        throw;
     }
  }

  // Constructeur de copie explicite
  fixed_vector( const fixed_vector<T,size>& other )
  : v (new_T[size])
  {
     try
     {
    copy( other.begin(),
           other.begin()+min(size,osize),
```

```
            begin() );
      }
      catch(...)
      {
         delete [] v_;
         throw;
      }
   }


   void Swap( fixed_vector<T,size>& other ) throw()
   {
     swap( v_, other.v_ );
   }

   // Modèle d'opérateur d'affectation
   template<typename O, size_t osize>
   fixed_vector<T,size>&
   operator=( const fixed_vector<O,osize>& other )

   {
     fixed_vector<T,size> temp( other );
     Swap( temp );                        // Ne peut pas lancer
     return *this;                        // d'exception...
   }

   // Opérateur d'affectation explicite
   operator=( const fixed_vector<T,size>& other )

   {
     fixed_vector<T,size> temp( other );
     Swap( temp );                        // Ne peut pas lancer
     return *this;                        // d'exception...
   }

   iterator       begin()       { return v_; }
   iterator       end()         { return v_+size; }
   const_iterator begin() const { return v_; }
   const_iterator end()   const { return v_+size; }

private:
  T* v_;
};
```

Erreur à éviter

Ne considérez pas la gestion des exceptions comme un détail d'implémentation. C'est au contraire un élément primordial à prendre en compte dès la conception de vos programmes.

En conclusion, cet problème a permis de démontrer l'utilité pratique des membres paramétrables des modèles de classe. Bien qu'ils ne soient pas, à l'heure actuelle, pris en charge par tous les compilateurs – cela ne saurait tarder, puisqu'ils font maintenant

partie de la norme C++ standard – les membres paramétrables permettent très souvent d'élargir le spectre d'utilisation possible des modèles de classe.

Pb n° 6. Objets temporaires — Difficulté : 5

Les compilateurs C++ ont, dans de nombreuses situations, recours à des objets temporaires. Ceux-ci peuvent dégrader les performances d'un programme, voire poser des problèmes plus graves s'ils ne sont pas maîtrisés par le développeur. Êtes-vous capable d'identifier tous les objets temporaires qui seront créés lors de l'exécution d'un programme ? Lesquels peut-on éviter ? Nous apporterons ici des réponses à ces questions ; le problème suivant, quant à lui, s'intéressera à l'utilisation des objets temporaires par la bibliothèque standard.

Examinez le code suivant :

```
string TrouverAdresse( list<Employe> emp, string nom )
{
  for( list<Employe>::iterator i = emp.begin();
       i != emp.end();
       i++ )
  {
    if( *i == nom )
    {
      return i->adresse;
    }
  }
  return "";
}
```

L'auteur de ces lignes provoque l'utilisation d'au moins trois objets temporaires. Pouvez-vous les identifier ? Les supprimer ?

Note : ne modifiez pas la structure générale de la fonction, bien que celle-ci puisse en effet être améliorée.

Solution

Aussi surprenant que cela puisse paraître, cette fonction provoque l'utilisation de pas moins de sept objets temporaires !

Cinq d'entre eux pourraient facilement être évités (trois sont faciles à repérer, les deux autres le sont moins). Les deux derniers sont inhérents à la structure de la fonction et ne peuvent pas être supprimés.

Commençons par les deux les plus faciles :

```
string TrouverAdresse( list<Employe> emp, string nom )
```

Au lieu d'être passés par valeur, ces deux paramètres devraient clairement être passés par références constantes (respectivement `const list<Employe>&` et `const string&`), un passage par valeur provoquant la création de deux objets temporaires pénalisants en terme de performance et tout à fait inutiles.

Recommandation

Passez les objets par référence constante (`const&`) plutôt que par valeur.

Le troisième objet temporaire est créé dans la condition de terminaison de la boucle :

```
for( /*...*/ ; i != emp.end(); /*...*/ )
```

La fonction `end()` du conteneur `list` renvoie une valeur (c'est d'ailleurs le cas pour la majorité des conteneurs standards). Par conséquent, un objet temporaire est créé et comparé à `i` lors de chaque boucle. C'est parfaitement inefficace et inutile, d'autant plus que le contenu de `emp` ne varie pas pendant l'exécution de la boucle : il serait donc préférable de stocker la valeur de `emp.end()` dans une variable locale avant le début de la boucle et d'utiliser cette variable dans l'expression de la condition de fin.

Recommandation

Ne recréez pas plusieurs fois inutilement un objet dont la valeur ne change pas. Stockez-le plutôt dans une variable locale que vous réutiliserez.

Passons maintenant à un cas plus difficile à repérer :

```
for( /*...*/ ; i++ )
```

L'emploi de l'opérateur de post-incrémentation provoque l'utilisation d'un objet temporaire. En effet, contrairement à l'opérateur de pré-incrémentation, l'opérateur de post-incrémentation doit mémoriser dans une variable temporaire la valeur « avant incrémentation », afin de pouvoir la retourner à l'appelant :

```
const T T::operator++(int)()
{
  T old( *this ); // Mémorisation de la valeur originale
   ++*this;

  // Toujours implémenter la post-incrémentation
  // en fonction de la pré-incrémentation.

  return old;     // Renvoi de la valeur originale
}
```

Il apparaît ici clairement que l'opérateur de post-incrémentation est moins efficace que l'opérateur de pré-incrémentation : le premier fait non seulement appel au second, mais doit également stocker et renvoyer la valeur originale.

Recommandation

Afin d'éviter tout risque de divergence dans votre code, implémentez systématiquement l'opérateur de post-incrémentation en fonction de l'opérateur de pré-incrémentation.

Dans notre exemple, la valeur originale de `i` avant incrémentation n'est pas utilisée : il n'y a donc aucune raison d'utiliser la post-incrémentation plutôt que la pré-incrémentation. Signalons, au passage, que de nombreux compilateurs remplacent souvent de manière implicite et à titre d'optimisation, la post-incrémentation par une pré-incrémentation, lorsque cela est possible (voir plus loin le paragraphe consacré à ce sujet).

Recommandation

Réservez l'emploi de la post-incrémentation aux cas où vous avez besoin de récupérer la valeur originale de la variable avant incrémentation. Dans tous les autres cas, préférez la pré-incrémentation.

```
if( *i == nom )
```

Cette instruction nous indique que la classe `Employe` dispose soit d'un opérateur de conversion vers le type `string`, soit d'un constructeur de conversion prenant une variable de type `string` en paramètre. Dans un cas comme dans l'autre, ceci provoque la création d'un objet temporaire afin de permettre l'appel d'une fonction `operator==()` comparant des `strings` (si `Employe` a un opérateur de conversion) ou des `Employes` (si `Employe` a un constructeur de conversion).

Notons que le recours à cet objet temporaire peut être évité si on fournit une fonction `operator==()` prenant un opérande de type `string` et un autre opérande de type `Employe` (solution peu élégante) ou bien si `Employe` implémente une conversion vers une référence de type `string&` (solution préférable).

Recommandation

Prenez garde aux objets temporaires créés lors des conversions implicites. Pour les éviter au maximum, évitez de doter vos classes d'opérateurs de conversion et spécifiez l'attribut `explicit` pour les constructeurs susceptibles de réaliser des conversions.

```
    return i->addr;
// (ou)
    return "";
```

Ces instructions `return` provoquent chacune la création d'un objet temporaire de type `string`.

Il serait techniquement possible d'éviter la création de ces objets en ayant recours à la création d'une variable locale :

```
string ret; // Par défaut, vaut “”
```

```
  if( *i == nom )
  {
    ret = i->adresse;
  }

  return ret;
```

Seulement, est-ce intéressant en terme de performance ?

Si cette deuxième version peut paraître plus claire, elle n'est pas nécessairement plus efficace à la compilation : ceci dépend du compilateur que vous utilisez.

Dans le cas où l'employé serait trouvé et son adresse renvoyée, nous avons, dans la première version, une construction « de copie » d'un objet `string` et, dans la seconde version, une construction par défaut suivi d'une affectation.

En pratique, la première version « deux `return` » s'avère plus rapide[1] que la version « variable locale ». Il n'est donc ici pas souhaitable de supprimer les objets temporaires.

```
string TrouverAdresse( /* ... */)
```

L'emploi d'un retour par valeur provoque la création d'un objet temporaire. Ce dernier pourrait, estimez-vous, être supprimé grâce à l'emploi d'un type référence (`string&`) en valeur de retour... Erreur ! Ceci signifierait, dans notre exemple, renvoyer une référence vers une variable locale à la fonction ! À l'issue de l'exécution de `TrouverAdresse`, cette référence ne serait plus valide et son utilisation provoquerait immanquablement une erreur à l'exécution.

Recommandation

Ne renvoyez jamais une référence vers une variable locale à une fonction !

Pour être honnête, il y a une technique possible permettant de renvoyer une référence valide et d'éviter, par là-même, la création d'un objet temporaire. Cela consiste à avoir recours une variable locale statique :

```
const string&
TrouveAdresse( /* emp et nom passés par référence */ )
{
  for( /* ... */ )
  {
    if( i->nom==nom )
    {
      return i->adresse;
    }
  }
  static const string vide;
  return vide;
}
```

1. Des tests effectués sur un compilateur très répandu du marché ont prouvé que la première version est de 5 % à 40 % plus rapide (en fonction du degré d'optimisation)

Si l'employé est trouvé, on renvoie une référence vers une variable `string` membre d'un objet `Employe` contenu dans la liste. Cette solution n'est ni très élégante ni très sûre, car elle repose sur le fait que l'appelant soit bien au courant de la nature et de la durée de vie de l'objet référence. Par exemple, le code suivant provoquera une erreur :

```
string& a = TrouverAdresse( emp, "Jean Dupont" );
emp.clear();
cout << a; // Erreur !
```

L'emploi d'une référence non valide ne provoque pas systématiquement une erreur à l'exécution : cela dépend du contexte du programme, de votre chance... ceci rend ce type de bogue d'autant plus difficile à diagnostiquer.

Une erreur de ce type couramment répandue est l'utilisation d'itérateurs invalidés par une opération effectuée sur le conteneur qu'ils permettent de manipuler (voir le problème n° 1 à ce sujet).

Voici, pour finir, une nouvelle implémentation de la fonction `TrouverAdresse`, dans laquelle tous les objets temporaires superflus ont été supprimés (d'autres optimisations auraient été possibles, on ne s'y intéressera pas dans ce problème). Remarquons que comme `list<Employe>` est dorénavant passé en paramètre constant, il faut utiliser des itérateurs constants.

```
string TrouverAdresse( const list<Employee>& emp,
                       const string&         nom )
{
  list<Employe>::const_iterator end( emp.end() );
  for( list<Employe>::const_iterator i = emp.begin();
       i != end;
       ++i )
  {
    if( i->nom == nom )
    {
      return i->adresse;
    }
  }
  return "";
}
```

Optimisation de la post-incrémentation par les compilateurs

Lorsque que vous employez un opérateur de post-incrémentation sans utiliser la valeur originale, vous perdez en efficacité par rapport à l'emploi d'une simple pré-incrémentation.

Le compilateur est-il autorisé, à titre d'optimisation, à remplacer une post-incrémentation non justifiée par une pré-incrémentation ?

La réponse est en général non, sauf dans certains cas bien précis, comme les types standard prédéfinis `int` et `complex` que le compilateur peut traiter d'une manière spécifique.

Pour les types non prédéfinis, le compilateur n'est par défaut pas autorisé à effectuer ce type d'optimisation, car il ne maîtrise pas a priori la syntaxe d'utilisation d'une classe implémentée par un développeur. À la limite, rien ne permet de présumer du fait que les opérateurs de post-incrémentation et pré-incrémentation réalisent la même opération, à la valeur retournée près (bien qu'évidemment, il soit plus que souhaitable que cette situation totalement incohérente soit évitée, sous peine de rendre très dangereuse l'utilisation de la classe en question).

Il y a néanmoins une solution pour forcer l'optimisation : elle consiste à implémenter en-ligne (`inline`) l'opérateur de post-incrémentation (lequel doit, rappelons-le, faire appel à l'opérateur de pré-incrémentation). Ceci aura pour effet de rendre visibles les objets temporaires dans le code appelant, permettant ainsi au compilateur de les supprimer dans le cadre classique des optimisations.

Cette dernière solution n'est pas recommandée, l'emploi de fonctions en-ligne n'étant jamais idéal. La meilleure option consiste évidemment à prendre l'habitude d'utiliser systématiquement la pré-incrémentation lorsque la récupération de la valeur originale n'est pas requise.

Pb n° 7. Algorithmes standards — Difficulté : 5

La capacité à réutiliser l'existant fait partie des qualités requises pour un bon développeur. La bibliothèque standard regorge de fonctionnalités très utiles, qui ne sont malheureusement pas assez souvent exploitées. Pour preuve, nous allons voir comment il est possible d'améliorer le programme du problème précédent en réutilisant un algorithme existant.

Reprenons le code du problème précédent :

```
string TrouverAdresse( list<Employe> emp, string nom )
{
  for( list<Employe>::iterator i = emp.begin();
       i != emp.end();
       i++ )
  {
    if( *i == nom )
    {
      return i->adresse;
    }
  }
  return "";
}
```

Peut-on simplifier cette fonction en faisant appel à des éléments de la bibliothèque standard ? Quels avantages peut-on retirer de cette modification ?

Solution

L'emploi de l'algorithme standard `find()` permet d'éliminer la boucle de parcours des éléments (il aurait également été possible d'utiliser la fonction `find_if`) :

```
string FindAddr( list<Employee> emps, string name )
{
  list<Employee>::iterator i(
    find( emps.begin(), emps.end(), name )
    );
  if( i != emps.end() )
  {
    return i->addr;
  }
  return "";
}
```

Cette implémentation est plus simple et plus efficace que l'implémentation originale.

Recommandation

Réutilisez le code existant – surtout celui de la bibliothèque standard. C'est plus rapide, plus facile et plus sûr.

On a toujours intérêt à réutiliser les fonctionnalités de la bibliothèque standard plutôt que de perdre du temps à réécrire des algorithmes ou des classes existantes. Le code de la bibliothèque standard a toutes les chances d'être bien plus optimisé que le nôtre et de comporter moins d'erreurs – en effet il a été développé depuis longtemps et déjà utilisé par un grand nombre de développeurs.

En combinant l'emploi de `find()` et la suppression des objets temporaires vue lors du problème précédent, nous obtenons une fonction largement optimisée :

```
string TrouverAdresse( const list<Employe>& emp,
                       const string&        nom )
{
    list<Employee>::const_iterator i(
    find( emp.begin(), emp.end(), nom )
    );
  if( i != emp.end() )
  {
    return i->adresse;
  }
  return "";
}
```

Gestion des exceptions

Commençons par un bref historique des publications qui ont inspiré ce chapitre.

En 1994, Tom Cargill publia un article intitulé « Gestion des exceptions : une fausse impression de sécurité » (Cargill94)[1], dans lequel il présenta plusieurs exemples de code pouvant se comporter de manière incorrecte en présence d'exceptions. Il soumit également à ses lecteurs un certain nombre de problèmes, dont certains restèrent sans solution valable pendant plus de trois ans, mettant ainsi en évidence le fait qu'à l'époque, la communauté C++ maîtrisait imparfaitement la gestion des exceptions.

Il fallut attendre 1997 et la publication de « *Guru of the Week n° 8* » sur le groupe de discussion Internet *comp.lang.c++.moderated* pour que soit enfin fournie une solution complète au problème de Cargill. Cet article, qui eut un certain retentissement, fit l'objet, un an plus tard, d'une seconde parution dans les numéros de septembre, novembre et décembre 1998 de « *C++ Report* ». Cette seconde version, adaptée pour être conforme aux dernières évolutions du standard C++, ne présentait pas moins de trois solutions complètes au problème initial (tous ces articles seront repris prochainement dans un ouvrage à paraître prochainement : *C++ Gems II* [Martin00]).

Au début de l'année 1999, Scott Meyers proposa dans « *Effective C++ CD* » (Meyers99) une version retravaillée du problème original de Cargill, enrichie d'articles issus de ses autres ouvrages *Effective C++* et *More Effective C++*.

Nous présentons dans ce chapitre une synthèse de toutes ces publications, enrichies notamment par Dave Abrahams et Greg Colvin qui sont, avec Matt Austern, les auteurs de deux rapports soumis au Comité de Normalisation C++ ayant conduit à la nouvelle version de la bibliothèque standard, mieux adaptée à la gestion des exceptions.

1. Disponible sur Internet à l'adresse http://cseng.awl.com/bookdetail.qry?ISBN=0-201-63371-X&ptype=636.

Pb n° 8. Écrire du code robuste aux exceptions (1re partie) Difficulté : 7

La gestion des exceptions est, avec l'utilisation des modèles, l'une des fonctionnalités les plus puissantes du C++ ; c'est aussi l'une des plus difficiles à maîtriser, notamment dans le contexte de modèles de classe ou de fonction, où le développeur ne connaît à l'avance ni les types manipulés, ni les exceptions qui sont susceptibles de se produire.

Dans ce problème, nous étudierons, par l'intermédiaire d'un exemple mettant en oeuvre exceptions et modèles de classe, les techniques permettant d'écrire du code se comportant correctement en présence d'exceptions. Nous verrons également comment réaliser des conteneurs propageant correctement toutes les exceptions vers l'appelant, ce qui est plus facile à dire qu'à faire.

Nous repartirons de l'exemple initial soumis par Cargill : un conteneur de type « Pile » (`Stack`) dans lequel l'utilisateur peut ajouter (`Push`) ou retirer (`Pop`) des éléments. Au fur et à mesure de l'avancement du chapitre, nous ferons évoluer progressivement ce conteneur, le rendant de plus en plus apte à gérer correctement les exceptions, en diminuant progressivement les contraintes imposées sur le type `T` contenu. Nous aborderons notamment le point particulier des zones mémoires allouées dynamiquement, particulièrement sensibles aux exceptions.

Ceci nous permettra, au passage, de répondre aux questions suivantes :

- Quels sont les différents degrés de qualité possibles dans la gestion des exceptions ?
- Un conteneur générique peut-il (et doit-il) propager toutes les exceptions lancées par le type contenu vers le code appelant ?
- Les conteneurs de la bibliothèque standard se comportent-ils correctement en présence d'exceptions ?
- Le fait de rendre un conteneur capable de gérer correctement les exceptions a-t-il un impact sur son interface publique ?
- Les conteneurs génériques doivent-ils utiliser des spécificateurs d'exception (`throw`) au niveau de leur interface ?

Nous présentons ci-dessous la déclaration du conteneur `Stack` tel qu'il a été initialement proposé par Cargill. Le but du problème est de voir s'il se comporte correctement en présence d'exceptions ; autrement dit, de s'assurer qu'un objet `Stack` reste toujours dans un état valide et cohérent, quelles que soient les exceptions générées par ses fonctions membres, et que ces exceptions sont correctement propagées au code appelant – le seul capable de les gérer étant donné que la définition du type contenu (`T`) n'est pas connue au moment de l'implémentation de `Stack`.

```
template <class T> class Stack
{
public:
  Stack();
  ~Stack();
```

```
  /*...*/
private:
  T*     v_;      // Pointeur vers une zone mémoire
                  // allouée dynamiquement
  size_t vsize_;  // Taille totale de la zone mémoire
  size_t vused_;  // Taille actuellement utilisée
};
```

Implémentez le constructeur par défaut et le destructeur de `Stack`, en vous assurant qu'ils se comportent correctement en présence d'exceptions, en respectant les contraintes énoncées plus haut.

Solution

Il est clair que le point le plus sensible concernant la classe `Stack` est la gestion de la zone mémoire allouée dynamiquement. Nous devons absolument nous assurer qu'elle sera correctement libérée si une exception se produit. Pour l'instant, nous considérons que les allocations / désallocations de cette zone sont gérées directement depuis les fonctions membres de `Stack`. Dans un second temps, nous verrons une autre implémentation possible, faisant appel à une classe de base privée.

Constructeur par défaut

Voici une proposition d'implémentation pour le constructeur par défaut :

```
// Ce constructeur se comportera-t-il correctement
// en présence d'exceptions ?

template<class T>
Stack<T>::Stack()
 : v_(0),
   vsize_(10),
   vused_(0)           // Au départ, rien n'est utilisé
{
  v_ = new T[vsize_];  // Allocation initiale
}
```

Ce constructeur se comportera-t-il correctement en présence d'exceptions ? Pour le savoir, identifions les fonctions susceptibles de lancer des exceptions et voyons ce qui se passerait dans ce cas-là ; toutes les fonctions doivent être prises en compte : fonctions globales, constructeurs, destructeurs, opérateurs et autres fonctions membres.

Le constructeur de `Stack` affecte la valeur `10` à la variable membre `vsize_`, puis alloue dynamiquement un tableau de `vsize_` objets de type `T`. L'instruction « `new T[vsize_]` » appelle l'opérateur global `::new` (ou l'opérateur `new` redéfini par la classe `T`, s'il y en existe un) et le constructeur de `T`, et ceci autant de fois que nécessaire (10 fois, en l'occurrence). Chacun de ces opérateurs peut échouer : d'une part, l'opé-

rateur `new` peut lancer une exception `bad_alloc`, d'autre part, le constructeur de `T` peut lancer n'importe quelle exception. Néanmoins, dans les deux cas, nous sommes assurés que la mémoire allouée sera correctement désallouée par un appel à l'opérateur `delete[]` adéquat, et que, par conséquent, l'implémentation proposée ci-dessus se comportera parfaitement en présence d'exceptions.

Ceci vous paraît un peu rapide ? Rentrons un petit peu dans le détail :

1. Toutes les exceptions éventuelles sont correctement transmises à l'appelant. Dans ce constructeur, nous n' interceptons aucune exception (pas de bloc `catch`) : nous sommes donc assurés que si l'instruction « `new T[vsize_]` » génère une exception, celle-ci sera correctement propagée au code appelant.

Recommandation

Une fonction ne doit pas bloquer une exception : elle doit obligatoirement la traiter et/ou la transmettre à l'appelant.

2. Il n'y a pas de risque de fuites mémoires. Détaillons en effet ce qui se passe en cas de génération d'exception : si l'opérateur `new()` lance une exception `bad_alloc`, comme le tableau d'objets `T` n'a pas encore été alloué, il n'y a donc pas de risque de fuite ; si la construction d'un des objets `T` alloué échoue, le destructeur de `Stack` est automatiquement appelé (du fait qu'une exception a été générée au cours de l'exécution de `Stack::Stack`), ce qui a pour effet d'exécuter l'instruction `delete[]`, laquelle effectue correctement la destruction et la désallocation des objets `T` ayant déjà été alloués.

On fait ici l'hypothèse que le destructeur de `T` ne lance pas d'exceptions, ce qui aurait pour effet catastrophique d'appeler la fonction `terminate()` en laissant toutes les ressources allouées. Nous argumenterons cette hypothèse lors du problème n° 16 « Les dangers des destructeurs lançant des exceptions ».

3. Les objets restent toujours dans un état cohérent, même en cas d'exceptions. Certains pourraient arguer que si une exception se produit lors de l'allocation du tableau dans `Stack::Stack()`, la variable membre `vsize_` se retrouve initialisée avec une valeur de `10` alors que le tableau correspondant n'existe pas, et que, donc, nous nous retrouvons dans un état incohérent. En réalité, cet argument ne tient pas car la situation décrite ci-dessus ne peut pas se produire : en effet, à partir du moment où une exception se produit dans le constructeur d'un objet, cet objet est automatiquement détruit et peut donc être considéré comme « mort-né ». Il n'y a donc aucun risque de se retrouver avec un objet `Stack` existant dans un état incohérent.

Recommandation

Assurez-vous que votre code se comporte correctement en présence d'exceptions. En particulier, organisez votre code de manière à désallouer correctement les objets et à laisser les données dans un état cohérent, même en présence d'exceptions.

Pour finir, signalons qu'il y a une autre manière, plus élégante, d'écrire le même constructeur :

```
template<class T>
Stack<T>::Stack()
  : v_(new T[10]),  // Allocation initiale
    vsize_(10),
    vused_(0)       // Au départ, rien n'est utilisé
{
}
```

Cette deuxième version, équivalente à la première en terme de fonctionnalités, est préférable car elle initialise tous les membres dans la liste d'initialisation du constructeur, ce qui est une pratique recommandable.

Destructeur

L'implémentation du destructeur est facile à partir du moment où nous faisons une hypothèse simple :

```
template<class T>
Stack<T>::~Stack()
{
  delete[] v_;      // Ne peut pas lancer d'exception
}
```

Pour quelle raison « `delete[] v` » ne risque-t-elle pas de lancer d'exception ? Cette instruction appelle `T::~T` pour chacun des objets du tableau, puis appelle l'opérateur `delete[]()` pour désallouer la mémoire.

La norme C++ indique qu'un opérateur `delete[]()` ne peut pas lancer d'exceptions, comme le confirme la spécification des prototypes autorisés pour cette fonction [1]:

```
void operator delete[]( void* ) throw();
void operator delete[]( void*, size_t ) throw();
```

Par conséquent, la seule fonction susceptible de lancer une exception est le destructeur de `T`. Or, nous avons justement fait précédemment l'hypothèse que cette fonction `T::~T()` ne lançait pas d'exceptions. Nous aurons l'occasion de démontrer, plus loin dans ce chapitre, que ce n'est pas une hypothèse déraisonnable. Cela ne constitue pas, en tous cas, une contrainte trop forte sur `T`. Nous allons simplement admettre dans un premier temps, qu'il ne serait pas possible de réaliser un programme correct allouant et désallouant dynamiquement des tableaux d'objets si le destructeur

1. Techniquement parlant, rien de ne vous empêche d'implémenter un opérateur `delete[]` susceptible de lancer des exceptions ; ce serait néanmoins extrêmement dommageable à la qualité de vos programmes.

des objets alloués est susceptible de lancer des exceptions[1]. Nous en détaillerons les raisons plus loin dans ce chapitre.

Recommandation

Assurez-vous que tous les destructeurs et les opérateurs `delete()` (ou `delete[]()`) que vous implémentez ne laissent pas remonter d'exceptions ; ils ne doivent pas générer d'exception eux-mêmes ni laisser remonter une exception reçue d'un niveau inférieur.

Pb n° 9. Écrire du code robuste aux exceptions (2e partie) Difficulté : 8

Les cas du constructeur et du destructeur de `Stack()` étant réglés, nous passons, dans ce problème, au constructeur de copie et à l'opérateur d'affectation, pour lesquels l'implémentation sera légèrement plus complexe à réaliser.

Repartons de l'exemple du problème précédent :

```
template <class T> class Stack
{
public:
  Stack();
  ~Stack();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
   /*...*/

private:
  T*     v_;      // Pointeur vers une zone mémoire
                  // allouée dynamiquement
  size_t vsize_;  // Taille totale de la zone mémoire
  size_t vused_;  // Taille actuellement utilisée
};
```

Implémentez le constructeur de copie et l'opérateur d'affectation de `Stack`, en vous assurant qu'ils se comportent correctement en présence d'exceptions. Veillez notamment à ce qu'ils propagent toutes les exceptions reçues vers le code appelant et laissent, quoi qu'il arrive, l'objet `Stack` dans un état cohérent.

1. C'est de toute façon une bonne habitude de programmation de ne pas lancer des exceptions depuis un destructeur ; l'idéal étant d'adjoindre la spécification `throw()`à chaque destructeur implémenté.

Solution

Nous fonderons l'implémentation du constructeur de copie et de l'opérateur d'affectation sur une seule et même fonction utilitaire `NewCopy()`, capable de réaliser une copie (et éventuellement, au passage, une réallocation avec augmentation de taille) d'un tableau dynamique d'objets `T`. Cette fonction prend en paramètre un pointeur vers un tableau existant (`src`), la taille du tableau existant (`srcsize`) et du tableau cible (`destsize`) ; elle renvoie à l'appelant un pointeur vers le tableau nouvellement alloué (dont l'appelant garde désormais la responsabilité). Si une exception se produit au cours de l'exécution de `NewCopy`, toutes les zones mémoires temporaires sont correctement désallouées et l'exception est correctement propagée à l'appelant.

```
template<class T>
T* NewCopy( const T* src,
            size_t   srcsize,
            size_t   destsize )
{
  assert( destsize >= srcsize );
  T* dest = new T[destsize];
  try
  {
    copy( src, src+srcsize, dest );
  }
  catch(...)
  {
    delete[] dest; // Ne peut pas lancer d'exception
    throw;         // Relance l'exception originale
  }
  return dest;
}
```

Analysons cette fonction :

1. La première source potentielle d'exception est l'instruction « `new T[destsize]` » : une exception de type `bad_alloc` peut se produire lors de l'appel à `new` ou bien une exception de type quelconque peut se produire lors de l'appel du constructeur de `T` ; dans les deux cas, rien n'est alloué et la fonction se termine en ne laissant aucune fuite mémoire et en propageant correctement les exceptions au niveau supérieur.

2. La deuxième source potentielle d'exception est la fonction `T::operator=()`, appelée par la fonction `copy()` : si elle génère une exception, celle-ci sera interceptée puis relancée par le bloc `catch{}`, lequel détruit au passage le tableau `dest` précédemment alloué ; ceci assure un comportement correct (pas de fuite mémoire, exceptions propagées à l'appelant). Nous faisons ici néanmoins l'hypothèse importante que la fonction `T::operator=()` est implémentée de telle sorte qu'en cas de génération d'exception, l'objet cible (`*dest`, dans notre cas) puisse être détruit sans dommage (autrement dit, qu'il n'ait pas été déjà partiellement détruit, ce qui provoquerait une erreur à l'exécution de « `delete[] dest` »)[1].

3. Si l'allocation et la copie ont réussi, le pointeur du tableau cible est renvoyé à l'appelant (qui en devient responsable) par l'instruction « `return dest` », qui ne peut pas générer d'exception (copie d'une valeur de pointeur).

Constructeur de copie

Nous obtenons ainsi facilement une implémentation du constructeur de copie de `Stack`, basée sur `NewCopy()` :

```
template<class T>
Stack<T>::Stack( const Stack<T>& other )
  : v_(NewCopy( other.v,
                other.vsize,
                other.vsize)),
    vsize_(other.vsize_),
    vused_(other.vused_)
{
}
```

La seule source potentielle d'exceptions est `NewCopy`, pour laquelle on vient de voir qu'elle les gère correctement.

Opérateur d'affectation

Passons maintenant à l'opérateur d'affectation :

```
template<class T>
Stack<T>&
Stack<T>::operator=( const Stack<T>& other )
{
  if( this != &other )
  {
    T* v_new = NewCopy( other.v_,
                        other.vsize_,
                        other.vsize_ );
    delete[] v_;  // Ne peut pas lancer d'exception
    v_ = v_new;   // Prise de contrôle du nouveau tableau
    vsize_ = other.vsize_;
    vused_ = other.vused_;
  }
  return *this;   // Pas de risque d'exception
                  // (pas de copie de l'objet)
}
```

Cette fonction effectue un test préliminaire pour éviter l'auto affectation, puis alloue un nouveau tableau, utilisé ensuite pour remplacer le tableau existant ; la seule

1. Nous verrons plus loin une version améliorée de `Stack` ne faisant plus appel à `T::operator=()`.

instruction susceptible de lancer une exception est la fonction `New Copy()`. Si une exception se produit, l'état de l'objet est inchangé et l'exception est correctement transférée à l'appelant.

Cet exemple permet d'illustrer la technique générale à utiliser pour une gestion correcte des exceptions.

Recommandation

Lorsque vous implémentez une fonction destinée à réaliser une certaine opération, séparez les instructions effectuant l'opération elle-même (susceptibles de générer une exception), du code réalisant la validation de cette opération, constitué d'instructions unitaires (ne risquant pas de générer des exceptions).

Pb n° 10. Écrire du code robuste aux exceptions (3e partie) Difficulté : 9 1/2

Passons maintenant à la dernière partie de l'implémentation de notre conteneur `Stack`, la moins facile des trois...

Cette fois encore, il s'agit d'ajouter des nouvelles fonctions membres au modèle de classe `Stack` :

```
template <class T> class Stack
{
public:
  Stack();
  ~Stack();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
  size_t Count() const;
  void Push(const T&);
  T Pop(); // Lance une exception si la pile est vide
private:
  T*     v_;      // Pointeur vers une zone mémoire
                  // allouée dynamiquement
  size_t vsize_;  // Taille totale de la zone mémoire
  size_t vused_;  // Taille actuellement utilisée
};
```

Implémentez les fonctions membres `Count()`, permettant de récupérer la taille de l'espace mémoire utilisé par la pile, `Push()`, permettant d'ajouter un élément à la pile, et `Pop()`, permettant de retirer l'élément le plus récemment ajouté à la pile (cette fonc-

tion lancera une exception si la pile est vide). Bien entendu, ces trois fonctions doivent se comporter correctement en présence d'exceptions !

SOLUTION

La fonction Count()

Cette fonction est, de loin, la plus facile à implémenter :

```
template<class T>
size_t Stack<T>::Count() const
{
  return vused_;  // Ne peut pas lancer d'exception
}
```

`Count()` renvoie la valeur d'un type prédéfini. Aucun problème.

La fonction Push()

Pour cette fonction, nous allons employer une technique similaire à celle utilisée pour l'implémentation du constructeur de copie et de l'opérateur d'affectation :

```
template<class T>
void Stack<T>::Push( const T& t )
{
  if( vused_ == vsize_ )  // Si nécessaire, on augmente
  {                       // la taille du tableau
    size_t vsize_new = vsize_*2+1; // Facteur arbitraire
    T* v_new = NewCopy( v_, vsize_, vsize_new );
    delete[] v_;  // Ne peut pas lancer d'exception
    v_ = v_new;   // Prise de contrôle du nouveau tableau
    vsize_ = vsize_new;
  }
  v_[vused_] = t;
  ++vused_;
}
```

La fonction commence par exécuter, si nécessaire, un bloc d'instructions destinées à augmenter la taille du tableau : on fait d'abord appel à la fonction `NewCopy()` – si cette fonction génère une exception, l'état de l'objet `Stack` reste inchangé et l'exception est correctement transmise à l'appelant – puis à un ensemble d'instructions unitaires effectuant la prise en compte de l'opération. Ce premier bloc se comporte donc correctement en présence d'exceptions.

Cette opération de réallocation (éventuellement) effectuée, la fonction copie l'élément reçu en paramètre dans le tableau interne, puis incrémente l'indicateur de taille utilisé. Ainsi, si l'opération de copie échoue, l'indicateur de taille n'est pas modifié.

Recommandation

Lorsque vous implémentez une fonction destinée à réaliser une certaine opération, séparez les instructions effectuant l'opération elle-même (susceptibles de générer une exception), du code réalisant la validation de cette opération, constitué d'instructions unitaires (ne risquant pas de générer des exceptions).

La fonction Pop()

Contrairement aux apparences, cette fonction va être relativement complexe à sécuriser. En effet, que penser de l'implémentation ci-dessous, celle qui viendrait naturellement à l'esprit ?

```
// Est-ce vraiment une bonne implémentation ??

template<class T>
T Stack<T>::Pop()
{
  if( vused_ == 0)
  {
    throw "Erreur : la pile est vide";
  }
  else
  {
    T result = v_[vused_-1];
    --vused_;
    return result;
  }
}
```

Si la pile est vide, une exception est générée et transmise à l'appelant ; c'est correct. Dans les autres cas, on copie la valeur du dernier élément du tableau dans une variable locale `result`, que l'on renvoie à l'appelant, après avoir diminué la valeur de la taille utilisée. Si la copie de l'objet `v[vused-1]` vers `T` provoque une exception, alors la taille de la pile n'est pas modifiée et l'exception est transmise à l'appelant ; c'est correct également.

Finalement, cette fonction semble donc correctement implémentée. En réalité, ce n'est pas le cas ; en effet, considérons le code client suivant, qui utilise une variable `Stack<string>` nommée `s` :

```
string s1(s.Pop());
string s2;
s2 = s.Pop();
```

Si une exception se produit lors de l'appel à `s.Pop()` à la construction de l'objet `s1`, cet objet restera dans un état incohérent (non initialisé). Pire, si une exception se produit lors de l'appel à `Pop()` dans l'instruction `s2=s.Pop()`, non seulement l'objet `s2` ne sera pas correctement initialisé, mais, dans certains cas, il peut se produire des fuites mémoires irrécupérables. En effet, cette instruction fait appel à un objet tempo-

raire, retourné par la fonction[1], dans lequel est copié l'objet retiré de la pile, et qui sera à son tour copié dans `s2`. Si une exception se produit, lors de la copie de l'objet temporaire, le pointeur vers l'objet `T` venant d'être dépilé sera irrémédiablement perdu, laissant une zone mémoire impossible à désallouer. Ce code ne se comporte donc pas correctement en présence d'exceptions. Néanmoins, il faut noter que l'utilisateur de `Stack` n'avait aucune possibilité d'implémenter un code correct et que le problème est inhérent à la conception même de la fonction `Stack::Pop()` et au fait qu'elle renvoie un objet `T` par valeur, ce qui est la source de nos soucis (voir également le problème n° 19 sur conception des fonctions et gestion des exceptions).

Le principal enseignement de cet exemple est que la gestion correcte des exceptions n'est pas un détail d'implémentation, mais un élément devant être pris en compte dès la *conception de la classe*.

Erreur à éviter

Ne considérez pas la gestion des exceptions comme un détail d'implémentation. C'est au contraire un élément primordial à prendre en compte dès la conception de vos programmes.

Deuxième version de la fonction Pop()

Voici une seconde proposition d'implémentation pour la fonction `Pop()`, obtenue suite à un petit nombre de changements[2] :

```
template<class T>
void Stack<T>::Pop( T& result )
{
  if( vused_ == 0)
  {
    throw "Erreur : la pile est vide";
  }
  else
  {
    result = v_[vused_-1];
    --vused_;
  }
}
```

1. Il est possible que certains compilateurs, à des fins d'optimisation, n'aient pas recours à un objet temporaire dans ce type de situation. Néanmoins, pour être portable, un code doit se comporter correctement, qu'il y ait un objet temporaire ou non.
2. Plus exactement, suite au nombre *minimal* de changements possibles. La solution consistant à implémenter une fonction `Pop()` renvoyant une référence vers l'objet `T` dépilé n'est pas viable, car cela signifierait avoir une référence vers un objet contenu physiquement dans `Stack` mais dont `Stack` n'a plus la responsabilité, ce qui est dangereux (`Stack` pouvant tout à faire décider de désallouer l'objet correspondant, vu qu'il ne fait plus partie de la pile)

Cette implémentation assure que l'objet dépilé arrive correctement au niveau du code appelant, sauf en cas de pile vide, bien entendu.

Il subsiste néanmoins un défaut dans cette version : la fonction `Pop()` a *deux* responsabilités : dépiler le dernier élément de la pile et le renvoyer à l'appelant. Il serait préférable de séparer ces deux rôles.

Recommandation

Efforcez-vous de toujours découper votre code de manière à ce que chaque unité d'exécution (chaque module, chaque classe, chaque fonction) ait une responsabilité unique et bien définie.

Il serait préférable d'avoir deux fonctions distinctes, l'une renvoyant le dernier élément de la pile (`Top`), l'autre le dépilant (`Pop`).

```
template<class T>
T& Stack<T>::Top()
{
  if( vused_ == 0)
  {
    throw "Erreur : pile vide";
  }
  return v_[vused_-1];
}

template<class T>
void Stack<T>::Pop()
{
  if( vused_ == 0)
  {
    throw "Erreur : pile vide";
  }
  else
  {
    --vused_;
  }
}
```

On remarque au passage que cette même technique est utilisée par les conteneurs de la bibliothèque standard, dont les fonctions « `Pop` » (`list::pop_back`, `stack::pop`,...) ne permettent pas de récupérer la valeur venant d'être dépilée. Ce n'est pas un hasard; c'est, en réalité, le seul moyen de se comporter correctement en présence d'exceptions.

Les fonctions ci-dessus sont similaires aux fonctions membres `top` et `pop` du conteneur `stack<>` de la librairie standard. Là encore, il ne s'agit pas d'une coïncidence. Pour que le parallèle soit parfait, il faudrait rajouter deux fonctions membres à notre conteneur `Stack<>` :

```
template<class T>
const T& Stack<T>::Top() const
{
  if( vused_ == 0)
  {
    throw "Erreur : pile vide";
  }
  else
  {
    return v_[vused_-1];
  }
}
```

Version surchargée de la fonction `Top()` renvoyant une référence *constante* vers le dernier élément de la pile.

```
template<class T>
bool Stack<T>::Empty() const
{
  return( vused_ == 0 );
}
```

Fonction permettant de déterminer si la pile est vide ou non.

Nous obtenons ainsi un conteneur `Stack<>` dont l'interface publique est identique au `stack<>` de la bibliothèque standard (il faut néanmoins noter que le `stack<>` standard est différent puisqu'il sous-traite les fonctionnalités de conteneur à un objet interne, mais ce n'est qu'un détail d'implémentation).

Erreur à éviter

Ne sous-estimez pas l'importance de la gestion des exceptions, qui doit être prise en compte dès la phase de spécification d'un programme. Il peut être difficile, voire impossible, de réaliser un programme se comportant correctement en présence d'exceptions à partir de classes et fonctions mal conçues.

En particulier, comme nous venons de le voir, il est en général très difficile d'écrire du code correct en présence de fonctions effectuant deux tâches à la fois.

Nous étudierons plus loin dans ce chapitre un autre exemple classiquement générateur de problèmes liés aux exceptions : les constructeurs de copie et les opérateurs d'affectation incapables de gérer correctement l'auto-affectation (en fait ceux qui ont recours à un test préliminaire pour éviter l'auto-affectation mais ne fonctionneraient pas correctement sans ce test).

Pb n° 11. Écrire du code robuste aux exceptions (4e partie) Difficulté : 8

Ce problème fait la synthèse des trois problèmes précédents et tente de dégager des règles générales relatives à la gestion des exceptions.

Nous disposons à présent d'un conteneur `Stack<T>` se comportant correctement en présence d'exceptions.

Répondez le plus précisément possible aux questions suivantes :

- Quels sont les principaux éléments garantissant le bon comportement d'un programme en présence d'exceptions ?
- Quels sont les contraintes imposées au type contenu (`T`) pour le fonctionnement correct du conteneur `Stack<T>` ?

Solution

Il est certain qu'il existe plusieurs manières d'écrire du code se comportant correctement en présence d'exceptions. On peut néanmoins dégager les principes généraux suivants, initialement énoncés par Dave Abrahams :

- **Les zones mémoires dynamiques doivent être correctement désallouées en présence d'exceptions**. Dans le cas de notre conteneur `Stack`, ceci signifie que si une exception est lancée par un objet `T` contenu ou par toute autre opération effectuée au sein de `Stack`, les objets alloués dynamiquement doivent être correctement libérés afin de ne pas provoquer de fuites mémoires.
- **Toute opération échouant à cause d'une exception doit être annulée.** Autrement dit, une opération doit toujours être réalisée de manière atomique (soit entièrement effectuée, soit entièrement annulée) et ne jamais laisser le programme dans un état incohérent. C'est le type de sémantique qui est couramment utilisée dans le contexte des bases de données sous les termes « validation ou annulation » (« *commit* ou *rollback* ») . Prenons l'exemple d'un client de `Stack` appelant successivement la fonction `Pop()` pour récupérer une référence vers le dernier élément de la pile, puis la fonction `Push()`pour ajouter un élément : si cette deuxième fonction échoue à cause d'une exception, l'état de l'objet `Stack` doit rester inchangé et, en particulier, la référence obtenue par `Pop()` doit toujours être valide. Pour plus d'informations à ce sujet, voir la documentation de la SGI[1], une implémentation spécifique de la bibliothèque standard garantissant une gestion correcte des exceptions, réalisée par Dave Abrahams (http://www.stlport.org/doc/sgi_stl.html).

1. Silicon Graphics Implementation

Remarquons qu'en général, si le premier principe est respecté, il y a de fortes chances pour que le second le soit aussi, d'emblée[1]. Par exemple, dans le cas précis de notre conteneur `Stack`, nous nous sommes concentrés sur le fait d'obtenir une désallocation correcte des zones de mémoire dynamique en cas d'exceptions et avons automatiquement obtenu un code vérifiant pratiquement le second principe[2].

Dans certains cas, il peut être utile d'*exiger d'une fonction qu'elle ne lance d'exceptions dans aucun cas*. Il est parfois nécessaire d'avoir recours à une contrainte de ce type afin d'assurer le comportement correct d'un programme, comme nous l'avons vu à l'occasion du destructeur de `Stack`. Nous reviendrons sur ce sujet dans la suite de ce chapitre.

Recommandation

Connaissez et appliquez les principes généraux garantissant le bon comportement d'un programme en présence d'exceptions.

Concernant l'implémentation de `Stack`, il est intéressant de noter au passage que nous avons obtenu un modèle de classe parfaitement robuste aux exceptions en ne faisant appel qu'à un seul bloc `try/catch`. Nous verrons plus loin une seconde version de `Stack`, encore plus élégante, n'ayant, quant à elle, recours à *aucune* instruction `try/catch`.

Passons maintenant à la deuxième question. Les contraintes imposées au type contenu (`T`) pour le fonctionnement correct du conteneur `Stack<T>` sont les suivantes :

- existence d'un constructeur par défaut (nécessaire pour la bonne compilation de l'instruction « `new T[...]` ») ;
- existence d'un constructeur de copie (dans le cas où `Pop` retourne un objet par valeur) ;
- destructeur ne lançant pas d'exceptions (indispensable pour la robustesse du code) ;
- existence d'un opérateur d'affectation robuste aux exceptions (afin de garantir, qu'en cas d'échec de l'affectation, l'objet cible est inchangé). Notons au passage

1. Ce n'est pas systématique ; le conteneur `vector` de la bibliothèque standard est un contre-exemple bien connu.
2. Sauf dans un cas subtil : si la fonction `Push()` réalise une réallocation du tableau interne (augmentation de la taille) mais que l'affectation `v_[vused_]= t` échoue sur une exception, l'objet `Stack` reste dans un état cohérent, mais les références vers les objets internes initialement obtenues par appel à la fonction `Top()` ne sont plus valables, la zone mémoire interne ayant changé d'emplacement. Ce défaut pourrait être aisément corrigé en déplaçant quelques lignes de code et ajoutant un bloc `try/catch`, mais nous verrons, plus loin dans ce chapitre, une deuxième implémentation de `Stack` qui résout ce problème plus élégamment.

que c'est la seule fonction à laquelle il faut imposer d'être robuste aux exceptions pour obtenir une classe `Stack` qui soit elle-même robuste.

Dans la suite de ce chapitre, nous allons voir comment il est possible de diminuer les contraintes imposées à `T`, tout en conservant une parfaite robustesse aux exceptions. Nous nous intéresserons également à l'instruction « `delete[] x` » et à ses dangers.

Pb n° 12. Écrire du code robuste aux exceptions (5e partie) Difficulté : 7

Nous nous attaquons maintenant à la réalisation de deux nouvelles versions du conteneur `Stack`.

Nous disposons déjà d'une première version de `Stack` tout à fait robuste aux exceptions. Dans ce problème et les suivants, nous allons réaliser deux versions différentes de `Stack`, fournissant ainsi trois solutions complètes au problème original de Cargill.

Les deux nouvelles solutions permettront de pallier certaines imperfections de la version de `Stack` obtenue pour l'instant, sur laquelle on peut légitimement se poser les questions suivantes :

- N'existe un moyen plus efficace de gérer les zones de mémoire dynamiques et, en particulier, de supprimer le bloc `try/catch` ?
- Est-il possible de réduire le nombre des contraintes imposées au type contenu (`T`) ?
- Le modèle de classe `Stack` – et, plus généralement, tout conteneur générique – doit-il avoir recours à des spécifications d'exception dans sa déclaration ?
- Quelles opérations effectuent réellement les opérateurs `new[]` et `delete[]` ?

La réponse à cette dernière question pourra peut-être vous surprendre ; elle sera en tous cas l'occasion d'insister sur le fait qu'une bonne compréhension des mécanismes d'exception passe obligatoirement par une maîtrise parfaite des opérations exécutées par le code, notamment toutes les conversions implicites, invocations d'opérateurs redéfinis et utilisations d'objets temporaires pouvant se cacher derrière un banal appel de fonction, chacune de ces opérations pouvant potentiellement générer une exception[1].

Un premier point sur lequel on peut améliorer le fonctionnement de `Stack` est la gestion des zones de mémoire dynamique, qu'il est possible d'encapsuler dans une classe utilitaire séparée :

1. Sauf celles ayant recours explicitement à la spécification d'exception `throw()` dans leur déclaration, ainsi que quelques fonctions de la bibliothèque standard, dûment documentées.

```
template <class T> class StackImpl
{
/*????*/:
  StackImpl(size_t size=0);
  ~StackImpl();
  void Swap(StackImpl& other) throw();

  T*     v_;      // Pointeur vers une zone mémoire
                  // allouée dynamiquement
  size_t vsize_;  // Taille totale de la zone mémoire
  size_t vused_;  // Taille actuellement utilisée
};
private:
  // Privé et non défini. Pas de copie possible
  StackImpl( const StackImpl& );
  StackImpl& operator=( const StackImpl& );
};
```

On retrouve dans `StackImpl` toutes les variables membres initialement situées dans `Stack`. On note la présence d'une fonction `Swap()`permettant d'échanger la valeur de l'objet sur lequel elle est implémentée avec celle de l'objet passé en paramètre.

1. Implémentez les trois fonctions membres de `StackImpl`, en tenant compte cette fois de la contrainte complémentaire suivante : le nombre d'objets construits situés dans le tableau interne doit être *exactement* égal au nombre d'objets dans la pile (en particulier, l'espace inutilisé ne doit pas contenir d'objets construits).
2. Quel est le rôle de `StackImpl` ?
3. Par quoi peut-on remplacer le commentaire `/*????*/` ? Argumentez votre réponse.

Solution

Nous présentons ici une implémentation possible pour chacune des trois fonctions membres de la `StackImpl`. Les mécanismes de gestion des exceptions sont très similaires à ceux déjà rencontrés au début de ce chapitre ; à ce titre, nous ne les ré-exposerons pas en détail.

En revanche, nous commencerons par introduire quelques fonctions utilitaires courantes, qui seront utilisées pour l'implémentation de `StackImpl` et `Stack`, dans ce problème et les suivants.

Constructeur

Le constructeur est relativement facile à implémenter. Nous utilisons l'opérateur « `new sizeof(T)*size` » pour implémenter un tableau d'octets (car si nous avions utilisé l'instruction « `new T[size]` », le tableau aurait été initialisé avec des objets `T` construits, ce qui a été explicitement proscrit par l'énoncé du problème).

Quelques fonctions utilitaires

Nous présentons ici trois fonctions utilitaires que nous réutiliserons dans la suite du chapitre : `construct()`, qui permet de construire un objet à une adresse mémoire donnée en lui affectant une valeur donnée, `destroy()`, qui permet de détruire un objet (nous en verrons deux versions) et `swap()`, directement inspirée de la bibliothèque standard, qui échange deux valeurs.

```
// construct() : construit un nouvel objet
// à une adresse donnée, en l'initialisant
// avec une valeur donnée
template <class T1, class T2>
void construct( T1* p, const T2& value )
{
  new (p) T1(value);
}
```

Utilisé avec cette syntaxe, l'opérateur « `new` » construit un nouvel objet `T1` en le plaçant à l'adresse `p` (placement `new`) ; aucune mémoire n'est allouée dynamiquement. Un objet construit de cette façon doit, sauf cas particulier, être détruit par un appel explicite au destructeur, et non pas avec l'opérateur `delete`.

```
// destroy() : détruit un objet
// ou un tableau d'objets
//
template <class T>
void destroy( T* p )
{
  p->~T();
}

template <class FwdIter>
void destroy( FwdIter first, FwdIter last )
{
  while( first != last )
  {
    destroy( &*first );
    ++first;
  }
}
```

La première de ces deux fonctions est symétrique de la fonction `construct()` et effectue, comme préconisé, un appel explicite du destructeur. Quant à la deuxième, nous aurons l'occasion de voir sa grande utilité très prochainement.

```
// swap() : échange deux valeurs
//
template <class T>
void swap( T& a, T& b )
{
  T temp(a); a = b; b = temp;
}
```

```
template <class T>
StackImpl<T>::StackImpl( size_t size )
  : v_( static_cast<T*>
         ( size == 0
           ? 0
           : operator new(sizeof(T)*size) ) ),
    vsize_(size),
    vused_(0)
{
}
```

Destructeur

Le destructeur est la plus facile des trois fonctions. Nous l'implémentons ici en fonction de `destroy()`, vue plus haut et de l'opérateur `delete()`, dont il faut se rappeler ce qui a été vu dans les problèmes précédents.

```
template <class T>
StackImpl<T>::~StackImpl()
{
    destroy( v_, v_+vused_ ); // Ne peut pas lancer d'exception
    operator delete( v_ );
}
```

La fonction Swap()

Et, pour finir, la fonction `Swap()` qui nous sera d'une extrême utilité dans l'implémentation de `Stack` ; notamment pour la fonction `operator=()`, comme nous allons le voir bientôt.

```
template <class T>
void StackImpl<T>::Swap(StackImpl& other) throw()
{
    swap( v_,     other.v_ );
    swap( vsize_, other.vsize_ );
    swap( vused_, other.vused_ );
}
```

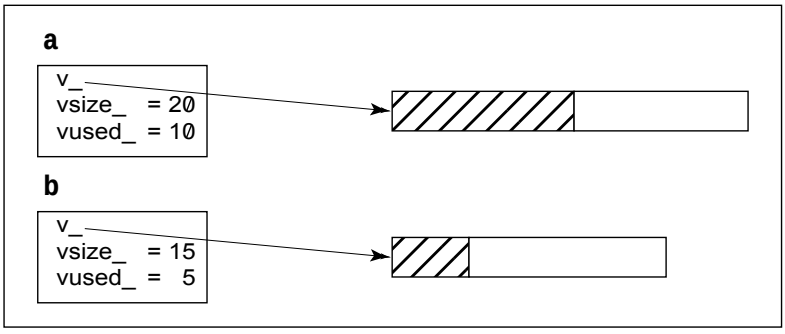


Figure 1. Deux objets `StackImpl<T>` a et b

Pour mieux visualiser le fonctionnement de `Swap()`, prenons l'exemple de deux objets `StackImpl<T>` `a` et `b`, représentés sur la figure 1.

Voici ce que deviennent ces deux objets après exécution de l'instruction `a.Swap(b)`:

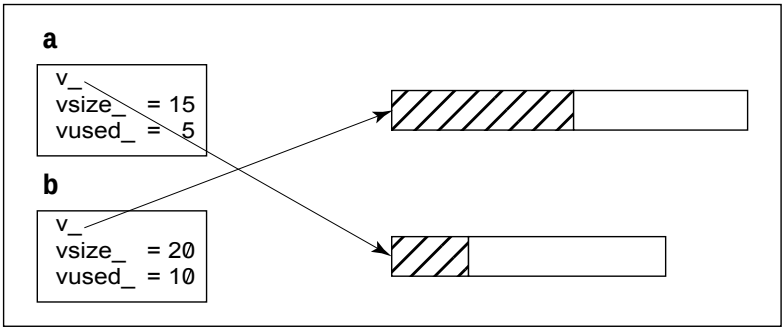


Figure 2. Les mêmes objets `StackImpl<T>`, après `a.Swap(b)`

On peut noter au passage que la fonction `Swap()` ne génèrera jamais d'exception, par construction. C'est d'ailleurs pour cela qu'elle sera la pierre angulaire de la robustesse aux exceptions de notre nouvelle version de `Stack`.

La raison d'être de `StackImpl` est simple : son rôle est de masquer à la classe externe `Stack` les détails d'implémentation liés à la gestion des zones de mémoire dynamique. D'une manière générale, il est toujours préférable de recourir le plus possible à l'encapsulation et de séparer les responsabilités.

Recommandation

Efforcez-vous de toujours découper votre code de manière à ce que chaque unité d'exécution (chaque module, chaque classe, chaque fonction) ait une responsabilité unique et bien définie.

Passons maintenant à la troisième question : par quoi peut-on remplacer le commentaire `/*????*/` inclus dans la déclaration de `StackImpl` ? La véritable question sous-jacente est plutôt : comment `StackImpl` sera-t-elle utilisée par `Stack` ? Il y a en C++ deux possibilités techniques pour mettre en oeuvre une relation « EST-IMPLEMENTE-EN-FONCTION-DE »: l'utilisation d'une classe de base privée ou l'utilisation d'une variable membre privée.

Technique n° 1 : classe de base privée. Dans ce cas de figure, le commentaire doit être remplacé par `protected`[1] (les fonctions privées ne pouvant pas être appelées depuis la classe dérivée). La classe `Stack` dériverait de la classe `StackImpl` de manière privée et lui déléguerait toute la gestion de la mémoire, assurant pour sa part,

1. L'emploi de public serait également techniquement possible, mais pas conforme à l'utilisation que l'on souhaite faire de `StackImpl` ici.

outre les diverses fonctionnalités du conteneur, la construction des objets `T` contenus. Le fait de séparer clairement l'allocation/désallocation du tableau de la construction/destruction des objets contenus présente de nombreux avantages : en particulier, nous sommes certains que l'objet `Stack` ne sera pas construit si l'allocation – effectuée dans le constructeur de `StackImpl` – échoue.

Technique n° 2 : membre privé. Dans cette deuxième solution, le commentaire doit être remplacé par `public`. Le conteneur `Stack` serait toujours « IMPLEMENTE-EN-FONCTION-DE » de la classe `StackImpl`, mais en lui étant cette fois lié par une relation de type « A-UN ». Nous avons, ici encore, une séparation nette des responsabilités : la classe `StackImpl` gérant les ressources mémoires tandis que la classe `Stack` gère la construction/destruction des objets contenus. Du fait que le constructeur d'une classe est appelé après les constructeurs des objets membres, nous sommes également assurés que l'objet `Stack` ne sera pas construit si l'allocation de la zone mémoire échoue.

Ces deux techniques sont très similaires mais présentent néanmoins quelques différences. Nous allons étudier successivement l'une puis l'autre, dans les deux problèmes suivants.

Pb n° 13. Écrire du code robuste aux exceptions (6e partie) Difficulté : 9

Nous étudions dans ce problème une nouvelle version de `Stack`, utilisant `StackImpl` comme classe de base, qui nous permettra de diminuer le nombre de contraintes imposées au type contenu et d'obtenir une implémentation très élégante de l'opérateur d'affectation.

Le but de ce problème est d'implémenter une deuxième version du modèle de classe `Stack`, dérivant de la classe `StackImpl` étudiée dans le problème précédent (dans laquelle on remplacera donc le commentaire par `protected`) :

```
template <class T>
class Stack : private StackImpl<T>
{
public:
  Stack(size_t size=0);
  ~Stack();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
  size_t Count() const;
  void   Push(const T&);
  T&     Top();   // Lance une exception si la pile est vide
  void   Pop();   // Lance une exception si la pile est vide
};
```

Proposez une implémentation pour chacune des fonctions membres de `Stack`, en faisant bien entendu appel aux fonctions adéquates de la classe de base `StackImpl`.

Bien entendu, la classe `Stack` doit toujours être parfaitement robuste aux exceptions. Signalons, au passage, qu'il existe une manière très élégante d'implémenter la fonction `operator=()`.

Solution

Constructeur par défaut

Voici une proposition d'implémentation pour le constructeur par défaut de `Stack` (on présente une implémentation en-ligne à des fins de concision) :

```
template <class T>
class Stack : private StackImpl<T>
{
public:
  Stack(size_t size=0)
    : StackImpl<T>(size)
  {
  }
```

Ce constructeur fait simplement appel au constructeur de la classe de base `StackImpl`, lequel initialise l'état de l'objet et effectue l'allocation initiale de la zone mémoire. La seule fonction susceptible de générer une exception est le constructeur de `StackImpl`. Il n'y a donc pas de risque de se retrouver dans un état incohérent : si une exception se produit, l'objet `Stack` ne sera jamais construit (voir le problème n° 8 pour plus de détails au sujet des constructeurs lançant des exceptions).

On peut noter au passage une légère modification de la déclaration du constructeur par défaut par rapport aux problèmes précédents : nous introduisons ici un paramètre `size`, avec une valeur par défaut de 0, permettant de spécifier la taille initiale du tableau et que nous aurons l'occasion d'utiliser lors de l'implémentation de la fonction `Push`.

Recommandation

Pour une meilleure robustesse aux exceptions, encapsulez le code gérant les ressources externes (zones de mémoire dynamique, connexions à des bases de données,...) en le séparant du code utilisant ces ressources.

Destructeur

Nous n'avons pas besoin d'implémenter de destructeur spécifique pour la classe `Stack`, ce qui est assez élégant. En effet, le destructeur de `StackImpl`, appelé par l'implémentation par défaut du destructeur de `Stack`, détruit correctement tous les objets contenus et effectue la désallocation du tableau interne.

Constructeur de copie

Voici une proposition d'implémentation, utilisant la fonction `construct()` détaillée lors du problème précédent :

```
Stack(const Stack& other)
  : StackImpl<T>(other.vused_)
{
  while( vused_ < other.vused_ )
  {
    construct( v_+vused_, other.v_[vused_] );
    ++vused_;
  }
}
```

Nous avons ici une implémentation efficace et élégante du constructeur de copie de `Stack` (dont on peut remarquer, au passage, qu'elle ne fait pas appel au constructeur de copie de `StackImpl`). Si le constructeur de `T` – appelé depuis la fonction `construct()`– lance une exception (c'est la seule source possible ici), le destructeur de `StackImpl` est appelé, ce qui a pour effet de détruire tous les objets `T` déjà construits et de désallouer correctement le tableau mémoire interne. Nous voyons ici l'un des grands avantages de l'utilisation de `StackImpl` comme classe de base : nous pouvons implémenter autant de constructeurs de `Stack` que nous le désirons, sans avoir à nous soucier explicitement des destructions et désallocations en cas d'échec de la construction.

Opérateur d'affectation

Nous présentons ici une version très élégante et très robuste de l'opérateur d'affectation de `Stack` :

```
Stack& operator=(const Stack& other)
{
  Stack temp(other); // Fait tout le travail
  Swap( temp );      // Ne peut pas lancer d'exception
  return *this;
}
```

Élégant, n'est-ce pas ? Nous avons déjà vu ce type de technique lors du problème n° 10, rappelons-en le principe fondateur :

Recommandation

Lorsque vous implémentez une fonction destinée à réaliser une certaine opération, séparez les instructions effectuant l'opération elle-même (susceptibles de générer une exception), du code réalisant la validation de cette opération, constitué d'instructions unitaires (ne risquant pas de générer des exceptions).

Nous construisons un objet `Stack` temporaire (`temp`), initialisé à partir de l'objet source (`other`) puis nous appelons `Swap` (membre de la classe de base) pour échanger l'état de notre objet temporaire avec celui de l'objet passé en paramètre; lorsque la fonction se termine, le destructeur de `temp` est automatiquement appelé, ce qui a pour effectuer de désallouer correctement toutes les ressources liées à l'ancien état de notre objet `Stack`.

Notons qu'implémenté de cette manière, l'opérateur d'affectation gère correctement, de manière native, le cas de l'auto-affectation (« `Stack s; s=s;` »). Voir le problème n° 38 pour plus d'informations sur l'auto-affectation, et notamment sur l'emploi du test « `if (this != &other)` » dans un opérateur d'affectation).

Les sources d'exceptions potentielles sont l'allocation ou la construction des objets `T`. Comme elles ne peuvent se produire que lors de la construction de l'objet `temp`, nous sommes assurés qu'en cas d'échec, l'impact sur notre objet `Stack` sera nul. Il n'y a pas non plus de risque de fuite mémoire liée à l'objet `temp`, car le constructeur de copie de `Stack` est parfaitement robuste de ce point de vue. Une fois que nous sommes certains que l'opération a été effectuée correctement, nous la validons en mettant à jour l'état de notre objet `Stack` grâce à la fonction `Swap()`, laquelle ne peut pas lancer d'exceptions (elle est déclarée avec une spécification d'exceptions `throw` et n'effectue, de toute façon, que des copies de valeurs de types prédéfinis).

Cette implémentation est largement plus élégante que celle proposée lors du problème n° 9; elle est également plus simple, ce qui permet de s'assurer d'autant plus facilement de son bon comportement en présence d'exceptions.

Il est d'ailleurs possible de faire encore plus compact, en utilisant un passage par valeur au lieu d'une variable temporaire :

```
Stack& operator=(Stack temp)
{
  Swap( temp );
  return *this;
}
```

La fonction Stack<T>::Count

C'est, de loin, la plus facile.

```
size_t Count() const
{
  return vused_;
}
```

La fonction Stack<T>::Push

Cette fonction est un peu plus complexe. Prenez le temps de l'étudier un moment avant de lire le commentaire qui suit.

```
void Push( const T& t )
{
  if( vused_ == vsize_ )  // grow if necessary
  {
    Stack temp( vsize_*2+1 );
    while( temp.Count() < vused_ )
    {
      temp.Push( v_[temp.Count()] );
    }
    temp.Push( t );
    Swap( temp );
  }
  else
  {
    construct( v_+vused_, t );
    ++vused_;
  }
}
```

Nous distinguons deux cas dans cette implémentation, suivant qu'il faut augmenter ou non la taille du tableau interne.

Commençons par le cas le plus simple : s'il reste de la place allouée mais non occupée dans le tableau interne, nous appelons la fonction `construct()` pour tenter de placer l'objet à ajouter dans ce tableau et, si l'opération réussit, nous mettons à jour la taille utilisée (`vused_`).

L'autre cas est un peu plus complexe : s'il est nécessaire d'agrandir le tableau, nous construisons un objet `Stack` temporaire (`temp`) auquel nous ajoutons tous les éléments actuellement contenus dans notre objet `Stack`, ainsi que l'objet `t` passé en paramètre de la fonction `Push()`; pour finir, nous utilisons la fonction `Swap()` pour échanger l'état de notre objet avec celui de l'objet temporaire.

Est-ce robuste aux exceptions ? La réponse est oui :

- Si la construction de `temp` échoue, l'état de notre objet `Stack` est inchangé et il n'y pas de fuite mémoire, ce qui est correct.
- Si l'une des opérations de chargement de l'objet `temp` échoue et lance une exception (soit lors de l'ajout à `temp` d'un objet existant, soit lors de la construction ou de l'ajout à `temp` du nouvel objet, copie de `t`), les ressources sont correctement libérées lors de la destruction de `temp`, qui se produit automatiquement lorsque la fonction `Push` se termine)
- Dans aucun cas, nous ne modifions pas l'état de notre objet `Stack` original tant que l'opération d'ajout du nouvel élément n'a pas été correctement effectuée.

Nous avons donc ici un solide mécanisme de « validation ou annulation », la fonction `Swap()` n'étant exécutée que si les opérations de réallocation éventuelle et d'ajout du nouvel élément se sont correctement terminées. En particulier, nous avons éliminé le risque de corruption des références vers les objets contenus qui pouvait survenir suite à un appel de `Push()` effectuant une réallocation suivie d'un échec de l'ajout du nouvel élément, ainsi que nous l'avons vu lors du problème n° 11.

La fonction Stack<T>::Top

La fonction `Top()` reste inchangée.

```
T& Top()
{
  if( vused_ == 0 )
  {
    throw "Pile vide";
  }
  return v_[vused_-1];
}
```

La fonction Stack<T>::Pop

La fonction `Pop()` également, à part l'appel supplémentaire à `destroy()`.

```
void Pop()
{
  if( vused_ == 0 )
  {
    throw "Pile vide";
  }
  else
  {
    --vused_;
    destroy( v_+vused_ );
  }
}
```

En résumé, l'utilisation de `StackImpl` comme classe de base nous a permis de simplifier l'implémentation des fonctions membres de `Stack`. L'un des plus grands avantages de la gestion séparée des ressources mémoires effectuée par `StackImpl` se manifeste dans l'implémentation des constructeurs et du destructeur de `Stack` : d'une part, nous pouvons implémenter autant de constructeurs de `Stack` que nous le désirons, sans avoir à nous soucier explicitement des destructions et désallocations en cas d'échec de la construction ; d'autre part, il n'est pas nécessaire d'implémenter un destructeur pour `Stack`, le destructeur par défaut convient.

Pour finir, il est intéressant de noter que nous avons éliminé le bloc `try/catch` contenu dans la première version, prouvant ainsi qu'il est tout à fait possible d'implémenter une classe parfaitement robuste aux exceptions sans utiliser un seul bloc `try/catch`.

Pb n° 14. Écrire du code robuste aux exceptions (7e partie) Difficulté : 5

Deuxième variante de `Stack`, utilisant cette fois `StackImpl` en tant qu'objet membre.

Dans ce problème, on implémentera une troisième version du modèle de classe `Stack` utilisant un objet membre de type `StackImpl` (décrite lors du problème n° 12 ; on remplacera le commentaire de la déclaration par `public`).

```
template <class T>
class Stack
{
public:
  Stack(size_t size=0);
  ~Stack();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
  size_t Count() const;
  void   Push(const T&);
  T&     Top();   // Lance une exception si la pile est vide
  void   Pop();   // Lance une exception si la pile est vide
private:
  StackImpl<T> impl_;  // Objet privé
};
```

Proposez une implémentation pour chacune des fonctions membres de `Stack`. N'oubliez pas la robustesse aux exceptions.

Solution

L'implémentation étant extrêmement similaire à la version précédente, nous présentons directement le code (les fonctions sont en-lignes, à des fins de concision) :

```
template <class T>
class Stack
{
public:
  Stack(size_t size=0)
    : impl_(size)
  {
  }
  Stack(const Stack& other)
    : impl_(other.impl_.vused_)
  {
    while( impl_.vused_ < other.impl_.vused_ )
    {
      construct( impl_.v_+impl_.vused_,
                 other.impl_.v_[impl_.vused_] );
```

```
      ++impl_.vused_;
    }
  }
  Stack& operator=(const Stack& other)
  {
    Stack temp(other);
    impl_.Swap(temp.impl_); // this can't throw
    return *this;
  }
  size_t Count() const
  {
    return impl_.vused_;
  }
  void Push( const T& t )
  {
    if( impl_.vused_ == impl_.vsize_ )
    {
      Stack temp( impl_.vsize_*2+1 );
      while( temp.Count() < impl_.vused_ )
      {
        temp.Push( impl_.v_[temp.Count()] );
      }
      temp.Push( t );
      impl_.Swap( temp.impl_ );
    }
    else
    {
      construct( impl_.v_+impl_.vused_, t );
      ++impl_.vused_;
    }
  }
  T& Top()
  {
    if( impl_.vused_ == 0 )
    {
      throw "Pile vide";
    }
    return impl_.v_[impl_.vused_-1];
  }
  void Pop()
  {
    if( impl_.vused_ == 0 )
    {
      throw "Pile vide";
    }
    else
    {
      --impl_.vused_;
      destroy( impl_.v_+impl_.vused_ );
    }
  }
private:
  StackImpl<T> impl_; // Objet privé
};
```

Nous avons ainsi, dans ce problème et le précédent, obtenu deux nouvelles implémentations possibles de `Stack`. Chacune a ses avantages et ses inconvénients, ainsi que nous allons le voir dans le problème suivant.

Pb n° 15. Écrire du code robuste aux exceptions (8e partie) — Difficulté : 9

Prenons un moment pour comparer les trois implémentations différentes de `Stack` obtenues jusqu'ici.

1. Vaut-il mieux utiliser `StackImpl` comme classe de base ou comme objet membre ?
2. Dans quelle mesure les deux dernières versions de `Stack` sont-elles réutilisables ? Quelles contraintes imposent-elles à `T`, le type contenu ? Note : plus les contraintes sur `T` sont faibles, plus le conteneur est réutilisable de manière générique.
3. La classe `Stack` doit-elle avoir recours à des spécifications d'exceptions dans les déclarations de ses fonctions membres ?

Solution

Répondons aux questions dans l'ordre :

1. *Vaut-il mieux utiliser* `StackImpl` *comme classe de base ou comme objet membre ?*

Ces deux méthodes sont relativement similaires et permettent toutes les deux d'obtenir un partage net des responsabilités entre allocation/désallocation et construction/destruction.

En règle générale, lorsqu'on a besoin de mettre en oeuvre une relation du type « EST-IMPLEMENTE-EN-FONCTION-DE », il est préférable d'utiliser la composition (utilisation d'un objet membre) plutôt que la dérivation (utilisation d'une classe de base), lorsque cela est possible. En effet, la composition présente l'avantage de masquer complètement la classe `StackImpl`. Ainsi, le code client ne voit que l'interface publique de `Stack` et n'est par conséquent pas dépendant de `StackImpl`. Il est néanmoins parfois obligatoire de faire dériver la classe utilisatrice de la classe utilitaire dans certains cas :

- Besoin d'accéder à une fonction protégée de la classe utilitaire.
- Besoin de redéfinir une fonction virtuelle de la classe utilitaire.
- Besoin de construire l'objet utilitaire avant d'autres objets de base de la classe utilisatrice.[1]

1. On pourrait également rajouter une quatrième raison, un peu moins défendable, de confort d'écriture (le fait d'utiliser une classe de base permettant de s'affranchir de nombreux « `impl_.` »)

2. *Dans quelle mesure les deux dernières versions de* `Stack` *sont-elles réutilisables ? Quelles contraintes imposent-elles à* `T`*, le type contenu ?*

Lorsque vous implémentez un modèle de classe – et, qui plus est, lorsque ce modèle est destiné à servir de conteneur – posez-vous toujours la question de la réutilisabilité de votre code. Plus votre modèle imposera de contraintes au niveau du code client, moins il sera réutilisable. Dans le cas de notre exemple `Stack`, il s'agit de savoir qu'elles sont les contraintes imposées à `T`, le type contenu (et, en particulier, si le nombre de ces contraintes a été réduit par rapport à la première version de `Stack`).

Nous avons déjà vu que la différence principale entre les deux nouvelles versions de `Stack` et la version précédente se situe au niveau de la gestion de la mémoire (séparation des responsabilités d'allocation / désallocation et construction / destruction). Cette évolution permet une meilleure robustesse aux exceptions, mais ne change pas grand chose aux contraintes imposées à `T`.

Une autre différence est que les nouvelles versions de `Stack` construisent et détruisent les objets contenus au fur et à mesure de leur ajout/suppression de la pile, à l'inverse de la première version qui remplissait son tableau interne avec des objets construits par le constructeur par défaut. Ceci augmente la performance de la classe `Stack`, mais surtout, permet de diminuer les contraintes imposées au type contenu.

Rappelons quelles étaient ces contraintes avec la première version de `Stack` :

- Existence d'un constructeur par défaut (nécessaire pour la bonne compilation de l'instruction « `new T[...]` »)
- Existence d'un constructeur de copie (dans le cas où `Pop` retourne un objet par valeur)
- Destructeur ne lançant pas d'exceptions (indispensable pour la robustesse du code)
- Existence d'un opérateur d'affectation robuste aux exceptions (afin de garantir, qu'en cas d'échec de l'affectation, l'objet cible est inchangé).

Dans les nouvelles versions de `Stack`, seul le constructeur de copie de `T` est utilisé : il n'est donc pas nécessaire d'imposer à `T` d'avoir un constructeur par défaut et un opérateur d'affectation. Les contraintes imposées à `T` par les nouvelles versions ne sont donc qu'au nombre de deux :

- Existence d'un constructeur de copie
- Destructeur ne lançant pas d'exceptions

Ceci améliore la réutilisabilité de notre conteneur `Stack`, par rapport à la version originale. Il existe en effet de nombreuses classes n'ayant pas de constructeur par défaut ou pas d'opérateur d'affectation (comme, par exemple, les classes contenant des membres de type référence, pour lesquelles on souhaite parfois interdire l'affectation). Ce type de classe peut maintenant être stocké dans notre nouveau conteneur `Stack`.

Recommandation

Pensez dès la conception à la réutilisabilité de votre code.

3. *La classe* `Stack` *doit-elle avoir recours à des spécifications d'exceptions dans les déclarations de ses fonctions membres ?*

La réponse est non. La raison principale en est que l'auteur de `Stack` n'a a priori aucune idée du comportement que va avoir le type contenu `T` ; il ignore en particulier si les fonctions membres de `T` vont générer des exceptions et, si c'est le cas, quels types d'exceptions. Introduire des spécifications d'exception irait donc totalement à l'encontre de la réutilisabilité de `Stack`, imposant des contraintes supplémentaires sur le type contenu.

D'un autre côté, il y a certaines fonctions membres pour lesquelles on est certain qu'elles ne lanceront pas d'exception (par exemple, `Count()`). On pourrait être tenté de spécifier « `throw()` » au niveau de la déclaration de ces fonctions. Ce n'est pas souhaitable pour deux raisons :

- Appliquer une spécification `throw()` à une fonction membre impose des contraintes supplémentaires à votre code : vous vous interdisez, par exemple, de substituer à l'implémentation de cette fonction une nouvelle version susceptible de générer des exceptions, sous peine de perturber fortement les clients, s'attendant à ce qu'aucune exception ne soit générée. Autre exemple, appliquer « `throw()` » à une fonction virtuelle impose que les fonctions redéfinies ne génèrent pas d'exception. Il ne faut certes pas en conclure qu'il ne faut *jamais* utiliser throw() ; il faut néanmoins en faire un usage prudent.
- Les spécifications d'exceptions peuvent être coûteuses en terme de performances (bien que, sur ce point, les compilateurs s'améliorent de jour en jour). Il est donc préférable de les éviter pour les fonctions et classes utilisées fréquemment, comme par exemple, les conteneurs génériques.

Pb n° 16. Écrire du code robuste aux exceptions (9e partie) Difficulté : 8

Connaissez-vous bien les opérations effectuées l'instruction `delete[]`et leur impact sur la robustesse aux exceptions ?

Considérez l'instruction « `delete [] p` », `p` pointant vers un tableau en mémoire globale ayant été correctement alloué et initialisé par une instruction « `new[]` ».

1. Détaillez précisément les opérations effectuées par l'instruction « `delete[] p` ».
2. Ces opérations présentent-elles des dangers ? Argumentez votre réponse.

Solution

Ce problème va être l'occasion d'aborder les dangers de l'instruction « `delete []` », dont l'apparente innocence est trompeuse.

Les dangers des destructeurs lançant des exceptions

Rappelons-nous la fonction utilitaire `destroy()` vue lors du problème n° 12 :

```
template <class FwdIter>
void destroy( FwdIter first, FwdIter last )
{
  while( first != last )
  {
    destroy( &*first );
    ++first;
  }
}
```

Lors des problèmes précédents, cette fonction se comportait correctement car nous avions fait l'hypothèse que le destructeur de `T` ne lançait pas d'exceptions. Si cette condition n'est pas respectée et qu'une exception est générée, disons, lors du premier `destroy()` d'une série de cinq, la fonction s'arrêtera prématurément, laissant en mémoire quatre objets impossibles à détruire, ce qui n'est évidemment pas une bonne solution.

On pourrait certes arguer qu'il doit être possible d'écrire une fonction `destroy()` capable de gérer correctement le cas d'une exception lancée depuis le destructeur de `T`, avec une implémentation du type :

```
template <class FwdIter>
void destroy( FwdIter first, FwdIter last )
{
  while( first != last )
  {
    try
    {
      destroy( &*first );
    }
    catch(...)
    {
      /* que faire ici ? */
    }
    ++first;
  }
}
```

Bien évidemment, toute la question est ici de décider quel traitement on va effectuer dans le bloc `catch`. Il y a trois solutions possibles : relancer l'exception ; la convertir puis relancer une exception différente ; ne pas relancer d'exception et continuer la boucle.

1. Si le bloc « `catch` » relance l'exception, la fonction `destroy()` sera certes partiellement robuste aux exceptions du fait qu'elle retransmettra correctement les exceptions à l'appelant, mais en revanche tout à fait insatisfaisante du fait qu'elle laissera en mémoire des objets qui ne pourront jamais être détruits. Il faut donc exclure cette première solution.

2. Si le bloc « `catch` » convertit l'exception et en relance une différente, non seulement la fonction `destroy()` provoquera toujours des fuites mémoires mais, de plus, elle ne retransmettra même pas correctement les exceptions à l'appelant. Cette solution est donc encore pire que la première.

3. Si le bloc « `catch` » absorbe l'exception et continue la boucle, le problème inverse sera produira : la fonction `destroy()` ne risquera pas de provoquer de fuites mémoire[1], mais elle ne permettra pas à l'appelant de recevoir correctement les exceptions générées (même si un traitement correct et/ou un enregistrement de l'erreur est effectué dans le bloc « `catch` », ceci ne doit pas dispenser la fonction de retransmettre les exceptions à l'appelant).

D'aucuns suggèrent d'utiliser le bloc « `catch` » pour « enregistrer » temporairement l'exception, ce qui permet à la boucle de se terminer normalement, puis de relancer l'exception en question à la fin de l'exécution de la fonction. Ce n'est pas une bonne solution car, si plusieurs exceptions se produisent au cours de la boucle, il n'est possible d'en relancer qu'une seule à la fin, même si elles ont été toutes enregistrées. Toutes les autres alternatives diverses qu'on pourrait imaginer s'avèrent toutes, croyez-moi, déficientes d'un point de vue ou d'un autre. La conclusion est simple : il n'est pas possible d'écrire du code robuste en présence de destructeurs susceptibles de lancer des exceptions.

Ceci nous conduit naturellement à un sujet connexe : le comportement des opérateurs `new[]` et `delete[]` en présence d'exceptions.

Considérons par exemple le code suivant :

```
T* p = new T[10] ;
delete[] p ;
```

Ceci est sans aucun doute du C++ standard, comme vous avez sûrement eu l'occasion d'en écrire tous les jours. Néanmoins, vous êtes-vous déjà posé la question de savoir ce qui se passerait si l'un des destructeurs de `T` lançait une exception ? Quand bien même vous vous la seriez posée, il est probable que vous n'eussiez pas trouvé de réponse satisfaisante ; pour la bonne raison qu'il n'y en a pas : en effet, la norme C++ ne précise pas le comportement que doit adopter l'instruction `delete[]` dans le cas où le destructeur d'un des objets détruits lance une exception. Aussi surprenant que celui puisse paraître aux yeux de certains, un échec de « `delete[] p` » conduira donc à un état indéterminé.

Supposons, par exemple, que toutes les constructions des objets se passent correctement mais que la destruction du cinquième objet échoue. L'opérateur `delete[]` se retrouve alors face au même dilemme vu précédemment : soit il continue la désallocation des quatre objets restants, évitant ainsi les fuites mémoires, mais interdisant la retransmission correcte des exceptions à l'appelant (en particulier, s'il se produisait

1. À moins que le destructeur de `T` ne libère pas correctement toutes les ressources mémoires dynamique le concernant au moment où il lance une exception ; mais, dans ce cas, c'est la robustesse de `T` aux exceptions qui est en cause, pas celle de `destroy()`.

plusieurs exceptions, seule une pourrait être retransmise) ; soit il retransmet immédiatement l'exception survenue, laissant en mémoire des objets impossibles à détruire.

Autre cas de figure : supposons que la construction d'un objet, le cinquième par exemple, échoue. Les destructeurs des objets déjà construits sont alors appelés, dans l'ordre inverse de la construction (le quatrième, puis le troisième, etc.). Mais que se passe t-il si l'un des destructeurs appelés lance à son tour une exception ? Faut-il continuer à détruire les autres objets, opérations risquant, à leur tour, de lancer d'autres exceptions ? Faut-il retransmettre immédiatement l'exception en laissant des objets non détruits en mémoire ? Une nouvelle fois, la situation est inextricable.

Si un destructeur est susceptible de lancer des exceptions, il n'est pas possible d'avoir recours aux opérateurs `new[]` et `delete[]`, sous peine d'obtenir du code non robuste. La conclusion est simple : *n'implémentez jamais un destructeur susceptible de générer ou de retransmettre une exception*[1]. Tous les destructeurs doivent être implémentés comme s'ils étaient déclarés avec l'attribut `throw()` ; c'est-à-dire qu'ils ne doivent, en aucun cas, propager la moindre exception.

Recommandation

Ne laissez jamais une exception s'échapper d'un destructeur ou d'un opérateur `delete()` ou `delete[]()` redéfini, sous peine de compromettre la robustesse de votre code. Implémentez toutes les fonctions de ce type comme si elles étaient dotées de l'attribut `throw()`.

Ceci peut paraître décevant, quand on pense que les exceptions ont été originalement introduites en C++ pour permettre aux constructeurs et aux destructeurs de signaler des erreurs – étant donné qu'ils ne disposent pas de valeur de retour. En réalité, l'utilité pratique est principalement restreinte aux constructeurs – les destructeurs n'ayant, en général, que très peu de risques de mal s'exécuter. Il est donc peu dommageable d'interdire à un destructeur de transmettre des exceptions. Quant au cas de constructeurs lançant des exceptions, il est heureusement parfaitement géré en C++, même dans le cas de la construction de tableaux dynamiques d'objets avec `new[]`.

Du bon usage des exceptions

Ce problème et les précédents ont mis en avant un certain nombre de risques pouvant se présenter en cas de mauvaise utilisation des exceptions. N'en concluez pas que les exceptions sont dangereuses ! Utilisées correctement, elles restent un outil extrêmement performant de remontée d'erreurs. Il suffit de respecter un certain nombre de

1. Lors de l'adoption de la version finale de la norme C++ en novembre 1997 à Morristown (New Jersey), le comité de normalisation C++ a notamment édicté qu' « aucun des destructeurs défini dans la bibliothèque standard C++ ne doit lancer d'exception » et que « les conteneurs standards ne doivent pas pouvoir être instanciés avec une classe dont le destructeur est susceptible de lancer des exceptions ». Il existe également un certain nombre de propriétés supplémentaires que nous allons voir dans le problème suivant.

règles simples : encapsuler le code gérant les ressources externes (zones dynamiques de mémoire, connexions à des bases de données), effectuer les opérations sur des objets temporaires avant de les « valider » en mettant à jour l'état de l'objet principal, ne jamais laisser une exception s'échapper d'un destructeur.

Recommandation

Pour une bonne robustesse de votre code aux exceptions, respectez toujours les trois règles suivantes :
(1) Ne laissez jamais une exception s'échapper d'un destructeur ou d'un opérateur `delete()` ou `delete[]()` redéfini. Implémentez toutes les fonctions de ce type comme si elles étaient dotées de l'attribut `throw()`.
(2) Encapsulez le code gérant les ressources externes (zones de mémoire dynamique, connexions à des bases de données,...) en le séparant du code utilisant ces ressources.
(3) Lorsque vous implémentez une fonction destinée à réaliser une certaine opération, séparez les instructions effectuant l'opération elle-même (susceptibles de générer une exception), du code réalisant la validation de cette opération, constitué d'instructions unitaires (ne risquant pas de générer des exceptions).

Pb n° 17. Écrire du code robuste aux exceptions (10e partie) Difficulté : 9 1/2

Pour conclure cette série de problèmes, nous nous intéressons à la robustesse aux exceptions de la bibliothèque standard C++.

Un travail particulièrement important a été effectué sur la bibliothèque standard C++ et de sa robustesse aux exceptions. Les principaux artisans en sont Dave Abrahams, Greg Colvin et Matt Austern, qui ont travaillé durement – qu'ils en soient remerciés ici – pour produire une spécification complète sur le sujet quelques jours seulement avant l'adoption finale de la norme ISO WG21/ANSI J16 en novembre 1997 à Morristown (New Jersey).

La question à laquelle il vous est proposé de répondre est la suivante : dans quelle mesure la bibliothèque standard C++ est-elle robuste aux exceptions ? Argumentez votre réponse.

Solution

Pour répondre succinctement à la question : les conteneurs de la bibliothèque C++ standard sont parfaitement robustes aux exceptions – nous en verrons les justifications ci-dessous ; le reste de la bibliothèque (en particulier, les `iostreams` et les `facets`) ne garantissent qu'une robustesse minimale.

Robustesse aux exceptions des conteneurs standards

Tous les conteneurs de la bibliothèque standard sont implémentés de manière à garantir qu'ils se comportent correctement en présence d'exceptions et transmettent toutes les exceptions à l'appelant :

- Tous les itérateurs obtenus à partir d'un conteneur standard sont robustes aux exceptions et peuvent être copiés sans risque de génération d'exception.
- Tous les conteneurs standards satisfont aux garanties de base de la robustesse aux exceptions : ils peuvent être détruits sans générer d'exceptions et restent toujours dans un état cohérent, même en présence d'exceptions.
- Pour rendre possible le point précédent, la bibliothèque standard requiert d'un certain nombre de fonctions qu'elles ne génèrent pas d'exceptions, en aucune circonstance : la fonction `swap()`, dont l'importance pratique a pu être démontrée lors des problèmes précédents ; la fonction `allocator<T>::delete()`, vue au moment de la discussion relative à l'opérateur `delete()` au début de ce chapitre ; ainsi que les destructeurs des types contenus (voir à ce sujet le paragraphe « Les dangers des destructeurs lançant des exceptions »).
- Toutes les fonctions de la bibliothèque standard (à deux exceptions près) garantissent que toutes les zones de mémoire dynamiques seront correctement désallouées et que les objets resteront toujours dans un état cohérent en cas d'exception. Ces fonctions sont toutes implémentées suivant la logique « valider ou annuler » : toute opération doit être entièrement exécutée ou, dans le cas contraire, laisser inchangé l'état des objets manipulés.
- Les deux exceptions à cette dernière règle sont les suivantes : d'une part, l'insertion simultanée de plusieurs éléments n'est pas robuste aux exceptions, et ce quel que soit le type de conteneur ; d'autre part, les opérations d'insertion et de suppression pour les conteneurs `vector<T>` et `deque<T>` ne le sont que si le constructeur de copie et l'opérateur d'affectation du type contenu ne lancent pas d'exception, et ceci qu'il s'agisse d'insertion simple ou multiple. En particulier, l'insertion ou la suppression d'éléments dans un `vector<string>` ou un `vector<vector<int>>` peut ne pas se comporter correctement en présence d'exceptions.

Ces limitations sont le résultat d'un compromis entre performance et sécurité : imposer dans la norme C++ une robustesse parfaite aux exceptions sur les dernières opérations citées aurait été nécessairement pénalisant en terme de performances ; le comité de normalisation a préféré l'éviter. Par conséquent, si une exception se produit alors que vous effectuez une opération de ce type, les objets manipulés risquent de se retrouver dans un état incohérent. Seule solution lorsque vous souhaitez réaliser une insertion multiple ou une insertion/suppression dans `vector<T>` ou `deque<T>` alors que le constructeur de copie et/ou l'opérateur d'affectation de `T` sont susceptibles de générer des exceptions : effectuer les opérations sur une copie du conteneur, puis, une fois qu'on est certain que tout s'est déroulé correctement, échanger la copie et l'original avec la fonction `swap()`, dont on sait qu'elle ne risque pas de générer d'exception.

Pb n° 18. Complexité du code (1re partie) Difficulté : 9

Ce problème aborde une question intéressante, dont la réponse vous paraîtra peut-être surprenante : combien y a-t-il de chemins d'exécution possibles dans une banale fonction de 3 lignes ?

Combien la fonction ci-dessous comporte-t-elle de chemins d'exécution possibles ?

```
String EvaluerSalaireEtRenvoyerNom( Employe e )
{
  if( e.Fonction() == "PDG" || e.Salaire() > 100000 )
  {
     cout<<e.Prenom()<<" "<<e.Nom()<<"est trop payé"<<endl;
  }
  return e.Prenom() + " " + e.Nom();
}
```

On fera les hypothèses suivantes :

- Les fonctions évaluent toujours les paramètres dans le même ordre.
- Les destructions s'effectuent toutes correctement (aucune exception ne peut être générée par un destructeur[1]).
- Les appels de fonctions sont atomiques (une fonction implémentant une opération la réalise soit complètement, soit pas du tout).
- On nomme « chemin d'exécution » une séquence de fonctions exécutée dans un ordre donné et d'une manière donnée.

Solution

Combien la fonction `EvaluerSalaireEtRenvoyerNom` *comporte-t-elle de chemins d'exécution possibles ?*

La réponse est : 23.

Vous avez trouvé...	*Votre note*
3	Moyen
4-14	Bon
15-23	Excellent

Ces 23 chemins d'exécution se divisent en :

- 3 chemins normaux
- 20 chemins « exceptionnels », c'est-à-dire survenant en présence d'exceptions.

1. Ce qui, entre nous, est une bonne chose. Voir, à ce sujet, le paragraphe : « *les dangers des destructeurs lançant des exceptions* » dans le problème n° 16.

Chemins d'exécution normaux

Si vous avez trouvé le bon nombre de chemins d'exécutions normaux, c'est que vous maîtrisez la technique employée par le C++ pour optimiser l'évaluation des expressions conditionnelles :

```
if( e.Fonction() == "PDG" || e.Salaire() > 100000 )
```

1. Si la première partie de l'expression est vérifiée, il n'est pas nécessaire d'évaluer la seconde partie (`e.Salaire()` n'est pas appelée et on passe directement à l'intérieur du bloc `if`). Remarque : cette optimisation n'aurait pas lieu si l'un des deux opérateurs `||` ou > était redéfini, cas que l'on exclut ici.
2. Si la première partie de l'expression n'est pas vérifiée mais que la seconde l'est, les deux parties de l'expression sont évaluées et on exécute l'intérieur du bloc `if`.
3. Si aucune des deux parties de l'expression n'est vérifiée, on évalue les deux parties de l'expression mais on ne rentre pas le bloc `if`.

Chemins d'exécution « exceptionnels »

```
String EvaluerSalaireEtRenvoyerNom( Employe e )
 ^4(bis)^                                ^4^
```

4. L'argument `e` étant passé par valeur, le constructeur de copie `Employe` est appelé. Cette opération est susceptible de générer une exception.

4bis. La fonction retournant une valeur, il est possible que le constructeur de copie de `String`, invoqué pour copier le contenu de l'objet temporaire renvoyé vers le code appelant, échoue et génère une exception. On ne comptera pas ce cas comme un chemin d'exécution supplémentaire, vu qu'il peut être considéré comme externe à la fonction.

```
if( e.Fonction() == "PDG" || e.Salaire() > 100000 )
          ^5^       ^7^  ^6^ ^11^   ^8^  ^10^  ^9^
```

5. L'appel à `Fonction()` peut générer une exception, provoquée soit par son implémentation interne, soit par l'échec de l'opération de copie de l'objet temporaire retourné.
6. La chaîne `"PDG"` est susceptible d'être converti en un objet temporaire (du même type que celui retourné par `Fonction`) afin de pouvoir être transmis en paramètre de l'opérateur == adéquat. Cette opération de construction peut échouer.
7. Si `operator==()` est une fonction redéfinie, elle est susceptible de générer une exception.
8. L'appel à `Salaire()` peut générer une exception, de manière tout à fait similaire à ce qui a été vu au (5).
9. La valeur `100000` peut faire l'objet d'une conversion, laquelle peut générer une exception, de manière similaire à ce qui a été vu au (6).
10. Si elle est redéfinie, la fonction `operator>()` peut générer une exception, de manière similaire à ce qui a été vu au (7).

11. Même remarque pour la fonction `operator||()`.

```
cout<<e.Prenom()<<" "<<e.Nom()<<"est trop payé"<<endl;
    ^12^   ^17^      ^13^ ^14^ ^18^ ^15^            ^16^
```

12-16. Comme le spécifie la norme C++, chacun des cinq appels à `operator<<()` peut être à l'origine d'une exception.

17-18. De manière similaire à (5), les fonctions `Prenom()` et `Nom()` peuvent générer une exception ou renvoyer un objet temporaire dont la copie peut provoquer une exception.

```
return e.Prenom()  +        " "      +    e.Nom();
           ^19^  ^22^      ^21^    ^23^    ^20^
```

19-20. Même remarque que 17-18.

21. Même remarque que 6.

22-23. Même remarque que pour 7, appliquée à l'opérateur `+`.

Recommandation

Sachez où et quand des exceptions sont susceptibles d'être générées.

Nous avons pu voir dans ce problème dans quelle mesure il est parfois complexe de maîtriser parfaitement tous les chemins d'exécution possibles. Nous allons voir dans le problème suivant quel impact peut avoir cette complexité invisible sur la fiabilité et la facilité de test d'un code.

Pb n° 19. Complexité du code (2e partie) Difficulté : 7

Dans ce problème, il s'agit de modifier le code du problème précédent de manière à le rendre robuste aux exceptions.

La fonction ci-dessous, étudiée lors du problème précédent, est-elle robuste aux exceptions ? Autrement dit, se comporte-t-elle correctement en présence d'exceptions et transmet-elle toutes les exceptions au code appelant ?

```
String EvaluerSalaireEtRenvoyerNom( Employe e )
{
  if( e.Fonction() == "PDG" || e.Salaire() > 100000 )
  {
     cout<<e.Prenom()<<" "<<e.Nom()<<"est trop payé"<<endl;
  }
  return e.Prenom() + " " + e.Nom();
}
```

Argumentez-votre réponse. Si la fonction n'est pas robuste aux exceptions, peut-on au moins assurer qu'en cas d'exception, il ne se produira pas de fuite mémoire et/ou que l'état du programme sera inchangé ? Peut-on, à l'inverse, assurer que cette fonction ne génèrera jamais d'exception ?

On considérera que toutes les fonctions appelées et les objets utilisés (incluant les objets temporaires) sont robustes aux exceptions (c'est-à-dire qu'ils peuvent générer des exceptions, mais se comportent correctement dans ce cas-là et, en particulier, ne laissent pas de ressources mémoires non désallouées).

Solution

Une remarque préliminaire au sujet des hypothèses de travail : nous avons considéré que toutes les fonctions appelées étaient robustes aux exceptions ; en pratique, ce n'est pas toujours le cas pour les flux. Les fonctions `operator<<` des classes « flux » peuvent en effet avoir des effets secondaires indésirables : il peut tout à fait arriver qu'elles traitent une partie de la chaîne qui leur est passée en paramètre, puis échouent sur une exception. Nous négligerons ce type de problèmes dans la suite.

Reprenons le code de la fonction `EvaluerSalaireEtRenvoyerNom` et voyons si elle est robuste aux exceptions :

```
String EvaluerSalaireEtRenvoyerNom( Employe e )
{
  if( e.Fonction() == "PDG" || e.Salaire() > 100000 )
  {
     cout<<e.Prenom()<<" "<<e.Nom()<<"est trop payé"<<endl;
  }
  return e.Prenom() + " " + e.Nom();
}
```

Telle qu'elle est écrite, cette fonction garantit de ne pas provoquer de fuite mémoire en présence d'exceptions. En revanche, elle n'assure pas que l'état du programme sera inchangé si une exception se produit (autrement dit, si la fonction échoue sur une exception, elle pourra ne s'être exécutée que « partiellement »).

En effet :

- Si une exception se produit entre le début et la fin de la transmission des informations à `cout` (par exemple, si le quatrième opérateur << génère une exception), le message n'aura été que partiellement affiché[1].
- Si le message s'affiche correctement mais qu'une exception se produit lors de la suite de l'exécution de la fonction (lors de la préparation de la valeur de retour, par exemple), alors la fonction aura affiché un message alors qu'elle ne se sera pas exécutée entièrement.

1. Ce n'est pas dramatique dans le cas d'un message affiché à l'écran, mais pourrait l'être nettement plus, par exemple, dans le cas d'une transaction bancaire.

Enfin, la fonction ne garantit évidemment pas qu'elle ne laissera échapper aucune exception, au contraire : chacune des opérations effectuées en interne est susceptible de générer une exception et il n'y a aucun bloc `try/catch` !

Recommandation

Connaissez et appliquez les principes généraux garantissant le bon comportement d'un programme en présence d'exceptions.

Tentons maintenant de modifier la fonction de manière à assurer son caractère atomique (exécution complète ou sans effet, mais pas partielle).

Voici une première proposition :

```
// 1ère proposition : est-ce mieux ?

String EvaluerSalaireEtRenvoyerNom( Employe e )
{
  String result = e.Prenom() + " " + e.Nom();

  if( e.Fonction() == "PDG" || e.Salaire() > 100000 )
  {
     String message = result+"est trop payé\n";
     cout << message;
  }
  return result;
}
```

Cette solution n'est pas mauvaise. Nous limitons totalement le risque d'affichage partiel, grâce à l'emploi de la variable `message` et du caractère « `\n` » au lieu de « `endl` ». Il subsiste pourtant un problème, illustré dans le code client suivant :

```
// Un problème
//
String leNom ;
leNom = EvaluerSalaireEtRenvoyerNom( Robert ) ;
```

La fonction renvoyant un objet par valeur, le constructeur de copie de `String` est appelé pour copier la valeur retournée dans l'objet « `leNom` ». Si cette copie échoue, la fonction se sera exécutée mais le résultat aura été irrémédiablement perdu ; on ne peut donc pas considérer qu'il s'agit là d'une fonction atomique.

Une deuxième solution serait d'utiliser une référence non constante passée en paramètre plutôt qu'une valeur de retour :

```
// 2ème proposition : est-ce la bonne ?

void EvaluerSalaireEtRenvoyerNom( Employe e, String& r )
{
  String result = e.Prenom() + " " + e.Nom();

                if( e.Fonction() == "PDG" || e.Salaire() > 100000 )
  {
```

```
        String message = result+"est trop payé\n";
       cout << message;
   }
  r = result;
}
```

Cette solution peut paraître meilleure, du fait qu'elle évite l'utilisation d'une valeur de retour et élimine, de ce fait même, les risques liés à la copie. En réalité, elle pose exactement le même problème que la précédente, déplacé à un autre endroit : c'est maintenant l'opération d'affectation qui risque d'échouer sur une exception, ce qui aura également pour conséquence une exécution partielle de la fonction.

Pour finalement résoudre le problème, il faut avoir recours à un pointeur pointant vers un objet `String` alloué dynamiquement ou, mieux encore, à un pointeur automatique (`auto_ptr`) :

```
// 3ème proposition : la dernière et la bonne !

auto_ptr<String>
EvaluerSalaireEtRenvoyerNom( Employe e )
{
  auto_ptr<String> result
      = new String (e.Prenom() + " " + e.Nom());

  if( e.Fonction() == "PDG" || e.Salaire() > 100000 )
  {
     String message = result+"est trop payé\n";
     cout << message;
  }
  return result; // ne risque pas de lancer d'exceptions
}
```

Cette dernière solution est la bonne. Il n'y a plus aucun risque de génération d'exception au moment de la transmission d'une valeur de retour ou au moment d'une affectation. Cette fonction satisfait tout à fait à la technique du « valider ou annuler » : si elle est interrompue par une exception, elle annule totalement les opérations en cours (aucun message affiché à l'écran, aucune valeur reçue par l'appelant). L'utilisation d'un `auto_ptr` comme valeur de retour permet de gérer correctement la transmission de la chaîne de caractère à l'appelant : si l'appelant récupère correctement la valeur de retour, il prendra le contrôle de la chaîne allouée et aura pour responsabilité de la désallouer ; si, au contraire, l'appelant ne récupère pas correctement la valeur de retour, la chaîne orpheline sera automatiquement détruite au moment de la destruction de la variable automatique `result`. Nous n'avons obtenu cette robustesse aux exceptions qu'au prix d'une petite perte de performance due à l'allocation dynamique. Les bénéfices retirés au niveau de la qualité du code en valent largement la peine.

Nous avons finalement réussi, avec la troisième proposition, à obtenir une implémentation de la fonction se comportant de manière atomique[1] (c'est-à-dire dont l'exé-

1. Comme nous l'avons indiqué précédemment, nous négligeons ici le fait que les opérateurs sur les flux peuvent générer des exceptions.

cution est soit totale, soit sans effet). Ceci a été possible car nous avons réussi à assurer que les deux actions externes (affichage d'une chaîne à l'écran, renvoi d'une valeur chaîne au code appelant) de la fonction soient exécutées toutes les deux en cas de réussite ou ne soient ni l'une ni l'autre exécutées en cas d'exception.

Il n'est pas toujours possible d'arriver à ce résultat, surtout avec des fonctions ayant des actions externes trop nombreuses ou trop découplées (par exemple, une fonction réalisant une écriture sur `cout` puis une écriture sur `cerr` poserait problème). Dans ce type de cas, la seule solution viable est souvent la scission de la fonction à traiter en plusieurs fonctions. C'est donc, une nouvelle fois, l'occasion d'insister sur les bénéfices apportés par une bonne modularité du code.

Recommandation

Efforcez-vous de toujours découper votre code de manière à ce que chaque unité d'exécution (chaque module, chaque classe, chaque fonction) ait une responsabilité unique et bien définie.

En conclusion :

L'obtention d'une robustesse importante aux exceptions ne s'effectue généralement (mais pas toujours) qu'au prix d'un sacrifice en terme de performances.

Il est général difficile de rendre « atomique » une fonction ayant plusieurs actions extérieures. En revanche, le problème peut être en général être résolu par le découpage de la fonction à traiter en plusieurs fonctions.

Il n'est pas toujours indispensable de rendre une fonction parfaitement robuste aux exceptions. En l'occurrence, la première des trois propositions présentées plus haut sera a priori amplement suffisante pour la majorité des utilisateurs et permettra de plus d'éviter la légère perte de performance qu'impose la troisième proposition.

Dernière petite remarque : dans tout ce problème, nous avons négligé le fait que la fonction `operator<<()` de la classe `ostream` puisse ne pas se comporter correctement en présence d'exceptions (c'est d'ailleurs le cas pour toutes les classes de type « flux »). Notre hypothèse initiale selon laquelle toutes les fonctions internes à `EvaluerSalaireEtRenvoyerNom` étaient robustes aux exceptions n'était donc pas tout à fait pertinente. Pour être parfaite, notre implémentation devrait gérer par un bloc `try/catch` les exceptions pouvant être générées par l'opérateur `<<` puis les relancer vers l'appelant en ayant, au passage, réinitialisé le statut d'erreur de l'objet `cout`.

Conception de classes, héritage

Pb n° 20. Conception d'une classe — Difficulté : 7

Êtes-vous capable de concevoir des classes techniquement solides ? Le but de cet exemple est de mettre en exergue les détails permettant d'implémenter des classes d'un niveau professionnel, plus robustes et plus faciles à maintenir.

Étudiez le code suivant. Il contient un certain nombre d'imperfections et d'erreurs. Lesquelles ? Comment peut-on les améliorer ?

```
class Complex
{
public:
  Complex( double real, double imaginary = 0 )
    : _real(real), _imaginary(imaginary)
  {
  }
  void operator+ ( Complex other )
  {
    _real = _real + other._real;
    _imaginary = _imaginary + other._imaginary;
  }
  void operator<<( ostream os )
  {
    os << "(" << _real << "," << _imaginary << ")";
  }
  Complex operator++()
  {
    ++_real;
    return *this;
  }
```

```
  Complex operator++( int )
  {
    Complex temp = *this;
    ++_real;
    return temp;
  }
private:
  double _real, _imaginary;
};
```

SOLUTION

Cette classe contient un grand nombre de problèmes, qui ne seront d'ailleurs pas tous traités ici, le but de cet exemple étant plutôt d'aborder les questions relatives à la mécanique de la classe (quel est le prototype à utiliser pour l'opérateur << ? L'opérateur + doit-il être une fonction membre ?…) plutôt que les détails du style de l'implémentation.

Avant même de s'intéresser à ces problèmes, on peut se demander s'il est vraiment opportun de développer une classe `Complex`, alors qu'une classe équivalente existe dans la bibliothèque standard du C++. Plutôt que de perdre du temps à améliorer la classe `Complex` présentée ici, il serait plus judicieux de réutiliser le modèle de classe existant (`std ::complex`) qui a toutes les chances d'être bien plus exempt d'erreurs et plus optimisé que notre classe – étant donné le fait qu'il a été développé depuis un grand nombre d'années et déjà utilisé par un grand nombre de développeurs.

Recommandation

Réutilisez le code existant– surtout celui de la librairie standard. C'est plus rapide, plus facile et plus sûr.

Cette remarque préliminaire étant faite, intéressons-nous tout de même aux problèmes de notre classe `Complex`.

1. Le constructeur autorise une conversion implicite.

```
Complex( double real, double imaginary = 0 )
  : _real(real), _imaginary(imaginary)
{
}
```

Son second paramètre ayant une valeur par défaut, ce constructeur peut être utilisé pour effectuer une conversion implicite de `double` vers `Complex`. Ca ne prête peut-être pas à conséquence dans ce cas, mais, comme nous l'avons vu dans le problème n° 6, les conversions implicites non contrôlées peuvent induire des problèmes. Il est préférable de prendre l'habitude de spécifier par défaut le mot-clé `explicit` pour un constructeur, à moins d'être certain que la conversion implicite qu'il induit est sans risque (voir également le problème n° 19, relatif aux conversions implicites).

CONSEIL

Prenez garde aux variables temporaires générées de manière cachée par les conversions implicites. Pour cela, spécifiez, lorsque cela est possible, le mot-clé « `explicit` » pour les constructeurs et évitez d'implémenter des opérateurs de conversions.

2. La fonction `operator+` n'est pas optimisée.

```
void operator+ ( Complex other)
  {
      _real = _real + other._real;
      _imaginary = _imaginary + other._imaginary;
  }
```

Il aurait fallu, pour une meilleure efficacité, passer une référence constante plutôt qu'une valeur.

Recommandation

Passez les objets par référence constante (`const&`) plutôt que par valeur.

D'autre part, concernant l'addition des parties réelles et imaginaires, il aurait été préférable d'utiliser `a+=b` plutôt que `a = a+b` (le gain de performance n'est pas énorme lorsqu'il s'agit de `double`s, néanmoins, l'amélioration peut être notable pour des variables de type objet).

CONSEIL

Utilisez « `a op= b` » plutôt que « `a = a op b` » (`op` désignant n'importe quel opérateur). C'est plus clair et souvent plus efficace.

L'opérateur += est plus efficace car il travaille directement sur l'objet situé à gauche de l'opérateur et renvoie uniquement une référence, alors que l'opérateur + renvoie un objet temporaire, comme l'indique l'exemple suivant :

```
T& T::operator+=( const T& other )
{
  //...
  return *this;
}
const T operator+( const T& a, const T& b )
{
  T temp( a );
  temp += b;
  return temp;
}
```

Notez bien la relation entre les deux opérateurs + et +=. Le premier doit faire appel au second dans son implémentation. Ainsi, le code sera plus simple à écrire et à main-

tenir (les deux opérateurs feront la même chose et la probabilité qu'ils divergent au cours de la maintenance sera moindre).

Recommandation

Si vous implémentez un opérateur simple (par exemple, `operator+`), implémentez toujours l'opérateur d'affectation correspondant (par exemple, `operator+=`) et basez l'implémentation du premier sur le second. D'une manière générale, deux opérateurs `op` et `op=` doivent toujours se comporter de manière identique (`op` désignant n'importe quel opérateur).

3. La fonction `operator+` ne devrait pas être une fonction membre.

```
void operator+ ( Complex other )
{
  _real = _real + other._real;
  _imaginary = _imaginary + other._imaginary;
}
```

Dans notre exemple, le fait qu'`operator+` soit une fonction membre rend impossible un certain type d'additions entre `Complex` et `double`, opérations que l'utilisateur serait pourtant naturellement tenté de faire, étant donné que la classe `Complex` autorise les conversions implicites à partir du type `double`. En effet, s'il est possible d'écrire « `a=b+1.0` », il n'est pas possible d'écrire « `a= 1.0 + b` » car la fonction membre `operator+` attend un `Complex` comme opérande situé à gauche du signe +.

Pour bien faire, il serait plus judicieux de définir deux fonctions globales : `operator+(const Complex&, double)` et `operator+(double, const Complex&)`, au lieu de la fonction membre définie ici.

CONSEIL

Quelques règles pour déterminer si un opérateur doit être implémenté ou non en tant que fonction membre :

- Les opérateurs =,`()`,`[]` et `->` doivent toujours être des fonctions membres.
- Les opérateurs `new`, `new[]`,`delete` et `delete[]` redéfinis pour une classe doivent toujours être des fonctions membres statiques de cette classe.
- Pour toutes les autres fonctions :
  - Si l'opérateur a besoin d'une conversion de type pour son opérande gauche, s'il s'agit d'un opérateur s'appliquant à un flux d'entrée-sortie (<< ou >>) ou si l'opérateur ne fait appel qu'à l'interface publique de la classe, implémentez-le en tant que fonction globale (non-membre), éventuellement amie (déclarées `friend`), dans les deux premiers cas.
  - Si l'opérateur doit avoir un comportement virtuel, ajoutez une fonction virtuelle à la classe et implémentez l'opérateur en tant que fonction membre faisant appel à cette fonction virtuelle.
  - Sinon, si l'opérateur ne rentrant dans aucun des cas précédents, implémentez-le en tant que fonction membre.

4. La fonction `operator+` ne devrait pas modifier la valeur de l'objet et devrait renvoyer un objet `Complex` temporaire contenant la somme.

```
void operator+ ( Complex other )
{
   _real = _real + other._real;
   _imaginary = _imaginary + other._imaginary;
}
```

Pour être encore plus précis, cette fonction devrait renvoyer un « `const Complex` » (et non pas simplement un « `Complex` »), afin d'interdire les instructions du type « `a+b=c` ».

5. Il faudrait définir la fonction `operator+=`. En effet, ainsi que nous l'avons mentionné plus haut : « si vous implémentez un opérateur simple, implémentez toujours l'opérateur d'affectation correspondant ». Étant donné qu'`operator+` a été défini, il faudrait donc également définir `operator+=` (en basant d'ailleurs son implémentation sur celle d'`operator+`).

6. La fonction `operator<<` ne devrait pas être membre de la classe.

```
void operator<<( ostream os )
{
   os << "(" << _real << "," << _imaginary << ")";
}
```

La situation est ici similaire au cas de la fonction `operator+`, traité au point (3). Cette fonction devrait plutôt être globale (non membre), avec les paramètres suivants : « `(ostream&, const Complex&)` ». En pratique, on définit d'ailleurs généralement une fonction membre – souvent virtuelle – nommée `Print()`sur laquelle on base l'implémentation de la fonction globale `operator<<`.

Pour bien faire, il faudrait également s'assurer que l'opérateur << fonctionne avec les manipulateurs de formatage classiques de la bibliothèque standard. Référez-vous à la documentation de `iostream` pour plus de détails.

7. La fonction operator<< devrait retourner un `ostream&`, afin d'autoriser le chaînage d'appel (par exemple : « `cout << a << b` »).

Recommandation

Les fonctions `operator<<` et `operator >>` doivent toujours renvoyer une référence vers un flux.

8. Le type de retour de l'opérateur de pré-incrémentation est incorrect.

```
Complex operator++()
{
   ++_real;
   return *this;
}
```

Cette fonction devrait renvoyer une référence non constante vers un `Complex`, sous peine de provoquer un comportement inattendu de certaines utilisations de l'opérateur et de pénaliser inutilement la performance du code.

9. Le type de retour de l'opérateur de post-incrémentation est incorrect.

```
Complex operator++( int )
{
  Complex temp = *this;
  ++_real;
  return temp;
}
```

Cette fonction devrait renvoyer une référence constante vers un `Complex`. En interdisant que l'objet renvoyé soit modifié, on s'assure que les instructions du type « `a++++` », qui ne réalisent pas ce à quoi l'utilisateur inexpérimenté s'attend, sont interdites par le compilateur.

10. L'implémentation de l'opérateur de post-incrémentation devrait être basée sur celle de l'opérateur de pré-incrémentation (voir le problème n° 6 au sujet de l'implémentation idéale d'un opérateur de post-incrémentation).

Recommandation

Basez toujours l'implémentation de l'opérateur de post-incrémentation sur celle de l'opérateur de pré-incrémentation, afin de réduire au maximum les risques d'incohérence dans le programme.

11. Évitez d'utiliser des noms réservés

```
private :
double _real, _imaginary ;
```

L'habitude, malheureusement parfois conseillée dans certains livres comme *Design Patterns* (Gamma95), consistant à faire débuter les noms des variables membres par un caractère souligné (*underscore*) n'est pas recommandable, car la norme C++ réserve justement, pour l'implémentation de la bibliothèque standard, certains noms de variables ayant cette syntaxe. À moins de connaître explicitement ces noms réservés afin de pouvoir les éviter – tâche risquant de s'avérer ardue ! – le plus sage est de proscrire l'utilisation de noms de variables membres commençant par un caractère souligné[1]. En ce

1. Le lecteur attentif pourrait arguer ici que les variables réservées commençant par un caractère souligné sont en l'occurrence des variables non-membres, et que par conséquent cela ne pose pas de problème si le développeur utilise une syntaxe de ce type pour des variables membres. Cette affirmation doit être nuancée car il subsiste un risque fort lié à l'utilisation, dans l'implémentation de la bibliothèque standard, de macros `#define` portant des noms commençant par un caractère souligné. Le compilateur ne contrôlant pas la portée des macros, même une macro écrite dans le but d'implémenter une variable membre peut entrer en conflit avec une variable non membre du même nom. L'idéal est donc d'éviter systématiquement les caractéres soulignés au début des noms de variables.

qui me concerne, j'ai l'habitude d'utiliser des noms de variables *terminés* par un caractère souligné.

Pour finir, voici une version corrigée de la classe `Complex`, tenant compte de toutes les remarques ci-dessus :

```
class Complex
{
  public:
    explicit Complex( double real, double imaginary = 0 )
      : real_(real), imaginary_(imaginary)
    {
    }

    Complex& operator+=( const Complex& other )
    {
      real_ += other.real_;
      imaginary_ += other.imaginary_;
      return *this;
    }

    Complex& operator++()
    {
      ++real_;
      return *this;
    }

    const Complex operator++( int )
    {
      Complex temp( *this );
      ++*this;
      return temp;
    }

    ostream& Print( ostream& os ) const
    {
      return os << "(" << real_ << "," << imaginary_ << ")";
    }

  private:
    double real_, imaginary_;
  };

  const Complex operator+( const Complex& lhs, const Complex& rhs )
  {
    Complex ret( lhs );
    ret += rhs;
    return ret;
  }

  ostream& operator<<( ostream& os, const Complex& c )
  {
    return c.Print(os);
  }
}
```

Pb n° 21. Redéfinition de fonctions virtuelles Difficulté : 6

Le mécanisme des fonctions virtuelles est très pratique et très souvent utilisé en C++. Néanmoins, mal maîtrisé, il peut occasionner des comportements inattendus pouvant faire perdre beaucoup de temps au développeur inexpérimenté. Ce problème vous permet de tester votre degré de connaissance des subtilités liées aux fonctions virtuelles.

Examinez le code suivant, visiblement écrit par un développeur voulant tester l'utilisation des fonctions virtuelles. Quel est, à votre avis, le résultat auquel ce développeur s'attendait ? Quel est le résultat qui va se produire en réalité?

```
#include <iostream>
#include <complex>
using namespace std;
class Base
{
public:
  virtual void f( int );
  virtual void f( double );
  virtual void g( int i = 10 );
};
void Base::f( int )
{
  cout << "Base::f(int)" << endl;
}
void Base::f( double )
{
  cout << "Base::f(double)" << endl;
}
void Base::g( int i )
{
  cout << i << endl;
}
class Derived: public Base
{
public:
  void f( complex<double> );
  void g( int i = 20 );
};
void Derived::f( complex<double> )
{
  cout << "Derived::f(complex)" << endl;
}
void Derived::g( int i )
{
  cout << "Derived::g() " << i << endl;
}
void main()
{
  Base    b;
```

```
  Derived d;
  Base*   pb = new Derived;
  b.f(1.0);
  d.f(1.0);
  pb->f(1.0);
  b.g();
  d.g();
  pb->g();
  delete pb;
}
```

Solution

Avant d'aborder les fonctions virtuelles proprement dites, voyons déjà une petite remarque de style et une erreur grave.

1. « `void main()` » n'est pas du C++ standard, et donc, n'est pas portable.

```
void main()
```

La fonction `main()` apparaît sous cette forme dans de nombreux ouvrages. Bien que certains auteurs indiquent que cette écriture fait partie intégrante du C++ standard, ce n'est malheureusement pas le cas – comme cela ne l'a jamais été non plus pour le C, d'ailleurs.

Bien que « `void main()` » ne soit pas une forme autorisée par la norme C++, de nombreux compilateurs l'acceptent. Abuser du laxisme des compilateurs de ce type, c'est écrire du code qui risque de ne pas être portable sur d'autres compilateurs. L'idéal est donc de se limiter aux deux formes officielles – et portables – de `main` :

```
int main()
int main(int argc, char* argv[])
```

Une petite remarque au sujet de l'utilisation de `return` dans la fonction `main()`. Dans notre exemple, aucune instruction `return` n'est utilisée – ce qui est normal, étant donné que le type de retour de notre fonction est `void`. Néanmoins, il faut savoir que, même dans le cas d'une fonction `main()` correcte renvoyant un `int`, il n'est pas obligatoire de retourner explicitement une valeur. La norme C++ spécifie qu'en l'absence d'instruction `return` explicite, le compilateur doit tout de même accepter le code et ajouter un « `return 0` » implicite. Ceci étant, il n'est pas recommandé de prendre cette habitude de programmation car, indépendamment du fait qu'il peut être utile de renvoyer des codes d'erreurs à l'appelant, il s'avère qu'en pratique, un grand nombre de compilateurs n'implémentent pas cette règle et émettent des avertissements lorsque aucun `return` n'est présent dans la fonction `main()`.

2. « `delete pb ;` » est une instruction dangereuse.

```
delete pb ;
```

Cette instruction semble anodine. Elle le serait effectivement si la classe B comportait un destructeur virtuel. Comme ce n'est pas le cas, cette tentative de destruction

d'un objet de classe `Derived` référencé par un pointeur de type `Base*` va faire appel au mauvais destructeur – celui de la classe de base – ce qui va avoir pour conséquence d'appeler l'opérateur `delete()` en spécifiant une taille d'objet incorrecte.

Recommandation

Dotez systématiquement toute classe de base d'un destructeur virtuel (à moins d'être absolument certain que personne ne tentera jamais de détruire un objet dérivé, référencé à partir d'un pointeur sur cette classe).

Passons maintenant aux fonctions virtuelles. Pour cela, précisons avant tout la signification de trois termes couramment utilisés dans ce contexte :

- **Surcharger** *(to overload)* une fonction `f()` signifie définir une nouvelle fonction avec le même nom (`f`) mais des paramètres différents, ayant la même portée que `f()`. Lorsqu'une fonction `f()` est appelée, le compilateur recherche, parmi les fonctions disponibles, celle qui correspond le mieux aux paramètres fournis.
- **Redéfinir** *(to override)* une fonction virtuelle `f()`membre d'une classe signifie définir une nouvelle fonction ayant le même nom (`f`) mais des paramètres différents dans une classe dérivée.
- **Masquer** *(to hide)* une fonction `f()` ayant une portée donnée (classe de base, classe extérieure, espace de nommage) signifie définir une nouvelle fonction portant le même nom (`f`) mais ayant une portée différente, à partir de laquelle la fonction `f()` initiale sera inaccessible (classe dérivée, classe interne, ou autre espace de nommage).

3. La fonction `Derived::f()` masque `Base::f()`

```
void Derived::f(complex<double>)
```

La fonction `Derived::f()` ne surcharge pas `Base::f()`, elle la masque. Il est important de bien comprendre la différence : dans notre exemple, cela signifie que `Base::f(int)` et `Base::f(double)` ne sont plus visibles lorsqu'on se situe à l'intérieur de `Derived` (aussi étonnant que cela puisse paraître, de nombreux compilateurs, même parmi les plus populaires, n'émettent même pas d'avertissement pour signaler les problèmes de ce genre ; c'est donc d'autant plus important que nous insistions ici sur ce point).

Si le masquage les fonctions `f` de `Base` était intentionnel, alors le code de l'exemple est correct[1]. Pour éviter le masquage, souvent source de mauvaises surprises, il

1. Bien qu'en général, pour faciliter la lisibilité et la maintenance du code, il soit plus clair, lorsqu'on masque délibérément un nom de la classe de base, de le spécifier en utilisant une instruction `using` située dans la partie privée de la classe dérivée.

faut rendre explicitement les noms de ces fonctions accessibles depuis l'intérieur de `Derived` en utilisant l'instruction « `using Base::f` ».

Recommandation

Si vous définissez une fonction membre portant le même nom qu'une ou plusieurs fonctions héritées, assurez-vous de rendre accessibles ces fonctions héritées depuis la classe dérivée en utilisant une déclaration de type « `using` », à moins que vous ne soyez certains de vouloir délibérément les masquer.

4. La fonction `Derived::g()` redéfinit la fonction `Base::g()` mais change la valeur par défaut du paramètre

```
void g(int i = 20)
```

Ce n'est vraiment pas une bonne idée ! À moins de vouloir délibérément semer le trouble et la confusion, il n'est vraiment pas utile de changer les valeurs par défaut des paramètres des fonctions héritées que vous redéfinissez (d'ailleurs, on peut se demander si, d'une manière générale, il ne vaut pas mieux préférer la surcharge de fonctions à l'utilisation de paramètres par défaut – mais c'est un autre sujet). En conclusion, ce code est correct et sera accepté par le compilateur, mais il est vraiment recommandé d'éviter pareilles pratiques ! Nous verrons un peu plus loin un exemple des conséquences que la redéfinition de cette valeur peut avoir.

Recommandation

Ne changez jamais les valeurs par défaut des paramètres des fonctions héritées que vous redéfinissez.

Penchons-nous à présent sur la fonction `main()` et voyons si elle effectue vraiment ce à quoi l'auteur de ces lignes s'attendait :

```
void main()
{
  Base    b;
  Derived d;
  Base*   pb = new Derived;
  b.f(1.0);
```

Pas de problème. C'est la fonction `Base::f(double)` qui est appelée.

```
  d.f(1.0);
```

Cette fois, c'est la fonction `Derived::f(complex<double)` qui est appelée, et non pas, comme certains auraient pu s'y attendre, la fonction `Base::f(double)`. Pourquoi, à votre avis ? Souvenez-vous de ce que nous avons vu plus haut : la classe `Derived` ne contient pas de déclaration « `using Base::f()` » ; par conséquent, le fait de définir la fonction `Derived::f(complex<double>)` masque les fonctions `Base::f(int)` et `Base::f(double)`. Ces fonctions sont donc hors de la portée de la classe `Derived` et

ne sont pas prises en compte par l'algorithme de surcharge effectuant le choix de la meilleure fonction `f`.

La classe `complex<double>` implémentant une conversion implicite depuis le type `double`, notre exemple n'est en que plus trompeur : le fait que la fonction `Base::f(double)` soit masquée ne provoque aucune erreur de compilation lorsque le développeur, pensant l'appeler, écrit « `d.f(1.0)` ». En effet, la conversion implicite aidant, le compilateur interprète cet appel de la manière suivante : « `Derived::f(complex<double>(1.0))` ».

```
pb->f(1.0);
```

Bien que `pb` soit un pointeur de type `Base*` pointant vers un objet `Derived`, c'est la fonction `Base::f(double)` qui est appelée. Le fait que cette fonction soit virtuelle ne change rien, la résolution des noms s'effectuant parmi les fonctions déclarées dans `Base` (le type statique ou type du pointeur) et non dans `Derived` (le type dynamique ou type du « pointé »). En l'absence de fonction `Derived::f(double)`, c'est donc la fonction correspondante de `Base` qui est appelée. Dans le même ordre d'idée, l'appel « `pb->f(complex<double>(1.0)) ;` » provoquerait une erreur de compilation car il n'y a pas fonction correspondante dans la classe `Base`.

```
b.g() ;
```

Affiche « `10` », valeur par défaut de la fonction `Base::g(int)`. Pas de surprise.

```
d.g() ;
```

Affiche « `Derived::g() 20` », valeur par défaut de la fonction `Derived::g(int)`. Pas de surprise non plus.

```
pb->g() ;
```

Affiche « `Derived ::g() 10` ».

Cela mérite un petit commentaire. Ce serait donc la fonction `Derived :g(int)` qui serait appelée, mais avec le paramètre par défaut de `Base::g(int)`! Aussi étrange que cela puisse paraître, c'est effectivement ce qui se produit[1]. En effet, comme pour la résolution des noms, c'est le type statique qui est pris en compte par le compilateur pour les paramètres par défaut. Il est donc normal qu'on obtienne ici la valeur « 10 ». En revanche, comme `Base::g(int)` est virtuelle, c'est bien la fonction `Derived :g(int)` qui est appelée.

Si vous avez bien compris ces derniers paragraphes, et notamment tout ce qui concerne le masquage des fonctions et les rôles respectifs des types statique et dynamique, alors vous avez une bonne maîtrise du mécanisme des fonctions virtuelles. Félicitations !

```
delete pb ;
```

1. Bien qu'on puisse, encore une fois, blâmer l'auteur de la classe `Derived` pour son choix inepte de changer la valeur par défaut du paramètre lors de la redéfinition de `g()`.

Pour finir, cette instruction n'effectue pas correctement la désallocation de l'objet vers lequel pointe pb, car le destructeur de `Base` n'est pas virtuel – se référer au début de la solution, où ce point a été vu en détail.

Pb n° 22. Relations entre classes (1re partie) Difficulté : 5

Nous aborderons dans ce problème une faute de conception, malheureusement couramment commise par de trop nombreux développeurs.

Une application réseau utilise deux types de sessions de communication, dont chacune utilise un protocole particulier. Les deux protocoles présentent un certain nombre de similarités (certaines opérations et même certains messages leur sont communs). Le développeur a donc décidé de rassembler ces opérations et messages communs dans une classe de base `BasicProtocol` :

```
class BasicProtocol
{
public:
  BasicProtocol();
  virtual ~BasicProtocol();
  bool BasicMsgA( /*...*/ );
  bool BasicMsgB( /*...*/ );
  bool BasicMsgC( /*...*/ );
};

class Protocol1 : public BasicProtocol
{
public:
  Protocol1();
  ~Protocol1();
  bool DoMsg1( /*...*/ );
  bool DoMsg2( /*...*/ );
  bool DoMsg3( /*...*/ );
  bool DoMsg4( /*...*/ );
};
class Protocol2 : public BasicProtocol
{
public:
  Protocol2();
  ~Protocol2();
  bool DoMsg1( /*...*/ );
  bool DoMsg2( /*...*/ );
  bool DoMsg3( /*...*/ );
  bool DoMsg4( /*...*/ );
  bool DoMsg5( /*...*/ );
};
```

Les fonctions `DoMsg...()` sous-traitent leur travail, autant que faire se peut, aux fonctions `BasicProtocol::Basic()`, mais effectuent elles-mêmes les transmissions de message. Chaque classe pourrait avoir des membres supplémentaires, mais ce n'est pas le sujet de ce problème.

Commentez ce code, notamment au niveau des relations entre les classes. Argumentez vos remarques.

Solution

Le code ci-dessus présente un défaut majeur, malheureusement trop fréquemment rencontré, au niveau de la conception des relations entre classes.

Récapitulons les données du problème : les classes `Protocol1` et `Protocol2` dérivent toutes les deux, de manière publique, de la classe `BasicProtocol`, dont elles utilisent les fonctions, comme le précise l'énoncé :

> Les fonctions `DoMsg...()` sous-traitent leur travail, autant que faire se peut, aux fonctions `BasicProtocol::Basic()`, mais effectuent elles-mêmes les transmissions de message.

Autrement dit, les classes `Protocol1` et `Protocol2` font appel à la classe `BasicProtocol` dans leur implémentation. Cette relation de type « EST-IMPLEMENTE-EN-FONCTION-DE » devrait se traduire, en C++, par l'utilisation d'une dérivation privée ou d'un objet membre (composition). Malheureusement, de trop nombreux développeurs choisissent dans ce cas d'effectuer une dérivation publique, preuve qu'ils effectuent une confusion entre hériter d'une implémentation et hériter d'une interface, alors que ce sont deux concepts bien distincts.

Recommandation

N'abusez pas du mécanisme de dérivation publique. Ce type de dérivation doit être réservé aux cas où la relation entre classe dérivée et classe de base est de type « EST-UN » ou « FONCTIONNE-COMME-UN », au sens du principe de substitution de Liskov.

Cette mauvaise habitude qui consiste à utiliser une dérivation publique pour hériter d'une implémentation finit par donner naissance à de gigantesques hiérarchies de classes, dont la maintenance est complexe et l'utilisation malaisée – les utilisateurs souhaitant utiliser une classe dérivée bien déterminée se voyant obligés d'apprendre les interfaces des nombreuses classes de base. Sans compter l'utilisation accrue de la mémoire due à l'ajout de nombreuses « vtables » (tables de fonctions virtuelles) et l'impact sur les performances dû aux nombreuses indirections lors des appels des fonctions virtuelles. Ayez à l'esprit ces quelques paragraphes chaque fois que vous concevez un programme C++ : les lourdes hiérarchies de classes sont une mauvaise habitude; rarement nécessaires, elles présentent généralement plus d'inconvénients que d'avantages. Ne soyez pas victime du discours selon lequel « la programmation

orientée objet repose avant tout sur la notion d'héritage ». À titre de contre exemple, étudiez l'architecture de la bibliothèque standard et vous verrez.

Recommandation

N'utilisez pas l'héritage public pour réutiliser le code de la classe de base; utilisez l'héritage public pour être réutilisé par des objets externes utilisant le polymorphisme de la classe de base.[a]

a Je remercie Marshall Cline, co-auteur du désormais classique C++ FAQs (Cline 95), à qui j'ai emprunté ce conseil.

Voici d'ailleurs divers indices qui prouvent bien l'existence d'un défaut de conception :

1. `BasicProtocol` n'a pas de fonctions virtuelles (à part le destructeur, dont nous parlerons plus loin).[1] Ceci indique que cette classe n'est pas conçue pour être utilisée de manière polymorphique, ce qui limite grandement l'intérêt d'une dérivation publique à partir de cette classe.

2. `BasicProtocol` n'a aucun membre protégé, autrement dit, elle n'a aucune « interface de dérivation », ce qui limite grandement l'intérêt d'une dérivation à partir de cette classe, quelle qu'elle soit d'ailleurs (publique ou privée).

3. `BasicProtocol` implémente un certain nombre de fonctionnalités utilisées par les classes dérivées. En revanche, elle n'effectue pas le même type de travail que ses dérivées puisque, comme l'indique l'énoncé, elle ne réalise pas de transmission de messages. Autrement dit, la relation entre un objet `BasicProtocol` et un objet `Protocol` n'est ni du type FONCTIONNE-COMME ni du type EST-UTILISABLE-EN-TANT-QUE. L'héritage public ne doit être utilisé que dans un cas et un seul : la modélisation d'une relation EST-UN obéissant au principe de substitution de Liskov (c'est-à-dire FONCTIONNE-COMME ou EST-UTILISABLE-EN-TANT-QUE).[2]

4. Les classes `Protocol1` et `Protocol2` font uniquement appel à l'interface publique de `BasicProtocol`. En d'autres termes, elles n'exploitent pas du tout leur caractère de classe dérivée et pourraient tout aussi bien utiliser `BasicProtocol` sous la forme d'un objet externe.

1. Même si `BasicProtocol` était elle-même dérivée d'une autre classe de base contenant, elle, des fonctions virtuelles, nous serions arrivé à la même conclusion : c'est cette autre classe de base qui aurait pu être utilisée de manière polymorphique et c'est donc de cette classe qu'il aurait fallu, dans ce cas, dériver les classes `Protocol`.
2. Il faut reconnaître qu'il n'est pas toujours nécessaire d'adopter l'approche puriste « une responsabilité par classe » et que, parfois, en effectuant une dérivation publique depuis une classe de base pour hériter d'une interface, on hérite également d'une implémentation. Néanmoins, cette approche est pratiquement toujours possible à réaliser (voir le problème suivant).

En conclusion, `BasicProtocol` n'est clairement pas conçue pour servir de classe de base. Par conséquent, il serait plus judicieux de lui donner un nom du type `MessageCreator` ou `MessageHelper`, d'ôter au destructeur son attribut « `virtual` » et de revoir la relation de cette classe avec les classes `Protocol`. Pour modéliser cette relation (« EST-IMPLEMENTE-EN-FONCTION-DE »), il y a deux possibilités : héritage privé ou objet membre (composition). Laquelle choisir ? La réponse est claire :

Recommandation

Pour modéliser une relation du type « EST-IMPLEMENTE-EN-FONCTION-DE », préférez systématiquement l'utilisation d'un objet membre (composition). N'utilisez l'héritage privé que lorsque c'est absolument nécessaire – besoin d'accéder aux membres protégés ou de redéfinir des fonctions virtuelles. N'utilisez jamais l'héritage public pour réutiliser un code existant.

L'utilisation d'objets membres simplifie la relation entre classes – la classe « cliente » n'ayant accès qu'à l'interface publique de la classe « serveur ». Prenez l'habitude d'éviter l'héritage lorsqu'il n'est pas indispensable. Votre code n'en sera que plus clair et facile à lire – autrement dit, moins cher à maintenir.

Pb n° 23. Relations entre classes (2e partie) Difficulté : 6

Les schémas de conception (design patterns) jouent un rôle important dans l'écriture de code réutilisable. Saurez-vous reconnaître, et éventuellement améliorer, les schémas de conception utilisés dans ce problème ?

Considérons un programme manipulant une base de données, qui effectue fréquemment des opérations sur tous les enregistrements (ou sur un ensemble d'enregistrements) de tables de la base. Le programme fonctionne de la manière suivante : il effectue une première passe, en lecture seule, sur l'ensemble de la table, afin d'établir la liste des enregistrements à traiter. Puis, s'appuyant sur cette liste, conservée en mémoire, il effectue une deuxième passe pour réaliser effectivement les opérations sur les enregistrements concernés.

Le développeur ayant réalisé ce programme s'est appuyé sur une classe abstraite `GenericTableAlgorithm`, qui fournit un canevas réutilisable pour chacune des opérations faites sur les tables : cette classe implémente la logique des deux passes – la première établissant une liste d'enregistrements à traiter, la seconde parcourant cette liste en effectuant des opérations sur ces enregistrements. Les fonctions spécifiques à chaque manipulation de la base (suivant quels critères déterminer si un enregistrement est à traiter ou non, quelle opération réaliser sur chaque enregistrement) sont implémentées dans autant de classes dérivées de cette classe abstraite.

```
//-------------------------------------------------
// Fichier gta.h (GenericTableAlgorithm)
//-------------------------------------------------
class GenericTableAlgorithm
{
public:
  GenericTableAlgorithm( const string& table );
  virtual ~GenericTableAlgorithm();

  // La fonction Process() parcourt tous les enregistrements
  // de la table, appelant la fonction Filter() pour chacun,
  // afin de déterminer s'il doit être traité ou non.
  // Puis, dans une deuxième passe, elle appelle ProcessRow()
  // pour chacun des enregistrements de la liste établie
  // lors de la première passe.
  // Elle renvoie « true » si le traitement s'est bien passé.

  bool Process();

private:
  // La fonction Filter() examine si l'enregistrement
  //  passé en paramètre est à traiter ou non.
  // Le comportement par défaut est de renvoyer « true »
  // (inclusion de tous les enregistrements)
  //
  virtual bool Filter( const Record& );


  // La fonction ProcessRow() est appelée pour chacun
  // des enregistrements devant être traités.
  // Cette fonction étant virtuelle pure, elle doit
  // obligatoire être redéfinie dans les classes dérivées,
  // qui implémentent les détails de l'opération à réaliser.
  // (Note: avec cette architecture, chaque enregistrement
  // à traiter sera donc lu deux fois; ce n'est peut être
  // pas optimal du point de vue des performances, mais,
  // pour l'exercice, considérons que c'est nécessaire)

  virtual bool ProcessRow( const PrimaryKey& ) =0;

  struct GenericTableAlgorithmImpl* pimpl_; // Implémentation
};
```

Voici un exemple de code client créant et utilisant une classe dérivée de `GenericTableAlgorithm` :

```
class MonAlgorithme : public GenericTableAlgorithm
{
  // ... Redéfinir Filter() and ProcessRow() afin
  //     d'implémenter des opérations spécifiques...
};

int main()
{
```

```
    MonAlgorithme a( "Client" );
    a.Process();
}
```

1. Ce programme, relativement bien conçu, implémente un schéma de conception bien connu. Lequel ? En quoi est-il utile ici ?

2. Commentez la manière dont le passage de la conception – qu'on ne remettra pas en cause – à l'implémentation a été réalisé. L'auriez-vous réalisé différemment ? Quel est le rôle de l'objet membre `pimpl_`?

3. À bien y réfléchir, l'architecture de ce programme pourrait être améliorée. Quels sont les rôles joués par `GenericTableAlgorithm` ? S'ils sont multiples, ne pourrait-on pas les implémenter dans plusieurs classes ? Quelles conséquences cela aurait-il sur la réutilisabilité et l'extensibilité de ces classes ?

Solution

Prenons les questions dans l'ordre :

1. Ce programme, relativement bien conçu, implémente un schéma de conception bien connu. Lequel ? En quoi est-il utile ici ?

Le schéma implémenté ici est la « Méthode du Modèle » (*Template Method*) [Gamma95], qui, rappelons-le, n'a aucun lien avec les modèles C++. Ce schéma est utile dans les cas où il s'agit d'implémenter des opérations suivant toujours le même ordre d'exécution et pour lesquelles seule la teneur varie, les détails de chaque opération spécifique étant implémentés dans des classes dérivées. Ce schéma peut être répété à plusieurs niveaux d'une hiérarchie de classes, une classe dérivée faisant elle-même appel, dans l'implémentation de ses fonctions virtuelles, à des nouvelles fonctions virtuelles devant être redéfinies dans une classe dérivée de niveau inférieur.

Remarque : on aurait pu également noter que la technique du « Pimpl » également utilisée ici, est en partie similaire au schéma de conception « Méthode du Pont » (*Brigde Method*). Néanmoins, le « Pimpl » de notre exemple est utilisé uniquement à des fins de masquage de l'implémentation de la classe GenericTableAlgorithm (pare-feu logiciel), et non pas en tant que « pont ». La technique du « Pimpl » sera étudiée en détail dans les problèmes 26 à 30.

Recommandations

Évitez d'utiliser des fonctions virtuelles publiques; préférez l'utilisation de la « Méthode du Modèle » (Template Method).
Apprenez à connaître et à utiliser les schémas de conception.

2. Commentez la manière dont le passage de la conception – qu'on ne remettra pas en cause – à l'implémentation a été réalisé. L'auriez-vous réalisé différemment ? Quel est le rôle de l'objet membre `pimpl_`?

Pour les fonctions réalisant un traitement, il a été choisi d'utiliser des codes de retour de type `bool`. C'est peut être suffisant dans ce cas, mais il faut noter que pour reporter des erreurs à l'appelant, il est souvent préférable d'utiliser soit un code de retour pouvant contenir plusieurs valeurs, soit, encore mieux, une exception.

Le pointeur membre « `pimpl_` », au nom intentionnellement prononçable, masque l'implémentation de la partie interne de la classe. Il pointera vers une classe interne `GenericTableAlgorithmImpl` contenant les variables et fonctions privées de la classe `GenericTableAlgorithm`. Ainsi, on évite de recompiler inutilement le code client lorsqu'une modification est apportée à la partie privée de la classe. Cette technique très utile, documentée entre autres par Lakos (Lakos96), permet, au prix d'une modification mineure du code, d'augmenter considérablement la modularité de ce code.

Recommandation

Lorsque vous implémentez une classe destinée à être largement utilisée, utilisez la technique du pare-feu logiciel (ou technique du « Pimpl ») afin de masquer la partie privée de la classe. Pour cela, déclarez un pointeur membre « `struct XxxxImpl* pimpl_` », vers une structure qui sera définie dans l'implémentation de la classe et contiendra les variables et fonctions membres privés. Par exemple : « `class Map { private : struct MapImpl* pimpl_; };` ».

3. À bien y réfléchir, l'architecture de ce programme pourrait être améliorée. Quels sont les rôles joués par `GenericTableAlgorithm` ? S'ils sont multiples, ne pourrait-on pas les implémenter dans plusieurs classes ? Quelles conséquences cela aurait-il sur la réutilisabilité et l'extensibilité de ces classes ?

Ce programme pourrait être amélioré sur un point bien précis. Actuellement, la classe `GenericTableAlgorithm` effectue deux tâches :

- d'une part, elle fournit une interface publique destinée à être utilisée par le code client.
- d'autre part, elle fournit une interface abstraite destinée à être implémentée par des classes dérivées spécialisées.

Ces deux tâches étant tout à fait indépendantes l'une de l'autre, il est préférable de les confier à deux classes séparées.

Recommandation

Dans la mesure du possible, attribuez à chaque entité de votre code – chaque module, chaque classe, chaque fonction – une tâche unique et clairement définie.

Voici ce que donnerait le code avec deux classes séparées :

```
//-------------------------------------------------
// Fichier gta.h (GenericTableAlgorithm)
//-------------------------------------------------
// Tâche n° 1 : Fournir une interface publique
// à l'usage des utilisateurs externes de cette classe.
// Cette tâche étant totalement indépendante de la notion
// d'héritage, il est préférable de l'implémenter
// dans une classe spécifique qui n'est pas destinée
// à être utilisée comme classe de base.
//

class GTAClient;

class GenericTableAlgorithm
{
public:
  // Le constructeur prend maintenant en paramètre
  // une classe contenant l'implémentation des opérations
  // à effectuer sur la base.
  //
  GenericTableAlgorithm( const string& table,
                         GTAClient&    worker );

  // Cette classe n'étant pas destinée à servir de classe
  // de base, il n'est plus nécessaire d'avoir un
  // destructeur virtuel
  //
  ~GenericTableAlgorithm();

  bool Process(); // inchangée

private:
  struct GenericTableAlgorithmImpl* pimpl_; // Implémentation
};

//-------------------------------------------------
// Fichier gtaclient.h
//-------------------------------------------------
// Tâche n° 2 : Fournir une interface abstraite
// qui sera implémentée par des classes dérivées.
// Cette tâche étant totalement découplée
// de l'utilisation de la classe GenericTableAlgorithm
// par des client externes, il est préférable
// de l'implémenter dans une classe séparée,
// dont le rôle sera de servir de classe de base.
// Les classes dérivées, réalisant l'implémentation des
// opérations sur les données, seront utilisées
// par la classe GenericTableAlgorithm.

//
class GTAClient
{
```

```
public:
  virtual ~GTAClient() =0;
  virtual bool Filter( const Record& );
  virtual bool ProcessRow( const PrimaryKey& ) =0;
};

//-------------------------------------------------
// Fichier gtaclient.cpp
//-------------------------------------------------
bool GTAClient::Filter( const Record& )
{
  return true;
}
```

Il est préférable que ces deux nouvelles classes soient définies dans deux fichiers en-tête séparés.

Voici à quoi ressemble le code du client, après application de ces changements :

```
class MaClasseUtilitaire: public GTAClient
{
  // ... Redéfinir Filter() and ProcessRow() afin
  //     d'implémenter des opérations spécifiques...
};

int main()
{
          GenericTableAlgorithm a( "Client", MaClasseUtilitaire() );
  a.Process();
}
```

Ceci peut paraître très similaire à la version précédente; néanmoins, trois améliorations ont été apportées :

1. L'interface publique de `GenericTableAlgorithm` est isolée du reste du programme : en particulier, si un nouveau membre public lui est ajouté, il n'est pas nécessaire de recompiler les fonctions implémentant les opérations sur la base de données, comme c'était le cas dans la version précédente.

2. L'interface abstraite déclarant les opérations sur la base de données est, elle aussi, isolée du reste du programme : si une modification est apportée à cette interface, il n'est pas nécessaire de recompiler les programmes client utilisant `GenericTableAlgorithm`, comme c'était le cas dans la version précédente.

3. Les classes implémentant les opérations sur les données pourront, si nécessaire, être réutilisées depuis tout classe « algorithme » faisant appel aux fonctions `Filter()` et `ProcessRow()`- et non plus uniquement depuis `GenericTableAlgorithm`.

En résumé, garder à l'esprit la règle d'or de l'informatique selon laquelle « il n'y a pratiquement aucun problème qui ne puisse se résoudre par l'ajout d'un niveau d'indirection », tout en veillant constamment à ne pas rendre les choses plus complexes que nécessaires, vous permettra généralement de produire un code plus modulaire et donc, plus facile à réutiliser et à maintenir.

Voyons pour finir deux ou trois points relatifs à la généricité en général.

On pourrait encore faire évoluer la classe `GenericTableAlgorithm` en remplaçant la fonction membre `Process()` par une fonction `operator()()`effectuant le même traitement, cette classe effectuant une opération unique. Mieux encore, on pourrait purement simplement remplacer la classe `GenericTableAlgorithm` par une fonction du même nom, pour la bonne raison qu'il n'y a aucun besoin de conserver un contexte entre deux appels successifs à `Process()`, donc aucun besoin d'instances séparées de `GenericTableAlgorithm` :

```
bool GenericTableAlgorithm(
  const string& table,
  GTAClient&    method
  )
{
  // ... mettre ici le code précédemment contenu dans Process()
}

int main()
{
  GenericTableAlgorithm( "Client", MaClasseUtilitaire() );
}
```

Nous obtenons donc ici une fonction dont le comportement est spécialisé par la classe passée en paramètre. Mieux encore, sachant que les objets `method` ne contiennent que des fonctions virtuelles, à l'exclusion de variables membres, et que donc toutes leurs instances sont fonctionnellement équivalentes, on peut transformer `GenericTableAlgorithm` en modèle de fonction :

```
template<typename ClasseUtilitaire>
bool GenericTableAlgorithm( const string& table )
{
  // ... mettre ici le code précédemment contenu dans Process()
}


int main()
{
  GenericTableAlgorithm<MaClasseUtilitaire>( "Customer" );
}
```

Nous aboutissons donc ici, pour finir, à une fonction générique. C'est probablement l'implémentation la plus adéquate dans le cas du problème qui nous intéresse. Néanmoins, il faut éviter de tomber dans le piège de la généricité excessive, dont on

abuse parfois sous prétexte de vouloir simplifier le code client, ce qui n'est pas une raison suffisante en soi.

Pb n° 24. Utiliser l'héritage sans en abuser Difficulté : 6

« Quand utiliser l'héritage ? »
L'héritage est une fonctionnalité du C++ souvent utilisée de manière abusive, même par des développeurs expérimentés. Il est recommandé de toujours minimiser le couplage entre classes : lorsqu'il y a plusieurs manières d'implémenter une relation entre classes, il faut systématiquement choisir celle qui crée la dépendance la plus faible. L'héritage est une des relations les plus fortes que l'on puisse créer en C++ (avec la relation d'amitié) : il faut vraiment la réserver aux cas où il n'y a pas d'autre choix.
Dans ce problème, nous aborderons les différents types d'héritage : l'héritage privé, méconnu, l'héritage protégé, rare, pour lequel nous étudions un cas concret d'utilisation et enfin l'héritage public, très employé, pour lequel nous préciserons quand il doit vraiment être utilisé. Au passage, nous établirons la listes des raisons souvent invoquées pour justifier l'emploi d'une relation d'héritage et nous verrons lesquelles sont valables et lesquelles ne le sont pas.

Le modèle de classe ci-dessous permet de gérer une liste d'éléments (notamment l'ajout d'un élément et la récupération de la valeur d'un élément)

```
// Exemple 1
  //
  template <class T>
  class Liste
  {
  public:
    bool   Insert( const T&, size_t index );
    T      Access( size_t index ) const;
    size_t Size() const;
  private:
    T*     buf_;
    size_t bufsize_;
  };
```

Considérez maintenant le code ci-dessous, qui montre deux manières d'implémenter une classe `MaListe` basée sur le modèle de classe `Liste` :

```
  // Exemple 1(a)
  //
  template <class T>
  class MaListe1 : private Liste<T>
  {
  public:
    bool   Add( const T& ); // appelle Insert()
    T      Get( size_t index ) const;
                           // appelle Access()
```

```
  using MyList<T>::Size;
  //...
};

// Exemple 1(b)
//
template <class T>
class MaListe2
{
public:
  bool   Add( const T& ); // appelle impl_.Insert()
  T      Get( size_t index ) const;
                          // appelle impl_.Access()
  size_t Size() const;    // appelle impl_.Size();
  //...
private:
  MyList<T> impl_;
};
```

Comparez ces deux propositions, en répondant notamment aux questions suivantes :

- Quelles sont les différences entre `MaListe1` et `MaListe2` ?
- D'une manière plus générale, quelles sont les différences entre héritage non-public (privé ou protégé) et composition ? Établissez une liste de raisons pouvant justifier l'emploi de l'héritage non-public plutôt que la composition.
- Quelle est la meilleure version : `MaListe1` ou `MaListe2` ?
- Pour finir et en se replaçant dans un contexte général, établissez la liste des cas pour lesquels vous utiliseriez l'héritage public.

Solution

Ce problème aborde diverses questions relatives à l'héritage, notamment le choix entre héritage non-public (privé ou protégé) et composition.

La réponse à la première question (« Quelles sont les différences entre `MaListe1` et `MaListe2` ? ») est relativement simple : il n'y a aucune différence notable entre `MaListe1` et `MaListe2` ; ces deux classes sont fonctionnellement identiques.

La deuxième question permet de rentrer un peu plus dans le vif du sujet : « D'une manière plus générale, quelles sont les différences entre héritage privé et composition? Établissez une liste de raisons pouvant justifier l'emploi de l'héritage non-public (privé ou protégé) plutôt que la composition »

- L'héritage non-public (privé ou protégé) devrait toujours être utilisé pour traduire une relation du type « EST-IMPLEMENTE-EN-FONCTION-DE » (sauf dans un cas très particulier que nous détaillons un peu plus loin). Il rend la classe utilisatrice dépendante des parties publique *et protégée* de la classe utilisée.

- La composition traduit toujours une relation du type « A-UN », et donc « EST-IMPLEMENTE-EN-FONCTION-DE ». Elle rend la classe utilisatrice uniquement dépendante de la partie publique de la classe utilisée.

Il est facile de démontrer que la composition est un sous-ensemble de l'héritage, c'est-à-dire qu'il n'y a rien qu'on puisse faire avec un objet membre `Liste<T>` et qui ne puisse pas être fait avec une classe dérive de `Liste<T>`. D'un autre côté, avec l'héritage, nous sommes limités à un seul objet `Liste<T>` (l'objet de base), alors qu'en utilisant la composition, il est possible d'avoir plusieurs instances de `Liste<T>`.

Recommandation

Pour implémenter une relation du type « EST-IMPLEMENTE-EN-FONCTION-DE », préférez l'utilisation de la composition à celle de l'héritage.

D'une manière plus générale, quelles fonctionnalités supplémentaires apporte l'héritage non-public par rapport à l'utilisation de la composition ? Autrement dit, quels sont les cas où l'emploi de l'héritage non-public est nécessaire ? En voici un certain nombre, classés par fréquence décroissante :

- **Besoin de redéfinir des fonctions virtuelles.** Lorsque la classe utilisée comporte des fonctions virtuelles pouvant être spécialisées par la classe utilisatrice, l'héritage se justifie. C'est d'ailleurs la principale raison d'être du mécanisme d'héritage. Dans le cas où la classe utilisée serait abstraite (si elle comporte au moins une fonction virtuelle pure), la création d'une classe dérivée est obligatoire puisqu'on ne peut pas créer d'instance de la classe utilisée.

- **Besoin d'accéder à des membres protégés.** La classe utilisatrice a besoin de faire appel à une ou plusieurs fonctions membres protégées[1]. Se justifie, en particulier lorsque la classe utilisée comporte un constructeur protégé devant être appelée par la classe utilisatrice.

- **Besoin de construire l'objet « utilisé » avant un autre objet de base (et/ou de le détruire après).** Dans le cas spécifique où la classe utilisatrice a elle-même une classe de base dont les instances doivent être construites après (et détruites avant) l'objet « utilisé », l'héritage est la seule solution possible. Ce type de besoin peut se produire, par exemple, lorsque l'objet « utilisé » maintient un verrou (section critique, transaction de base de données, …) qui doit couvrir l'intégralité de la durée de vie d'un objet de base de l'objet utilisateur.

- **Besoin de partager une classe de base virtuelle ou de redéfinir le constructeur d'une classe de base virtuelle.** Lorsque la classe utilisée a une classe de base virtuelle et que la classe utilisatrice souhaite, soit hériter de cette classe de base, soit

1. Je dis « fonctions membres » car, bien entendu, il serait tout à fait maladroit d'implémenter une classe comportant des variables membres publiques ou protégées, bien qu'on puisse rencontrer de telles pratiques dans certains exemples de code bien maladroits.

spécialiser la construction de cette classe de base[1], il est indispensable de faire dériver la classe utilisatrice de la classe dérivée.

- **Optimisation des performances par l'emploi d'une « classe de base vide ».** Lorsque la classe « utilisée » ne comporte que des fonctions membres (et donc aucune variable membre), le fait d'utiliser l'héritage à la place de la composition peut permettre d'économiser de l'espace mémoire, en vertu du principe de la « classe de base vide ». Ce principe autorise les compilateurs à ne réserver aucun espace mémoire pour une classe de base ne contenant que des fonctions membres, alors qu'ils sont obligés d'allouer un espace mémoire non nul pour tout objet membre, mais lorsque celui-ci ne contient aucune variable membre.

```
class B { /* ... ne contient que des fonctions membres... */ };

// Composition  : occupation d'espace mémoire superflu
   //
   class D
    {
         B b_; // b_ occupera au moins un octet en mémoire,
   };          // même si B est une classe vide

// Héritage : limite l'occupation d'espace mémoire superflu
   //
   class D : private B
   {       // l'objet de base B peut n'occuper
   };      // aucune place en mémoire
```

Pour plus de détails, voir l'excellent article de Nathan Myers sur ce sujet dans *Dr. Dobb's Journal* (Myers97).

Pour être réaliste, cette optimisation – qui n'est d'ailleurs pas implémentée par tous les compilateurs – n'apporte pas grand chose : il faut véritablement qu'il y ait un nombre énorme d'instances créées (disons, quelques dizaines de milliers) pour qu'elle présente un intérêt pratique. Il n'est en général pas judicieux d'introduire une relation d'héritage – et par là-même un couplage fort entre classes – uniquement pour profiter de cette optimisation (à moins d'être certain que votre compilateur l'implémente et que le nombre d'objets alloués par votre programme est suffisamment important pour que vous en retiriez un avantage certain).

Il existe un cas supplémentaire qui peut justifier le recours à une dérivation non publique (au passage, c'est le seul qui ne traduise pas une relation du type « EST-IMPLEMENTE-EN-FONCTION-DE ») :

- **Mise en œuvre de « polymorphisme partiel » (implémentation d'un certain type de relation « EST-UN »).** Pour implémenter une relation « EST-UN » au sens du principe de substitution de Liskov[2], on utilise, dans la très grande majorité

1. Rappel sur l'héritage virtuel : c'est la classe située le plus bas dans la hiérarchie qui est responsable de l'initialisation de toutes les classes de base virtuelles.
2. Vous trouverez de nombreux articles consacrés au principe de substitution de Liskov (*Liskov Substitution Principle*) sur le site www.objectmentor.com

des cas, l'héritage public. Néanmoins, et cela, la plupart des gens l'ignorent, on peut utiliser l'héritage non-public (privé ou protégé) pour implémenter un certain type de relation « EST-UN ». Considérons une classe `Derivee` , dérivée de manière privée de la classe `Base`. Bien évidemment, les instances de cette classe ne peuvent pas être utilisées de manière polymorphique par le biais d'un pointeur de type `Base*` depuis du code extérieur, car tous les membres de `Derivee` sont privés. Cependant, il est possible d'utiliser `Derivee` de manière polymorphique depuis l'*intérieur* de `Derivee` elle-même (c'est-à-dire depuis l'intérieur des fonctions membres et amies de `Derivee`). En remplaçant la dérivation privée par une dérivation protégée, on étend la possibilité d'utiliser ce polymorphisme interne aux éventuelles classes dérivées de `Derivee`, mettant ainsi en œuvre un type particulier de relation « EST-UN » qui peut être utile dans certains cas.

Nous avons à présent recensé toutes les raisons pouvant justifier l'emploi de l'héritage privé ou protégé. L'héritage public, quant à lui, n'est employé que pour modéliser une relation du type « EST-UN » (nous reviendrons sur ce point à l'occasion de la dernière question du problème).

À la lumière des points précédents, nous pouvons maintenant répondre à la troisième question : « Quelle est la meilleure version : `MaListe1` ou `MaListe2` ? »

Prenons l'exemple de code n° 1 et voyons s'il remplit un ou plusieurs des critères requis pour l'emploi de l'héritage privé ou protégé :

- Nécessité d'accéder aux membres protégés de la classe utilisée : non (`Liste` n'a pas de membres protégés)
- Nécessité de redéfinir des fonctions virtuelles : non (`Liste` n'a pas de fonctions virtuelles).
- Nécessité de construire l'objet utilisé avant l'objet utilisateur (et/ou de le détruire après) : non (la classe `MaListe1` n'a pas d'autre classe de base, il n'y a donc pas de problèmes liés aux durées de vie relatives de l'objet `Liste` et d'un objet de base de `MaListe1`).
- Nécessité de partager une classe de base virtuelle ou de redéfinir le constructeur d'une classe de base virtuelle : non (la classe `Liste` n'a pas de classe de base virtuelle).
- Nécessité d'optimiser les performances par emploi d'une « classe de base vide » : non (la classe `Liste` n'est pas vide).
- Nécessité de mise en oeuvre du « polymorphisme partiel » : non (la classe `MaListe` n'est aucunement liée à la classe `Liste` par une relation de type EST-UN, même depuis l'intérieur des fonctions membres et amies de `MaListe`). Ce dernier point mérite qu'on s'y attarde un instant car il met en évidence un des inconvénients (mineurs) de l'héritage : au cas où un des critères ci-dessus aurait été rempli et que donc la dérivation de `MaListe` depuis `Liste` aurait été justifiée et utilisée, cela aurait fait apparaître le danger de l'utilisation polymorphique accidentelle d'objets `MaListe` considérés comme des objets `Liste` à partir des fonctions membres et amies de `MaListe`; possibilité certes rare mais potentielle source de confusion pour le développeur inexpérimenté.

En résumé, la meilleure solution est celle employant la composition (`MaListe2`). Utiliser l'héritage sans avoir une bonne raison de le faire introduit des couplages inutiles et des dépendances coûteuses entre classes. Malheureusement, de trop nombreux développeurs, même parmi les plus expérimentés, ont souvent recours à l'héritage alors que la composition suffit.

Toutefois, le lecteur attentif aura peut être noté un avantage mineur (et sans grand intérêt) de la solution employant l'héritage sur celle employant la composition : dans le premier cas, il suffit d'écrire une instruction « `using` » pour avoir accès à la fonction `Size()`; dans le second cas, il faut implémenter explicitement une nouvelle fonction relayant l'appel à la fonction `Size()` de l'objet contenu.

Voyons maintenant un cas où il est nécessaire d'utiliser l'héritage :

```
// Exemple 2 : Cas où l'utilisation d'héritage est justifiée
//
class Base
{
public:
  virtual int Fonction1();
protected:
  bool Fonction2();
private:
  bool Fonction3(); // fait appel à Fonction1
};
```

Dans cet exemple, nous considérons une classe `Base` dotée d'une fonction virtuelle `Fonction1()` et d'une fonction protégée `Fonction2()`. Le seul moyen de redéfinir cette fonction virtuelle ou d'avoir accès à cette fonction protégée est de dériver une classe de `Base`. Notons au passage qu'il s'agit ici d'un exemple où la redéfinition d'une fonction virtuelle n'est pas effectuée uniquement à des fins de polymorphisme mais également pour modifier le comportement de la classe de base (la fonction `Fonction3()` faisant appel à `Fonction1()` dans son implémentation).

L'utilisation de l'héritage est donc justifiée. Ceci étant, quelle est la meilleure manière de mettre en œuvre cet héritage ? Commentez l'exemple suivant :

```
// Exemple 2(a)
//
class Derivee : private Base // nécessaire ?
{
public:
  int Fonction1();
  // ... suivent plusieurs fonctions, dont
  //     certaines font appel a Base::Fonction2()
  //     et d'autres non.
};
```

Ce code permet la redéfinition de `Fonction1()`, ce qui est une bonne chose. En revanche, il donne accès à la fonction `Base ::Fonction2()`à *tous les membres* de `Derivee`. Par conséquent, il rend tous les membres de `Derivee` dépendants de l'interface protégée de `Base`, ce qui est regrettable.

L'exemple 2(a) présente donc l'inconvénient d'introduire un couplage trop fort entre `Derivee` et `Base`. Néanmoins, il y a un moyen plus judicieux de réaliser cette dérivation :

```
// Exemple 2(b)
//
class DeriveeImpl : private Base
{
public:
  int Fonction1();
  // ... Suivent des fonctions utilisant Base::Fonction2()
};

class Derivee
{
 // ... Suivent des fonctions n'utilisant pas Base::Fonction2()
private:
  DeriveeImpl impl_;
};
```

Cette deuxième solution est bien meilleure car elle encapsule dans deux classes différentes les deux types de dépendances établies avec la classe `Base` : `Derivee` dépend uniquement de l'interface publique de `Base` et ne dépend plus de son interface protégée. Alors que dans l'exemple 2(a), la classe `Derivee` jouait deux rôles (spécialiser `Base` tout en faisant appel à elle), l'exemple 2(b) sépare bien les rôles, se conformant ainsi à la règle d'or de la conception objet : « une classe, une responsabilité ».

Voyons maintenant quelques avantages apportés par l'utilisation de la composition :

D'une part, la composition permet de disposer de plusieurs instances de la classe « utilisée », ce qui est difficile, voire parfois impossible à réaliser avec l'emploi de l'héritage. Si vous avez besoin à la fois d'utiliser une relation d'héritage et d'avoir plusieurs instances, il faut utiliser une technique similaire à celle décrite dans l'exemple 2(b) : créez une classe utilitaire dérivée (comme `DeriveeImpl`) puis agrégez, dans la classe utilisatrice, plusieurs instances de cette classe utilitaire.

D'autre part, l'utilisation de la composition apporte davantage de flexibilité :

- La classe de l'objet membre « utilisé » peut être facilement masquée derrière un pare-feu logiciel par la technique du « Pimpl[1] », alors que la définition d'une classe de base sera toujours visible.
- L'objet membre peut également être facilement remplacé par un pointeur, ce qui permet de changer facilement la nature de l'objet utilisé à l'exécution, ce qui n'est évidemment pas possible lorsqu'on utilise une classe de base.
- Utilisée conjointement avec des modèles de classes, la composition permet d'atteindre un très bon niveau de généricité.

1. Voir les problèmes 26 à 30

Pour illustrer ce troisième point, voyons une version légèrement modifiée de notre exemple 1 :

```
// Exemple 1(c): Composition générique
//
template <class T, class Impl = Liste<T> >
class MaListe3
{
public:
  bool   Add( const T& ); // appelle impl_.Insert()
  T      Get( size_t index ) const;
                                    // appelle impl_.Access()
  size_t Size() const;    // appelle impl_.Size();
  // ...
private:
  Impl impl_;
};
```

Au lieu d'avoir une classe `MaListe` « IMPLEMENTEE-EN-FONCTION-DE » `Liste`, nous avons à présent une classe « IMPLEMENTABLE-EN-FONCTION-DE » n'importe quelle classe dont l'interface publique contient les fonctions `Add()`, `Get()` et `Size()`. C'est d'ailleurs une technique utilisée par la bibliothèque standard dans son implémentation des modèles `stack` et `queue`, qui sont par défaut « IMPLEMENTES-EN-FONCTION-DE » `dequeue`, mais sont également « IMPLEMENTABLE-EN-FONCTION-DE » toute classe comportant la même interface que `dequeue`.

En pratique, cette généricité est utile car elle permet de spécialiser le comportement de la classe « utilisée ». Par exemple, pour un programme amené à effectuer un très grand nombre d'insertions, on instanciera le modèle de classe `MaListe3` en passant, en deuxième paramètre, une classe avec une fonction `Insert()` spécialement optimisée. L'utilisation d'une valeur par défaut pour ce paramètre assure une compatibilité ascendante (`MaListe3<int>` étant synonyme de `MaListe3<int,Liste<int>>`).

Ce niveau de flexibilité est difficile à atteindre avec l'héritage, avec lequel les décisions d'implémentation sont figées au moment de la conception d'une classe plutôt que lors de son utilisation. Par exemple, faire dériver `MaListe3` de `Liste<T>` aurait introduit un couplage supplémentaire inutile.

Pour finir, penchons-nous sur la dernière question de notre problème : « En se replaçant dans un contexte général, établissez la liste des cas lesquels vous utiliseriez l'héritage public. »

Concernant l'héritage public, il y a une règle simple mais fondamentale qu'il faut avoir en permanence à l'esprit : l'*unique* cas justifiant l'emploi de l'héritage public est la mise en oeuvre d'une relation de type « EST-UN » au sens du principe de substitution de Liskov[1]. Ce principe énonce que dans le contexte d'un code client utilisant un objet de la classe de base, la substitution d'un objet dérivé en lieu et en place de l'objet de base doit être sans effet sur le comportement du code client [Remarque :

1. Vous trouverez de nombreux articles consacrés au principe de substitution de Liskov (*Liskov Substitution Principle*) sur le site www.objectmentor.com

nous abordons une des rares exceptions à cette règle – ou plus être plus précis une *extension* – dans le problème n° 3].

Prenez soin notamment d'éviter deux erreurs courantes :

- **Ne jamais utiliser l'héritage public lorsque l'héritage privé ou protégé est suffisant.** L'héritage public ne doit jamais être utilisé pour traduire une relation du type « EST-IMPLEMENTE-EN-FONCTION-DE » en l'absence de relation « EST-UN ». De trop nombreux développeurs ne respectent malheureusement pas cette règle, pourtant fondamentale. Il est inutile et coûteux d'utiliser une relation d'héritage public lorsque l'héritage non-public fait l'affaire. Encore une fois, lorsqu'il y a plusieurs manières d'implémenter une relation entre classes, il faut systématiquement choisir celle qui crée la dépendance la plus faible.

- **Ne jamais utiliser l'héritage public pour implémenter une relation du type « EST-PRESQU'UN ».** Lorsqu'on met en oeuvre une relation d'héritage public entre deux classes, il faut s'assurer que *toutes* les fonctions virtuelles redéfinies dans la classe dérivée se comportent de manière similaire aux fonctions correspondantes de la classe de base. Certains développeurs, même parmi les expérimentés, n'assurent parfois cette similarité de comportement que pour *la plupart* des fonctions virtuelles redéfinies, ce qui présente une conséquence très gênante : un code client initialement conçu pour utiliser des objets de classes de base pourra ne pas avoir le même comportement avec des objets dérivés de cette classe. Un exemple souvent cité (Robert Martin) est celui du carré et du rectangle : il faudrait soi-disant dériver « `Carre` » de « `Rectangle` » car « un carré est un rectangle particulier ». C'est vrai en mathématique, ça ne l'est pas pour des classes. En effet, admettons que la classe `Rectangle` contienne une fonction virtuelle `SetLargeur(int)` et que cette fonction soit redéfinie dans la classe `Carre`. A priori, l'implémentation dans `SetLargeur(int)` dans `Carre` mettra à jour à la fois la largeur et la hauteur avec la valeur passée en paramètre, afin d'assurer que le carré conserve sa nature « carrée ». Dès lors, un problème risque de se poser si du code client utilise un objet `Rectangle` de manière polymorphique : en effet, ce code ne s'attendra pas à ce que la hauteur soit modifiée lorsqu'il fixe la largeur, ce qui est pourtant ce qui se passera ! Ceci est un bon exemple de mauvaise utilisation de l'héritage public. Les classes `Carre` et `Rectangle` ne respectent pas le Principe de Substitution de Liskov car la classe dérivée ne se comporte pas de la même manière que la classe de base.

Lorsque je rencontre ce type de relation « EST-PRESQU'UN », j'attire systématiquement l'attention du développeur sur les risques que ce type d'implémentation présente, en insistant en particulier sur le fait que l'utilisation polymorphique d'objets dérivés est susceptible de produire un résultat inattendu. Je m'entends fréquemment répondre que ce n'est pas grave, qu'il s'agit là d'une incompatibilité mineure et que le risque qu'un code client provoque un problème est faible. C'est généralement vrai lorsque le rédacteur du code client est au courant des emplois à éviter. Mais, des années plus tard, un développeur réalisant une opération de maintenance peut, en

apportant des modifications –même mineures– à ce code, faire resurgir le problème et passer des heures, voire des jours, à le repérer et à le corriger.

Donc, soyez fermes : une classe dérivée qui ne se comporte pas comme une classe de base n' « EST PAS » une classe de base. Évitez donc systématiquement d'utiliser l'héritage public dans ce cas-là.

Recommandations

N'utilisez l'héritage public que dans les cas où la relation entre classe dérivée et classe de base est de type « EST-UN » ou « FONCTIONNE-COMME-UN », au sens du Principe de Substitution de Liskov.
N'utilisez pas l'héritage public pour réutiliser le code de la classe de base; utilisez l'héritage public pour être réutilisé par des objets externes utilisant le polymorphisme de la classe de base.

Conclusion

N'abusez pas de l'héritage. Pour modéliser une relation du type « EST-IMPLEMENTE-EN-FONCTION-DE », préférez systématiquement l'emploi de la composition, lorsque cela suffit. N'employez l'héritage public que pour modéliser une relation du type « EST-UN ». Évitez l'héritage multiple lorsque l'héritage simple suffit. Gardez à l'esprit que les lourdes hiérarchies de classe sont difficiles à comprendre et à maintenir, que l'héritage oblige à fixer des choix dès la conception, ce qui réduit en conséquence la flexibilité à l'exécution.

Contrairement à une opinion communément répandue, il est tout à possible de programmer « Orienté-Objet » sans utiliser systématiquement l'héritage. Lorsque plusieurs solutions sont possibles, utilisez toujours la plus simple. Votre code n'en sera que plus stable et plus facile à maintenir.

Pb n° 25. Programmation orientée objet — Difficulté : 4

Abandonnons pour quelques instants les exemples de code pour nous poser une question plus générale, à laquelle il est souvent répondu trop catégoriquement par l'affirmative : « Le C++ est-il un langage orienté-objet ? ».

Discutez la phrase suivante :

« Le C++ est un langage puissant présentant de nombreuses fonctionnalités orientées objet, notamment l'encapsulation, la gestion des exceptions, l'héritage, les modèles, le polymorphisme, le typage fort des données et la gestion des modules de code. »

Solution

Le but de ce problème était d'amener le lecteur à réfléchir sur les réelles possibilités du langage C++ : ses avantages mais également ses éventuelles insuffisances. Cette vaste question mériterait d'être débattue longuement. Néanmoins, trois points principaux peuvent être notés :

1. Il n'existe pas de définition précise du concept « orienté objet ». Bien que ce concept existe maintenant depuis de nombreuses années, personne ne s'entend sur sa signification précise. Interrogez dix personnes et vous obtiendrez dix réponses différentes. Pratiquement une grande majorité s'accordera à dire que la programmation orientée objet s'articule autour des concepts d'héritage et de polymorphisme; la plupart incluront également l'encapsulation; d'autres citeront peut-être la gestion des exceptions; certains, plus rares, ajouteront la notion de modèle (*template*); chacun saura défendre son point de vue avec force arguments, prouvant ainsi qu'il n'y a pas de définition unique de l' « Orienté-Objet »

2. Le langage C++ est un langage multi-facettes. C++ n'est pas *uniquement* un langage orienté objet. Bien qu'il présente un grand nombre de fonctionnalités orientées objet, il n'impose pas leur utilisation. Il est tout à fait possible d'écrire des programmes non orienté objet en C++; de nombreux développeurs le font.

Tous les efforts de normalisation du langage ont convergé vers un but commun : doter le C++ de fortes capacités d'*abstraction*, lui permettant de réduire au maximum la complexité des logiciels (Martin95)[1]. C++ n'est pas limité à la programmation orientée objet : il autorise plusieurs styles de programmations. Parmi ces styles, les plus importants sont la programmation orientée objet et la programmation générique, qui, par leurs capacités d'abstraction, permettent la réalisation de programmes modulaires. La programmation orientée objet, en permettant le regroupement des variables d'état caractérisant une entité et des fonctions les manipulant, ainsi que l'encapsulation et l'héritage, permet d'écrire du code plus clair, plus modulaire et plus facile à réutiliser. La programmation générique, plus récente, permet d'écrire des fonctions et de classes manipulant des objets dont le type est inconnu à l'avance, offrant ainsi un moyen de diminuer drastiquement le couplage entre les différents éléments d'un programme. Peu de langages, à l'heure actuelle, sont aussi complets en matière de programmation générique. Les modèles (*templates*) C++ sont d'ailleurs à l'origine de la programmation générique moderne.

En conclusion, le langage C++ offre aujourd'hui, grâce au travail réalisé par les comités de normalisation, deux techniques majeures : programmation orientée objet et programmation générique. Leur combinaison permet d'atteindre un niveau d'abstraction et de flexibilité inégalé.

1. Cet excellent ouvrage démontre en quoi l'un des avantages principaux de la POO est la possibilité de réduire la complexité des logiciels en gérant finement les dépendances au sein du code.

3. Aucun langage n'est parfait. Si C++ est le langage qui vous utilisez le plus aujourd'hui, qui peut prédire si vous ne trouvez pas mieux demain ? C++ n'est pas parfait : il ne permet pas la gestion fine des modules, n'a pas de « ramasse-miettes » mémoire, implémente un typage statique des données mais pas véritablement de typage « fort ». Tout langage a ses avantages et ses inconvénients. Ne soyez pas irrémédiablement inconditionnel d'un unique langage: sachez choisir, en fonction des circonstances le langage le mieux adapté à vos besoins.

Pare-feu logiciels

La robustesse d'un programme est souvent très liée à la manière dont sont gérées les dépendances au sein du code. Le langage C++ dispose de deux puissantes méthodes d'abstraction, la programmation orientée objet et la programmation générique (Sutter98). Les fonctionnalités qu'elles offrent – encapsulation, polymorphisme, généricité – sont autant d'outils utilisables pour réduire les dépendances entre modules et, par conséquent, minimiser la complexité des programmes.

Pb n° 26. Éviter les compilations inutiles (1re partie) — Difficulté : 4

La gestion des dépendances ne concerne pas uniquement ce qui se passe à l'exécution du programme – comme les interactions entre classes – mais également la phase de compilation, sur laquelle nous nous concentrerons dans ce problème. Nous nous intéresserons dans cette première partie à l'élimination des fichiers en-tête inutiles.

De nombreux développeurs ont la mauvaise habitude d'inclure souvent plus de fichiers en-tête que nécessaire. Ceci peut alourdir considérablement les temps de compilation, notamment lorsqu'un fichier en-tête en inclut de nombreux autres.

Examinez le code ci-dessous. Identifiez et supprimez – ou remplacez – les instructions `#include` superflues, sans que cela nécessite de modifier le reste du code. Les commentaires sont importants.

```
//  x.h: Fichier original
//
#include <iostream>
#include <ostream>
#include <list>
```

```
// Les classes A, B, C, D et E ne sont pas
// des modèles de classe.
// A et C comportent des fonctions virtuelles.

#include "a.h"  // classe A
#include "b.h"  // classe B
#include "c.h"  // classe C
#include "d.h"  // classe D
#include "e.h"  // classe E

class X : public A, private B
{
public:
     X( const C& );
  B  f( int, char* );
  C  f( int, C );
  C& g( B );
  E  h( E );
  virtual std::ostream& print( std::ostream& ) const;
private:
  std::list<C> clist_;
  D            d_;
};
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
  return x.print(os);
}
```

Solution

Le premier fichier en-tête peut être purement et simplement supprimé, tandis que le second peut être remplacé par un autre fichier plus léger.

1. Supprimer <iostream>

```
#include <iostream>
```

Beaucoup de développeurs ont la mauvaise habitude d'inclure systématiquement <iostream>, dès que leur code fait appel à quelque chose ressemblant de près ou de loin à un flux. Dans notre exemple, la classe X ne fait appel qu'à ostream et donc l'inclusion de <ostream> est suffisante, bien qu'encore légèrement superflue comme nous allons le voir immédiatement.

Recommandation

Ne jamais inclure de fichiers en-tête inutiles.

2. Remplacer <ostream> par <iosfwd>

```
#include <ostream>
```

Dans notre exemple, il n'est pas nécessaire de *définir* le type du paramètre et de la valeur de retour de la fonction `print` pour que la compilation de `x.h` s'effectue correctement : la déclaration « en avance » d'`ostream` suffit.

On peut donc s'affranchir de « `#include <iostream>` ». En revanche, on ne peut pas le remplacer par une déclaration de type « `class ostream ;` » : cette syntaxe fut possible à une époque, mais ne l'est plus aujourd'hui pour les deux raisons suivantes :

- `ostream` est maintenant contenue dans l'espace de nommage `std`, à l'intérieur duquel les développeurs ne sont pas autorisés à déclarer des classes.
- En réalité, `ostream` n'est pas une classe mais une définition de type (`typedef`) : `basic_ostream<char>`. Par conséquent, avant de déclarer `ostream`, il faudrait également déclarer `basic_ostream<>`, ce qui pourrait poser problème car les implémentations de la bibliothèque standard utilisent souvent des paramètres supplémentaires à usage interne pour les modèles de classes – c'est d'ailleurs la principale raison de l'interdiction faite aux développeurs d'effectuer des déclarations au sein de `std`.

Heureusement, nous pouvons utiliser le fichier en-tête standard `<iosfwd>`, qui contient les déclarations en avance de tous les modèles de classe relatifs aux flux (dont `basic_ostream`) ainsi que leur `typedef` associé (dont `ostream`).

En conclusion, nous pouvons remplacer « `#include <iostream>` » par « `#include <iosfwd>` ».

Recommandation

Lorsqu'une déclaration « en avance » suffit, utilisez `<iosfwd>` plutôt que les autres fichiers en-tête relatifs aux flux.

Il faut noter qu'il n'existe pas d'équivalent de `<iosfwd>` dans les autres modèles de classe de la bibliothèque standard (on aurait pu imaginer `<listfwd>` pour `list` ou `<stringfwd>` pour `string`...). En effet, le cas d'`<iosfwd>` est spécifique puisqu'il a été créé pour faciliter la compatibilité ascendante lors de l'introduction de la version d'`<iostream>` basée sur des modèles de classe.

Arrivé à ce point, le lecteur averti aurait pu s'étonner du fait que le fichier `x.h` ne se contente pas de faire référence à `ostream` en tant que paramètre d'entrée ou valeur retournée mais *utilise* bel et bien le type `ostream` (dans l'opérateur `<<`) et que, par conséquent, une *définition* de ce type s'impose.

Ceci aurait été une remarque sensée, mais néanmoins inexacte. En effet, considérons à nouveau la fonction en question :

```
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
  return x.print(os);
}
```

Cette fonction utilise `ostream&` en paramètre d'entrée et valeur de retour (ce qui, comme la majorité des développeurs le savent, ne requiert pas la définition de

ostream). Dans son implémentation, la fonction passe à son tour le paramètre reçu à une autre fonction ce qui, et cela beaucoup moins de gens le savent, ne requiert pas non plus de définition.

Comme ce sont là les seules opérations effectuées avec ostream, il n'est pas nécessaire d'en inclure la définition. Bien entendu, si la fonction operator<< avait fait appel à une fonction membre de os, nous aurions eu besoin de la définition.

3. Remplacer « e.h » par une déclaration « en avance »

```
#include « e.h » // classe E
```

Le code ne fait référence à la classe E que comme paramètre d'entrée ou valeur de retour, par conséquent la définition complète de E n'est pas nécessaire et nous pouvons remplacer « #include « e.h » » par « class E ; »

Recommandation

Ne jamais inclure un fichier en-tête lorsqu'une déclaration « en avance » suffit.

Pb n° 27. Éviter les compilations inutiles (2e partie) Difficulté : 6

Les fichiers en-tête superflus étant éliminés, passons maintenant à l'étape suivante : rendre le code le moins dépendant possible des parties privées des classes.

Repartons de l'exemple du fichier précédent, à présent affranchi de quelques fichiers en-tête inutiles. Identifier les fichiers en-tête supplémentaires dont on peut se passer, à condition d'effectuer quelques modifications à la classe X. Attention : vous ne pouvez pas modifier la liste des classes dont hérite X ni la partie publique de X; les modifications effectuées ne doivent pas avoir d'impact sur le code client utilisant X – si ce n'est une recompilation.

```
//  x.h: Après suppression des en-têtes inutiles
//
 #include <iosfwd>
 #include <list>


// Les classes A, B, C, D et E ne sont pas
// des modèles de classe.
// A et C comportent des fonctions virtuelles.

#include "a.h"  // classe A
#include "b.h"  // classe B
#include "c.h"  // classe C
#include "d.h"  // classe D
```

```
#include "e.h"  // classe E
class E;
class X : public A, private B
{
public:
     X( const C& );
  B  f( int, char* );
  C  f( int, C );
  C& g( B );
  E  h( E );
  virtual std::ostream& print( std::ostream& ) const;
private:
  std::list<C> clist_;
  D            d_;
};
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
  return x.print(os);
}
```

Solution

Examinons les fichiers en-tête que nous n'avions pas pu supprimer lors du problème précédent :

- « `a.h` » et « `b.h` » contiennent les définitions des classes `A` et `B` – qui sont des classes de base de `X`. Nous n'avions pas pu les supprimer car le compilateur en a besoin pour déterminer, en particulier, la taille des objets X et les fonctions virtuelles. Nous ne pourrons pas non plus les supprimer dans ce problème, au vu des contraintes imposées par l'énoncé. Néanmoins, nous verrons comment il est possible (et pourquoi il est opportun) de le faire à l'occasion du problème suivant.
- « `c.h` » contient la définition de la classe `C`, qui est utilisée pour instancier le modèle de classe `list<C>`. Dans leur grande majorité, les compilateurs imposent le fait que la définition de `T` soit visible lorsque l'on instancie un modèle de classe `list<T>` (bien qu'elle ne soit pas imposée par la norme C++, cette contrainte devient tellement répandue qu'elle finira pas être normalisée), pourtant nous allons trouver un moyen de supprimer « `#include c.h` »
- « `d.h` » contient la définition de la classe `D`, dont le type est utilisé pour une variable membre privée de `X`. Nous allons également nous passer de « `#include d.h` »

Nous allons maintenant utiliser une technique particulière qui va nous permettre de supprimer les fichiers en-tête « `c.h` » et « `d.h` » : la technique du « Pimpl ».

Le langage C++ permet facilement de contrôler l'accès à certains membres d'une classe, à travers la notion de partie privée. En revanche, il est beaucoup moins facile de rendre un code client d'une classe indépendant de la partie privée de cette classe. Dans l'idéal, l'encapsulation devrait permettre de découpler totalement le code client et les détails d'implémentation d'une classe. Par le mécanisme des variables privées,

le C++ rend possible en partie de cette encapsulation; néanmoins le fait que la partie privée d'une classe soit déclarée dans le fichier en-tête de cette classe – héritage du C – rend le code client dépendant des types utilisés dans cette partie privée.

Un des meilleurs moyens pour encapsuler totalement la partie privée d'une classe est l'utilisation d'un pare-feu logiciel (Coplien92, Lakos96, Meyers98, Meyers99, Murray93) ou « Pimpl » (ainsi nommé en raison de l'utilisation d'un pointeur `pimpl_`[1]).

Un Pimpl est un pointeur membre particulier utilisé pour masquer la partie privée d'une classe. Il pointe vers une structure définie dans l'implémentation de la classe. Autrement dit, au lieu d'écrire :

```
// Fichier x.h
class X
{
  // Membres publics et protégés
private:
  // Membres privés. S'ils changent,
  // tout le code client doit être recompilé
};
```

on écrira :

```
// Fichier x.h

class X
{
  // Membres publics et protégés
private:
  struct XImpl* pimpl_;
    // Pointeur vers une structure qui sera définie plus tard
};

// Fichier x.cpp
struct X::XImpl
{
  // contient les membres privés. Désormais,
  // il est possible de le changer sans avoir
  // à recompiler le code client
};
```

Chaque objet `X` alloue dynamiquement un objet interne `XImpl`. D'une certaine manière, nous avons déplacé la partie privée de l'objet pour la cacher derrière un pointeur opaque, le « Pimpl ».

L'avantage principal de cette technique est de réduire les dépendances au sein du code et donc de minimiser les risques de recompilation :

1. Comme nous allons le voir, le terme « Pimpl » fait référence à l'utilisation d'un *pointeur* membre (d'où *p*) vers une structure privée contenant les détails de l'*impl*émentation, d'où « *pimpl* ».

- Les types utilisés dans l'implémentation de la classe n'ont pas besoin d'être vus depuis le code client, ce qui permet de réduire le nombre de fichiers en-tête inclus et donc les temps de compilation.
- L'implémentation d'une classe peut être modifiée sans que le code client ait besoin d'être recompilé.

En revanche, cette technique dégrade un peu les performances :

- Chaque construction / destruction s'accompagne d'une allocation / désallocation
- Chaque accès à un membre privé nécessite au moins une indirection supplémentaire (si le membre privé auquel on accède appelle lui-même une fonction dans la partie visible de la classe, cela nécessitera plusieurs indirections)

En appliquant cette technique à notre exemple de code, nous pouvons éliminer trois fichiers en-tête qui n'étaient utilisés que par la partie privée de X :

```
#include <list>
#include « c.h » // classe C
#include « d.h » // classe D
```

Le fichier « `c.h` » doit être remplacé par une déclaration « `class C;` » car le type `C` est utilisé comme paramètre et type de retour dans la partie publique de la classe. En revanche, les deux autres fichiers peuvent disparaître complètement.

Recommandation

Lorsque vous implémentez une classe destinée à être largement utilisée, utilisez la technique du pare-feu logiciel (ou technique du « Pimpl ») afin de masquer la partie privée de la classe. Pour cela, déclarez un pointeur membre « `struct XxxxImpl* pimpl_` », vers une structure qui sera définie dans l'implémentation de la classe et contiendra les variables et fonctions membres privés. Par exemple : « `class Map { private : struct MapImpl* pimpl_; };` ».

Voici à quoi ressemble le code après modification :

```
//  x.h: Après mise en place d'un "Pimpl"
//
#include <iosfwd>
#include "a.h"  // classe A (a des fonctions virtuelles)
#include "b.h"  // classe B (n'a pas de fonctions virtuelles)
class C;
class E;
class X : public A, private B
{
public:
     X( const C& );
  B  f( int, char* );
  C  f( int, C );
  C& g( B );
  E  h( E );
  virtual std::ostream& print( std::ostream& ) const;
```

```
private:
  struct XImpl* pimpl_;
    // Pointeur vers une structure définie dans
    // l'implémentation de la classe
};
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
  return x.print(os);
}
```

Le code client n'est plus dépendant de la partie privée de `X`, qui se retrouve dans le fichier source « `x.cpp` » :

```
//  Fichier source : x.cpp
//
struct X::XImpl
{
  std::list<C> clist_;
  D            d_;
};
```

En conclusion, nous avons réussi à éliminer trois fichiers en-tête, ce qui n'est pas négligeable. Nous allons pouvoir faire encore mieux dans le problème suivant, où nous allons être autorisés à modifier plus largement encore la structure de `X`.

Pb n° 28. Éviter les compilations inutiles (3e partie) — Difficulté : 7

Les fichiers en-tête inutiles ont été supprimés, la partie privée de la classe a été correctement encapsulée... est-il possible de découpler encore plus fortement notre classe du reste du programme ? Réponse dans ce problème, qui nous permettra de revenir sur quelques principes de base de la conception de classes.

Repartons une nouvelle fois de l'exemple étudié dans les deux problèmes précédents. Nous nous sommes à présent affranchis d'un grand nombre de fichiers en-tête superflus. Est-il possible de supprimer encore d'autres directives « `#include` » ? Si oui, comment ?

Vous pouvez apporter des modifications à la classe `X`, dans la mesure où son interface publique est inchangée et où le code client ne nécessite aucun changement – à part une recompilation. Encore une fois, les commentaires sont importants.

```
//  x.h: Après mise en place d'un "Pimpl"
//
#include <iosfwd>
#include "a.h"  // classe A (a des fonctions virtuelles)
#include "b.h"  // classe B (n'a pas de fonctions virtuelles)
```

```
class C;
class E;
class X : public A, private B
{
public:
     X( const C& );
  B  f( int, char* );
  C  f( int, C );
  C& g( B );
  E  h( E );
  virtual std::ostream& print( std::ostream& ) const;
private:
  struct XImpl* pimpl_;
    // Pointeur vers une structure définie dans
                  // l'implémentation de la classe
};
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
  return x.print(os);
}
```

Solution

De trop nombreux développeurs ont la mauvaise habitude d'utiliser l'héritage plus souvent que nécessaire. Comme nous avons pu le voir dans le problème n° 24, la relation d'héritage (qui signifie notamment « EST-UN ») est beaucoup plus forte qu'une relation du type « A-UN » ou « UTILISE-UN ». Pour minimiser les dépendances au sein du code, préférez systématiquement la composition (utilisation d'un objet membre) à l'héritage, lorsque cela suffit. Pour paraphraser Albert Einstein : « Utilisez un couplage aussi fort que nécessaire, mais pas plus fort. »

Dans notre exemple, `X` dérive de `A` de manière publique et de `B` de manière privée. Comme nous l'avons vu dans le chapitre précédent, l'héritage public ne doit être utilisé que pour implémenter une relation « EST-UN » et satisfaire au principe de substitution de Liskov [1].

La classe `A` comportant des fonctions virtuelles, on peut considérer que la relation entre `X` et `A` est de type « EST-UN » et que donc l'héritage public se justifie. En revanche, `B` est une classe de base privée de `X` ne comportant pas de fonctions virtuelles. L'unique raison qui pourrait justifier l'utilisation de l'héritage privé plutôt que la composition serait le besoin pour `X` de redéfinir des fonctions virtuelles ou d'accéder à des membres protégés de `B`[2]. Si on considère que ce n'est pas le cas de la classe `X`, alors il

1. On peut trouver un certain nombre d'articles relatifs au LSP (Liskov Substitution Principe) sur le site www.objectmentor.com. Voir aussi Martin95.
2. Il peut exister d'autres situations nécessitant une relation d'héritage, néanmoins elles sont très peu nombreuses. Voir Sutter98(a) et Sutter99 pour un examen exhaustif des (rares) situations justifiant l'emploi de l'héritage. Ces articles exposent clairement les raisons pour lesquelles il faut en général préférer la composition à l'héritage.

est préférable de remplacer la relation d'héritage par une relation de composition, à savoir remplacer la classe de base `B` par un objet membre de type `B`.

Cet objet membre pouvant être placé dans la partie privée de `X` (`pimpl_`), nous pouvons supprimer un fichier en-tête supplémentaire :

```
#include "b.h"  // classe B (n'a pas de fonctions virtuelles)
```

En revanche, nous avons toujours besoin d'une déclaration « en avance » de B, dont le type est mentionné dans la déclaration d'une fonction membre.

Recommandation

Ne jamais utiliser l'héritage lorsque la composition suffit.

Nous aboutissons pour finir à un code très simplifié par rapport à l'original :

```
//  x.h: Après élimination de la classe de base B
//

#include <iosfwd>
#include "a.h"  // class A

class B;
class C;
class E;

class X : public A
{
public:
     X( const C& );
  B  f( int, char* );
  C  f( int, C );
  C& g( B );
  E  h( E );
  virtual std::ostream& print( std::ostream& ) const;
private:
  struct XImpl* pimpl_; // contient un objet de type B
};

inline std::ostream& operator<<( std::ostream& os, const X& x )
{
  return x.print(os);
}
```

En trois passes progressives et sans modifier l'interface publique de `X`, nous avons finalement réduit le nombreux de fichiers en-tête utilisés de huit à deux !

Pb n° 29. Pare-feu logiciels — Difficulté : 6

Utiliser la technique du « Pimpl » permet de réduire fortement les dépendances au sein du code et, par conséquent, les temps de compilation. Néanmoins, quelle est la meilleure façon d'utiliser un objet `pimpl_` et, en particulier, que faut-il mettre à l'intérieur ?

Un des inconvénients du C++ est l'obligation de recompiler tout le code utilisant une classe lorsque la partie privée de cette classe est modifiée. Pour éviter cela, on utilise couramment la technique du « Pimpl » qui consiste à masquer les détails de l'implémentation d'une classe derrière un pointeur membre :

```
class X
{
public:
  /* ... membres publics ... */
protected:
  /* ... membres protégés (?) ... */
private:
  /* ... membres privés (?) ... */
  struct XImpl* pimpl_;
    // Pointeur vers une structure définie dans
    // l'implémentation de la classe

};
```

1. Quels éléments de `X` faut-il mettre dans `XImpl` ?

- Toutes les variables privées (mais pas les fonctions) de `X`
- Tous les membres privés (variables et fonctions) de `X`
- Tous les membres privés et protégés de `X`
- Tous les membres de `X`, y compris les membres publics (l'interface publique de `X` ne contenant plus que des fonctions de transfert vers les fonctions correspondantes implémentées dans `XImpl`).

  Discutez les avantages et inconvénients de chacune de ces solutions.

2. `XImpl` doit-il contenir un pointeur vers l'objet `X` ?

Solution

La classe X utilise une technique de masquage de l'implémentation initialement documentée par Coplien (Coplien92), consistant à séparer la partie visible d'une classe et son implémentation (`XImpl`). Utilisée au départ pour gérer le partage d'une implémentation entre plusieurs utilisateurs, cette technique est maintenant couramment utilisée pour réduire les dépendances au sein d'un programme. Comme nous l'avons vu dans les problèmes 28 et 29, utiliser la classe interne `XImpl` permet de

réduire les temps de compilation grâce à l'élimination d'un certain nombre de « `#include` » et de rendre le code client d'une classe totalement indépendant de la partie privée de cette classe (il est possible d'ajouter / supprimer des membres situés dans `XImpl` sans avoir à recompiler le code client).

Prenons les questions dans l'ordre :

1. Quels éléments de `X` faut-il mettre dans `XImpl` ?

Option 1 (Note : 6/10) : Toutes les variables privées (mais pas les fonctions) de `X`. Cette option permet rendre le code client de classe indépendant des *types* utilisés dans la partie privée de `X` (dans le fichier en-tête de la classe, on peut remplacer les `#include` déclarant ces types par des déclarations « en avance »). En revanche, le code client est toujours dépendant des modifications apportées aux fonctions privées.

Option 2 (Note : 10/10) : Tous les membres privés (variables et fonctions) de `X`. Cette option permet de rendre le code client totalement indépendant de la partie privée de la classe X. C'est en général la meilleure solution, à deux détails près :

- Les fonctions virtuelles privées ne seront plus visibles depuis le code extérieur : par conséquent, elles ne pourront plus être appelées par un code client, ni redéfinies dans une classe dérivée. Bien entendu, il est rare que l'on définisse des fonctions virtuelles privées, mais cela peut arriver (elles ne seront alors appelables que par des fonctions ou des classes *amies*).
- Il sera parfois nécessaire d'inclure dans `XImpl` un pointeur vers l'objet externe (souvent appelé `self_`) afin de pouvoir faire appel aux membres non-privés, ce qui introduit un niveau d'indirection supplémentaire pouvant dégrader les performances. Pour éviter cela, un bon compromis est d'inclure également dans `XImpl` ce type de membres (voir la réponse à la question 2, consacrée à ce problème).

Recommandation

Lorsque vous implémentez une classe destinée à être largement utilisée, utilisez la technique du pare-feu logiciel (ou technique du « Pimpl ») afin de masquer la partie privée de la classe. Pour cela, déclarez un pointeur membre « `struct XxxxImpl* pimpl_` », vers une structure qui sera définie dans l'implémentation de la classe et contiendra les variables et fonctions membres privés. Par exemple : « `class Map { private : struct MapImpl* pimpl_; };` »

Option 3 (Note : 0/10) : Tous les membres privés et protégés de `X`. Cette option est à éviter absolument : il ne faut jamais masquer les membres protégés d'une classe dans un « Pimpl », car cela rend impossible leur utilisation par les classes dérivées, ce qui est pourtant leur finalité !

Option 4 (Note : 10/10 dans certaines situations) : Tous les membres de `X`, y compris les membres publics (l'interface publique de `X` ne contenant plus que des fonctions de transfert vers les fonctions correspondantes implémentées dans `XImpl`). Cette option peut être utile dans certains cas, dans la mesure où elle permet de se passer de pointeur vers l'objet externe, toutes les fonctions de la classe étant accessibles directement depuis `XImpl`. Le revers de la médaille est que ce sont maintenant les fonctions publiques transférant les appels vers `XImpl` qui subissent un niveau d'indirection supplémentaire.

2. `XImpl` doit-il contenir un pointeur vers l'objet `X` ?

Parfois, oui. Un pointeur vers l'objet externe est nécessaire dans le cas où une fonction privée – implémentée dans `XImpl` – aurait besoin d'accéder à une fonction non-privée de la classe, ce qui peut tout à fait arriver. Pour éviter l'utilisation d'un pointeur de ce type, qui introduit un niveau d'indirection supplémentaire, on peut utiliser une solution intermédiaire entre les options 2 et 4 : mettre dans `XImpl` tous les membres privés et les fonctions publiques et protégées appelées depuis les fonctions privées.

Pb n° 29. La technique du « Pimpl Rapide » Difficulté : 6

Réduction des dépendances rime malheureusement souvent avec dégradation des performances. Dans ce problème, nous étudions une technique particulière de « Pimpl » efficace tant au niveau de la compilation que de l'exécution.

La technique du « Pimpl » telle que nous l'avons étudiée dans les problèmes précédents nécessite l'allocation dynamique d'un objet membre contenant l'implémentation de la classe. Toute opération d'allocation dynamique (par `new` ou `malloc`) étant relativement coûteuse – notamment par rapport à un appel classique de fonction, ceci peut avoir un impact négatif sur la performance d'un programme :

```
// Technique originale
//

// Fichier y.h

class X;
class Y
{
  /*...*/
  X* px_;
};


// file y.cpp

#include "x.h"
Y::Y() : px_( new X ) {}
Y::~Y() { delete px_; px_ = 0; }
```

Une variante possible serait d'utiliser un véritable objet membre :

```
// Variante n° 1
//

// Fichier y.h
```

```
#include "x.h"

class Y
{
  /*...*/
  X x_;
};

// Fichier y.cpp
Y::Y() {}
```

Malheureusement, ceci rend le code client de `Y` dépendant de la déclaration de `X`, ce qui annule tout le bénéfice apporté par le « Pimpl ».

Voici une seconde variante, permettant d'éliminer à la fois la déclaration de `X` et le recours à l'allocation dynamique :

```
// Variante n° 2

// Fichier y.h

class Y
{
  /*...*/
  static const size_t sizeofx = /* une certaine valeur*/;
  char x_[sizeofx];
};

// Fichier y.cpp

#include "x.h"

Y::Y()
{
  assert( sizeofx >= sizeof(X) );
  new (&x_[0]) X;
}
Y::~Y()
{
  (reinterpret_cast<X*>(&x_[0]))->~X();
}
```

1. Quelle est l'occupation mémoire supplémentaire induite par la « Technique du Pimpl » par rapport à une implémentation classique ?
2. En quoi la « Technique du Pimpl » dégrade-t-elle les performances par rapport à une implémentation classique ?
3. Commentez la variante n° 2. Vous inspire-t-elle un moyen encore meilleur de s'affranchir de l'occupation mémoire supplémentaire et de la dégradation de performance générées par l'utilisation d'un « Pimpl » ?

Pour plus d'informations sur la technique du Pimpl, reportez-vous au problème n° 29.

Solution

1. *Quelle est l'occupation mémoire supplémentaire induite par la technique du Pimpl par rapport à une implémentation classique ?*

L'emploi d'un « Pimpl » induit une occupation mémoire égale au minimum à la taille d'un pointeur, parfois deux lorsque l'objet interne contient un pointeur vers l'objet externe. Avec un compilateur utilisant des pointeurs sur 32 bits – le plus courant actuellement, ceci représente théoriquement un espace mémoire de 4 ou 8 octets. En pratique, ces 8 octets peuvent se transformer en 14 octets ou plus, en raison de contraintes d'alignements mémoire. Pour mieux comprendre ce dernier point, considérez l'exemple suivant :

```
struct X { char c; struct XImpl* pimpl_; };
struct X::XImpl { char c; };
int main()
{
  cout << sizeof(X::XImpl) << endl
       << sizeof(X) << endl;
}
```

Avec la plupart des compilateurs classiques utilisant des pointeurs codés sur 32 bits, ce code va produire le résultat suivant :

```
1
8
```

Autrement dit, l'occupation mémoire générée par l'emploi de `pimpl_` est de 7 octets (et non pas 4, comme on pourrait s'y attendre). Ce qui justifie ces trois octets supplémentaires est le fait qu'en général, les compilateurs alignent les pointeurs sur des adresses multiples de 4 (pour des raisons d'optimisation). La structure `X` comportant un premier membre type `char` (1 octet), le compilateur introduit 3 octets supplémentaires vides avant le pointeur `pimpl_`, qui représente donc globalement un « coût » de 7 octets. Si la structure `XImpl` contient elle-même un pointeur vers l'objet externe, ce coût peut s'élever jusqu'à 14 octets dans le cas d'une machine 32 bits et 30 octets pour une machine 64 bits.

Il n'est en général pas possible de s'affranchir de ces octets supplémentaires, sinon au prix de manœuvres dangereuses et peu recommandables que je détaillerai plus loin, dans le paragraphe séparé « Les dangers de l'optimisation à tout va ».

Dans les cas très spécifiques où la minimisation de la taille mémoire a vraiment une importance cruciale pour votre programme, un moyen – malheureusement *non portable* – de supprimer en partie ces octets vides est le recours aux directives `#pragma` du compilateur permettant de redéfinir l'alignement par défaut pour une classe donnée. Ainsi, dans le cas où votre compilateur autoriserait le contrôle de l'alignement par l'utilisateur, vous pouvez gagner jusqu'à 6 octets par objet X – au prix d'une dégradation (minuscule) des performances due au fait que le pointeur ne sera plus optimisé. Autant dire que le gain d'espace mémoire est ridiculement faible (sauf dans le cas d'un nombre énorme d'instances) eu égard au fait que le code a été rendu

non portable. D'une manière générale, il est sage d'éviter d'optimiser plus que nature un programme au détriment de sa lisibilité ou de sa portabilité.

2. *En quoi la technique du Pimpl dégrade-t-elle les performances par rapport à une implémentation classique ?*

La technique du Pimpl est pénalisante à deux niveaux :

- D'une part, chaque construction/destruction d'un objet `X` s'accompagne d'une allocation/désallocation d'un objet `XImpl` – ce qui est une opération coûteuse[1].
- D'autre part, chaque accès à un membre de `XImpl` depuis un membre de la classe `X` passe au minimum par un niveau d'indirection (plusieurs niveaux dans le cas où `XImpl` contiendrait un pointeur vers l'objet externe qu'elle utilise pour appeler elle-même des fonctions dans la classe `X`).

Pour diminuer le coût en performance, il est possible d'utiliser la technique du « Pimpl Rapide » que nous allons étudier maintenant.

3. *Commentez la variante n° 2 . Vous inspire-t-elle un moyen encore meilleur de s'affranchir de l'occupation mémoire supplémentaire et de la dégradation de performance générées par l'utilisation d'un « Pimpl » ?*

Cette variante n° 2 est incorrecte et dangereuse ! Ne l'utilisez PAS ! Elle présente un grand nombre de problèmes et risques qui seront décrits dans le paragraphe « Les dangers de l'optimisation à tout va » situé à la fin de ce chapitre.

Néanmoins, l'idée sous-jacente de cette variante consistant à utiliser une zone mémoire « pré-allouée » pour réduire le coût en performance induit par des allocations dynamiques répétées est tout à fait intéressante. Nous allons voir maintenant comment mettre en oeuvre correctement ce type de mécanisme en utilisant la technique dite du « Pimpl Rapide », qui consiste à redéfinir pour la structure `XImpl`, un opérateur `new()` spécifique utilisant des blocs pré-alloués :

```
// Fichier x.h

class X
{
  /*...*/
  struct XImpl* pimpl_;
};
// Fichier x.cpp

#include "x.h"
struct X::XImpl
{
  /*...mettre ici les membres privés de X */
  static void* operator new( size_t )   { /*...*/ }
  static void  operator delete( void* ) { /*...*/ }
};
X::X() : pimpl_( new XImpl ) {}
```

1. ...en comparaison avec une opération classique comme un appel de fonction. Il est fait ici référence aux opérateurs d'allocation prédéfinis ::`operator new()` et `malloc()`.

```
X::~X() { delete pimpl_; pimpl_ = 0; }
```

Les opérateurs `new()` et `delete()` redéfinis pour la structure `XImpl` doivent être implémentés de manière à gérer un ensemble de blocs pré-alloués et à renvoyer, lors d'une demande d'allocation, l'adresse d'un bloc libre : ceci permet de gagner nettement en performance par rapport à une allocation dynamique normale. À des fins de réutilisation, on implémente en général une classe utilitaire séparée contenant des fonctions d'allocation et de désallocation auxquelles l'implémentation des opérateurs `new()` et `delete()` redéfinis fait appel :

```
template<size_t S>
class FixedAllocator
{
public:
  void* Allocate(); // Renvoie l'adresse d'un bloc de taille S
  void  Deallocate( void* ); // 'Libére' le bloc
private:
  /*...Liste de blocs mémoires définis (« static »)...*/
};
```

L'implémentation ne sera pas détaillée ici, cette technique courante de pré-allocation étant décrite dans de nombreux ouvrages C++.

Néanmoins, il faut signaler un danger potentiel de la classe `FixedAllocator` ci-dessus : si la partie privée utilise des variables statiques pour les blocs mémoires pré-alloués, ceci peut poser un problème si la fonction `Deallocate()` est appelée depuis le destructeur d'un objet statique – l'ordre de destruction des variables statiques lors de la fin de l'exécution d'un programme étant indéterminé. Une meilleure implémentation, éliminant ce type de risque, serait d'utiliser d'un objet membre gérant lui-même une liste de blocs mémoire statiques :

```
class FixedAllocator
{
public:
  static FixedAllocator& Instance();
  void* Allocate( size_t );
  void  Deallocate( void* );
private:
  /*...objet membre gérant une liste de blocs ...*/
 };
```

Pour une efficacité optimale, il est judicieux de gérer plusieurs listes de blocs pré-alloués de taille différente (par exemple, de 8, 16, 32 octets...) : on parle souvent alors d'arène mémoire (*memory arena*). Enfin, l'idéal est de définir une classe de base implémentant les opérateurs `new()`et `delete()` en fonction de la classe `FixedAllocator`, de laquelle on fera dériver `XImpl` :

```
struct FastArenaObject
{
  static void* operator new( size_t s )
  {
    return FixedAllocator::Instance()->Allocate(s);
  }
```

```
  static void operator delete( void* p )
  {
    FixedAllocator::Instance()->Deallocate(p);
  }
};
```

Nous aboutissons ainsi à un ensemble de classes réutilisables permettant d'implémenter la technique dite du « Pimpl rapide » :

```
//  Il suffit de faire dériver XImpl de
//  la classe FastArenaObject :

struct X::XImpl : FastArenaObject
{
  /*...Membres privés de X...*/
};
```

Ce qui permet d'obtenir une version optimisée de notre exemple initial :

```
// Fichier y.h

class X;
class Y
{
  /*...*/
  X* px_;
};

// Fichier y.cpp

#include "x.h" // À présent, X dérive de FastArenaObject

Y::Y() : px_( new X ) {}
Y::~Y() { delete px_; px_ = 0; }
```

Cette technique permet de diminuer nettement le temps utilisé pour les opérations d'allocation. Mais attention, ce n'est pas une solution parfaite ! La contrepartie de ce gain de performance est une utilisation mémoire superflue et une fragmentation accrue dues aux blocs de mémoire pré-alloués.

En conclusion, réservez l'emploi de la technique du « Pimpl » en général et de celle du « Pimpl rapide » en particulier aux situations où les gains apportés sont notables par rapport aux contreparties imposées. Ce conseil vaut d'ailleurs pour n'importe quel type d'optimisation.

Recommandation

Évitez les optimisations superflues sauf lorsque la recherche de la performance maximale est une contrainte majeure.

Les dangers de l'optimisation à tout-va

Nous détaillons ici les raisons pour lesquelles la variante n° 2 est extrêmement dangereuse et doit absolument être évitée. Ce paragraphe a été clairement séparé de la solution du problème afin de bien signaler au lecteur qu'il s'agit d'un code cité à titre de contre-exemple, qu'il ne faut surtout pas employer.

Reprenons le code en question :

```
// Fichier y.h : EXEMPLE À NE PAS SUIVRE

class Y
{
  /*...*/
  static const size_t sizeofx = /* une certaine valeur*/;
  char x_[sizeofx];
};

// Fichier y.cpp : EXEMPLE À NE PAS SUIVRE

#include "x.h"

Y::Y()
{
  assert( sizeofx >= sizeof(X) );
  new (&x_[0]) X;
}
Y::~Y()
{
  (reinterpret_cast<X*>(&x_[0]))->~X();
}
```

CE CODE EST DANGEREUX ! Certes, il optimise l'espace mémoire et les temps d'allocation[1]. Certes, il est possible qu'il fonctionne correctement sur votre compilateur. Cependant, il n'est pas portable, difficile à maintenir et surtout, il peut s'avérer catastrophique à l'exécution, et ceci de manière erratique et imprévisible. En voici les raisons :

1. *Alignement.* La mémoire alloué dynamiquement par `new()` ou `malloc()` est correctement alignée. En revanche, il n'est pas du tout assuré qu'un tableau statique le sera :

```
char* buf1 = (char*)malloc( sizeof(Y) );
char* buf2 = new char[ sizeof(Y) ];
char  buf3[ sizeof(Y) ];
new (buf1) Y;     // OK, buf1 est alloué dynamiquement (a)
new (buf2) Y;     // OK, buf2 est alloué dynamiquement (b)
new (&buf3[0]) Y; // Erreur, buf3 risque de ne pas être
                  // correctement aligné (c)
(reinterpret_cast<Y*>(buf1))->~Y(); // OK
(reinterpret_cast<Y*>(buf2))->~Y(); // OK
(reinterpret_cast<Y*>(&buf3[0]))->~Y(); // Erreur !
```

1. En revanche, il limite un peu les avantages apportés par le `pimpl_`, dans la mesure où le code client doit être recompilé si la valeur de `sizeofximpl` change !

Les lignes `(a)` et `(b)` ne sont pas idéales. En revanche, elles sont techniquement correctes, alors que la ligne `(c)` présente un risque d'erreur à l'exécution, d'autant plus sournois qu'il est possible que le code s'exécute *parfois* correctement aux moments où vous avez de la chance[1].

2. *Fragilité*. Une telle implémentation impose de nombreuses contraintes au niveau de la classe X ! Par exemple, l'auteur de X doit obligatoirement redéfinir l'opérateur d'affectation – ou alors, interdire l'affectation[2]. D'une manière générale, il faudrait s'assurer, pour chaque fonction membre, que son implémentation est compatible avec l'emploi du tableau statique `pimpl_`, au risque de rendre la classe X inutilisable !

3. *Coût de maintenance*. Lorsque la taille de `XImpl` varie, il faut modifier en conséquence la valeur de `sizeofximpl` et recompiler le code client, ce qui ne facilite pas la maintenance.

4. *Inefficacité*. Pour limiter le coût de maintenance, on pourrait fixer pour `sizeofximpl` une valeur délibérément grande, ce n'est pas non plus une bonne solution car cela gaspille inutilement de l'espace mémoire.

5. *Mauvais style de programmation*. Ce n'est pas certes pas un argument technique, néanmoins, il n'est en général pas bon signe de voir un développeur faisant appel à des trucs et astuces telle l'allocation d'objets au sein d'un tableau de caractères ou l'affectation en utilisant un opérateur `new()`qui force l'adresse d'allocation et l'appel explicite d'un destructeur (voir à ce sujet le problème n° 41 pour une liste des conséquences fâcheuses pouvant survenir).

En conclusion, l'emploi d'un tableau statique d'octets dans lequel on alloue un objet dynamiquement en forçant l'adresse d'allocation est à éviter absolument !

1. Pour être honnête, il existe un moyen de résoudre ce problème d'alignement : il suffit de remplacer la variable membre `pimpl_` par une union du type « `union { max_align m, char pimpl_[sizeofximpl]; };` ». Néanmoins, ceci ne résoud pas les autres problèmes exposés ci-dessous !
2. Il faut également s'assurer que cet opérateur se comporte correctement en présence d'exceptions. À ce sujet, voir les problèmes 8 à 17.

Résolution de noms, espaces de nommage, principe d'interface

Pb n° 31. Résolution de noms, principe d'interface (1re partie) — Difficulté : 9 1/2

Lorsque vous appelez une fonction qui a été surchargée – autrement dit, qui existe sous plusieurs occurrences portant le même nom – le compilateur détermine quelle fonction appeler suivant un algorithme de résolution de noms, dont la logique réserve parfois quelques surprises.

Examinez l'exemple de code ci-dessous. Déterminez quelles sont les fonctions appelées et pourquoi. Quelles conclusions pouvez-vous en tirer ?

```
namespace A
{
  struct X;
  struct Y;
  void f( int );
  void g( X );
}

namespace B
{
  void f( int i )
  {
    f( i );   // Quelle fonction f() ?
  }
```

```
    void g( A::X x )
    {
      g( x );   // Quelle fonction g() ?
    }
    void h( A::Y y )
    {
      h( y );   // Quelle fonction h() ?
    }
  }
```

Solution

Deux des trois cas sont relativement simples. En revanche, le troisième requiert une bonne maîtrise des mécanismes de résolution des noms en C++, notamment la connaissance de la « règle de Koenig ».

Commençons par les cas faciles :

```
namespace A
{
  struct X;
  struct Y;
  void f( int );
  void g( X );
}
namespace B
{
  void f( int i )
  {
    f( i );   // Quelle fonction f() ?
  }
}
```

La fonction `f()` s'appelle elle-même de manière récursive, car l'autre fonction `f()` déclarée au sein de l'espace de nommage `A` n'est pas visible depuis `B`. Si l'auteur du programme avait ajouté « `using A` » ou « `using A::f` », alors la fonction `A::f(int)` aurait été prise en compte lors de résolution de noms – ce qui aurait d'ailleurs conduit à une ambiguïté que le compilateur n'aurait pas réussi à résoudre.

Le cas de la fonction `g()` est un peu plus subtil :

```
  void g( A::X x )
  {
    g( x );   // Quelle fonction g() ?
  }
```

Cet appel de fonction est ambigu et provoquera une erreur de compilation, à moins que le développeur ne précise explicitement laquelle des deux fonctions `A::g()` ou `B::g()` il souhaite appeler.

Pourquoi cette différence avec le premier cas ? À première vue, on pourrait penser qu'en l'absence de déclaration « `using A` » dans `B`, il n'y a aucune raison que `A ::g()`

soit vue depuis B. C'est pourtant le cas, en vertu d'une règle particulière utilisée lors de la résolution de noms :

> **Règle de Koenig**[1] **(simplifiée) :**
> *Lors de l'appel d'une fonction prenant en paramètre un argument de type objet (en l'occurrence* X, de type A::X), le compilateur fait également intervenir, dans l'algorithme de résolution de noms, les fonctions de l'espace de nommage où est déclaré le type de cet argument (en l'occurrence A).

Voici un autre exemple d'application de la règle de Koenig :

```
namespace NS
{
  class T { };
  void f(T);
}

NS::T parm;
int main()
{
  f(parm);    // Appelle NS::f()
}
```

Si elle peut avoir des implications relativement subtiles dans certains cas (notamment au niveau de l'isolation des espaces de nommage et de l'analyse des dépendances entre classes, que nous aurons l'occasion d'aborder dans le problème suivant), la règle de Koenig n'en demeure pas moins essentielle (pour vous en convaincre, il suffit de remplacer, dans l'exemple de code ci-dessus, NS par std, T par string et f par operator<< et d'imaginer ce qui se passerait si cette règle n'existait pas)

Finissons par le troisième cas, facile celui-ci :

```
  void h( A::Y y )
  {
    h( y );   // Quelle fonction h()?
  }
}
```

Il n'y a qu'une seule fonction h(), donc la fonction h() s'appelle de manière récursive, à l'infini. La règle de Koenig s'applique – la fonction h()prenant en argument un paramètre de type A::Y déclaré dans A, les fonctions de A sont étudiées au moment de la résolution des noms, mais aucune ne correspond à la signature de B::h().

Quelles conclusions peut-on en tirer?

Résumons la situation : le comportement d'une fonction contenue dans un espace de nommage B est rendu dépendant d'une fonction contenue dans un autre espace de nommage A, complètement indépendant du premier, par le simple fait qu'elle prend un argument dont le type est déclaré dans A. Ceci signifie que les espaces de nommage

1. Du nom de son auteur Andrew Koenig, membre éminent de l'équipe C++ d'AT&T et du comité du normalisation C++. Voir aussi Koenig97.

ne sont pas aussi indépendants qu'il n'y parait ! Bien entendu, il ne s'agit pas ici de remettre en question cette fonctionnalité très utile du langage – qui remplit en général très bien son rôle d'isolation – mais uniquement de signaler, dans ce problème et le suivant, les quelques rares cas dans lesquels les espaces de nommage ne sont pas hermétiquement indépendants.

Pb n° 32. Résolution de noms, principe d'interface (2e partie) Difficulté : 9

Quel signifie véritablement le terme « classe » en C++ ? Qu'est-ce est l'« interface » d'une classe ?

Considérons la définition traditionnelle d'une classe :

> « Une classe est constituée d'un ensemble de données et de fonctions manipulant ces données »

Quelles sont les fonctions qui, selon vous, font partie d'une classe ? Quelles sont celles qui constituent, au sens large, l'interface de cette classe ?

Indice n° 1 : Il est clair que les fonctions membres non statiques font partie intégrante de la classe et que les fonctions membres publiques non statiques constituent, au moins en partie, l'interface de la classe. Qu'en est-il des fonctions membres statiques et des fonctions globales ?

Indice n° 2 : Inspirez-vous de la solution du problème n° 31

Solution

Au-delà de la question posée, nous examinerons également, pour approfondir le sujet, un ensemble de questions qui lui sont relatives :

- Peut-on programmer « objet » en C ?
- La notion d'interface de classe est-elle cohérente avec la règle de Koenig et avec l'exemple de Myers (qui sera décrit dans ce problème) ?
- Quel impact cette notion d'interface a-t-elle sur l'analyse des dépendances entre classes d'un programme et sur la conception orienté-objet en général ?

> « Une classe est constituée d'un ensemble de données et de fonctions manipulant ces données »

Les développeurs ont souvent tendance à mal interpréter cette définition, la résumant hâtivement par : « *Une classe, c'est un ensemble de variables membres et de fonctions membres* ».

Ce n'est pas si simple que cela. Prenez l'exemple suivant :

```
//*** Exemple 1 (a)
class X { /*...*/ };
/*...*/
void f( const X& );
```

À la question « `f()` fait-il partie de la classe `X` », beaucoup de développeurs répondent immédiatement : « Non, car `f()` n'est pas une fonction membre de `X` ». D 'autres réalisent que, finalement, l'exemple ci-dessus n'est pas foncièrement différent de celui-ci :

```
//*** Exemple 1 (b)
class X
{
  /*...*/
public:
  void f() const;
};
```

Aux droits d'accès près[1], la fonction `f()` est conceptuellement identique dans les deux exemples. La seule différence de forme est que, dans le premier cas, elle prend en paramètre une référence constante *explicite* vers `X`, alors que dans le second cas, elle prend un pointeur constant *implicite* vers `X` (`this`).

Conceptuellement parlant, la fonction `f()` du premier exemple semble fortement liée à `X` : elle effectue une opération sur `X` et est située dans le même fichier en-tête que la déclaration de `X`.

C'est surtout ce deuxième point qui est déterminant, si la fonction `f()`n'avait pas été située dans le même fichier en-tête que `X`, ce ne serait qu'une fonction globale utilisatrice de `X` comme une autre (ce n'est pas parce qu'on écrit une fonction prenant en paramètre une classe déclarée dans la bibliothèque standard qu'on modifie l'interface de cette classe).

Ceci nous conduit à l'énonciation du « Principe d'interface » :

> L'interface d'une classe `X` est constituée de toutes les fonctions, membres ou globales, qui « font référence à `X` » et « sont fournies avec `X` ».

En d'autres termes, l'interface d'une classe est l'ensemble des fonctions qui sont conceptuellement rattachées à cette classe.

Notons que toutes les fonctions membres d'une classe font partie de l'interface de cette classe : en effet, chaque fonction membre « fait référence à `X` » (les fonctions membres non statiques prennent un paramètre implicite de type `X*` ou `const X*`, les fonctions membres statiques sont dans la portée de `X`) et « est fournie avec `X` » (fait partie de la définition de `X`).

Étudions maintenant le cas de notre fonction `f()`de l'exemple `1(a)`: `f()` « fait référence à `X` » (elle prend un paramètre de type `const X&`) et f() est « fournie avec `X` » car elle est dans le même fichier en-tête que la déclaration de `X`. Par conséquent, on peut considérer que `f()` fait partie de la classe `X`.

1. Et encore, il serait possible de déclarer `f()` amie de `X` dans l'exemple 1(a).

Remarque : on peut considérer qu'une fonction globale « fait partie » d'une classe lorsqu'elle est déclarée dans le même fichier en-tête ou dans le même espace de nommage que cette classe[1].

Ainsi, l'interface de classe est l'ensemble des fonctions qui font *conceptuellement* partie de cette classe. Comme le montre l'exemple suivant, il est donc tout à fait possible qu'une fonction globale (non membre, donc) fasse « partie » d'une classe :

```
//*** Exemple 1 (c)
class X { /*...*/ };
/*...*/
ostream& operator<<( ostream&, const X& );
```

Ici, l'auteur de la classe `X` fournit une implémentation spécifique de l'opérateur `<<` permettant d'afficher les valeurs d'un objet de type `X`. Cet opérateur doit être implémenté sous la forme d'une fonction globale car il n'est pas possible de modifier la déclaration de la classe `ostream` de la bibliothèque standard. Il est clair que l'opérateur `<<` est rattaché conceptuellement à la classe `X` car sans lui, les fonctionnalités de la classe X seraient moindres. Eh bien cet exemple ne diffère en rien des deux précédents : `operator<<` fait référence à `X` et est fournie avec `X`, donc fait partie de `X`. Attention, les deux conditions doivent être remplies ! À titre de contre-exemple, `operator<<` fait référence à `ostream` mais n'est pas fournie avec `ostream`, donc `operator<<` ne fait pas partie de `ostream`[2].

Revenons à notre définition initiale du terme « classe » :

> « Une classe est constituée d'un ensemble de données et de fonctions manipulant ces données »

On ne peut, en conclusion, que constater que cette définition est parfaitement juste, les « fonctions » mentionnées pouvant tout à fait être membres ou ne pas l'être.

Continuons à nous interroger sur cette notion d'interface de classe. Est-elle spécifique au C++ ? Ou est-ce un concept orienté objet pouvant s'appliquer à d'autres langages ?

Intéressons-nous à l'exemple de code suivant, qui n'est pas écrit en C++ mais dans un autre langage – non orienté objet – qui devrait vous être familier :

```
/*** Exemple 2 (a) ***/
struct _iobuf { /*...contient les données...*/ };
typedef struct _iobuf FILE;
FILE* fopen ( const char* filename,
              const char* mode );
int   fclose( FILE* stream );
int   fseek ( FILE* stream,
              long  offset,
              int   origin );
```

1. Nous étudierons dans la suite de ce chapitre les relations entre interface de classe et espace de nommage, et notamment leur lien avec la règle de Koenig étudiée précédemment.
2. La similarité entre fonctions membres et non membres est encore plus marquée pour certains autres opérateurs. Par exemple, `a+b` peut faire référence soit `a.operator+(b)` soit à `operator+(a,b)` en fonction des types de `a` et `b`.

```
long  ftell ( FILE* stream );
/* etc. */
```

Nous avons ici un exemple typique de code « orienté-objet » écrit dans un langage qui ne l'est pas : une structure contenant la définition de données et un ensemble de fonctions permettant de manipuler ces données, qui prennent un argument ou renvoient un pointeur vers cette structure. Dans notre cas, la structure FILE représente un fichier et les fonctions associées permettent respectivement de construire (`fopen`), de détruire (`fclose`) ou de manipuler (`fseek`, `ftell`,...) un fichier.

Bien entendu, le langage C n'offre pas le même niveau de confort que le C++, notamment au niveau de l'encapsulation (rien n'empêche le code client de faire manipuler directement les données). Néanmoins, conceptuellement parlant, on peut tout à fait considérer que `FILE` est une classe et que les fonctions `fopen`, `fclose`, etc. font partie de la classe.

Reprenons le même exemple, en C++ cette fois :

```
//*** Exemple 2 (b)
class FILE
{
public:
  FILE( const char* filename,
        const char* mode );
 ~FILE();
  int  fseek( long offset, int origin );
  long ftell();
       /* etc. */
private:
  /*...contient les données...*/
};
```

Les paramètres `FILE*` des fonctions devenues membres ont été remplacés par des paramètres implicites `this`.

On aurait également tout à fait pu choisir de rendre certains fonctions non membres :

```
//*** Exemple 2 (c)
class FILE
{
public:
  FILE( const char* filename,
        const char* mode );
 ~FILE();
  long ftell();
       /* etc. */
private:
  /*...contient les données...*/
};
int fseek( FILE* stream,
           long  offset,
           int   origin );
```

La fonction `fseek()` fait référence à `FILE` et est fournie avec `FILE` ; elle fait par conséquent partie de l'interface de `FILE`.

Dans l'exemple 2(a), toutes les fonctions étaient globales car le langage C ne nous laissait pas d'autre choix. Mais en C++ également, il est tout à fait courant que certaines fonctions fassent partie de l'interface d'une classe sans pour autant être *membres* de cette classe : par exemple, l'opérateur `<<` ne peut pas être membre car son opérande de gauche est obligatoirement de type `ostream` ; de même, l'opérateur + ne doit en général pas être membre de manière à autoriser les conversions sur son opérande de gauche[1]. Ces opérateurs n'en font pas moins *partie* de la classe à laquelle ils se réfèrent.

Retour sur la règle de Koenig

La notion d'interface de classe telle que nous l'avons définie dans le paragraphe précédent est totalement cohérente avec la règle de Koenig.

Rappelons brièvement cette règle :

```
//*** Exemple 3 (a)
namespace NS
{
  class T { };
  void f(T);
}
NS::T parm;
int main()
{
  f(parm);    // Appelle NS::f()
}
```

Il parait ici « naturel » que ce soit la fonction `NS::f()` qui soit appelée depuis `main()`. Pourtant, ce n'est pas évident a priori, cette fonction étant située dans un autre espace de nommage (`NS`) et on pourrait s'attendre à ce que le compilateur exige une instruction « `using namespace NS` » pour prendre en compte `NS::f()` lors de la résolution de noms.

La règle de Koenig nous assure ici que la fonction `NS::f()` sera appelée sans qu'il soit nécessaire d'utiliser une instruction « `using` ». En effet, cette règle énonce que le compilateur doit, lors la résolution de noms pour l'appel d'une fonction, prendre en compte les fonctions contenues dans les espaces de nommage où sont déclarés les types des paramètres de cette fonction. Par exemple, le compilateur prend ici en compte les fonctions de `NS` car f() prend un paramètre de type `NS ::T`, défini dans `NS`.

Voici un autre exemple où la règle de Koenig s'avère bien utile :

```
//*** Exemple 3 (b)
#include <iostream>
#include <string>  //   contient la déclaration de
                   //   std::operator<< pour string
```

1. Voir le problème n° 20.

```
int main()
{
  std::string hello = "Hello, world";
  std::cout << hello;   // Appelle std::operator<<
                        // grâce à la règle de Koenig
}
```

Grâce à la règle de Koenig, le compilateur prend en compte automatiquement les fonctions de l'espace de nommage où est déclarée `string` (c'est-à-dire `std`) pour la résolution de l'appel « `std ::cout<<hello ;` ».

Sans cette règle, il y aurait la solution d'ajouter une instruction « `using std ::operator<< ;` » en haut de chaque module concerné, ce qui deviendrait vite fastidieux dans le cas de plusieurs opérateurs ; d'utiliser « `using namespace std ;` », ce qui aurait pour effet de copier tous les noms de `std` dans l'espace de nommage par défaut et, par là-même, de supprimer tout l'intérêt de l'utilisation des espaces de nommage ; ou bien encore de faire explicitement référence à la fonction « `std::operator<< (std::cout,hello) ;` » ce qui obligerait à se priver de la syntaxe habituelle des opérateurs, pourtant très confortable. Aucune de ces solutions n'est viable, ceci démontre bien à quel point la règle de Koenig est pratique !

En résumé, lorsque vous définissez, dans un même espace de nommage, une classe et une fonction globale faisant référence à cette classe[1], le compilateur va en quelque sorte établir une relation entre les deux.[2] Ceci nous ramène à la notion d'interface de classe, comme l'exemple de Myers va nous le montrer.

Règle de Koenig, suite et fin : l'exemple de Myers

Considérons l'exemple suivant – légèrement modifié par rapport au 3(a) :

```
//*** Exemple 4 (a)

// Fichier t.h
namespace NS
{
  class T { };
}

// Fichier main.cpp

void f( NS::T );

int main()
{
  NS::T parm;
  f(parm);     // OK: calls global f
}
```

1. Par valeur, référence, pointeur ou autre.
2. Certes moins forte que la relation existant entre une classe et une de ses fonctions membres. À ce sujet, voir le paragraphe « 'Être membre' ou 'faire partie' d'une classe » plus loin dans ce chapitre.

D'un côté, nous avons une classe `T` définie dans un espace de nommage `NS`, de l'autre, nous avons un code client utilisant T et définissant, pour ses propres besoins, une fonction globale `f()` utilisant `T`. Pour l'instant, rien à signaler...

Que se passe t'il si, un jour, l'auteur de « `t.h` » décide d'ajouter une fonction `f()` dans `NS` ?

```
 //*** Exemple 4 (b)
 // Fichier t.h
namespace NS
{
  class T { };
  void f( T ); // <-- Nouvelle fonction
}
void f( NS::T );
int main()
{
  NS::T parm;
  f(parm);     // Appel ambigu : Appelle NS::f
}              // ou la fonction globale f ?
```

Le fait d'ajouter une fonction `f()` à l'*intérieur* de l'espace de nommage `NS` a rendu inopérant du code client extérieur à `NS`, ce qui peut paraître plutôt gênant. Mais attendez, il y a pire :

```
//*** Exemple de Myers: "Avant"

namespace A
{
  class X { };
}
namespace B
{
  void f( A::X );
  void g( A::X parm )
  {
    f(parm);   // OK: Appelle B::f()
  }
}
```

Pour l'instant, tout va bien. Jusqu'au jour où l'auteur de `A` ajoute une nouvelle fonction :

```
//*** Exemple de Myers: "Après"

namespace A
{
  class X { };
  void f( X ); // <-- Nouvelle fonction
}

namespace B
{
  void f( A::X );
```

```
  void g( A::X parm )
  {
    f(parm);   // Appel ambigu : A::f ou B::f ?
  }
}
```

Là encore, le fait d'ajouter une fonction à l'intérieur de `A` peut rendre inopérant du code extérieur à `A`. À première vue, ceci peut paraître plutôt gênant et contraire aux principes même des espaces de nommage, dont la fonction est d'*isoler* différentes parties du code. C'est d'ailleurs d'autant plus troublant que le lien entre le code concerné et `A` n'est pas clairement apparent.

Si on y réfléchit un peu, ceci n'est pas un problème, au contraire ! Il se passe exactement ce qui doit se passer[1]. Il est tout à fait normal que la fonction `f(X)` de `A` intervienne dans la résolution de l'appel `f(parm)` – et, en l'occurrence, crée une ambiguïté. Cette fonction fait référence à `X` et est fournie avec `X`, donc, en vertu du principe de l'interface de classe, elle fait partie de la classe `X`. À la limite, peu importe que `f()` soit une fonction globale ou une fonction membre, l'important est qu'elle soit rattachée conceptuellement à `X`.

Voici une autre version de l'exemple de Myers :

```
//*** Exemple de Myers : "Après" (autre version)
namespace A
{
  class X { };
  ostream& operator<<( ostream&, const X& );
}

namespace B
{
  ostream& operator<<( ostream&, const A::X& );
  void g( A::X parm )
  {
    cout << parm; // Appel ambigu : A::operator<<
  }               // ou B::operator<< ?
}
```

Une fois encore, il est normal que l'appel `cout<<parm` soit ambigu. L'auteur de la fonction `g()` doit explicitement indiquer quelle fonction `operator<<` il souhaite appeler, celle de `B` – qu'il est normal de prendre en compte car elle est située dans le même espace de nommage – ou celle de `A` – qu'il est également normal de prendre en compte car elle « fait partie » de `X`, au nom du principe d'interface de classe :

> L'interface d'une classe `X` est constituée de toutes les fonctions, membres ou globales, qui « font référence à `X` » et « sont fournies avec `X` ».

L'exemple de Myers a le mérite de montrer que les espaces de nommage ne sont pas aussi indépendants qu'on le croit mais également que cette dépendance est *saine*

1. C'est cet exemple, cité à la réunion du Comité de Normalisation C++ en novembre 1997 à Morristown, qui m'a amené à réfléchir sur le sujet des dépendances en général.

et *voulue*. : ce n'est pas un hasard que la notion d'interface de classe soit cohérente avec la règle de Koenig. Cette règle a *justement* été établie en vertu de cette notion d'interface de classe.

Pour être complet sur le sujet, nous allons maintenant voir en quoi « être membre » d'une classe introduit une relation plus forte qu'être une fonction globale « faisant partie de la classe ».

« Être membre » ou « faire partie » d'une classe

Le principe d'interface de classe énonce que des fonctions membres et non-membres peuvent toutes deux faire conceptuellement « partie » d'une classe. Il n'en conclut pas pour autant que membres et non-membres sont équivalents. Il est clair qu'une fonction membre est liée de manière plus forte à une classe car elle a accès à l'ensemble des membres privés alors que ce n'est pas le cas pour une fonction globale – à moins qu'elle ne soit déclarée comme `friend`. De plus, lors d'une résolution de nom – faisant appel ou non à la règle de Koenig, une fonction membre prend toujours le pas sur une fonction globale :

```
// Ceci n'est PAS l'exemple de Myers

namespace A
{
  class X {} ;
  void f(X) ;
}

class B // 'class', pas 'namespace'
{
     void f(A::X);
     void g(A::X parm)
     {
        f(parm) ; // Appelle B::f(), pas d'ambiguïté
     }
};
```

Dans cet exemple, la classe `B` contenant une fonction *membre* `f(A::X)`, le compilateur fait appeler cette fonction, ne cherchant même pas à regarder ce qui est dans `A`, bien que f prenne un paramètre dont le type est déclaré dans `A`.

En résumé, une fonction membre est toujours plus fortement liée à une classe qu'une fonction non-membre – même faisant « partie » de la classe au nom de la notion d'interface de classe – notamment pour ce qui concerne l'accès aux membres privés et la résolution des noms.

Pb n° 33. Résolution de noms, interface de classe (3e partie) Difficulté : 5

Dans ce problème, nous allons mettre en pratique la notion d'interface de classe en l'appliquant tout d'abord à une question courante : quelle est la meilleure manière d'implémenter la fonction `operator<<()` ? Nous verrons également les conséquences de ce principe au niveau de la phase de conception des programmes.

Il y a en général deux manières courantes d'implémenter l'opérateur << : soit comme une fonction globale utilisant l'interface publique de classe (ou à la limite, l'interface non-publique, si elle est déclarée comme `friend`) ou comme une fonction globale appelant une fonction membre virtuelle `Print()`.

Quelle est la meilleure méthode ? Précisez les avantages et inconvénients de chacune.

Solution

Une des conséquences pratiques de la notion d'interface de classe est qu'elle permet d'identifier clairement les dépendances entre une classe et ses « clients ».

Nous allons voir ici l 'exemple concret de la fonction `operator<<`.

Classes et dépendances

Il y a classiquement deux manières d'implémenter l'opérateur <<.

La première consiste à utiliser une fonction globale faisant appel à l'interface de la classe :

```
//*** Exemple 5 (a) – sans fonction virtuelle
class X
{
  /*...aucune référence à ostream ...*/
};
ostream& operator<<( ostream& o, const X& x )
{
  /* Code réalisant l'affichage de X dans o */
  return o;
}
```

La seconde consiste à utiliser une fonction virtuelle :

```
//*** Exemple 5 (b) – avec fonction virtuelle
class X
{
  /*...*/
```

```
public:
  virtual ostream& print( ostream& );
};
ostream& X::print( ostream& o )
{
  /* Code réalisant l'affichage de X dans o */
  return o;
}
ostream& operator<<( ostream& o, const X& x )
{
  return x.print( o );
}
```

Traditionnellement, les développeurs C++ analysent ces deux options de la manière suivante :

- L'option (a) présente deux avantages : d'une part, elle minimise les dépendances – la classe `X` ne dépend pas (ou ne semble pas dépendre...) de `ostream`, car ce type n'apparaît pas dans l'interface de `X`, d'autre part, elle économise le surcoût d'un appel de fonction virtuelle.
- L'option (b) présente l'avantage d'être compatible avec les classes dérivées : un code client manipulant X de manière polymorphique (`X&` ou `X*` référençant un objet dérivé de `X`) se comportera correctement si la fonction `Print()` est redéfinie dans la classe dérivée.

Ceci est l'analyse traditionnelle, malheureusement trompeuse. Nous allons maintenant voir que le réexamen de la question à la lumière de la notion d'interface de classe conduit à une conclusion tout à fait différente : en réalité – ainsi que le commentaire en italique le laissait entendre – la classe `X` de l'option `(a)`est dépendante de `ostream`

Voici le raisonnement permettant d'aboutir à cette conclusion :

- Dans les options (a) et (b), la fonction `operator<<()` « fait référence » à la classe `X` et « est fournie avec » `X` : elle fait donc conceptuellement *partie* de la classe `X` dans les deux cas.
- Dans les deux cas, `operator<<()` fait référence à `ostream`.
- Par conséquent, dans les options (a) et (b), `X` dépend de `ostream` car, dans les deux cas, une fonction faisant partie de la classe fait référence à `ostream`.

Autrement dit, l'avantage principal de l'option (a) est caduc. Dans les deux cas, il est nécessaire d'avoir une déclaration de `ostream` dans le fichier en-tête « `x.h` » – comme en général, l'implémentation de l'opérateur << est dans le fichier source, une déclaration « en avance » suffit.

En résumé, l'unique avantage de l'option (a) est donc d'économiser le surcoût d'un appel de fonction virtuel. Ceci est plutôt mince au regard de l'avantage procuré par l'option (b). Ainsi, grâce à l'application du principe d'interface de classe, nous avons pu déterminer les véritables dépendances entre classes : il apparaît dès lors clairement que la meilleure des options est l'utilisation d'une fonction `Print()` virtuelle.

Nous avons vu ici un exemple pratique d'application de la notion d'interface de classe qui nous a permis d'analyser, au-delà de la nature de fonction membre ou non-membre, les fonctions faisant véritablement « partie » d'une classe.

Quelques résultats intéressants... et parfois surprenants

Soient `A` et `B` deux classes et `f(A,B)` une fonction globale :

- Si `f` est fournie avec `A`, alors `f` fait partie de `A` et, par conséquent, `A` dépend de `B`.
- Si `f` est fournie avec `B`, alors `f` fait partie de `B` et, par conséquent, `B` dépend de `A`.
- Si `A`, `B` et `f` sont fournies ensemble, alors `f` fait partie de `A` et `B` et, par conséquent, `A` et `B` sont interdépendantes. Ce qui était relativement jusqu'alors intuitif – si l'auteur d'une bibliothèque met à disposition deux classes et une fonction utilisant les utilisant toutes les deux, il est probable que ces trois éléments sont destinés à être utilisés ensemble – est maintenant clairement justifié par la notion d'interface de classe.

Pour finir, voyons un cas un peu plus subtil. Soient `A` et `B` deux classes et `A::g(B)` une fonction membre de `A` :

- `A` dépend de `B`, du fait de la fonction membre `A::g(B)`. Jusque là, rien de bien surprenant.
- Plus étonnant : si `A` et `B` sont fournies ensembles, alors `A::g(B)` fait partie de la classe `B`. En effet, dans ce cas, la fonction `A::g(B)` est « fournie avec » `B` et « fait référence » à `B` ; elle fait donc « partie de » `B`. Toujours dans ce cas, les deux classes `A` et `B` sont donc interdépendantes.

Il peut certes paraître étonnant d'avoir une fonction membre d'une classe « faisant partie » d'une autre classe. C'est néanmoins ce qui se produit lorsque `A` et `B` sont fournies ensemble. L'interdépendance entre `A` et `B` était intuitive – il est clair que deux classes réunies dans un même fichier en-tête sont a priori destinées à être utilisées ensemble et que les changements apportés à l'une ont un impact sur l'autre – elle est maintenant clairement identifiée par l'application du principe d'interface de classe.

Une petite remarque concernant les espaces de nommage et le fait d'être « fourni avec » :

```
//*** Exemple 6 (a)
//---Fichier a.h---
namespace N { class B; } // Déclaration en avance
namespace N { class A; } // Déclaration en avance
class N::A { public: void g(B); };
//---Fichier b.h---
namespace N { class B { /*...*/ }; }
```

Les clients de la classe `A` incluent le fichier « `a.h` » : pour eux, `A` et `B` sont fournies ensembles et sont donc interdépendantes. Les clients de la classe `B`, quant à eux, incluent « `b.h` » : pour eux, `A` et `B` ne sont pas fournies ensemble.

En conclusion de cette série de problèmes tournant autour de la notion d'interface de classe, voici les principaux points à retenir :

- La notion d'interface de classe : *l'interface d'une classe* `X` *est constituée de toutes les fonctions, membres ou globales, qui « font référence à* `X` *» et « sont fournies avec* `X` *».*

- Par conséquent, une classe peut être conceptuellement constituée de fonctions membres comme de fonctions *non-membres*. Néanmoins, les fonctions membre sont plus fortement liées à la classe.
- Pour une fonction, « être fourni avec » une classe signifie en général apparaître dans le même fichier en-tête et/ou le même espace de nommage que cette classe. Dans le premier cas, l'appartenance de la fonction à la classe se manifeste lors de l'étude des dépendances ; dans le second cas, elle se manifeste lors de la résolution des noms.

Pb n° 34. Résolution de noms, interface d'une classe (4e partie) Difficulté : 9

Pour finir cette série de problèmes consacrés à la notion d'interface de classe, nous allons étudier certains aspects subtils des mécanismes de résolution de nom.

1. Qu'appelle t-on le « masquage d'une fonction » ? Donnez un exemple de masquage de fonctions d'une classe de base par une classe dérivée.

2. Examinez le code ci-dessous. Va t-il se compiler correctement ? Sinon, pourquoi ?

```
// Exemple 2 : ce code va t-il se compiler correctement ?
//
// MonFicher.h :
namespace N { class C {}; }
int operator+(int i, N::C) { return i+1; }
// Main.cpp
#include <numeric>
int main()
{
  N::C a[10];
  std::accumulate(a, a+10, 0);
}
```

Solution

Le masquage de noms a déjà été étudié en détail dans le chapitre précédent, consacré à l'héritage. Nous en rappelons ici le principe, en réponse à la première question du problème.

Masquage de noms

Considérons le programme suivant :

```
// Exemple 1a: Exemple de fonction
//             masquée par une classe dérivée
```

```
//

struct B
{
  int f( int );
  int f( double );
  int g( int );
};

struct D : public B
{
private:
  int g( std::string, bool );
};
D   d;
int i;
d.f(i);  // OK : appelle B::f(int)
d.g(i);  // Erreur : g attend deux arguments !
```

Cet exemple produira une erreur de compilation à la ligne « `d.g(i)` » du fait que la fonction `B::g()`, masquée, n'est pas examinée lors de la résolution de noms. En effet, le fait de déclarer une fonction dans une classe dérivée masque systématiquement toutes les fonctions portant le même *nom* dans toutes les classes de base directes ou indirectes, quelle que soit leur signature. Dans notre exemple, le fait d'avoir une fonction `g` membre de `D` exclut l'examen par l'algorithme de résolution de noms de la fonction `B::f()` : ainsi, la fonction `D::g()` attendant deux arguments de type `string` et `bool` et étant, en outre, privée, une erreur de compilation se produit.

Pour mieux comprendre cette règle malheureusement ignorée par certains développeurs C++, voyons plus en détail ce qui se passe lors de la résolution de l'appel `d.g(i)` : en premier lieu, le compilateur recherche les fonctions `g()` existantes dans la portée de la classe `D`, sans se préoccuper de leur accessibilité ni de leur nombre de paramètres. S'il ne trouve *aucune* fonction nommée `g()`, et *uniquement dans ce cas-là*, le compilateur recherche des fonctions `g()` dans les portées proches (la classe de base `B`, en l'occurrence). L'opération est réitérée jusqu'à ce que le compilateur ait identifié une portée contenant au moins une fonction candidate – ou, jusqu'à ce que l'ensemble des portées possibles ait été examiné, sans succès. Si une portée contenant plusieurs fonctions candidates est identifiée, alors le choix entre ces différentes fonctions est effectué en fonction des droits d'accès et des paramètres.

Cet algorithme se justifie[1] : en effet, pour prendre un cas extrême, il paraît intuitivement incohérent qu'une fonction membre ayant pratiquement la bonne signature – à des conversions de types autorisées près – soit plus privilégiée qu'une fonction globale ayant exactement la bonne signature.

Éviter les masquage de noms involontaires

Il y a deux techniques courantes pour passer outre le masquage des noms de fonctions.

La première consiste à indiquer explicitement au compilateur le nom de la fonction à appeler :

```
// Exemple 1(b) : Appel explicite
//                d'une fonction de la classe de base
//
D   d;
int i;
d.f(i);    // Appelle B::f(int) (pas de masquage)
d.B::g(i); // Appelle explicitement B::g(int)
```

La deuxième consiste à indiquer au compilateur qu'il doit examiner `B::g()` lors de la résolution de nom à l'aide d'une instruction `using` :

```
// Exemple 1(c): Prise en compte
//              d'une fonction masquée
//
struct D : public B
{
  using B::g;
private:
  int g( std::string, bool );
};
```

Interface de classes et espaces de nommmage

Étudions maintenant la deuxième question du problème, qui va permettre de mettre en lumière les problèmes qui se présentent lorsqu'on déclare une classe et des fonctions globales qui lui sont liées (comme des opérateurs) dans deux espaces de nommage différents :

```
// Exemple 2: ce code va t'il se compiler correctement ?
//
// MonFicher.h :
namespace N { class C {}; }
int operator+(int i, N::C) { return i+1; }
// Main.cpp
#include <numeric>
int main()
```

1. On aurait pu imaginer des variantes à cet algorithme. Par exemple, examiner les portées successives tant qu'on n'a pas trouvé de fonction correspondant à la signature attendue – au lieu de se contenter d'avoir trouvé une fonction portant le bon nom. Ou encore, faire la liste des toutes les fonctions possibles présentes dans toutes les portées disponibles puis déterminer la fonction adéquate en fonction des paramètres et droits d'accès. Ces deux variantes présentent le risque de faire préférer le « lointain » au « proche ». Dans le premier cas, on risque, par exemple, de préférer une fonction « lointaine » ayant exactement la bonne signature à une fonction proche ayant une signature correcte à quelques conversions près. Dans la deuxième cas, on risque de provoquer une ambiguïté entre une fonction membre et une fonction globale ayant toutes les deux la bonne signature, plutôt que de préférer la fonction membre.

```
{
  N::C a[10];
  std::accumulate(a, a+10, 0);
}
```

Ce programme va t'il se compiler correctement ? Est-il portable[1] ?

Espaces de nommage et masquage de noms

À première vue, le code ci-dessus semble tout à fait correct. Pourtant, aussi surprenant que cela puisse paraître, il risque de ne pas se compiler correctement sur toutes les machines, ceci en fonction de l'implémentation de la bibliothèque standard dont vous disposez – même si cette implémentation est conforme à la norme C++.

Quelle est l'explication de ce mystère ? Pour le savoir, voyons de quelle manière est typiquement implémentée la fonction `accumulate`, qui est d'ailleurs en réalité un modèle de fonction :

```
namespace std
{
  template<class Iter, class T>
  inline T accumulate( Iter first,
                       Iter last,
                       T    value )
  {
    while( first != last )
    {
      value = value + *first;
      ++first;
    }
    return value;
  }
}
```

Le code de l'exemple (2) appelle la fonction `accumulate<N::C*,int>`. Par conséquent, lors de résolution de l'appel « `value + first` », le compilateur recherchera une fonction `operator+()` acceptant un `int` et un `N::C` en paramètres (ou des paramètres compatibles aux conversions autorisées près). A priori, c'est donc la fonction globale `operator+()` de l'exemple (2), correspondant à cette signature, qui *devrait* être appelée.

Malheureusement, il n'est pas *certain* que cette fonction soit effectivement appelée : ceci dépend des autres fonctions `operator+()` situées dans l'espace de nommage `std`, rencontrées par le compilateur avant l'instanciation de `accumulate<N::C*,int>`.

1. Il n'y a pas de problème de portabilité lié au fait que `std::accumulate()`risque d'appeler soit `operator+ (N::C,int)`, soit `operator+ (int,N::C)` en fonction de l'implémentation de la bibliothèque standard : la norme C++ spécifie clairement que c'est la première version qui doit être appelée. À ce titre là, l'exemple 2 est donc correct.

En effet, rappelons une nouvelle fois de quelle manière le compilateur effectue la résolution des noms lors d'un appel de fonction (on ne fait ici que reprendre les explications de la section précédente, adaptées à l'exemple en cours) : en premier lieu, le compilateur recherche les fonctions `operator+()` existantes dans l'espace de nommage `std`, sans se préoccuper de leur accessibilité ni de leur nombre de paramètres. S'il ne trouve aucune fonction nommée `operator+()` dans `std`, et *uniquement dans ce cas-là*, le compilateur recherche des fonctions `operator+()` dans les portées proches (l'espace de nommage global, en l'occurrence). L'opération est réitérée jusqu'à ce que le compilateur ait identifié une portée contenant au moins une fonction candidate – ou, jusqu'à ce que l'ensemble des portées possibles ait été examiné, sans succès. Si une portée contenant plusieurs fonctions candidates est identifiée, alors le choix entre ces différentes fonctions est effectué en fonction des droits d'accès et des paramètres.

Autrement dit, la réussite de la compilation de l'exemple 2 est totalement dépendante de l'implémentation du fichier en-tête standard `<numeric>` : si ce fichier déclare une fonction `operator+()`, quelle qu'elle soit, (indépendamment de son accessibilité ou de ses paramètres) ou s'il inclut un autre fichier en-tête déclarant `operator+()`, alors cet exemple de code ne se compilera pas. À l'inverse du C, la norme C++ ne spécifie pas explicitement la liste des fichiers en-tête standards inclus par un fichier en-tête standard donné : par exemple, en incluant `<numeric>`, il est probable – mais pas certain – que vous inclurez également `<iterator>`, fichier dans lequel sont définies plusieurs fonctions `operator+()`. Autre exemple, votre programme peut se compiler correctement jusqu'au jour où vous inclurez le fichier en-tête `<vector>`. En résumé, ce programme n'est pas portable : il peut très bien fonctionner avec certains compilateurs C++, mais refuser de se compiler avec d'autres.

Quand le compilateur s'en mêle...

Non seulement l'exemple 2 risque de ne pas de se compiler correctement, mais encore, l'erreur fournie par le compilateur a de fortes chances d'être une source de confusion supplémentaire pour le développeur. Voici, par exemple, le message d'erreur obtenu pour l'exemple 2 avec l'un des compilateurs populaires du marché :

```
error C2784: 'class std::reverse_iterator<'template-parameter-
1','template-parameter-2','template-parameter-3','template-
parameter-4','template-parameter-5'> __cdecl std::operator
+(template-parameter-5,const class
std::reverse_iterator<'template-parameter-1','template-parameter-
2','template-parameter-3','template-parameter-4','template-
parameter-5'>&)' : could not deduce template argument for
'template-parameter-5' from 'int'

error C2677: binary '+' : no global operator defined which takes
type 'class N::C' (or there is no acceptable conversion)
```

Ce message d'erreur complexe et peu exploitable est dû au modèle de fonction `operator+()` rencontré dans le fichier en-tête `<iterator>` (lui-même inclus par `<numeric>` dans cette implémentation de la bibliothèque standard).

Il n'est pas rare que les erreurs soient peu explicites dès que des modèles de classes et de fonction rentrent en ligne de compte. Les messages ci-dessus en sont un bon exemple :

- Le premier message n'est vraiment pas clair : si on comprend finalement, avec peine, que le compilateur indique qu'il a trouvé une fonction `operator+()` mais qu'il ne comprend pas comment l'utiliser, le premier réflexe à la lecture est de se demander : « Mais où donc ai-je bien pu faire appel à la classe `reverse_iterator` ? »
- Le second message est faux – une erreur du compilateur, à moitié excusable du fait que l'utilisation des espaces de nommage est récente et donc pas toujours maîtrisée en profondeur par tous les outils. Le message correct devrait être « aucun opérateur '+' global prenant en paramètre un `N::C` n'a été *trouvé* » plutôt que le message actuel « il n'existe aucun opérateur '+' global prenant en paramètre un `N::C` », car il en existe bel et bien un !

En résumé, les messages d'erreurs fournies par le compilateur dans ce genre de situation ne sont pas d'une grande aide. L'idéal serait de toute façon de modifier l'exemple 2 de manière à obtenir une implémentation portable. Nous allons voir tout de suite comment y arriver.

Comment s'en sortir ?

Nous avons déjà vu deux solutions possibles pour rendre « visible » une fonction d'une classe de base masquée depuis la classe dérivée : appeler explicitement la fonction (Exemple 1b) ou utiliser une instruction `using` (Exemple 1c). Aucune de ces deux solutions ne fonctionnera ici[1].

La seule solution viable est de mettre la fonction `operator+()` dans le même espace de nommage que la classe à laquelle elle se rattache :

```
// Exemple 2b : Solution
//
// MonFicher.h :
namespace N
{
   class C {};
   int operator+(int i, N::C) { return i+1; }
}
// Main.cpp
#include <numeric>
int main()
{
  N::C a[10];
  std::accumulate(a, a+10, 0); // Maintenant, ça fonctionne !
}
```

1. Pour être honnête, la première solution est techniquement possible mais très lourde pour le développeur, l'obligeant, à chaque appel, à faire référence explicitement au modèle de fonction `std::accumulate` en précisant ses paramètres d'instantation.

Ce code est, cette fois, tout à fait portable et fonctionne avec toute implémentation conforme de la bibliothèque standard, indépendamment des fonctions `operator+()` définies ou non dans l'espace de nommage `std` ou un autre. Comme `operator+()` est situé dans le même espace de nommage que son second paramètre, il est pris en compte lors de la résolution de l'appel `std::accumulate()` en vertu de la règle de Koenig, qui, rappelons-le, spécifie que le compilateur doit prendre en compte les espaces de nommage dans lesquels sont déclarés les types des paramètres d'une fonction lors de la résolution de l'appel de cette fonction. Ainsi, le compilateur trouve immédiatement le bon `operator+()`, évitant les problèmes de l'exemple 2 initial.

Il était prévisible que l'exemple 2 n'allait pas fonctionner, car il ne se conformait pas au Principe d'interface :

> L'interface d'une classe `X` est constituée de toutes les fonctions, membres ou globales, qui « font référence à `X` » et « sont fournies avec `X` »

Si une fonction globale (et, à plus forte raison, un opérateur) fait référence à une classe et est conceptuellement conçu pour « faire partie » de cette classe, alors il faut le « fournir avec » la classe, c'est-à-dire, entre autres choses, la déclarer dans le même espace de nommage que la classe. Tous les ennuis rencontrés dans l'exemple 2 provenaient du fait que la fonction `operator+()`, rattachée conceptuellement à la classe `C`, était située dans un espace de nommage différent de celui où est déclarée cette classe.

En conclusion, arrangez-vous pour toujours placer une classe et *toutes* les fonctions globales qui lui sont rattachées dans le même espace de nommage – lequel peut être l'espace de nommage global : ceci vous évitera de tomber sur des erreurs parfois subtiles liées à l'algorithme de résolution de noms.

Recommandation

Soyez prudents lorsque vous utilisez des espaces de nommage. Si vous placez une classe dans un espace de nommage, placez-y également toutes les fonctions globales rattachées à la classe; ceci vous évitera bien de mauvaises surprises.

Gestion de la mémoire

Pb n° 35. Gestion de la mémoire (1re partie) — Difficulté : 3

Maîtrisez-vous bien les différentes zones mémoires et leur utilité ? Ce problème va être l'occasion de tester vos connaissances.

Le C++ utilise plusieurs zones mémoires, ayant chacune des caractéristiques différentes.

Énumérez toutes les zones mémoires que vous connaissez, en précisant pour chacune les types de variables pouvant y être stockés – types prédéfinis et/ou objets – ainsi que la durée de vie des ces variables (par exemple, la pile contient des variables automatiques, qui peuvent être des objets ou des variables de type prédéfini).

Vous indiquerez également les performances relatives de ces différentes zones.

Solution

Le tableau ci-dessous recense les différentes zones mémoires utilisées par le C++. Notez bien la différence entre le tas (*heap*) et la mémoire globale (*free store*), qui sont deux zones bien distinctes, contenant chacune un certain type de variables allouées dynamiquement.

En tant que programmeur, il est important de bien différencier le *tas* de la *mémoire globale*, la norme C++ laissant délibérément la possibilité de les différencier ou non dans l'implémentation du compilateur.

Table 1: Zones mémoires utilisées par le C++

Zone mémoire	Caractéristiques – Durée de vie des variables contenues
Données constantes	La zone « Données constantes » contient toutes les variables constantes – autrement dit celles dont la valeur est connue à la compilation. Cette zone ne peut pas contenir d'objets. Les variables contenues sont crées lors du démarrage du programme et détruites à la fin de l'exécution. Ces variables sont en lecture seule (toute tentative de modification de l'une d'entre elles conduit à un résultat indéterminé, dépendant des optimisations mises en oeuvre par le compilateur)
Pile *(stack)*	La pile contient des variables automatiques (objets ou variables de types prédéfinis), qui sont allouées au début du bloc dans lequel elles sont définies et détruites à la fin de ce bloc. Les objets sont construits immédiatement après leur allocation et détruits immédiatement avant leur désallocation. Autrement dit, le programmeur n'a aucun moyen de manipuler des variables de la pile allouées mais non initialisées (à moins d'utiliser une destruction explicite suivi d'une réallocation forcée à une adresse donnée, pratique peu recommandable). Les allocations mémoires dans la pile sont nettement plus rapides que les allocations dynamiques dans le tas ou la mémoire globale (décrits plus bas), étant donné qu'il suffit de décaler un pointeur, au lieu d'opérer une recherche d'emplacement libre, plus complexe.
Mémoire globale *(free store)*	La mémoire globale contient les variables et tableaux alloués par l'opérateur `new` (et désalloués par `delete`). C'est l'une des deux zones d'allocation dynamique. Dans cette zone, la durée de vie d'un objet peut être inférieure à la durée d'allocation de la mémoire qui lui est associée : en effet, il peut arriver qu'un objet soit alloué sans être immédiatement construit et détruit sans être immédiatement désalloué. Durant la période où un objet est alloué mais inexistant (i.e. pas encore construit ou déjà détruit), la mémoire associée à cet objet peut être manipulée par l'intermédiaire d'un pointeur de type `void*`. Néanmoins, aucun des membres de l'objet (variable ou fonction) ne peut être manipulé.
Tas *(heap)*	Le tas est l'autre zone utilisée pour l'allocation dynamique. Elle contient les variables et tableaux alloués avec `malloc()` (et libérés avec `free()`). Le tas et la mémoire globale sont deux emplacements bien distincts : le résultat de la désallocation par `free()`de variables ou tableaux alloués par `new` – et vice versa – est imprévisible, bien que cela fonctionne avec les compilateurs pour lesquels `new` et `delete` sont implémentés en fonction de `malloc()` et `free()`, ce qui n'est pas le cas de tous. Une zone de mémoire allouée dans le tas peut être utilisée pour stocker un objet, à condition d'appliquer l'opérateur `new` en forçant l'adresse d'allocation (et d'effectuer une destruction explicite lors de la fin de l'utilisation de l'objet). Dans le cas d'une utilisation de ce type, les notes relatives à la durée de vie des objets dans la mémoire globale s'appliquent.
Variables globales et statiques	Les variables globales et statiques sont allouées au démarrage du programme. Néanmoins, contrairement aux variables constantes, elles peuvent n'être initialisées que plus tard, au cours de l'exécution du programme : par exemple, une variable statique définie au sein d'une fonction n'est initialisée que lors de la première exécution de la fonction. L'ordre d'initialisation des différentes variables globales et statiques des différents modules (y compris les membres statiques de classes) n'est pas défini : il est donc dangereux de faire l'hypothèse que telle variable s'initialisera avant telle autre. Là encore, les zones mémoires des objets alloués mais non initialisés peuvent être manipulées par l'intermédiaire d'un pointeur de type void*, mais aucun des membres de l'objet ne peut être manipulé (qu'il s'agisse d'une variable ou d'une fonction).

Ainsi, la section 18.4.1.1 de la norme C++ précise les points suivants :

> *L'implémentation des opérateurs* `new` et `new[]` par le compilateur est libre. En particulier, ces opérateurs sont libres de faire appel ou non aux fonctions C classiques `calloc()`, `malloc()` et `realloc()` déclarées dans `<cstdlib>`.

De plus, la norme C++ ne spécifie pas si `new` et `delete` doivent faire appel à `malloc()` et `free()` dans leur implémentation, bien qu'a contrario, il soit explicitement indiqué (20.4.6 §3 et 4) que `malloc()` et `free()`ne doivent pas faire appel à `new` et `delete` dans leur implémentation :

> *Les fonctions* `calloc()`, `malloc()` *et* `realloc()` *ne doivent pas faire appel à l'opérateur* `::new` *dans leur implémentation*
>
> *La fonction* `free()`*ne doit pas faire appel à l'opérateur* `::delete` *dans son implémentation.*

En pratique, le *tas* et la *mémoire globale* se comportent différemment : dans vos programmes, assurez-vous de ne pas les traiter comme une unique et même zone.

Recommandations

Retenez le fonctionnement des cinq zones mémoires utilisées par le C++ : pile (variables automatiques), mémoire globale (`new`/`delete`), tas (`malloc` / `free`), variables globales/statiques et données constantes.
Utilisez `new` et `delete` plutôt que `malloc` et `free`.

Pb n° 36. Gestion de la mémoire (2e partie) Difficulté : 3

Ce problème aborde différentes questions relatives à la redéfinition des opérateurs `new`/`delete` et `new[]`/`delete[]`.

Le code ci-dessous contient des exemples de classes effectuant elles-mêmes la gestion de leur propre allocation/désallocation. Examinez-le et identifiez les erreurs qu'il contient. Répondez également aux questions additionnelles.

1. Considérez le code suivant :

```
class B
{
public:
  virtual ~B();
  void operator delete  ( void*, size_t ) throw();
  void operator delete[]( void*, size_t ) throw();
  void f( void*, size_t ) throw();
};
```

```
class D : public B
{
public:
  void operator delete  ( void* ) throw();
  void operator delete[]( void* ) throw();
};
```

Pourquoi les opérateurs `delete` de `B` ont-ils deux paramètres, alors que ceux de `D` n'en ont qu'un ?

Est-il possible d'améliorer les déclarations des fonctions ?

2. Examinez maintenant le code suivant, utilisant les classes `B` et `D` définies plus haut. Indiquez, pour chacune des opérations de destruction, lequel des opérateurs `delete` est appelé (et avec quels paramètres).

```
D* pd1 = new D;
delete pd1;
B* pb1 = new D;
delete pb1;
D* pd2 = new D[10];
delete[] pd2;
B* pb2 = new D[10];
delete[] pb2;
```

3. Les affectations suivantes sont-elles correctes ?

```
B b;
typedef void (B::*PMF)(void*, size_t);
PMF p1 = &B::f;
PMF p2 = &B::operator delete;
```

Le code suivant est-il susceptible de poser des problèmes de gestion de mémoire ?

```
class X
{
public:
  void* operator new( size_t s, int )
                  throw( bad_alloc )
  {
    return ::operator new( s );
  }
};
class SharedMemory
{
public:
  static void* Allocate( size_t s )
  {
    return OsSpecificSharedMemAllocation( s );
  }
  static void  Deallocate( void* p, int i )
  {
    OsSpecificSharedMemDeallocation( p, i );
  }
};
```

```
class Y
{
public:
  void* operator new( size_t s,
                      SharedMemory& m ) throw( bad_alloc )
  {
    return m.Allocate( s );
  }
  void  operator delete( void* p,
                         SharedMemory& m,
                         int i ) throw()
  {
     m.Deallocate( p, i );
  }
};
void operator delete( void* p ) throw()
{
  SharedMemory::Deallocate( p );
}
void operator delete( void* p, std::nothrow_t&) throw()
{
  SharedMemory::Deallocate( p );
}
```

Solution

1. Considérez le code suivant :

```
class B
{
public:
  virtual ~B();
  void operator delete  ( void*, size_t ) throw();
  void operator delete[]( void*, size_t ) throw();
  void f( void*, size_t ) throw();
};
class D : public B
{
public:
  void operator delete  ( void* ) throw();
  void operator delete[]( void* ) throw();
};
```

Pourquoi les opérateurs `delete` *de* `B` *ont-ils deux paramètres, alors que ceux de* `D` *n'en ont qu'un ?*

Il n'y a pas d'autre réponse à cette question que « parce que le programmeur en a décidé ainsi ». En effet, les deux formes sont équivalentes et tout à fait valables (pour plus d'informations, se reporter à la norme C++ §3.7.3.2/3).

En revanche, il est à noter un défaut majeur dans l'implémentation des classes `B` et `D` : les opérateurs `delete` et `delete[]` ont été redéfinis sans que les opérateurs `new` et

new[] ne le soient également. C'est très dangereux, car rien n'indique que les opérateurs new et new[] par défaut, qui seront donc utilisés, effectuent des opérations symétriques de celles implémentées dans les opérateurs delete et delete[] de B et D.

Recommandation

Si vous redéfinissez des opérateurs de gestion mémoire pour une classe, n'implémentez jamais l'opérateur new (ou new[]) sans implémenter l'opérateur delete (ou delete[]), et vice-versa.

Est-il possible d'améliorer les déclarations des fonctions ?

Les opérateurs new et delete (ou new[] et delete[]) sont toujours considérés par le compilateur comme des fonctions statiques, même lorsqu'elles ne sont pas explicitement déclarées static. Il est recommandé de qualifier systématiquement ces opérateurs avec le mot-clé static, bien que ce ne soit pas exigé par le compilateur, afin de faciliter la maintenance du code.

Recommandation

Déclarez toujours explicitement les opérateurs new (new[]) et delete (delete[]) comme des fonctions statiques, nonobstant le fait que, si le mot-clé static est omis, les fonctions seront tout de même considérées comme statiques par le compilateur.

2. *Indiquez, pour chacune des opérations de destruction, lequel des opérateurs* `delete` *est appelé (et avec quels paramètres).*

```
D* pd1 = new D;
delete pd1;
```

C'est la fonction `D::operator delete(void*)` qui est appelée

```
B* pd1 = new D;
delete pd1;
```

C'est encore la fonction `D::operator delete(void*)` qui est appelée. En effet, le fait que le destructeur de B soit virtuel implique non seulement que le destructeur de D sera appelé, mais également que l'opérateur `D ::operator delete (void*)`le sera, bien que la fonction `B ::operator delete (void*)` n'est pas (et d'ailleurs, ne peut pas être) virtuelle.

Pour mieux comprendre ce dernier point, il faut savoir qu'en général, les compilateurs implémentent un destructeur comme une fonction recevant une variable booléenne signifiant « lorsque j'aurai détruit l'objet, devrai-je le désallouer ? », qui est positionnée à false dans le cas d'un objet automatique et à true dans le cas d'un objet alloué dynamiquement. La dernière opération que fait le destructeur est de tester la valeur de cette variable et, si elle vaut true, d'appeler l'opérateur delete de l'objet venant d'être détruit[1].

1. L'implémentation correspondante dans le cas d'un tableau est laissée aux bons soins du lecteur, à titre d'exercice.

C'est ainsi que l'opérateur `delete` donne l'impression de se comporter comme une fonction virtuelle alors que, dans les faits, il n'est pas virtuel (et ne peut pas l'être, puis qu'il est statique).

```
D* pd2 = new D[10];
delete pd2;
```

Appelle l'opérateur `D::operator delete[](void*)`

```
B* pd2 = new D[10];
delete pd2;
```

Le résultat que produit ce code est indéterminé. En effet, la norme C++ impose que le type du pointeur passé à une fonction `operator delete[]` soit le même que le type des objets vers lesquelles pointent ce pointeur. Pour plus d'informations sur ce sujet, voir Meyers99 : « Never Treat Arrays Polymorphically »

Recommandation

Ne considérez pas les tableaux comme des types polymorphiques.

3. *Les affectations suivantes sont-elles correctes ?*

```
B b;
typedef void (B::*PMF)(void*, size_t);
PMF p1 = &B::f;
PMF p2 = &B::operator delete;
```

La première affectation est correcte : nous affectons à un pointeur vers une fonction membre l'adresse d'une fonction membre ayant la bonne signature.

La seconde affectation est incorrecte : en effet, la fonction `operator::delete` n'est pas une fonction membre statique. Signalons encore une fois que les opérateurs `new` et `delete` sont obligatoirement des fonctions membres statiques, même si elles ne sont pas explicitement déclarées `static` dans le programme. Nous rappelons qu'il est recommandé de qualifier systématiquement ces opérateurs avec le mot-clé `static`, afin de faciliter la lisibilité et la maintenance du code.

Recommandation

Déclarez toujours explicitement les opérateurs `new` (`new[]`) et `delete` (`delete[]`) comme des fonctions statiques, nonobstant le fait que, si le mot-clé `static` est omis, les fonctions seront tout de même considérées comme statiques par le compilateur.

4. *Le code suivant est-il susceptible de poser des problèmes de gestion de mémoire ?*

Réponse courte : oui. Détaillons maintenant chacun des cas proposés :

```
class X
{
public:
  void* operator new( size_t s, int ) throw( bad_alloc )
  {
    return ::operator new( s );
  }
};
```

Cette classe est susceptible de provoquer des fuites mémoires, car l'opérateur `new` a été redéfini sans que l'opérateur `delete` correspondant ne l'ait été également.

Le même problème se pose pour la classe ci-dessous :

```
class SharedMemory
{
public:
  static void* Allocate( size_t s )
  {
    return OsSpecificSharedMemAllocation( s );
  }
  static void  Deallocate( void* p, int i )
  {
    OsSpecificSharedMemDeallocation( p, i );
  }
};
class Y
{
public:
  void* operator new( size_t s, SharedMemory& m )
     throw( bad_alloc )
  {
    return m.Allocate( s );
   }
```

Cette classe peut générer des fuites mémoires, car aucun opérateur `delete` correspondant à la signature de cet opérateur `new` n'a été redéfini. Si une exception se produisait au cours de la construction d'un objet alloué par cette fonction, le mémoire ne serait pas correctement libérée car aucun opérateur `delete` ne serait appelé.

Le code suivant, par exemple, pose problème :

```
SharedMemory shared ;
..
new (shared) Y ;
// Si Y::Y() lance une exception, la mémoire ne sera pas libérée.
```

Tout objet alloué par cet opérateur `new()` ne pourra jamais être détruit proprement, car la classe `Y` n'implémente pas d'opérateur `delete` classique.

```
void  operator delete( void* p,
                       SharedMemory& m,
                       int i ) throw()
{
```

```
        m.Deallocate( p, i );
    }
};
```

Cette fonction `Y::operator delete()` est inutile car elle ne pourra jamais être appelée.

```
void operator delete( void* p ) throw()
{
  SharedMemory::Deallocate( p );
}
```

Ce code risque de provoquer des erreurs graves à l'exécution, étant donné qu'il va désallouer avec la fonction `Deallocate()` de la mémoire ayant été allouée par l'opérateur `new`.

```
void operator delete( void* p, std::nothrow_t&) throw()
{
  SharedMemory::Deallocate( p );
}
```

Même remarque ici, bien que la situation d'erreur soit un peu subtile : un problème se produira si un appel à « `new (nothrow) T` » échoue sur une exception émise par le constructeur de `T`, ce qui provoquera, là encore, la désallocation d'un espace mémoire non alloué par `SharedMemory::Allocate()`.

Recommandation

Si vous redéfinissez des opérateurs de gestion mémoire pour une classe, n'implémentez jamais l'opérateur `new` (ou `new[]`) sans implémenter l'opérateur `delete` (ou `delete[]`), et vice-versa.

Pb n° 37. auto_ptr — Difficulté : 8

Ce problème aborde le fonctionnement et l'utilisation du pointeur intelligent `auto_ptr`, de la bibliothèque standard C++.

Avant tout, commençons par un bref historique de ce sujet. Ce problème est initialement paru sous une forme plus simple que celle présentée ici, dans une édition spéciale de *Guru of the Week* parue à l'occasion de l'adoption du *Final Draft International Standard for Programming Language C++*. Cette parution tardive – survenue la veille même du début du comité de normalisation – ayant soulevé des questions importantes, tout le monde se doutait que la fonctionnalité `auto_ptr` allait subir des modifications lors du comité final d'adoption de la norme. Lorsque celui-ci eut lieu (à Morristown, New Jersey en novembre 97), il prit effectivement en compte les améliorations proposées suite à la parution du problème en question.

C'est ici l'occasion de remercier tous ceux qui ont participé à la finalisation du chapitre `auto_ptr` de la norme C++, notamment Bill Gibbons, Steve Rumbay et surtout Greg Collins, qui a travaillé longuement sur les « pointeurs intelligents » en général, dans le but de les adapter aux différentes contraintes imposées par les nombreux comités de normalisation.

Nous présentons ici une solution complète et détaillée du problème original, qui illustre notamment la raison pour laquelle des changements de dernière minute ont été apportés à la norme et indique la meilleure manière d'utiliser `auto_ptr`.

Examinez le code suivant. Indiquez quelles instructions vous semblent correctes ou incorrectes.

```
auto_ptr<T> source()
{
  return auto_ptr<T>( new T(1) );
}
void sink( auto_ptr<T> pt ) { }
void f()
{
  auto_ptr<T> a( source() );
  sink( source() );
  sink( auto_ptr<T>( new T(1) ) );
  vector< auto_ptr<T> > v;
  v.push_back( auto_ptr<T>( new T(3) ) );
  v.push_back( auto_ptr<T>( new T(4) ) );
  v.push_back( auto_ptr<T>( new T(1) ) );
  v.push_back( a );
  v.push_back( auto_ptr<T>( new T(2) ) );
  sort( v.begin(), v.end() );
  cout << a->Value();
}
class C
{
public:    /*...*/
protected: /*...*/
private:
  auto_ptr<CImpl> pimpl_;
};
```

Solution

Nombreux sont ceux qui ont entendu parler du « pointeur intelligent » `auto_ptr`, mais rares sont ceux qui l'utilisent quotidiennement. C'est à déplorer, car `auto_ptr` facilite grandement l'écriture du code et, plus encore, contribue à rendre les programmes plus robustes. Cette section expose les divers avantages d'`auto_ptr` et met également en avant certains modes d'utilisations dangereux des « pointeurs intelligents », qui peuvent parfois conduire à des bogues difficiles à diagnostiquer.

À quoi sert `auto_ptr` ?

`auto_ptr` n'est qu'un exemple de « pointeur intelligent » parmi d'autres. De nombreuses bibliothèques du marché fournissent des pointeurs intelligents plus sophistiqués, dotés de fonctionnalités variées allant du compteur de référence jusqu'à des services de type « proxy » qu'on rencontre dans les gestionnaires d'objets distribués.

`auto_ptr` n'est qu'un type très simple de pointeur intelligent, mais il procure néanmoins d'indéniables avantages dans la programmation au quotidien : `auto_ptr` est un pointeur rattaché à un objet alloué dynamiquement, qui désalloue automatiquement cet objet lorsqu'il n'est plus utilisé.

```
// Exemple 1(a): Code original
  //
  void f()
  {
    T* pt( new T );
    /*....*/
    delete pt;
  }
```

La majorité d'entre nous écrit, chaque jour, du code ressemblant à celui-ci. Si la fonction `f()` ne comporte que quelques lignes, ça ne pose généralement pas de problème. Mais si `f()` est plus complexe et que les chemins d'exécution possibles sont plus difficiles à contrôler ou qu'une exception est susceptible d'être générée au cours de l'exécution de la fonction, il y a un risque plus important que l'appel à `delete` ne soit pas effectué lorsque le programme quitte la fonction, autrement dit un risque de fuite mémoire (objet alloué mais non détruit).

Une manière simple de rendre l'exemple 1(a) plus sûr est d'encapsuler le pointeur original dans un pointeur intelligent, dont la fonction sera de détruire automatiquement l'objet alloué lors de la fin de l'exécution de la fonction. Ce pointeur intelligent étant une variable automatique (automatiquement détruit à la fin du bloc à l'intérieur duquel il est déclaré), il est naturel de le qualifier de `auto_ptr`.

```
// Exemple 1(b): Code sécurisé grâce à auto_ptr
//
void f()
{
  auto_ptr<T> pt( new T );
  /*....*/
}
 // Lorsque la fonction se termine, le destructeur de pt est
 // appelé et l'objet associé est automatiquement détruit
```

Dans l'exemple ci-dessus, nous sommes assurés de la désallocation correcte de T, quelle que soit la manière dont se termine la fonction `f()` – fin normale ou exception – car le destructeur de pt sera automatiquement appelé lors de la désallocation des variables de la pile.

Pour reprendre le contrôle d'un objet rattaché à un `auto_ptr`, il suffit d'appeler `release()` – l'utilisateur reprend alors la responsabilité de la destruction de l'objet.

```
// Exemple 2: Utilisation d'auto_ptr
//
void g()
{
  T* pt1 = new T;
   // On passe le contrôle de l'objet alloué à pt1
  auto_ptr<T> pt2( pt1 );
  // On utilise pt2 comme on le ferait avec un pointeur normal
  *pt2 = 12;       // équivalent à  "*pt1 = 12;"
  pt2->SomeFunc(); // équivalent "pt1->SomeFunc();"
  // La fonction get() fournit la valeur du pointeur
  assert( pt1 == pt2.get() );
  // La fonction release() permet de reprendre le contrôle
  T* pt3 = pt2.release();
  // Nous devons détruire l'objet manuellement,
  // car il n'est plus contrôlé par l'auto_ptr
  delete pt3;
} // pt2 ne contrôle plus aucun objet, et ne va donc pas
  // essayer de désallouer quoi que ce soit.
```

On peut également utiliser la fonction `reset()` membre d'`auto_ptr` pour rattacher un nouvel objet à un `auto_ptr` existant. Si l'`auto_ptr` est déjà rattaché à un objet, il détruira d'abord ce premier objet (autrement dit, appliquer `reset()` à un `auto_ptr` est équivalent à détruire ce pointeur et à en créer un nouveau pointant vers le nouvel objet).

```
// Exemple 3: Utilisation de reset()
//
void h()
{
  auto_ptr<T> pt( new T(1) );
  pt.reset( new T(2) );
    // Détruit le premier objet T,
    // qui avait été alloué par "new T(1)"
} // À la fin de la fonction, pt est détruit,
  // et par conséquent, le second T l'est aussi
```

Encapsulation de variables membres de type pointeur

De manière similaire, `auto_ptr` peut être utilisé pour encapsuler une variable membre de type pointeur, comme le montre l'exemple suivant, qui utilise un « Pimpl[1] » (pare-feu logiciel).

1. Le principe du « Pimpl », ou pare-feu logiciel, est couramment utilisé pour réduire le temps de recompilation des projets en faisant en sorte que le code client d'une classe ne soit pas recompilé lorsque la partie privée de cette classe est modifiée. Pour plus d'informations sur ce sujet, se reporter aux problèmes 26 à 30.

```
// Exemple 4(a): un exemple de "Pimpl"
//
// Fichier c.h
//
class C
{
public:
  C();
  ~C();
  /*...*/
private:
  class CImpl; // déclaration différée
  CImpl* pimpl_;
};
// Fichier c.cpp
//
class C::CImpl { /*...*/ };
C::C() : pimpl_( new CImpl ) { }
C::~C() { delete pimpl_; }
```

Dans cet exemple, la partie privée de `C` est implémentée dans une classe secondaire `CImpl`, dont une instance est allouée dynamiquement lors de la construction de `C` et désallouée lors de sa destruction. La classe `C` contient une variable membre de type `CImpl*` pour conserver l'adresse de cette instance.

Voici comme cet exemple peut être simplifié grâce à l'utilisation d'`auto_ptr` :

```
// Exemple 4(b): "Pimpl" utilisant auto_ptr
//
// Fichier c.h
//
class C
{
public:
  C();
  ~C();
  /*...*/
private:
  class CImpl; // déclaration différée
  auto_ptr<CImpl> pimpl_;
  C& operator = ( const C& );
  C( const C& );
};
// Fichier c.cpp
//
class C::CImpl { /*...*/ };
C::C() : pimpl_( new CImpl ) { }
C::~C() {}
```

À présent, il n'est plus nécessaire de détruire explicitement `pimpl_` dans le destructeur de `C` : l'`auto_ptr` s'en chargera automatiquement. Mieux encore, la classe `C` n'a plus à se soucier de la détection et du traitement des éventuelles exceptions pouvant se produire dans l'exécution de son constructeur `C ::C()`car, contrairement à

l'exemple précédent, le pointeur `pimpl_` est automatiquement désalloué en cas de problème de ce type. Cette solution s'avère finalement plus sécurisée et plus simple que la gestion manuelle du pointeur. D'une manière générale, la pratique consistant à déléguer la gestion d'une ressource à un objet est recommandée – c'est ce que nous faisons ici avec `auto_ptr` gérant l'objet privé `pimpl_`. Nous reviendrons sur cet exemple à la fin du chapitre.

Appartenance, sources et puits

Nous avons déjà un premier aperçu des possibilités offertes par `auto_ptr`, mais nous n'en sommes qu'au début : voyons maintenant à quel point il est pratique d'utiliser des `auto_ptr` lors d'appels de fonctions, qu'il s'agisse d'en passer en paramètre ou bien d'en recevoir en valeur de retour.

Pour cela, intéressons-nous en premier lieu à ce qui se passe lorsqu'on copie un `auto_ptr` : l'opération de copie d'un `auto_ptr` source, pointant vers un objet donné, vers un `auto_ptr` cible, a pour effet de transférer l'appartenance de l'objet pointé du pointeur source au pointeur cible. Le principe sous-jacent est qu'un objet pointé ne peut appartenir qu'à un seul `auto_ptr` à la fois. Si l'`auto_ptr` cible pointait déjà vers un objet, cet objet est désalloué au moment de la copie. Une fois la copie effectuée, l'objet initial appartiendra donc à l'`auto_ptr` cible, tandis que l'`auto_ptr` source sera réinitialisé et ne pourra plus être utilisé pour faire référence à cet objet. C'est l'`auto_ptr` cible qui, le moment venu, détruira l'objet initial :

```
// Exemple 5: Transfert d'un objet
//            d'un auto_ptr à un autre
//
void f()
{
  auto_ptr<T> pt1( new T );
  auto_ptr<T> pt2;
  pt1->DoSomething(); // OK
  pt2 = pt1;  // Maintenant, c'est pt2 qui contrôle
              // l'objet T
  pt2->DoSomething(); // OK
} // Lorsque la fonction se termine, le destructeur
  // de pt2 détruit l'objet; pt1 ne fait rien
```

Faites bien attention de ne pas utiliser un `auto_ptr` auquel plus aucun objet n'appartient :

```
// Exemple 6: Utilisation d'un auto_ptr ne contrôlant rien
//            À NE PAS FAIRE !
//
void f()
{
  auto_ptr<T> pt1( new T );
  auto_ptr<T> pt2;
  pt2 = pt1;  // Maintenant, pt2 contrôle l'objet alloué
              // pt1 ne contrôle plus aucun objet.
```

```
  pt1->DoSomething();
              // Erreur : utilisation d'un pointeur nul !
}
```

Voyons maintenant à quel point `auto_ptr` est bien adapté à l'implémentation de sources et de puits. Une « source » est une fonction qui crée un nouvel objet puis abandonne le contrôle de cet objet à l'appelant. Un « puits » est une fonction qui, à l'inverse, prend le contrôle d'un objet existant (et généralement, détruit cet objet après l'avoir utilisé).

Dans l'implémentation de sources et de puits, plutôt que d'utiliser de simples pointeurs pointant vers l'objet manipulé, il peut s'avérer très pratique de renvoyer (resp. prendre en paramètre) un pointeur intelligent rattaché à cet objet.

Ceci nous amène finalement aux premières lignes de code de l'énoncé de notre problème :

```
auto_ptr<T> source()
{
  return auto_ptr<T>( new T(1) );
}
void sink( auto_ptr<T> pt ) { }
```

Non seulement ces lignes sont tout à fait correctes, mais, de plus, elles sont très élégantes :

1. La manière avec laquelle la fonction `source()` crée un nouvel objet et le renvoie à l'appelant est parfaitement sécurisée : l'`auto_ptr` assure que l'objet alloué sera détruit en temps voulu. Même si, par erreur, l'appelant ignore la valeur de retour de la fonction, l'objet sera également correctement détruit. Voir également le problème n° 19, qui démontre que la technique consistant à encapsuler la valeur retournée par une fonction dans un `auto_ptr` – ou un objet équivalent – est le seul moyen de vraiment s'assurer que cette fonction se comportera correctement en présence d'exceptions.

2. Lorsqu'elle est appelée, la fonction `sink()` prend le contrôle du pointeur qui lui est passé en paramètre car celui-ci est encapsulé dans un `auto_ptr`, ce qui assure la destruction de l'objet associé lorsque la fonction se termine (à moins que `sink()` n'ait, au cours de son exécution, transmis le contrôle du pointeur à une autre fonction). On peut noter que, dans notre exemple, la fonction `sink()` est équivalente à `pt.reset(0)`, dans la mesure où elle n'effectue aucune opération.

   La suite du code montre un exemple d'utilisation de `source()` et `sink()` :

```
void f()
{
  auto_ptr<T> a( source() );
```

Instruction tout à fait correcte et sans danger : la fonction `f()` prend le contrôle de l'objet alloué par `source()`, l'`auto_ptr` assurant qu'il sera automatiquement détruit lors de la fin de l'exécution de `f()`, qu'il s'agisse d'une fin normale ou d'une fin pré-

maturée suite à une exception. C'est un exemple tout à fait classique de l'utilisation d'`auto_ptr` comme valeur de retour d'une fonction.

```
sink( source() );
```

Là encore, instruction correcte et sans danger. Étant donné l'implémentation minimale de `source()` et `sink()`, ce code est équivalent à « `delete new T(1)` ». S'il est vrai que l'avantage procuré par `auto_ptr` n'est pas flagrant dans notre exemple, soyez convaincus que lorsque `source()`et `sink()` sont des fonctions complexes, l'utilisation d'un pointeur intelligent présente alors un intérêt certain.

```
sink( auto_ptr<T>( new T(1) ) );
```

Toujours correct et sans danger. Une nouvelle manière d'écrire « `delete new T(1)` » mais, encore une fois, une technique intéressante lorsque `sink()` est une fonction complexe qui prend le contrôle de l'objet alloué.

Nous avons présenté ici les emplois classiques d'`auto_ptr`. Attention ! Méfiez-vous des utilisations qui différent de celles exposées plus haut : `auto_ptr` n'est pas un objet comme les autres ! Certains emplois non contrôlés peuvent causer des problèmes, en particulier :

La copie d'un `auto_ptr` ne réalise pas de copie de l'objet pointé.

Ceci a pour conséquence qu'il est dangereux d'utiliser des `auto_ptr` dans un programme effectuant des copies en supposant implicitement – ce qui est généralement le cas – que les opérations de copie effectuent un clonage de l'objet copié.

Considérez par exemple le code suivant, régulièrement rencontré dans les groupes de discussions C++ sur l'Internet :

```
vector< auto_ptr<T> > v;
```

Cette instruction est incorrecte et présente des dangers ! D'une manière générale, il est toujours dangereux d'utiliser des `auto_ptrs` avec des conteneurs standards. Certains vous diront qu'avec leur compilateur et leur version de la bibliothèque standard, ce code fonctionne correctement, d'autres soutiendront que l'instruction ci-dessous est précisément citée à titre d'exemple dans la documentation d'un compilateur populaire : tous ont tort ! Le fait qu'une opération de copie entre `auto_ptrs` transmette le contrôle de l'objet pointé à l'`auto_ptr` cible au lieu de réaliser une copie de cet objet rend très dangereux l'utilisation des `auto_ptr` avec les conteneurs standards ! Il suffit, par exemple, qu'une implantation de `vector()` effectue une copie interne d'un des `auto_ptr` stocké pour que l'utilisateur voulant obtenir ultérieurement cet `auto_ptr` récupère un objet contenant un pointeur nul.

Ce n'est pas tout, il y a pire :

```
v.push_back( auto_ptr<T>( new T(3) ) );
v.push_back( auto_ptr<T>( new T(4) ) );
v.push_back( auto_ptr<T>( new T(1) ) );
v.push_back( a );
```

(Notez ici que le fait de la passer par valeur a pour effet d'ôter à la variable a le contrôle de l'objet vers lequel elle pointe. Nous reviendrons sur ce point ultérieurement.)

```
v.push_back( auto_ptr<T>( new T(2) ) );
sort( v.begin(), v.end() );
```

Ce code est incorrect et dangereux. La fonction `sort()`ainsi d'ailleurs que l'ensemble des fonctions de la bibliothèque standard qui effectuent des copies fait l'hypothèse que ces copies laissent inchangé l'objet source de la copie. S'il y a plusieurs implémentations possible de `sort()`, l'une des plus courantes a recours, en interne, à un élément « pivot » destiné à contenir des copies des certains des objets à trier. Voyons ce qui se passe lorsqu'on trie des `auto_ptrs` : lorsqu'un élement est copié dans l'élément pivot, le contrôle de l'objet pointé par cet `auto_ptr` est transmis à l'`auto_ptr` pivot, qui est malheureusement temporaire. Lorsque le tri sera terminé, le pivot sera détruit et c'est là que se posera le problème : un des objets de la séquence originale sera détruit !

C'est pour éviter ce genre de problèmes que le comité de normalisation C++ a décidé de procéder à des modifications de dernière minute : l'`auto_ptr` a été spécifiquement conçu pour provoquer une erreur de compilation lorsqu'il est utilisé avec un conteneur standard – règle qui a été mise en place, en pratique, dans la grande majorité des implémentations de la bibliothèque standard. Pour cela, le comité a spécifié que les fonctions `insert()` de chaque conteneur standard devaient prendre en paramètre des références constantes, interdisant par là même leur emploi avec les `auto_ptr`, pour lesquels le constructeur de copie et l'opérateur d'affectation prennent en paramètres des références non constantes.

Cas des `auto_ptr` ne contrôlant aucun objet

```
  // (après la copie de a vers un autre auto_ptr)
  cout << a->Value();
}
```

Même si, avec un peu de chance l'objet pointé peut, en dépit de la copie de `a`, ne pas avoir été détruit par l'objet `vector` ou la fonction `sort()`, ce code posera de toute manière un problème, car la copie d'un `auto_ptr` provoque non seulement le transfert de l'objet pointé vers l'`auto_ptr` cible, mais encore positionne à NULL la valeur de l'`auto_ptr` source. Ceci est fait spécifiquement dans le but d'éviter l'utilisation d'un `auto_ptr` ne contrôlant aucun objet. Le code ci-dessus provoquera donc immanquablement une erreur à l'exécution.

Voyons pour finir le dernier des usages courants d'`auto_ptr`

Encapsulation de pointeurs membres

Les `auto_ptrs` peuvent être utilisés pour encapsuler des variables membres de type pointeur :

```
class C
{
public:    /*...*/
protected: /*...*/
```

```
private:
  auto_ptr<CImpl> pimpl_;
};
```

Ainsi, de manière analogue à leur utilisation au sein de fonctions, ils évitent au développeur d'avoir à se soucier de la destruction de l'objet alloué, laquelle a ici lieu automatiquement lors de la destruction de `C`.

Il subsiste une contrainte – analogue à celle qu'on rencontrerait dans le cas de l'utilisation de pointeurs simples : il est obligatoire de redéfinir vos propres constructeur de copie et opérateur d'affectation pour la classe `C` (ou, au pire, d'interdire la copie et l'affectation en les déclarant `private` sans les implémenter), car les fonctions fournies par défaut par le compilateur poseraient problème.

`auto_ptr` et exceptions

Dans certains cas, `auto_ptr` peut être d'une grande utilité pour assurer qu'un programme se comporte correctement en présence d'exceptions. Pour preuve, considérez la fonction suivante :

```
// Que se passe t'il si une exception se produit ?
//
String f()
{
  String resultat ;
  resultat = "une valeur";
  cout << "hello !"
  return resultat ;
}
```

Cette fonction a deux actions extérieures : elle émet un message vers le flux de sortie (`cout`) et retourne une chaîne vers le code appelant. Sans entrer dans une discussion complète sur la robustesse aux exceptions[1], il faut savoir qu'une des caractéristiques requises pour le bon comportement d'une fonction en présence d'exceptions est son *atomicité* : la fonction doit s'exécuter complètement ou bien, si elle échoue, laisser l'état du programme inchangé (en l'occurrence, n'avoir aucune effet extérieur).

De ce point de vue là, la fonction `f()` présente un défaut, comme le montre l'exemple suivant :

```
String unNom ;
unNom = f() ;
```

La fonction `f()` renvoyant une valeur, le constructeur de copie de `String` est appelé pour copier la valeur retournée par la fonction dans l'objet `unNom`. Si ce constructeur échoue sur une exception, la fonction `f()` ne se sera exécutée que partiellement : un message aura été affiché, mais la valeur n'aura pas été correctement renvoyée à l'appelant.

1. À ce sujet, voir les problèmes n° 8 à 19.

Pour éviter ce problème, on peut utiliser une référence non constante passée en paramètre plutôt que retourner une valeur :

```
// Est-ce mieux ?
//
void f(String& result)
{
  cout << "hello !"
  resultat = "une valeur"
}
```

Cette solution peut paraître meilleure, du fait qu'elle évite l'utilisation d'une valeur de retour et élimine, par là même, les risques liés à la copie. En réalité, elle pose exactement le même problème que la précédente, déplacé à un autre endroit : c'est maintenant l'opération d'affectation qui risque d'échouer sur une exception, ce qui aura également pour conséquence une exécution partielle de la fonction.

Pour finalement résoudre le problème, il faut avoir recours à un pointeur pointant vers un objet `String` alloué dynamiquement ou, mieux encore, à un pointeur automatique (`auto_ptr`) :

```
// La bonne méthode
//
auto_ptr<String> f()
{
  auto_ptr<String> resultat = new String ;
  * resultat = "une valeur";

  cout << "hello !"
  return resultat ; // Ne peut pas générer d'exception
}
```

Cette dernière solution est la bonne. Il n'y a plus aucun risque de génération d'exception au moment de la transmission d'une valeur de retour ou au moment d'une affectation. Cette fonction satisfait tout à fait à la technique du « valider ou annuler » : si elle est interrompue par une exception, elle annule totalement les opérations en cours (aucun message affiché à l'écran, aucune valeur reçue par l'appelant). L'utilisation d'un `auto_ptr` comme valeur de retour permet de gérer correctement la transmission de la chaîne de caractère à l'appelant : si l'appelant récupère correctement la valeur de retour, il prendra le contrôle de la chaîne allouée et aura pour responsabilité de la désallouer ; si, au contraire, l'appelant ne récupère pas correctement la valeur de retour, la chaîne orpheline sera automatiquement détruite au moment de la destruction de la variable automatique `result`. Nous n'avons obtenu cette robustesse aux exceptions qu'au prix d'une petite perte de performance due à l'allocation dynamique. Les bénéfices retirés au niveau de la qualité du code en valent largement la peine.

Prenez l'habitude d'utiliser fréquemment les `auto_ptrs`. Ils permettent d'écrire du code plus simple et plus robuste aux exceptions ; de plus, ils sont standards et, par conséquent, portables.

Technique de l'`auto_ptr` constant

Pour finir ce chapitre, voyons une technique utile : l'utilisation d'un `auto_ptr` constant. Un `auto_ptr` constant ne perd jamais le contrôle de l'objet vers lequel il pointe. Les seules opérations permises sur un « `const auto_ptr` » sont l'application des opérateurs `*` et `->`, qui permettent de faire référence à l'objet pointé ou l'appel à la fonction membre `get()`, qui renvoie la valeur du pointeur stocké.

```
const auto_ptr<T> pt1( new T );
    // Le fait de déclarer pt1 "const" nous assure
    // qu'il ne peut pas être copié vers un autre_auto_ptr.
    // Il conserve ainsi en permanence le contrôle de "new T"
auto_ptr<T> pt2( pt1 ); // Interdit
auto_ptr<T> pt3;
pt3 = pt1;              // Interdit
pt1.release();          // Interdit
pt1.reset( new T );     // Interdit
```

L'utilisation d'un `const auto_ptr` est une technique couramment répandue, ainsi que le mentionne l'article de la solution originale de ce problème, paru dans *Guru of the Week*. Nous conclurons de même ce chapitre : « La technique de l'`auto_ptr` constant fait partie de ces choses devenues tellement courantes qu'on a l'impression de les avoir toujours connues. »

Quelques pièges à éviter

Pb n° 38. Auto-affectation — Difficulté : 5

Dans ce problème, nous allons étudier le problème de l'auto affectation et, plus généralement, voir comment déterminer si deux pointeurs pointent vers un même et unique objet.

Lorsqu'on redéfinit l'opérateur d'affectation pour une classe, il est courant qu'on implémente un test préliminaire pour éviter une « auto-affectation » :

```
T& T::operator=( const T& other )
{
  if( this != &other )    // Le test en question
  {
    // ...
  }
  return *this;
}
```

L'instruction « `this!&other` » utilisée ici est-elle nécessaire et/ou suffisante pour réaliser correctement le test ? Vous argumenterez votre réponse et proposerez, le cas échéant, une meilleure solution.

Solution

Réponse courte : un opérateur d'affectation correctement implémenté devrait bien se comporter dans toutes les situations, même en cas d'auto-affectation; le test proposé n'est donc pas indispensable.

Réponse longue : dans certains cas, il peut être justifié d'utiliser ce type de test, en prenant néanmoins garde au fait qu'il peut poser problème dans une situation bien précise (redéfinition de la fonction `operator&`)[1]

Éviter l'auto-affectation ?

Le test évitant l'auto-affectation ne devrait pas être *nécessaire* : autrement dit, l'implémentation de l'opérateur d'affectation devrait fonctionner correctement pour tout type d'affectation, y compris dans le cas particulier de l'auto-affectation :

Recommandation

Ne traitez jamais l'auto-affectation comme un cas particulier : l'implémentation générale d'un opérateur d'affectation doit être capable de gérer correctement l'auto-affectation.

Il peut être justifié, dans certains cas, d'utiliser un test de ce genre pour éviter une auto-affectation, inutile et pénalisante en terme en performances. Néanmoins, il faut en réserver l'emploi aux – rares – situations où le gain de performances est vraiment notable (opération d'affectation particulièrement coûteuse et/ou nombre d'auto-affectations particulièrement élevé).

Recommandation

Vous pouvez utiliser un test pour éviter l'auto-affectation dans les cas où cela procure un gain de performance.

Redéfinition de l'opérateur &

Il est tout à fait possible qu'une classe redéfinisse l'opérateur `&` (ou qu'il soit redéfini dans une des classes de base) en lui donnant un comportement tout à fait différent du comportement normal « obtenir l'adresse de... ». Auquel cas, le code « `this!=&other` » risque de donner un résultat différent de celui escompté. Soit dit en passant, l'auteur d'une classe ayant implémenté un test pour éviter l'auto-affectation serait un peu machiavélique de redéfinir la fonction `operator&` !

Au passage, on peut noter que si la classe redéfinit `T::operator!()`, ceci n'a pas d'influence sur le test « `this!=&other` » car cette fonction ne sera pas appelée (en effet, la fonction `T::operator!()`doit obligatoirement prendre au moins un paramètre de type `T`, et ne peut donc pas prendre deux paramètres de type `T*`).

1. Contrairement à ce que l'on peut parfois lire, il n'y a aucun problème lié à l'utilisation de l'héritage multiple.

Construction et auto-affectation

Dans le même ordre d'idées, voici un exemple de code parfois rencontré :

```
T::T( const T& other )
{
  if( this != &other )
  {
    // ...
  }
}
```

J'espère que vous avez trouvé l'erreur du premier coup[1].

Quelques compléments au sujet des pointeurs

Voici pour finir deux exemples de résultats inattendus pouvant se produire lors de comparaisons de pointeurs :

- Comparer deux pointeurs pointant vers des chaînes de caractère littérales peut donner des résultats inattendus : en particulier, si vous comparez deux pointeurs vers deux variables chaînes littérales distinctes ayant la même valeur, il est possible que ces deux pointeurs contiennent la même adresse. En effet, à des fins d'optimisation, la norme C++ autorise explicitement les compilateurs à ne pas stocker systématiquement chaque nouvelle chaîne littérale dans un espace mémoire séparé.
- La comparaison de valeurs de pointeurs à l'aide des opérateurs `<`, `<=`, `>` et `>=` produit en général un résultat indéterminé (sauf dans certains cas particuliers comme la comparaison de pointeurs vers des objets stockés dans un même tableau). Cette limitation peut être contournée par l'emploi du modèle de fonction `less<>` et de ses cousins, qui permettent d'établir une relation d'ordre entre des pointeurs. Ces fonctions sont utilisées, entre autres, lorsque l'on crée une table de correspondance utilisant un clé de type pointeur – par exemple `map<T*,U>` (qui est en fait, après résolution du paramètre par défaut : `map<T*,U,less<T*>>`).

1. Vérifier si on n'est pas en train d'effectuer une auto-affectation n'a aucun sens dans le cas d'un constructeur : l'objet « `other` » ne peut pas être égal à l'objet qu'on est en train de construire, pour la bonne raison que ce dernier n'existe pas encore ! Un de mes amis, Jim Hyslop, m'a fait remarquer que l'exemple suivant (techniquement illégal) utilisant un opérateur `new` avec « placement » (adresse d'allocation forcée) donnerait un sens, s'il était légal, à ce test dans le constructeur :
   ```
   T t ;
   new(&t) T(t) ; // Donnerait un sens au test si c'est légal
   ```
   Autre exemple dans lequel ce test serait utile : « `T t = t ;` », instruction acceptée par le compilateur mais qui provoquera immanquablement une erreur à l'exécution.

Pb n° 39. Conversions automatiques — Difficulté : 4

Les conversions automatiques d'un type à un autre sont extrêmement pratiques. Elles peuvent également être extrêmement dangereuses, comme nous allons le voir dans ce problème.

Comme vous le savez, la classe `string` de la bibliothèque standard C++ n'implémente pas de conversion automatique vers le type `const char*` mais fournit, à la place, une fonction membre `c_str()` renvoyant un `const char*` :

```
string s1("hello"), s2("world");
strcmp( s1, s2 );                // 1 (Erreur)
strcmp( s1.c_str(), s2.c_str() ) // 2 (OK)
```

Dans cet exemple, le premier appel à `strcmp` provoque une erreur de compilation car il n'existe pas de conversion de `string` vers `const char*`. Le second appel se compile correctement, mais est plus lourd à écrire; on est dès lors tenté de déplorer que la syntaxe du premier appel soit interdite...

À votre avis, pour quelle(s) raison(s) la classe `string` n'implémente-t-elle pas de conversion vers `const char` ?

Solution

Il est toujours dangereux d'implémenter des conversions implicites – qu'il s'agisse d'opérateurs de conversion ou de constructeurs à un argument (sauf s'ils sont déclarés `explicit`)[1]. En effet :

- Les conversions implicites peuvent être à l'origine de surprises lors de la résolution de noms.
- Les conversions implicites peuvent rendre possible la compilation d'un code pourtant « incorrect ».

Si la classe `string` implémentait une conversion automatique vers `const char*`, cette conversion risquerait d'être appelée de manière implicite par le compilateur sans que l'utilisateur ne s'en rende systématiquement compte, pouvant ainsi causer ainsi toutes sortes de problèmes parfois difficiles à repérer, car ne provoquant en général pas d'erreurs de compilation. De nombreux exemples pourraient être cités, en voici un :

```
string s1, s2, s3;
s1 = s2 - s3;   // Faute de frappe... '-' au lieu de '+'
```

1. Nous n'abordons ici que les problèmes courants liés aux conversions implicites. Il y a d'autres raisons qui justifient l'absence de conversion de `string` vers `const char*`. À ce sujet, voir Koenig A. and Moo B. Ruminations on C++ (Addison Wesley Longman, 1997), pages 290 à 292 ; Stroustrup Bjarne *The Design and Evolution of C++* (Addiso Wesley, 1994), page 83.

Si la classe `string` implémentait une conversion automatique vers `const char`, le code ci-dessus se compilerait correctement mais, à l'exécution, `s2` et `s3` seraient convertis vers des `const char*` et la soustraction des deux pointeurs résultants serait affectée à `s1` !

Recommandation

Évitez d'implémenter des opérateurs de conversion. Déclarez les constructeurs '`explicit`'

Pb n° 40. Durée de vie des objets (1re partie) Difficulté : 5

Allocation, construction, destruction, désallocation... À quel moment un objet est-il vraiment utilisable ?

Examinez le code suivant :

```
void f()
{
  T t(1);
  T& rt = t;
  //--- #1: on manipule t et rt ---
  t.~T();
  new (&t) T(2);
  //--- #2 : on manipule t et rt ---
} // t est détruit automatiquement
```

Le bloc de code `#2` va-t-il s'exécuter correctement ?

Solution

Le code proposé est légal et conforme à la norme C++ ; néanmoins, la fonction dans son ensemble n'est pas saine – elle peut poser des problèmes en présence d'exceptions – et utilise un style de programmation qu'il vaut mieux éviter.

Nous opérons ici une destruction explicite suivi d'une allocation « placée » (allocation à une adresse mémoire déterminée) : la norme C++ spécifie clairement qu'une référence (en l'occurrence `rt`) ne doit pas être altérée par une opération de ce type (bien entendu, on fait ici l'hypothèse que l'opérateur `&` n'a pas été redéfini pour la classe `T` et renvoie bien l'adresse de l'objet).

En revanche, la fonction `f()` peut se comporter de manière incorrecte en présence d'exceptions : en effet, si une exception se produit lors de la « reconstruction » de `t` (« `new (&t) T(2)` »), alors la fonction va se terminer et un appel au destructeur `T::~T` se produira, ce qui posera problème car la zone mémoire de `t` ne contiendra alors aucun objet construit. Autrement dit, en présence d'exception, `t` risque d'être construit

une fois mais détruit deux fois, ce qui provoquera à coup sûr une erreur à l'exécution.

Recommandation

Assurez-vous que votre code se comporte correctement en présence d'exceptions. En particulier, organisez votre code de manière à désallouer correctement les objets et à laisser les données dans un état cohérent, même en présence d'exceptions.

Indépendamment de ces questions relatives aux exceptions, il n'est pas recommandé de prendre l'habitude d'avoir recours à cette technique de destruction explicite / allocation placée. Si elle n'est pas dangereuse utilisée depuis du code client de `T`, elle peut en revanche poser problème depuis certaines fonctions membres :

```
// Avez-vous vu le danger ?
//
void T::DestroyAndReconstruct( int i )
{
  this->~T();
  new (this) T(i);
}
```

Ce code est dangereux ! Pour preuve, examinez le code suivant :

```
class U : public T { /* ... */ };
void f()
{
  /*AAA*/ t(1);
  /*BBB*/& rt = t;
  //--- #1: on manipule t et rt ---
  t.DestroyAndReconstruct(2);
  //--- #2: on manipule t et rt ---
} // t est détruit automatiquement
```

Si « `/*AAA*/` » vaut « `T` », le code du bloc #2 s'exécutera correctement, que « `/*BBB*/` » soit « `T` » ou une classe de base de « `T` ».

En revanche, si « `/*AAA*/` » vaut « `U` », il se produira à coup sûr une erreur à l'exécution, et ce, quelle que soit la valeur de « `/*BBB*/` » : en effet, l'appel à `DestroyAndReconstruct()`remplacera l'objet `t` de type `U` par un objet plus petit, de type `T`. La bonne exécution de la suite dépend de la partie de `U` utilisée par le code du bloc #2 : si ce code appelle des fonctions implémentées dans `U` et pas dans `T`, une erreur se produira.

En conclusion, même si elle fonctionne dans certains cas, cette technique dangereuse n'est pas recommandable.

Recommandation

Ne poussez pas le langage dans ses derniers retranchements. Les techniques les plus simples sont souvent les meilleures.

Pb n° 41. Durée de vie des objets (2e partie) Difficulté : 6

Ce problème étudie plus en détail la technique de destruction explicite / réallocation placée déjà entrevue au cours du problème précédent. Parfois utile, cette pratique présente néanmoins des dangers certains.

Examinez le code suivant :

```
T& T::operator=( const T& other )
{
  if( this != &other )
  {
    this->~T();
    new (this) T(other);
  }
  return *this;
}
```

1. Quel est l'intérêt d'implémenter l'opérateur d'affectation de cette manière ? Ce code présente-t-il des défauts ?

2. En faisant abstraction des défauts de codage, peut-on considérer que la technique utilisée ici est sans danger ? Sinon, quelle(s) autre(s) technique(s) pourrai(en)t permettre d'obtenir un résultat équivalent ?

Note : Voir aussi le problème n° 40.

Solution

La technique de destruction explicite / réallocation placée utilisée dans cet exemple est parfaitement légale et conforme à la norme C++ (où elle est même citée en exemple, voir plus loin la discussion sur ce sujet). Néanmoins, elle peut être à l'origine d'un grand nombre de problèmes, comme nous allons le montrer ici[1]. D'une manière générale, il est préférable de ne pas utiliser cette technique de programmation, d'autant plus qu'il existe un moyen plus sûr d'aboutir au même résultat.

L'intérêt majeur de cette technique est le fait que l'opérateur d'affectation est implémenté en fonction du constructeur de copie, assurant ainsi une identité de comportement entre ces deux fonctions. Ceci permet de ne pas à avoir répéter inutilement deux fois le même code et élimine les risques de désynchronisation lors des évolutions de la classe – par exemple, plus de risque d'oublier de mettre à jour une des deux fonctions lorsqu'on ajoute une variable membre à la classe.

1. Nous ne traiterons pas ici le cas trivial de la redéfinition de l'opérateur & (il est évident que cette technique ne fonctionne pas si l'opérateur & renvoie autre chose que « `this` » ; voir le problème n° 38 à ce sujet).

Cette technique s'avère également utile – voire indispensable – dans un cas particulier : si la classe `T` a une classe de base virtuelle comportant des variables membres, l'opération de destruction / réallocation assure que ces variables membres seront correctement copiées[1]. Néanmoins, ce n'est qu'un maigre argument car, d'une part une classe de base virtuelle ne devrait *pas* contenir de variables membres (voir, à ce sujet, Meyers98, Meyers99 et Barton94) et d'autre part le fait que `T` comporte une classe de base virtuelle indique très probablement qu'elle est elle-même destinée à servir de classe de base, ce qui va poser problème (comme nous allons le voir dans la section suivante).

Si l'utilisation de cette technique présente quelques rares avantages, elle induit malheureusement beaucoup de problèmes, que nous allons détailler maintenant.

Problème n° 1 : Troncature d'objets

Le code « `this->~T(); new(this) T(other);` » présente un défaut : il pose problème lorsque `T` est une classe de base dotée d'un destructeur virtuel et que la fonction `T::operator()=` est appelée depuis un objet dérivé. Dans ce cas de figure, l'opérateur d'affectation va détruire l'objet dérivé et le remplacer par un objet `T`, plus petit. Cette « troncature » de l'objet dérivé provoquera immanquablement des erreurs d'exécution dans la suite du code (voir également le problème n° 40 à ce sujet).

Il est d'usage, pour l'auteur d'une classe dérivée, d'implémenter l'opérateur d'affectation de sa classe en fonction de l'opérateur de la classe de base :

```
Derived&
Derived::operator=( const Derived& other )
{
  Base::operator=( other );
  // ...On effectue ici la copie des membres de Derived
  return *this;
}
```

Si la classe `T` de l'exemple était utilisée comme classe de base, voici ce que cela donnerait :

```
class U : T { /* ... */ };
U& U::operator=( const U& other )
{
  T::operator=( other );
                // ...On effectue ici la copie des membres de U
 // ... PROBLEME : 'this' n'est plus un U mais un T !
  return *this;  // Ne pointe pas vers un U !
}
```

1. Alors que sans cette technique, les variables en question seront, au mieux, copiées plusieurs fois et, au pire, copiées de manière incorrecte.

Non seulement la fonction `T::operator()=` perturbe la suite du code, mais encore elle le fait de manière silencieuse : il est en général très difficile de déboguer ce genre d'erreur si on n'a pas accès au code source de `T` (à moins que le destructeur de `U` n'affecte aux variables membres des valeurs permettant de détecter clairement que l'objet a été détruit, mais c'est une bonne habitude malheureusement trop peu répandue).

Il y a deux solutions pour corriger ce défaut du code :

- Remplacer l'appel explicite à `T::operator()=` par un appel à « `this->~T()` » suivi d'une réallocation de l'objet de base `T` à l'adresse `this`. Ceci permettrait de s'assurer que, lors d'un appel à cet opérateur depuis un objet dérivé `U`, seul l'objet de base `T` est détruit et reconstruit, évitant ainsi la troncature de l'objet `U`.
- Appliquer dans la fonction `U::operator()=` la même technique que celle employée dans `T::operator()=`. Cette solution est meilleure que la première mais met bien en évidence l'une des faiblesses de la technique destruction explicite / réallocation : si elle est utilisée dans une classe de base, elle doit être utilisée dans toutes les classes dérivées (ce qui est très difficile à mettre en oeuvre en pratique car il faut imposer aux auteurs des classes dérivées d'implémenter un opérateur d'affectation spécifique).

Si elles permettent de limiter les dégâts, ces deux solutions présentent néanmoins quelques pièges, sur lesquels nous allons avoir l'occasion de revenir bientôt.

Recommandations

Ne poussez pas le langage dans ses derniers retranchements. Les techniques les plus simples sont souvent les meilleures.
Évitez le code inutilement compliqué ou obscur, même s'il vous paraît simple et clair au moment où vous l'écrivez.

Problème n° 2 : Mauvais comportement en présence d'exceptions

Quand bien même le code de l'exemple aurait été corrigé, grâce à l'une des deux techniques présentées ci-dessus, il subsisterait de toute manière un certain nombre de problèmes impossibles à éliminer.

Le premier d'entre eux concerne le comportement du code en présence d'exceptions : si le constructeur de copie de `T` lance une exception lors de l'appel à « `new (this) T(other);` » (c'est le mode naturel qu'utilise un constructeur pour signaler des erreurs), alors la fonction `T::operator=()` se terminera prématurément en laissant l'objet `T` dans un état incohérent (l'ancien objet aura été détruit, mais n'aura pas été remplacé).

Il risque alors de se produire une erreur d'exécution dans la suite du code : tentative de manipulation d'un objet qui n'existe plus ou tentative de double destruction (au sujet de ce dernier point, voir le problème n° 40).

Recommandation

Assurez-vous que votre code se comporte correctement en présence d'exceptions. En particulier, organisez votre code de manière à désallouer correctement les objets et à laisser les données dans un état cohérent, même en présence d'exceptions.

Problème n° 3 : Altération de la notion de durée de vie

Un autre inconvénient majeur de la technique de destruction explicite / réallocation est qu'elle est incohérente avec la notion classique de durée de vie d'un objet. En particulier, elle risque de perturber le comportement de toute classe réalisant l' « acquisition » d'une ressource externe lors de sa construction et sa « libération » lors de sa destruction.

Prenons l'exemple d'une classe se connectant à une base de données lors de sa construction et s'en déconnectant lors de sa destruction : réaliser une opération d'affectation sur une instance de cette classe provoquera une déconnexion / reconnexion intempestive, risquant de laisser l'objet et ses clients dans un état incohérent – (ils utiliseront une connexion différente de celle qu'ils croiront utiliser). On aurait pu également prendre l'exemple d'une classe verrouillant une section critique lors de sa construction et la libérant lors de sa destruction.

Vous pourriez arguer que l'emploi d'opérations de ce genre est limité à un certain type de programmes et que vous ne l'utilisez pas dans vos classes. C'est possible, mais qui vous assure qu'une de vos classes de base ne le fait pas ? D'une manière générale, il ne faut pas utiliser de techniques de programmation imposant des contraintes sur les classes de bases parce qu'il n'est pas possible de *maîtriser* ce que contient une classe de base lorsqu'on implémente une classe dérivée.

Le problème de fond est cette technique est contradictoire avec le sens même de *construction* et *destruction* en C++. La construction doit correspondre au début de la vie de l'objet et la destruction à la fin de la vie, alors qu'ici on détruit un objet et on le remplace par un autre, tout en voulant faire croire au code extérieur que l'objet initial n'a pas cessé de « vivre » mais qu'il a simplement changé de valeur : ceci pose des problèmes lorsque le constructeur et le destructeur effectuent des opérations spécifiques comme l'acquisition et la libération de ressources externes.

Problème n° 4 : Perturbation des classes dérivées

La solution vue plus haut consistant à appeler explicitement le destructeur de T (`this ->T ::~T()`) pour éviter la troncature d'un objet dérivé présente l'inconvénient

d'aller à l'encontre du principe classique selon lequel les objets de base sont *construits avant* et *détruits après* l'objet dérivé. Dans de nombreuses situations, cela peut poser problème pour un objet dérivé de voir un de ses objets de base détruit puis remplacé sans qu'il le sache (bien que les conséquences soient limitées lorsque l'opération d'affectation n'effectue que des affectations « classiques » de membres... mais dans ce cas-là, quel est l'intérêt de le redéfinir ?).

Problème n° 5 : `this != &other`

L'exemple de code est totalement dépendant du test préliminaire « `this != &other` » : pour vous en convaincre, imaginez ce qui se passerait, sans ce test, dans le cas d'une auto-affectation. Par conséquent, nous avons le même problème que celui qui a été exposé dans le problème n° 38 : notre opérateur traite l'auto-affectation comme un cas *particulier*, alors qu'un opérateur d'affectation correctement implémenté devrait, sous sa forme *générale*, bien se comporter dans toutes les situations, même en cas d'auto-affectation[1].

À ce point du discours, nous avons découvert suffisamment de problèmes liés à l'utilisation de cette technique de destruction explicite / réallocation pour conclure sans scrupule que c'est une technique à éviter[2].

Nous allons à présent voir qu'il existe une autre solution pour mutualiser le code du constructeur de copie et de l'opérateur d'affectation sans subir les inconvénients de la technique précédente.

Cette solution est fondée sur l'utilisation d'une fonction membre `Swap()`, implémentée de manière à ne *jamais* lancer d'exception, réalisant l'*échange* de la valeur de l'objet sur lequel elle est appliquée avec celle de l'objet passé en paramètre :

```
T& T::operator=( const T& other )

{
  T temp( other );  // Prépare le travail
  Swap( temp );     // "Valide" le travail (sans risquer
  return *this;     // de lancer une exception)
}
```

Cette méthode implémente l'opérateur d'affectation en fonction du constructeur de copie; elle ne tronque pas les objets, se comporte correctement en présence d'exceptions, n'altère pas la notion de classe dérivée et traite sans problème le cas de l'auto-affectation. Nul doute qu'elle doit, sans conteste, être préférée à la méthode précédente !

1. Il est tout à fait possible d'utiliser un test de ce genre à des fins d'optimisation. En revanche, votre opérateur doit être implémenté de manière à traiter correctement l'auto-affectation, même en l'absence de test. Reportez vous au problème n° 38 pour plus d'informations.
2. Et encore, tous les problèmes n'ont pas été traités ici (notamment le comportement incohérent en présence d'opérateur d'affectation virtuel).

Pour plus d'informations à propos cette méthode en particulier et sur la gestion correcte des exceptions en général, voir également les problèmes 8 à 17[1].

Recommandations

Implémentez l'opérateur d'affectation en fonction du constructeur de copie en utilisant une fonction `Swap()` ne lançant pas d'exception :

```
// Bon
T& T::operator=( const T& other )
{
  T temp( other );
  Swap( temp );
  return *this;
}
```

N'utilisez jamais la technique consistant à implémenter l'opérateur d'affectation en fonction du constructeur de copie en ayant recours à une destruction explicite suivi d'une allocation placée – même si cette technique est parfois recommandée par certains.

Autrement dit, n'écrivez PAS :

```
// MAUVAIS
T& T::operator=( const T& other )
{
  if( this != &other)
  {
    this->~T();              // Dangereux !
    new (this) T( other );   // Dangereux !
  }
  return *this;
}
```

Du bon usage des exemples

La technique tant dénigrée dans les sections précédentes est citée comme exemple dans la norme C++ !

Nous reproduisons ici l'exemple en question (extrait de la section 3.8 §7, légèrement simplifié à des fins de clarté) afin de montrer qu'il est destiné qu'à illustrer quelques règles relatives à la durée de vie des objets et n'encourage en aucun cas l'usage systématique de la destruction / réallocation :

1. Cette méthode, il est vrai, ne fonctionne pas pour une classe ayant des membres de type référence. Cela ne remet pas en cause l'efficacité de la méthode car des objets contenant des membres de type référence devraient normalement ne pas pouvoir être copiés (si on désire pouvoir le faire, il faut utiliser des membres de type pointeur).

```
// Exemple extrait de la norme C++
  struct C {
    int i;
    void f();
    const C& operator=( const C& );
  };
  const C& C::operator=( const C& other)
  {
    if ( this != &other )
    {
      this->~C();     // Fin de la vie de *this
      new (this) C(other);
                      // Création d'un nouvel objet C
      f();            // S'exécute correctement
    }
    return *this;
  }
  C c1;
  C c2;
  c1 = c2; // S'exécute correctement
  c1.f();  // S'exécute correctement. c1 est un nouvel objet C
  c1.f();  // S'exécute correctement. c1 est un nouvel objet C
```

Au passage, on peut remarquer que cet exemple présente un petit défaut : l'opérateur `C::operator=()` renvoie un `const C&` plutôt que de renvoyer un `C&`. Bien que ce choix permette d'éviter des usages abusifs comme (« `(a=b)=c` »), il interdit l'utilisation des objets de type `C` avec les conteneurs de la bibliothèque standard, qui requièrent que l'opérateur d'affectation renvoient une référence non constante (voir Cline 95 :212 et Murray 93 : 32-33)

Éloge de la simplicité

Restez simples. N'utilisez pas les fonctionnalités avancées du langage C++, même si elles sont tentantes, à moins d'en maîtriser parfaitement les conséquences. En un mot, n'écrivez jamais du code que vous comprenez pas.

En premier lieu, cela risque de poser des problèmes de portabilité car, d'une part il est dangereux d'utiliser trop vite les nouvelles fonctionnalités d'un langage, les compilateurs ne les implémentant pas tous dans le même délai (aujourd'hui, certains compilateurs ne gèrent pas encore, par exemple, les arguments par défaut pour les modèles ni la spécialisation des modèles) et, d'autre part certains compilateurs font purement et simplement l'impasse sur quelques fonctionnalités avancées du langage, existant pourtant depuis de nombreuses années (la gestion des exceptions multiples, par exemple).

En second lieu, cela risque de compliquer la maintenance : l'une des forces, mais aussi l'un des dangers du langage C++ est la possibilité qu'il offre d'écrire du code très concis. Un code trop elliptique pourra vous paraître particulièrement élégant sur le moment mais s'avérer être un cauchemar pour la personne qui s'y replongera des mois ou des années plus tard pour en assurer la maintenance – et pensez que cette personne, ce peut être vous !

Par exemple, une bibliothèque graphique bien connue utilisait la surcharge des opérateurs pour implémenter une fonction `operator+()` permettant d'ajouter un contrôle graphique à une fenêtre :

```
Window  w( /*...*/ );
control c( /*...*/ );
w + c;
```

Certes, cette astuce a pu paraître élégante au premier abord mais, à l'usage elle s'est avérée être une grande source de confusion pour les développeurs. Les auteurs de cette bibliothèque auraient mieux fait de suivre le conseil avisé de Scott Meyer : « faites comme font les `int`s » (autrement dit, conformez-vous toujours à la sémantique des types prédéfinis lorsque vous redéfinissez des opérateurs).

N'écrivez que ce que vous maîtrisez, tout en expérimentant peu à peu de nouvelles choses : lorsque vous écrivez un programme, faites en sorte de maîtriser parfaitement 90% du code, et, utilisez les 10% restants pour acquérir de l'expérience sur des fonctionnalités que vous connaissez moins. Si vous êtes débutant en C++, écrivez au début 90% du programme en C et introduisez progressivement les améliorations apportées par le C++.

Maîtrisez ce que vous écrivez : soyez toujours conscient de toutes les opérations implicites effectuées par votre code, maîtrisez-en les conséquences éventuelles. Tenez-vous régulièrement au courant des pièges du C++ en lisant des livres comme celui que vous avez entre les mains, en consultant des groupes de discussion Internet comme *comp.lang.c++.moderated* ou *comp.std.c++* ou des magazines comme *C/C++ User's Journal* et *Dr. Dobb's Journal*. Ceci vous permettra de rester toujours au meilleur niveau et de maîtriser à fond le langage que vous utilisez quotidiennement.

Gardez toujours un regard critique face à ces sources d'informations. Que penser du fait que la technique dont nous venons d'étudier les défauts – implémentation de l'opérateur d'affectation en fonction du constructeur de copie en utilisant une destruction explicite suivie d'une allocation placée – est souvent citée en exemple dans des groupes de discussion Internet ? C'est bien la preuve que certains ne *maîtrisent pas ce qu'ils écrivent*, ignorant que cette technique peut présenter des dangers certains et qu'il faut lui préférer le recours à une fonction membre privée `Swap()` appelée depuis le constructeur de copie et l'opérateur d'affectation.

En conclusion, comme nous l'avons déjà dit, évitez de pousser le langage dans ses derniers retranchements. Ne succombez jamais à la tentation d'une apparente élégance, potentielle source de problèmes subtils. N'écrivez que ce que vous maîtrisez et maîtrisez ce que vous écrivez.

Compléments divers

Pb n° 42. Initialisation de variables — Difficulté : 3

Maîtrisez-vous toujours parfaitement ce que vous écrivez ? Pour le vérifier, examinez les quatre lignes de code proposées ici : leur syntaxe est très proche, mais elles ont toutes un comportement différent.

Quelle est la différence entre ces quatre lignes de code ?

```
T t;
T t();
T t(u);
T t = u;
```

(`T` désigne une classe)

Solution

Ces lignes illustrent trois différentes formes d'initialisation possibles pour un objet : initialisation par défaut, initialisation directe et initialisation par le constructeur de copie. Quant à la quatrième forme, il s'agit d'un petit piège à éviter !

Prenons les lignes dans l'ordre :

```
T t;
```

Il s'agit ici d'une *initialisation par défaut* : cette ligne déclare une variable de type `T`, nommée `t`, initialisée par le constructeur par défaut `T::T()`.

```
T t();
```

Voici le piège ! Même si elle ressemble à une initialisation de variable, cette ligne n'initialise rien du tout : elle est interprétée par le compilateur comme la déclaration

d'une fonction nommée `t`, ne prenant aucun paramètre et renvoyant un objet de type `T` (si cela ne vous semble pas évident, remplacez le type `T` par `int` et le nom `t` par `f` : cela donne « `int f();` » ce qui est clairement une déclaration de fonction).

D'aucuns suggèrent qu'il est possible d'utiliser la syntaxe « `auto T t();` » pour bien spécifier au compilateur qu'on souhaite déclarer et initialiser un objet automatique `t` de type `T`, et non pas une fonction nommée `t` renvoyant un `T`. Ce n'est pas une pratique recommandable, pour deux raisons. La première est que cela ne marchera qu'avec certains compilateurs – au passage non conformes à la norme C++, un compilateur correctement implémenté devant rejeter cette ligne en indiquant qu'il n'est pas possible de spécifier le qualificatif '`auto`' pour un type de retour de fonction. La seconde est qu'il y a une technique beaucoup plus simple pour obtenir le même résultat : écrire « `T t ;` ».Ne cherchez pas la complication lorsque vous écrivez un programme ! La maintenance de votre code n'en sera que plus facile.

```
T t(u);
```

Il s'agit ici de l'*initialisation directe*. L'objet `t` est initialisé dès sa construction à partir de la valeur de la variable `u`, par appel du constructeur `T::T(u)`.

```
T t = u;
```

Il s'agit, pour finir, de l'*initialisation par le constructeur de copie*. L'objet `t` est initialisé par le constructeur de copie de `T`, lui-même éventuellement précédé par l'appel d'une autre fonction.

Erreur courante

Il ne faut pas confondre initialisation par le constructeur de copie et affectation. En dépit de la présence d'un signe =, l'instruction « `T t=u;` » n'appelle pas `T::operator=()`.

Cette dernière initialisation fonctionne de la manière suivante :

- Si `u` est de type `T`, cette instruction est équivalente à « `T::t(u)` » (le constructeur de copie de `T` est appelé directement).
- Si `u` n'est pas de type `T`, cette instruction est équivalente à « `T t(T(u));` » (`u` est converti en un objet temporaire de type `T`, lui-même passé en paramètre au constructeur de copie de `T`). Il faut savoir qu'en fonction du niveau d'optimisation demandé, certains compilateurs pourront éventuellement supprimer cet appel au constructeur de copie pour le remplacer par une initialisation directe du type « `T t(u)` ». Il ne faut donc pas que la cohérence de votre code repose sur l'hypothèse que le constructeur de copie sera systématiquement appelé dans une initialisation de ce type.

Recommandation

Préférez, lorsque cela est possible, l'emploi d'une initialisation de type « `T t(u)` » au lieu de « `T t = u;` ». Ces deux instructions sont fonctionnellement équivalentes, mais la première présente plus d'avantages – comme, en particulier, la possibilité de prendre plusieurs paramètres.

Pb n° 43. Du bon usage de const — Difficulté : 6

const est un outil puissant, pouvant contribuer nettement à la stabilité et à la sécurité d'un programme. Il faut néanmoins être judicieux dans son utilisation. Ce problème présente quelques cas où const doit être évité ou au contraire, utilisé.

Examinez le programme ci-dessus. Sans en changer la structure – légèrement condensée à des fins de clarté – commentez l'utilisation des mots-clés const.

Proposez une version corrigée du programme à laquelle vous aurez ajouté ou ôté des const (et effectué les éventuels changements mineurs corrélatifs)

Question supplémentaire : y a t'il, dans le programme original, des instructions pouvant provoquer des erreurs à l'exécution en raison de const oubliés ou, au contraire, superflus ?

```
class Polygon
{
  public:
    Polygon() : area_(-1) {}
    void  AddPoint( const Point pt ) { InvalidateArea();
                                       points_.push_back(pt); }
    Point GetPoint( const int i )    { return points_[i]; }
    int   GetNumPoints()             { return points_.size(); }
    double GetArea()
    {
      if( area_ < 0 ) // Si l'aire n'a pas été calculée...
      {
        CalcArea();   // ...on la calcule
      }
      return area_;
    }
  private:
    void InvalidateArea() { area_ = -1; }
    void CalcArea()
    {
      area_ = 0;
      vector<Point>::iterator i;
      for( i = points_.begin(); i != points_.end(); ++i )
        area_ += /* calcul de l'aire (non détaillé ici) */;
    }
    vector<Point> points_;
    double        area_;
};

Polygon operator+( Polygon& lhs, Polygon& rhs )
{
   Polygon ret = lhs;
   int last = rhs.GetNumPoints();
   for( int i = 0; i < last; ++i ) // concaténation des points
   {
     ret.AddPoint( rhs.GetPoint(i) );
```

```
    }
    return ret;
}

void f( const Polygon& poly )
{
  const_cast<Polygon&>(poly).AddPoint( Point(0,0) );
}

void g( Polygon& const rPoly ) { rPoly.AddPoint( Point(1,1) ); }

void h( Polygon* const pPoly ) { pPoly->AddPoint( Point(2,2) ); }

int main()
{
  Polygon poly;
  const Polygon cpoly;

  f(poly);
  f(cpoly);
  g(poly);
  h(&poly);
}
```

SOLUTION

Ce problème est l'occasion de signaler, d'une part, des erreurs courantes dans l'utilisation de `const`, et également, d'autre part, des situations plus subtiles pouvant requérir – ou non – l'utilisation de `const` (voir notamment le paragraphe « `const` et `mutable` : des amis qui vous veulent du bien »)

```
class Polygon
{
  public:
    Polygon() : area_(-1) {}
    void  AddPoint( const Point pt ) { InvalidateArea();
                                       points_.push_back(pt); }
```

1. L'objet `Point` étant passé par valeur, il n'y a aucun intérêt à le déclarer `const`, étant donné que la fonction n'aura, de toute façon, aucune possibilité de modifier l'objet original.

Signalons au passage que deux fonctions ayant la même signature et ne différant que par les attributs `const` des paramètres passés par valeur sont considérées comme une seule et même fonction par le compilateur (pas de surcharge) :

```
int f( int );
int f( const int );  // re-déclaration f(int)
                     // Pas de surcharge, une seule fonction f
int g( int& );
int g( const int& ); // Surcharge
                     // Pas la même fonction que g(int&)
```

Recommandation

Il n'est pas nécessaire de spécifier l'attribut `const` au niveau de la déclaration de la fonction pour un paramètre passé par valeur. En revanche, si ce paramètre n'est pas modifié par la fonction, spécifiez l'attribut `const` au niveau de la définition.

```
Point GetPoint( const int i )    { return points_[i]; }
```

2. Même commentaire. Il est inutile de déclarer `const` un paramètre passé par valeur.

3. En revanche, la fonction `GetPoint()` devrait être déclarée `const` car elle ne modifie pas l'état de l'objet.

4. `GetPoint()` devrait plutôt retourner un « `const Point` », afin d'éviter que le code appelant ne modifie l'objet temporaire renvoyé lors de l'exécution de la fonction. C'est une remarque générale s'appliquant à toutes les fonctions renvoyant un paramètre par valeur (sauf lorsqu'il s'agit d'un type prédéfini comme `int` ou `long`)[1].

Au passage, on pourrait se demander pourquoi `GetPoint()` ne renvoie pas une référence plutôt qu'une valeur, permettant ainsi aux appelants de placer la valeur retournée à gauche d'un opérateur d'affectation (comme, par exemple, dans l'instruction : « `poly.GetPoint(i) = Point(2,2);` »). Certes, ce type d'écriture est pratique, mais l'idéal est tout de même de renvoyer une valeur constante ou encore mieux, une référence constante, comme nous allons le voir plus loin, notamment au moment de l'étude de la fonction `operator+()`.

Recommandation

Les fonctions retournant un objet par valeur doivent en général renvoyer une valeur constante (sauf s'il s'agit d'un type prédéfini, comme `int` ou `long`).

```
int   GetNumPoints()             { return points_.size(); }
```

5. Même commentaire qu'au point (3) : cette fonction devrait être déclarée `const` car elle ne modifie pas l'état de l'objet.

```
double GetArea()
{
  if( area_ < 0 ) // Si l'aire n'a pas été calculée...
  {
    CalcArea();     // ...on la calcule
  }
  return area_;
}
```

1. Il n'y a aucun intérêt à renvoyer une valeur constante pour un type prédéfini, au contraire. Le fait q'une fonction renvoit un « `const int` » plutôt qu'un « `int` » est non seulement inutile (un type de retour de type prédéfini ne peut de toute façon pas être placé à gauche d'une affectation), mais en plus dangereux (cela peut gêner l'instanciation des modèles de classe ou de fonction). À ce sujet, voir aussi Lakos 96 (p. 618), auteur à propos duquel il faut d'ailleurs signaler qu'il n'est pas, contrairement à moi, favorable à l'emploi de valeur de retour `const` même pour les types objets.

6. Cette fonction devrait, elle-aussi, être déclarée `const`. Ce n'est pas si évident car elle modifie effectivement l'état de l'objet, seulement, il s'agit de l'état *interne* de l'objet (mise à jour de la variable membre `area_`, utilisée ici pour mettre une valeur en cache, à des fins d'optimisation). D'un point de vue externe, l'appel de cette fonction laisse l'objet inchangé. Autrement dit, la fonction `GetArea()` modifie l'objet d'un point de vue *physique* mais laisse l'objet inchangé d'un point de vue *logique*.

La meilleure chose à faire ici est donc de déclarer `const` la fonction `GetArea()` et d'appliquer l'attribut `mutable` à la variable `area_`, la rendant ainsi modifiable depuis une fonction membre constante. Voir à ce sujet le paragraphe « `const` et `mutable` : des amis qui vous veulent du bien ».

Si votre compilateur n'implémente pas encore `mutable`, contournez le problème en utilisant l'opérateur `const_cast` sur la variable `area_` (et insérez un commentaire dans votre code en prévision du jour où `mutable` sera disponible)

```
  private:
    void InvalidateArea() { area_ = -1; }
```

7. Cette fonction devrait également être déclarée constante, par cohérence avec le point précédent, car elle ne modifie pas l'état *externe* de l'objet. En effet, à partir du moment où il est décidé que le mécanisme de mise en cache de l'aire dans la variable `area_` est un détail d'implémentation interne, il ne doit se manifester dans aucune des fonctions de l'interface de la classe, fût-elle privée.

```
    void CalcArea()
    {
      area_ = 0;
      vector<Point>::iterator i;
      for( i = points_.begin(); i != points_.end(); ++i )
        area_ += /* calcul de l'aire (non détaillé ici) */;
    }
```

8. Là encore, cette fonction devrait être déclarée constante car elle ne modifie pas l'état interne de l'objet. De plus, cette fonction doit pouvoir être appelée depuis la fonction `GetArea()`,constante elle-aussi.

9. L'itérateur utilisé pour parcourir la liste ne doit pas modifier l'état des objets `points_`. Il devrait être par conséquent être remplacé par un `const_iterator`.

```
    vector<Point> points_;
    double        area_;
};

Polygon operator+( Polygon& lhs, Polygon& rhs )
```

10. Les paramètres `lhs` et `rhs` devraient être des références constantes, car les objets correspondants ne sont pas modifiés par la fonction `operator+()`.

11. Le type de retour de la fonction devrait être `const Polygon` ou `const Polygon&` (voir le point n° 4)

```
  Polygon ret = lhs;
  int last = rhs.GetNumPoints();
```

12. « int last » devrait être remplacé par « const int last », cette variable n'étant pas amenée à changer au cours de l'exécution de la fonction.

```
    for( int i = 0; i < last; ++i ) // concaténation des points
    {
      ret.AddPoint( rhs.GetPoint(i) );
    }
    return ret;
}
```

On remarque qu'il n'est possible de rendre la fonction operator+() constante que si GetPoint()renvoie une valeur (ou une référence) constante.

```
void f( const Polygon& poly )
{
  const_cast<Polygon&>(poly).AddPoint( Point(0,0) );
}
```

Cette fonction déclare un paramètre de type référence constante, semblant ainsi indiquer à l'appelant qu'elle ne modifiera pas l'objet référencé. Or, dans son implémentation interne, elle supprime le caractère constant du paramètre à l'aide d'un const_cast afin d'appeler la fonction AddPoint()! Soyez cohérent : si un paramètre n'est pas effectivement constant, ne le déclarez pas const !

```
void g( Polygon& const rPoly ) { rPoly.AddPoint( Point(1,1) ); }
```

13. Ce const est non seulement inutile (par définition, une référence pointe systématiquement toujours vers le même objet), mais, qui plus est, il n'est pas conforme à la syntaxe C++.

```
void h( Polygon* const pPoly ) { pPoly->AddPoint( Point(2,2) ); }
```

14. Cette fois, la syntaxe est correcte, mais le const n'en est pas moins inutile, pour une raison légèrement différente : à partir du moment où le pointeur pPoly est passé par valeur, la fonction h() n'a aucune possibilité de faire pointer le pointeur original vers un objet différent, puisqu'elle ne dispose que d'une *copie* de ce pointeur.

```
int main()
{
  Polygon poly;
  const Polygon cpoly;

  f(poly);
```

Pas de problème.

```
  f(cpoly);
```

Le résultat de l'exécution de cette ligne est indéterminé, la fonction f() essayant de forcer la modification d'un objet déclaré constant. Voir le point n° 12.

```
  g(poly);
```

Pas de problème.

```
  h(&poly);
}
```

Pas de problème.

Pour finir, voici la version révisée de notre exemple (les divers points de style non relatifs au mot-clé `const`, notamment l'emploi abusif de fonctions en-ligne à des fins de concision, n'ont pas été corrigés).

```
class Polygon
{
public:
  Polygon() : area_(-1) {}
  void        AddPoint( Point pt ) { InvalidateArea();
                                     points_.push_back(pt); }
  const Point GetPoint( int i ) const  { return points_[i]; }
  int         GetNumPoints() const { return points_.size(); }
  double GetArea() const
  {
    if( area_ < 0 ) // Si l'aire n'a pas été calculée...
    {
      CalcArea();   // ... on la calcule
    }
    return area_;
  }
private:
  void InvalidateArea() const { area_ = -1; }
  void CalcArea() const
  {
    area_ = 0;
    vector<Point>::const_iterator i;
    for( i = points_.begin(); i != points_.end(); ++i )
    {
      area_ += /* calcul de l'aire (non détaillé ici) */;
    }
  }
  vector<Point>  points_;
  mutable double area_;
};

const Polygon operator+( const Polygon& lhs,
                         const Polygon& rhs )
{
  Polygon ret = lhs;
  const int last = rhs.GetNumPoints();
  for( int i = 0; i < last; ++i ) // concaténation des points
  {
    ret.AddPoint( rhs.GetPoint(i) );
  }
  return ret;
}

void f( Polygon& poly )
{
  poly.AddPoint( Point(0,0) );
}
```

```
void g( Polygon& rPoly ) { rPoly.AddPoint( Point(1,1) ); }
void h( Polygon* pPoly ) { pPoly->AddPoint( Point(2,2) ); }


int main()
{
  Polygon poly;
  f(poly);
  g(poly);
  h(&poly);
}
```

const et mutable : des amis qui vous veulent du bien

Il n'est pas rare de rencontrer des développeurs réticents à l'idée d'utiliser const dans leurs programmes, sous prétexte qu'à partir du moment où on commence à déclarer const quelques arguments et quelques fonctions, il faut en général revoir, de proche en proche, l'intégralité du programme.

Il est vrai que l'utilisation de const représente un (petit) travail supplémentaire au niveau du développement – surtout lorsqu'il s'agit de réviser un programme existant n'utilisant que peu ou pas const. Néanmoins, les bénéfices que vous pourrez en retirer en terme de fiabilité, clarté et sécurité de votre code sont tels qu'il ne faut pas un instant hésiter à faire cet investissement !

Une des grandes forces du langage C++ est la puissance d'analyse du compilateur, capable de repérer un très grand nombre d'incohérences ou d'erreurs lors de la phase de développement, réduisant d'autant le risque d'erreurs à l'exécution. Mais pour bénéficier de cette puissance, encore faut-il utiliser les fonctionnalités correspondantes du langage : const est l'une d'entre elles.

En résumé, l'emploi de const permet d'accroître significativement la qualité du code. Les développeurs qui, par paresse, omettent de l'utiliser ou – ce sont souvent les mêmes – négligent les avertissements (*warnings*) du compilateur s'exposent inutilement à des risques supplémentaires.

Sachez utiliser, lorsque c'est nécessaire, le mot-clé mutable qui permet d'autoriser qu'une variable membre soit modifiée par une fonction membre constante. Ceci permet, comme dans l'exemple vu plus haut, de déclarer comme *constantes* des fonctions modifiant l'état interne de l'objet sans en modifier l'état externe.

Il est vrai que certaines bibliothèques du marché n'utilisent pas const comme elles le devraient. Si vous êtes contraint d'utiliser une de ces bibliothèques, n'imitez surtout pas ses défauts ! Prenons l'exemple d'une fonction membre déclarée non constante dans la bibliothèque alors qu'elle aurait dû l'être, vu son rôle. Cette erreur vous empêche d'invoquer la fonction en question sur un objet constant situé dans votre code. Il y a alors deux solutions : la mauvaise consiste à supprimer les const de votre code (quelle erreur !), la bonne consiste à utiliser l'opérateur const_cast sur votre objet, en attendant une meilleure version de la bibliothèque.

Finalement, l'une des meilleures justifications de l'importance de `const` est sans nul doute son introduction très précoce au sein du langage C, dès l'époque des premiers comités de normalisation ANSI. Le `const` du C permet de qualifier des variables; pour la petite histoire, voici comment est né le `const` des fonctions membres : Lorsque au début des années 1980, un jeune chercheur nommé Stroustrup inventa le « C avec classes », il introduisit dès le début le mot-clé `readonly` permettant de qualifier les membres d'une classe ne modifiant pas l'état de l'objet. L'idée plut à l'équipe des Bell Laboratories, qui préféra néanmoins le mot-clé `const`. Vous connaissez la suite... (voir Stroustrup94, page 90)

En résumé, comme aime à le rappeler Scott Meyers (Meyers98, problème n° 21) : « Utilisez `const` chaque fois que c'est possible ». Votre code ne sera que plus fiable, plus clair et plus sûr.

Pb n° 44. Opérateurs de casting — Difficulté : 6

Ce problème va vous permettre de tester votre connaissance des opérateurs de casting C++. Employés à bon escient, ils peuvent être d'une très grande utilité.

Les nouveaux opérateurs de casting (transtypage) apportés par le C++ offrent plus de possibilités que l'ancienne technique utilisée en C (coercition de type en faisant précéder une variable par le type cible, entre parenthèses). Les connaissez-vous bien ?

Dans toute la suite de ce problème, on se référera à l'ensemble de classes et de variables globales suivant :

```
class  A { public: virtual ~A(); /*...*/ };
A::~A() { }
class B : private virtual A  { /*...*/ };
class C : public  A          { /*...*/ };
class D : public B, public C { /*...*/ };
A a1; B b1; C c1; D d1;
const A  a2;
const A& ra1 = a1;
const A& ra2 = a2;
char c;
```

Ce problème comporte quatre questions.

1. Parmi les nouveaux opérateurs de casting C++, rappelés ci-dessous, lesquels n'ont pas d'équivalents dans l'ancienne technique de casting utilisée en C ?

```
const_cast
dynamic_cast
reinterpret_cast
static_cast
```

2. Réécrivez les instructions suivantes en utilisant les nouveaux opérateurs de casting C++. Y a t'il, parmi elles, des instructions illégales sous leur forme originale ?

```
void f()
{
  A* pa; B* pb; C* pc;
  pa = (A*)&ra1;
  pa = (A*)&a2;
  pb = (B*)&c1;
  pc = (C*)&d1;
}
```

3. Commentez toutes les instructions suivantes. Indiquez en particulier les instructions non valides et celles dont le style est discutable.

```
void g()
{
  unsigned char* puc = static_cast<unsigned char*>(&c);
  signed char* psc = static_cast<signed char*>(&c);
  void* pv = static_cast<void*>(&b1);
  B* pb1 = static_cast<B*>(pv);
  B* pb2 = static_cast<B*>(&b1);
  A* pa1 = const_cast<A*>(&ra1);
  A* pa2 = const_cast<A*>(&ra2);
  B* pb3 = dynamic_cast<B*>(&c1);
  A* pa3 = dynamic_cast<A*>(&b1);
  B* pb4 = static_cast<B*>(&d1);
  D* pd = static_cast<D*>(pb4);
  pa1 = dynamic_cast<A*>(pb2);
  pa1 = dynamic_cast<A*>(pb4);
  C* pc1 = dynamic_cast<C*>(pb4);
  C& rc1 = dynamic_cast<C&>(*pb2);
}
```

4. Dans quels cas est-il utile d'utiliser `const_cast` pour convertir un objet non constant en objet constant ? Donnez un exemple précis.

Solution

1. *Parmi les nouveaux opérateurs de casting C++, rappelés ci-dessous, lesquels n'ont pas d'équivalents dans l'ancienne technique de casting utilisée en C ?*

Seul l'opérateur `dynamic_cast` n'a pas d'équivalent. Tous les autres peuvent être exprimés de manière équivalente avec l'ancienne technique de casting utilisée en C.

Recommandation

Préférez les nouveaux opérateurs disponibles en C++ à l'ancienne technique de casting utilisée en C.

2. *Réécrivez les instructions suivantes en utilisant les nouveaux opérateurs de casting C++. Y a t'il, parmi elles, des instructions illégales sous leur forme originale ?*

```
void f()
{
  A* pa; B* pb; C* pc;
  pa = (A*)&ra1;
```

Pour cette instruction, on utilisera l'opérateur `const_cast` :

```
pa = const_cast<A*>(&ra1);
```

La deuxième instruction pose problème :

```
pa = (A*)&a2;
```

Cette instruction ne peut pas être exprimée à l'aide d'un opérateur de casting C++. Le candidat le mieux placé aurait été `const_cast`, mais il aurait risqué de produire un résultat indéterminé à l'exécution, `a2` étant un objet constant.

```
pb = (B*)&c1;
```

Pour cette troisième instruction, on utilisera `reinterpret_cast` :

```
pb = reinterpret_cast<B*>(&c1);
```

Et pour finir :

```
pc = (C*)&d1;
```

Cette dernière instruction est illégale en C. En revanche, elle peut être écrite directement en C++, sans recourir à aucun opérateur de casting :

```
pc = &d1;
```

3. *Commentez toutes les instructions suivantes. Indiquez en particulier les instructions non valides et celles dont le style est discutable.*

Avant de rentrer dans le détail, une petite remarque d'ordre général : tous les appels à `dynamic_cast` dans l'exemple suivant seraient invalides si les classes mises en jeu ne comportaient pas de fonction virtuelle. Par chance, A en comporte une.

```
void g()
{
  unsigned char* puc = static_cast<unsigned char*>(&c);
  signed char* psc = static_cast<signed char*>(&c);
```

Erreur ! Il faut utiliser `reinterpret_cast` pour ces deux lignes. En effet, `char`, `signed` et `unsigned char` sont considérés en C++ comme trois types bien distincts (bien qu'il existe des conversions implicites entre eux).

```
  void* pv = static_cast<void*>(&b1);
  B* pb1 = static_cast<B*>(pv);
```

Ces deux lignes sont correctes, mais dans la première, l'opérateur `static_cast` est superflu, car pour tout pointeur, il existe déjà une conversion implicite vers `void*`.

```
B* pb2 = static_cast<B*>(&b1);
```

L'emploi de `static_cast` est ici correct, mais sans intérêt, `&b1` étant déjà de type `B*`.

```
A* pa1 = const_cast<A*>(&ra1);
```

La syntaxe correcte n'occulte pas un mauvais style de programmation. Dans la grande majorité des cas, il est recommandé de ne pas altérer le caractère *constant* d'un objet. Si c'est pour appeler une fonction membre non constante, alors il est préférable de rendre la fonction constante et d'employer le mot-clé `mutable`, comme nous l'avons vu dans le problème n° 43.

Recommandation

N'utilisez pas `const_cast` pour supprimer le caractère constant d'une référence. Modifiez plutôt l'objet référencé en spécifiant l'attribut `mutable` pour les variables membres qui le nécessitent.

```
A* pa2 = const_cast<A*>(&ra2);
```

Erreur ! contrairement au cas précédent, c'est l'objet référencé par `ra2` qui est constant, et non plus la référence. Par conséquent, l'utilisation du pointeur `pa2` ainsi obtenu risque de provoquer une erreur à l'exécution (l'objet constant `a2` aura pu, par exemple, être stocké par le compilateur dans une zone de mémoire en lecture seule). D'une manière générale, il est dangereux de rendre non constant un objet constant.

Recommandation

N'utilisez pas `const_cast` pour rendre non constant un objet constant.

```
B* pb3 = dynamic_cast<B*>(&c1);
```

Erreur ! Cette instruction va affecter à `pb3` la valeur `NULL`, pour la bonne raison que `c1` n'est pas un objet dérivé de `B`. Il s'ensuivra une erreur à l'exécution si on tente d'utiliser le `pb3` dans la suite du code. Le seul autre opérateur dont la syntaxe est correcte ici aurait été `reinterpret_cast`, dont il n'est même pas besoin de préciser les conséquences désastreuses dans cette situation.

```
A* pa3 = dynamic_cast<A*>(&b1);
```

Erreur aussi ! C'est moins évident de prime abord, car `B` dérive effectivement de `A`. Mais cette dérivation étant privée, `b1` « N'EST-PAS-UN » `A` ; par conséquent, le résultat de cette opération sera un pointeur nul, à moins que `g()` soitune fonction amie de `B`.

```
B* pb4 = static_cast<B*>(&d1);
```

Cette instruction est correcte, mais l'emploi de `static_cast` est superflu car on peut toujours convertir implicitement un pointeur vers un objet dérivé en un pointeur du type d'une des classes de base.

```
D* pd = static_cast<D*>(pb4);
```

Cette instruction est correcte. L'emploi d'un `dynamic_cast` aurait certes été plus judicieux, mais il est tout à fait autorisé d'utiliser un `static_cast` pour transformer un pointeur vers n'importe quel autre pointeur *accessible* de la même hiérarchie de classe. Il aurait néanmoins fallu préférer ici un `dynamic_cast`, du fait qu'il limite le risque d'erreurs. En effet, si le pointeur à convertir pointe vers un objet dérivé de la classe du pointeur cible (« casting descendant »), tout se passe bien. Mais si, en revanche, il pointe vers un objet de base, cela risque de provoquer des erreurs à l'exécution difficiles à repérer dans le cas d'un `static_cast` (pointeur « partiellement valable ») alors que dans le cas d'un `dynamic_cast`, toute conversion invalide résultera en un pointeur nul.

Recommandation

Utilisez `dynamic_cast` plutôt que `static_cast` pour réaliser des « casting descendants ».

```
pa1 = dynamic_cast<A*>(pb2);
pa1 = dynamic_cast<A*>(pb4);
```

Ces deux lignes sont relativement similaires, dans la mesure où elles tentent toutes les deux de convertir un `B*` en `A*`. Pourtant, le résultat de la première conversion sera un pointeur nul, tandis que la deuxième se déroulera correctement. Nous avons déjà vu la raison plus haut : B dérive de A de manière privée, et, par conséquent, l'objet `b1`, vers lequel pointe `pb2`, « N'EST-PAS-UN » `A`, ce qui rend l'opération de conversion impossible.

Dans ces conditions, pourquoi la deuxième ligne s'exécute-t-elle correctement ? Il se trouve que `pb4` pointe vers l'objet `d1` et que `D` dérive publiquement de `A` via la hiérarchie `D`, `C`, `A`. L'opérateur `dynamic_cast` est capable de naviguer au sein de la hiérarchie de classe afin d'atteindre la classe cible : en l'occurrence, il aura effectué les conversions suivantes : `B* -> D* -> C* -> A*`.

```
C* pc1 = dynamic_cast<C*>(pb4);
```

Instruction correcte, en vertu de ce que nous venons de voir : la capacité de `dynamic_cast` à naviguer au sein d'une hiérarchie de classe.

```
  C& rc1 = dynamic_cast<C&>(*pb2);
}
```

Pour finir, une instruction qui générera une exception à l'exécution. En effet, il est clair qu'il n'est pas possible de convertir (`*pb2`) en une référence vers `C` car (`*pb2`) « N'EST-PAS-UN » `C`. Ce qui change, en revanche, c'est qu'avec des références, l'opérateur n'a pas de moyen simple d'indiquer qu'une erreur produite. Ne pouvant pas renvoyer une hypothétique « référence nulle », l'opération génère donc une exception `bad_cast`.

4. *Dans quels cas est-il utile d'utiliser `const_cast` pour convertir un objet non constant en objet constant ? Donnez un exemple précis.*

Dans les trois questions précédentes, nous n'avons utilisé const_cast que pour convertir des objets constants en objets non-constants.

Il faut savoir qu'il est également possible d'utiliser cet opérateur pour effectuer l'opération inverse, à savoir rendre constant des objets non-constants. C'est d'un emploi rare; cela peut néanmoins être utile dans le cas de fonctions surchargées ne différant que par le caractère constant ou non des paramètres :

```
void f( T& );
void f( const T& );
template<class T> void g( T& t )
{
  f( t );                         // Appelle f(T&)
  f( const_cast<const T&>(t) ); // Appelle f(const T&)
}
```

Dans cet exemple, l'emploi de const_cast pour convertir la référence à T en une référence constante est le seul moyen d'appeler spécifiquement la fonction f(const T&).

Remarquons au passage que si f() avait été un modèle de fonction (template<class T> void f(T);), il aurait été maladroit de l'appeler en utilisant la syntaxe f(const_cast<const T&>(t)).Une instanciation spécifique de la bonne version du modèle aurait été plus judicieuse (f<const T&>(t)) .

Pb n° 45. Le type bool — Difficulté : 7

L'introduction du type bool en C++ se justifie t'elle vraiment ? N'aurait-il pas été possible de l'émuler en utilisant les autres fonctionnalités du langage ?

Le C++ introduit deux nouveaux types prédéfinis par rapport au C : bool, utilisé pour stocker les variables booléennes, et wchar_t (qui était un typedef en C), utilisé pour stocker les caractères codés sur deux octets.

Était-il nécessaire d'introduire ce type bool ? Indiquez les diverses solutions qui auraient permis d'émuler le type bool en utilisant les fonctionnalités existantes du langage, en précisant les limites de chacune.

Solution

Il n'est pas possible d'obtenir une implémentation exactement équivalente au type bool en utilisant les autres fonctionnalités du langage. C'est d'ailleurs la raison fondamentale pour laquelle ce type, ainsi que les mots-clés true et false, ont été ajoutés au C++.

Il existe néanmoins des implémentations approchantes, que nous allons présenter ici en indiquant, pour chacune, ses inconvénients.

Il y a quatre principales options : utilisation de `typedef`, utilisation de `#define`, utilisation d'`enum` ou création d'une classe `bool`.

Option 1 : `typedef` (note : 8.5/10)

Cette option consiste à définir un nouveau nom de type, correspondant à un type entier existant, `int` par exemple :

```
// Option 1: typedef
//
typedef int bool;
const bool true  = 1;
const bool false = 0;
```

Cette solution n'est pas mauvaise, mais elle présente quelques inconvénients :

- Le type `bool` n'est pas considéré comme un type séparé du point de vue de la surcharge des fonctions :

```
// Fichier f.h
void f( int  ); //
void f( bool ); // Redéclare la même fonction !

// Fichier f.cpp
void f( int  ) { /*...*/ }   // OK
void f( bool ) { /*...*/ }   // Erreur, redéfinition de f !
```

- Le champ des valeurs possibles pour une variable `bool` n'est pas restreint à `true` et `false` :

```
void f( bool b )
{
  assert( b != true && b != false );
 // Cette assertion peut potentiellement échouer !
}
```

Cette option est donc valable, mais pas idéale.

Option 2 : `#define` (note : 0/10)

Cette option consiste à définir utiliser l'instruction `#define` du préprocesseur :

```
// Option 2: #define
//
#define bool int
#define true  1
#define false 0
```

Cette solution est bien entendu catastrophique. Elle présente non seulement les

défauts de l'option 1, mais également tous les problèmes habituels liés à l'utilisation de `#define` (effets de bords, écrasements de noms,...). À éviter.

Option 3 : `enum` (note : 9/10)

Cette option consiste à définir utiliser un type énuméré :

```
// Option 3: enum
//
enum bool { false, true };
```

Cette solution est relativement bonne, dans la mesure où elle pallie les défauts de l'option 1 relatifs à la surcharge de fonction et aux champs des valeurs possibles.

Elle pose malheureusement problème dans le cas particulier des expressions conditionnelles :

```
bool b;
b = ( i == j ); // Erreur de compilation !
```

Cela ne fonctionne pas, car il n'est pas possible de convertir implicitement un `int` en un `enum`.

Option 4 : `class` (note : 9/10)

Comme, après tout, nous utilisons un langage orienté-objet, pourquoi ne pas créer une classe `bool` ?

```
class bool
{
public:
  bool();
  bool( int );            // Pour les conversions implicites
  bool& operator=( int ); // depuis le type 'int'
  //operator int();   // À voir...
  //operator void*(); // À voir...
private:
  unsigned char b_;
};
const bool true ( 1 );
const bool false( 0 );
```

Cette classe répond pratiquement à nos besoins. Elle pose néanmoins un petit problème au sujet des opérateurs de conversion marqués « *à voir* » :

- Si ces opérateurs sont définis, il risque de survenir les ennuis classiques liés à l'utilisation d'opérateurs de conversion et/ou de constructeurs de conversion non explicites, notamment lorsqu'il s'agit de conversions de et vers des types simples (interférences lors de la résolution des appels de fonctions surchargées, notamment). Voir les problèmes n° 20 et n° 39 pour plus de détails à ce sujet.

- Si ces opérateurs ne sont *pas* définis, notre type `bool` risque de ne pas se comporter naturellement dans certains situations pourtant classiques :

```
bool b;
/*...*/
if( b ) // Erreur si b ne dispose pas d'une conversion
{       // automatique vers int
    /*...*/
}
```

Nous nous trouvons donc face un petit dilemme : soit nous implémentons ces opérateurs, au risque de perturber la surcharge des fonctions, soit nous les omettons, nous exposant ainsi à d'autres types de problèmes. Dans aucun des cas, nous n'obtenons une classe `bool` équivalente au type `bool` prédéfini du C++.

Résumons la situation :

- L'utilisation d'un `typedef` présente quelques inconvénients au niveau de la surcharge des fonctions.
- Un `#define` présente non seulement les inconvénients du `typedef` mais également tous les problèmes usuels liés au préprocesseur.
- Un `enum` interdit la conversion implicite du résultat d'une expression conditionnelle vers un `bool` (comme « `b= (i == j)` »)
- Une classe `bool` interdit soit la surcharge des fonctions, soit l'emploi d'un booléen dans une instruction conditionnelle (du type « `if(b)` »), en fonction de l'implémentation ou non des opérateurs de conversions vers `int` ou `void*`.

Il apparaît donc clairement que le langage C++ avait vraiment besoin d'un type `bool` prédéfini, lequel est, signalons-le au passage, également utilisé comme type de retour des expressions conditionnelles (ce qui n'aurait pas été possible de réaliser avec nos implémentations – à part, à la rigueur, avec la quatrième).

Pb n° 46. Transferts d'appel — Difficulté : 3

Il n'est pas rare d'avoir besoin de « fonctions de transferts » dont l'unique rôle est de transférer l'appel à une autre fonction, à laquelle le travail est sous-traité. Les fonctions de ce type ne sont pas très complexes à implémenter ; ce problème, qui leur est consacré, sera pourtant l'occasion d'aborder une petite subtilité des compilateurs à ce sujet.

Les fonctions de transferts sont couramment utilisées pour sous-traiter une tâche à une autre fonction ou à un objet. Il est important que ces fonctions soit performantes, sous peine de pénaliser inutilement la vitesse d'exécution du programme.

Examinez la fonction de transfert suivante :

```
// Fichier f.cpp
//
#include "f.h"
/*...*/

bool f( X x )
{
  return g( x );

}
```

Vous paraît-elle correctement implémentée ? Sinon, que changeriez-vous ?

Solution

La fonction `f()` doit avant tout être performante. À ce titre, elle n'est pas optimale sous la forme présentée ici et pourrait être améliorée sur deux points :

1. **Passer le paramètre sous la forme d'une référence constante, et non pas par valeur.**

   La fonction `f()` fait une copie de l'objet `x` qui lui est passé en paramètre, puis passe à son tour cette copie à la fonction `g()`. Cette copie inutile de `x` pénalise les performances, alors qu'elle pourrait facilement être évitée par l'emploi d'une référence constante au lieu du passage de paramètre par valeur.

   C'est l'occasion ici de faire une petite remarque d'ordre historique : jusqu'en 1997, il n'était pas obligatoire d'utiliser un passage de paramètre par référence afin d'éviter cette copie inutile. En effet, la norme C++ spécifiait alors qu'un compilateur était autorisé à éliminer les paramètres d'une fonction dont l'unique utilisation était d'être transférés à une autre fonction : le compilateur pouvait, à la place, passer directement les paramètres en question à la fonction appelée. Prenons un exemple :

   ```
   X my_x;
   f( my_x );
   ```

   Face à un code de ce type, les compilateurs étaient autorisés à passer `my_x` directement à `g`, au lieu de créer une variable, copie de `my_x`, au sein de la fonction `f()` puis de passer cette variable à `g()`.

   Le problème est que cette technique d'optimisation, autorisée mais pas imposée, n'était pas systématiquement implémentée par tous les compilateurs. C'est la raison pour laquelle elle a été remise en question, puis interdite, lors d'une réunion du comité de normalisation C++ à Londres en 1997. Elle a en effet été jugée dangereuse dans la mesure où il peut arriver qu'un programme utilise, à des fins d'implémentation interne, le nombre d'appels effectués à un constructeur de copie. Ce nombre pouvait varier d'un compilateur à un autre, en fonction de la présence ou non de l'optimisation en question.

À l'heure actuelle, les seules situations où le compilateur est autorisé, à des fins d'optimisation, à passer outre l'appel d'un constructeur de copie sont les paramètres de retour de fonction et les objets temporaires.

Dans le cas de notre fonction `f()`, la seule solution pour éviter une copie inutile est donc de passer en paramètre une référence constante au lieu d'une valeur.

Recommandation

Plutôt que de passer des paramètres par valeur, utilisez, lorsque cela est possible, des références constantes.

Même si, avant 1997, le recours à une référence était optionnel (étant donné l'optimisation réalisée alors par la majorité des compilateurs), il était tout de même conseillé, à des fins de portabilité.

Recommandation

Ne fondez pas le bon fonctionnement de votre programme sur des optimisations réalisées par le compilateur.

2. Implémenter la fonction `f()` en ligne

Cette optimisation est plus discutable et peut dépendre du contexte : le fait d'écrire une fonction en ligne (*inline*) augmentant sa rapidité d'exécution mais augmentant également la taille du programme.

Dans le cas des fonctions de transfert, pour lesquels le code est minimal et la rapidité d'exécution, en général, fondamentale, l'écriture en ligne s'impose.

Mais attention ! L'emploi des fonctions en ligne doit être réservé à des cas particuliers comme celui-ci.

Recommandation

N'utilisez les fonctions en ligne (inline) que dans les cas très spécifiques où la recherche de la performance est une contrainte majeure.

Au passage, il faut signaler que le fait de déclarer `f()` en ligne présente l'inconvénient supplémentaire de rendre le code client dépendant de l'implémentation de `f()` et, en particulier, du prototype de `g()`, et donc, par conséquent, des éventuelles déclarations de types des paramètres de `g()`. Ceci augmente les dépendances au sein du code, ce qui est d'autant plus dommageable que le code client n'avait absolument pas besoin d'appeler, ni donc de connaître `g()`.

En conclusion, les fonctions en ligne ont leurs avantages et leurs inconvénients. Elles doivent être utilisées avec parcimonie, lorsque le contexte l'impose.

Pb n° 47. Contrôle du flot d'exécution Difficulté : 6

Une bonne connaissance de l'ordre dans lequel les instructions d'un programme seront exécutées est primordiale pour éviter les erreurs d'exécution. C'est le sujet de ce dernier problème.

Examinez le code suivant et relevez toutes les instructions risquant de provoquer des problèmes dus à une mauvaise maîtrise de leur ordre d'exécution.

```
#include <cassert>
#include <iostream>
#include <typeinfo>
#include <string>

using namespace std;

//  Les lignes suivantes proviennent d'autres fichiers en-tête
//
char* itoa( int valeur, char* buffer, int base );
extern int CompteurDeFichiers;

// Fonctions et classes et macros utilitaires
// pour vérifier la taille du tableau
//
template<class T>
inline void VerifierTaille( T& p )
{
  static int NumeroDuFichier= ++CompteurDeFichiers;
  if( !p.VerifierTaille() )
  {
    cerr << "Echec de la vérification de taille"
     << "fichier : " << NumeroDuFichier
     << ", " << typeid(p).name()
     << " à l'adresse "
     << static_cast<void*>(&p) << endl;
    assert( false );
  }
}

template<class T>
class Verificateur
{
public:
  Verificateur( T& p ) : p_(p) { VerifierTaille( p_ ); }
  ~Verificateur()              { VerifierTaille( p_ ); }
private:
  T& p_;
};

#define VERIFIER_TAILLE Verificateur<TypeTableau> V( *this )
```

```
//-------------------------------------------------------

// TableauBase et conteneur sont des classes de base
             // non détaillées ici

template<class T>
class Tableau : private TableauBase, public conteneur
{
  typedef Tableau TypeTableau;
public:
  Tableau( size_t TailleInitiale = 10 )
  : conteneur ( TailleInitiale ),
    TableauBase( conteneur::GetType() ),
    TailleUtilisee_(0),
    TailleTotale_(TailleInitiale),
    buffer_(new T[TailleTotale_])
  {
    VERIFIER_TAILLE;
  }

  void Redimensionner( size_t NouvelleTaille )
  {
    VERIFIER_TAILLE;
    T* oldBuffer = buffer_;
    buffer_ = new T[NouvelleTaille];
    copy( oldBuffer, oldBuffer+
                     min(TailleTotale_,NouvelleTaille), buffer_ );
    delete[] oldBuffer;
    TailleTotale_ = NouvelleTaille;
  }

  string AfficherTailles()
  {
    VERIFIER_TAILLE;
    char buf[30];
    return string("Taille totale = ")
    + itoa(TailleTotale_,buf,10)
    + ", taille utilisée = "
    + itoa(TailleUtilisee_,buf,10);
  }

  bool VerifierTaille()
  {
    if( TailleUtilisee_ > 0.9*TailleTotale_ )
          Redimensionner( 2*TailleUtilisee_ );
    return TailleUtilisee_ <= TailleTotale_;
  }
private:
  T*     buffer_;
  size_t TailleUtilisee_, TailleTotale_;
};

int f( int& x, int y = x ) { return x += y; }
int g( int& x )            { return x /= 2; }
```

```
void main( int, char*[] )
{
  int i = 42;
    cout << "f(" << i << ") = " << f(i) << ", "
         << "g(" << i << ") = " << g(i) << endl;
  Tableau<char> a(20);
  cout << a.AfficherTailles() << endl;
}
```

Solution

Cet exemple de code comporte un très grand nombre de problèmes potentiels. Examinons-le ligne par ligne :

```
#include <cassert>
#include <iostream>
#include <typeinfo>
#include <string>

using namespace std;

//  Les lignes suivantes proviennent d'autres fichiers en-tête
//
char* itoa( int valeur, char* buffer, int base );
extern int CompteurDeFichiers;
```

Cette déclaration fait référence à une variable globale déclarée dans un autre module, visiblement utilisée pour gérer un compteur de fichiers. En C++, l'ordre dans lequel les variables globales (incluant les membres statiques de classe) sont initialisées est indéterminé. Il y a donc un risque d'utiliser `CompteurDeFichiers` dans notre code avant que son initialisation, située dans un autre module, n'ait eu lieu.

Recommandation

Évitez l'utilisation de variables globales ou de variables membres statiques. Si vous êtes contraints d'en utiliser, soyez conscients des règles qui régissent leur initialisation.

```
// Fonctions et classes et macros utilitaires
// pour vérifier la taille du tableau
//
template<class T>
inline void VerifierTaille( T& p )
{
  static int NumeroDuFichier= ++CompteurDeFichiers;
```

Voici justement un cas pratique pouvant poser problème. Rien n'indique que la variable globale `CompteurDeFichiers` sera initialisée avant la variable membre statique `NumeroDuFichier`. Si, par exemple, la variable globale est initialisée avec une valeur non nulle (`int CompteurDeFichiers = 100 ;`) mais que l'initialisation de la

variable membre statique a lieu avant celle de la variable globale, alors le compte des fichiers commencera à 0 (valeur d'initialisation par défaut pour les types prédéfinis) au lieu de 100.

```
  if( !p.VerifierTaille() )
  {
    cerr << "Echec de la vérification de taille"
     << "fichier : " << NumeroDuFichier
     << ", " << typeid(p).name()
     << " à l'adresse "
     << static_cast<void*>(&p) << endl;
    assert( false );
  }
}

template<class T>
class Verificateur
{
public:
  Verificateur( T& p ) : p_(p) { VerifierTaille( p_ ); }
  ~Verificateur()              { VerifierTaille( p_ ); }
private:
  T& p_;
};

#define VERIFIER_TAILLE Verificateur<TypeTableau> V( *this )
```

L'idée consistant à employer une macro `VERIFIER_TAILLE` créant un objet local de type `Verificateur<TypeTableau>`, où `TypeTableau` est un `typedef` correspondant au type de l'instance de modèle de classe `Tableau`, lequel fait appel à la fonction `VerifierTaille()` qui provoque un arrêt du programme en cas d'incohérence (taille utilisée du tableau supérieure à la taille allouée) est foncièrement bonne.

Ce code présente néanmoins le défaut de ne s'exécuter correctement qu'en mode *debug*, vu qu'il repose sur l'emploi de la fonction `assert()`. L'emploi d'`assert()` peut certes constituer un aide supplémentaire, mais il aurait fallu prévoir un système de remontée d'erreur s'exécutant aussi bien en mode *debug* qu'en mode *non-debug*.

```
// TableauBase et conteneur sont des classes de base
              // non détaillées ici

template<class T>
class Tableau : private TableauBase, public conteneur
{
  typedef Tableau TypeTableau;
public:
  Tableau( size_t TailleInitiale = 10 )
  : conteneur ( TailleInitiale ),
    TableauBase( conteneur::GetType() ),
```

Si `conteneur::GetType()` est une fonction membre statique (n'ayant par définition accès qu'à des variables statiques), cet appel fonctionnera correctement. S'il s'agit, en revanche, d'une fonction membre normale retournant la valeur d'une varia-

ble membre normale, il se posera un problème du fait que l'objet de base `conteneur` ne sera pas encore initialisé au moment de l'appel du constructeur de `TableauBase` : en effet, les classes de base non virtuelles sont initialisées dans l'ordre de leur déclaration, de gauche à droite (et, dans notre cas, `TableauBase` est déclarée avant `conteneur`).

Recommandation

Dans la liste d'initialisation d'un constructeur, faites toujours appel aux construc-teurs des classes de base dans l'ordre de leur déclaration.

```
  TailleUtilisee_(0),
  TailleTotale_(TailleInitiale),
  buffer_(new T[TailleTotale_])
```

Cette fois, il s'agit d'une grave erreur ! L'allocation de `buffer_` ne se fera absolument pas correctement, car l'initialisation des variables s'effectue dans l'ordre où elles sont *déclarées* dans la classe, en l'occurrence:

```
  buffer_(new T[TailleTotale_]),
  TailleUtilisee_(0),
  TailleTotale_(TailleInitiale)
```

Sous cette forme, qui correspond effectivement à l'ordre d'appel, il apparaît clairement que la variable `TailleTotale_` n'est pas initialisée au moment de l'allocation de `buffer_`, ce qui conduit à un tableau d'une taille complètement aléatoire !

Recommandation

Dans la liste d'initialisation d'un constructeur, initialisez toujours les variables membres dans l'ordre de leur déclaration dans la classe.

```
{
  VERIFIER_TAILLE;
}
```

La fonction `VerifierTaille()` est appelée deux fois, depuis le constructeur et le destructeur de `Verificateur` : c'est une fois de trop. Néanmoins, en comparaison des autres problèmes vus jusqu'ici, il s'agit là plus d'un détail d'optimisation que d'une véritable erreur.

```
void Redimensionner( size_t NouvelleTaille )
{
   VERIFIER_TAILLE;
  T* oldBuffer = buffer_;
  buffer_ = new T[NouvelleTaille];
  copy( oldBuffer, oldBuffer+
     min(TailleTotale_,NouvelleTaille), buffer_ );
  delete[] oldBuffer;
  TailleTotale_ = NouvelleTaille;
```

```
}
```

Cette fonction est mal implémentée car elle ne se comportera pas correctement en présence d'exceptions. Ce n'est pas l'instruction « `new T` » qui est en cause (si celle-ci lance une exception `bad_alloc`, les variables du programme resteront dans un état cohérent) mais l'opérateur d'affectation de `T` invoqué... par la fonction `copy()`, bien sûr. Si une exception est générée au cours de cette opération de copie, la fonction `Redimensionner()` se terminera en laissant l'objet dans un état incohérent (deux tableaux alloués, l'ancien et le nouveau ; plus de pointeur vers l'ancien tableau, donc plus de moyen de le désallouer).

Recommandation

Assurez-vous que votre code se comporte correctement en présence d'exceptions. En particulier, organisez votre code de manière à désallouer correctement les objets et à laisser les données dans un état cohérent, même en présence d'exceptions.

```
string AfficherTailles()
{
     VERIFIER_TAILLE;
    char buf[30];
  return string("Taille totale = ")
  + itoa(TailleTotale_,buf,10)
  + ", taille utilisée = "
  + itoa(TailleUtilisee_,buf,10);
}
```

La fonction `itoa()` qui réalise la conversion d'un entier vers une chaîne de caractère utilise le tableau de caractères `buf` passé en paramètres pour placer la chaîne résultat.

```
char* itoa( int valeur, char* buffer, int base );
```

La fonction `AfficherTailles()` risque de poser problème du fait que les divers opérandes des opérateurs `+` ne seront pas toujours évalués dans le même ordre – en fonction du compilateur utilisé. Les appels successifs à `itoa` vont ainsi écraser à chaque fois la variable `buf` provoquant, dans certains cas, des résultats incohérents. Tout dépendra de l'ordre dans lequel seront évalués les paramètres de la fonction `operator+()`.

Pour le voir plus clairement, réécrivons l'instruction `return` en utilisant la syntaxe complète de la fonction `operator+` (les appels sont toujours évalués de gauche à droite) :

```
return
  operator+(
    operator+(
      operator+( string("Taille totale = "),
                 itoa(TailleTotale_,buf,10) ) ,
      ", taille utilisée = " ) ,
    itoa(TailleUtilisee_,buf,10) );
```

Considérons, par exemple, que la taille totale vaut 10, la taille utilisée vaut 5 et intéressons-nous à l'ordre d'évaluation des paramètres de la fonction `operator+()` la plus extérieure. Si c'est le premier paramètre qui est évalué d'abord, alors l'affichage sera correct (« `Taille totale = 10, taille utilisée= 5` ») car le résultat du premier appel `itoa()` sera stocké dans un tableau temporaire, du fait de la composition de plusieurs opérateurs + dans l'opérande de gauche; ainsi, le fait que le second appel `itoa()` écrase la valeur de `buf` sera sans conséquence. Si, en revanche, c'est le second paramètre qui est évalué en premier, alors l'affichage produit sera incorrect (« `Taille totale = 10, taille utilisée= 10` »), car le second appel à `itoa()` écrasera la valeur de `buf` récupérée à l'issue du premier appel, laquelle n'aura pas, cette fois, été stockée dans un tableau temporaire.

Erreur à éviter

N'écrivez jamais du code dont l'exécution dépend de l'ordre d'évaluation des paramètres d'une fonction.

```
bool VerifierTaille()
{
  if( TailleUtilisee_ > 0.9*TailleTotale_ )
        Redimensionner( 2*TailleUtilisee_ );
  return TailleUtilisee_ <= TailleTotale_;
}
```

Cet appel à la fonction `Redimensionner()` présente deux problèmes :

- Il y a, d'une part, un risque d'appel récursif à l'infini lorsque la condition (`TailleUtilisee_ > 0.9*TailleTotale_`) est vérifiée : en effet, la première chose que fait la fonction `Redimensionner()` est d'appeler la fonction `VerifierTaille()`, pour laquelle la condition en question sera toujours vérifiée, ce qui va appeler à nouveau `Redimensionner()`, etc.
- Il y a, d'autre part, une différence de comportement gênante entre le mode *debug* et les autres modes, du fait de l'utilisation de `assert()` dans le modèle de fonction globale `VerifierTaille()` comme unique moyen de remontée d'erreurs. Comme nous l'avons déjà mentionné plus haut, il aurait fallu prévoir un système de remontée d'erreurs fonctionnant quel que soit le mode de compilation, *debug* ou non. C'est particulièrement important dans le cas d'une fonction comme `Redimensionner()` qui risque fortement de ne pas être appelée aux mêmes moments d'une exécution à l'autre, étant donné qu'elle dépend de l'utilisation qui est faite de `Tableau` par le code client.

```
private:
  T*     buffer_;
  size_t TailleUtilisee_, TailleTotale_;
};

int f( int& x, int y = x ) { return x += y; }
```

Cette déclaration de fonction est illégale et un compilateur conforme à la norme C++ devrait normalement la rejeter. Nous considérerons, dans la suite, que la valeur par défaut de `y` est `1`.

```
int g( int& x )            { return x /= 2; }

void main( int, char*[] )
{
  int i = 42;
    cout << "f(" << i << ") = " << f(i) << ", "
         << "g(" << i << ") = " << g(i) << endl;
```

Là encore, voici un exemple de code dont l'exécution dépend de l'ordre d'évaluation des paramètres : il n'est pas possible de prévoir l'ordre dans lequel seront appelées les fonctions `f()` et `g()`, ni la valeur de `i` aux moments où on tentera de l'afficher. Un compilateur bien connu, produit, par exemple, le résultat suivant : « `f(22)=22, g(21)=21` »).

Ce n'est pas le compilateur qui est en tort, mais bien l'auteur de ces lignes.

Erreur à éviter

N'écrivez jamais du code dont l'exécution dépend de l'ordre d'évaluation des paramètres d'une fonction.

```
  Tableau<char> a(20);
  cout << a.AfficherTailles() << endl;
}
```

Ceci devrait normalement afficher « `Taille totale = 20, taille utilisée = 0` ». Malheureusement, il se peut que le résultat soit différent, du fait des problèmes de la fonction `AfficherTailles()`, détaillés plus haut.

En conclusion, une bonne connaissance des mécanismes de flot d'exécution (notamment, de ce qui est indéterminé) est une condition nécessaire pour l'écriture de bons programmes C++.

Post-scriptum

Si les problèmes présentés dans cet ouvrage vous ont plu, sachez que vous pourrez régulièrement retrouver des articles similaires sur le groupe de discussion Internet `comp.lang.c++.moderated` et consulter les numéros passés de *Guru Of The Week* sur http://www.peerdirect.com/resources (ce livre est inspiré des problèmes n° 1 à 30 ; à la date de septembre 2000, nous en sommes au problème n° 73).

Voici un petit aperçu des sujets traités dernièrement :

- Une étude approfondie de l'utilisation d'`auto_ptr`, des espaces de nommage et de la gestion des exceptions, dans la lignée des éléments déjà étudiés dans les problèmes n° 8 à 17, 31 à 34 et 37 de ce livre)
- Une série de trois problèmes relatifs au comptage de référence, notamment dans le contexte d'applications multi-thread, abordant des sujets rarement discutés auparavant.
- De nombreux problèmes relatifs à la bibliothèque standard, notamment à l'utilisation et l'extensibilité des conteneurs (`vector`, `map`) et des flux d'entrée-sorties, dans la lignée du problème n° 3 de ce livre.
- Un jeu de Master-Mind écrit avec le minimum d'instructions possible.

Et ce ne sont là que quelques exemples parmi d'autres.

Par ailleurs, sachez qu'il est dans mes projets d'en préparer une suite, inspirée des nouveaux problèmes parus sur *Guru Of The Week* et des divers articles et éditoriaux que j'écris pour *C/C++ Users Journal* et d'autres magazines.

J'espère que ce livre vous a plu et que vous continuerez à me faire part de vos desiderata en matière de sujets traités. Sachez au passage que plusieurs des sujets abordés dans ce livre l'ont été suite à des demandes de lecteurs parvenues par le biais du site Internet cité plus haut.

Merci à tous ceux qui m'ont exprimé leur intérêt et leur soutien au sujet de *Guru Of The Week* et de ce livre. Puisse-t-il être votre compagnon au quotidien dans la réalisation de programmes plus fiables, plus robustes et plus performants.

Bibliographie

Barton94 • John Barton et Lee Nackman, *Scientific and Engineering C++*. Addison-Wesley, 1994.

Cargill92 • Tom Cargill, *C++ Programming Style.* Addison-Wesley, 1992.

Cargill94 • Tom Cargill, "Exception Handling: A False Sense of Security." *C++ Report*, 9(6), Nov.–Dec. 1994.

Cline95 • Marshall Cline and Greg Lomow, *C++ FAQs*. Addison-Wesley, 1995.

Cline99 • Marshall Cline, Greg Lomow et Mike Girou, *C++ FAQs, Second Edition.* Addison Wesley Longman, 1999.

Coplien92 • James Coplien, *Advanced C++ Programming Styles and Idioms*. Addison- Wesley, 1992.

Ellis90 • Margaret Ellis et Bjarne Stroustrup, *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

Gamma95 • Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Keffer95 • Thomas Keffer, *Rogue Wave C++ Design, Implementation, and Style Guide.* Rogue Wave Software, 1995.

Koenig97 • Andrew Koenig et Barbara Moo, *Ruminations on C++*. Addison Wesley Longman, 1997.

Lakos96 • John Lakos, *Large-Scale C++ Software Design*. Addison-Wesley, 1996.

Lippman97 • Stan Lippman (ed.), *C++ Gems.* SIGS / Cambridge University Press, 1997.

Lippman98 • Stan Lippman et Josée Lajoie, *C++ Primer, Third Edition.* Addison Wesley Longman, 1998.

Martin95 • Robert C. Martin, *Designing Object-Oriented Applications Using the Booch Method*. Prentice-Hall, 1995.

Martin00 • Robert C. Martin (ed.), *C++ Gems II*. SIGS / Cambridge University Press, 2000.

Meyers97 • Nathan Meyers, "The Empty Base C++ Optimization." *Dr. Dobb's Journal*, Aug. 1997.

Meyers96 • Scott Meyers, *More Effective C++*. Addison-Wesley, 1996.

Meyers98 • Scott Meyers, *Effective C++, Second Edition*. Addison Wesley Longman, 1998.

Meyers99 • Scott Meyers, *Effective C++ CD*. Addison Wesley Longman, 1999. (See also http://meyerscd.awl.com)

Murray93 • Robert Murray, *C++ Strategies and Tactics*. Addison-Wesley, 1993.

Stroustrup94 • Bjarne Stroustrup, *The Design and Evolution of C++*. Addison-Wesley, 1994.

Stroustrup97 • Bjarne Stroustrup, *The C++ Progamming Language, Third Edition*. Addison Wesley Longman, 1997.

Sutter98 • Herb Sutter, "C++ State of the Union." *C++ Report*, 10(1), Jan. 1998.

Sutter98(a) • Herb Sutter, "Uses and Abuses of Inheritance, Part 1." *C++ Report*, 10(9), Oct. 1998.

Sutter99 • Herb Sutter, "Uses and Abuses of Inheritance, Part 2." *C++ Report*, 11(1), Jan. 1999.

Index