Herb Sutter

http://bibliolivre.com

Mieux programmer en C++

47 problèmes pratiques résolus

Eyrolles

Mieux programmer en C++

47 problèmes pratiques résolus

Télécharger la version complète
Sur http://bibliolivre.com

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

Télécharger la version complète
Sur http://bibliolivre.com

Traduit de l'anglais par Thomas Pétillon

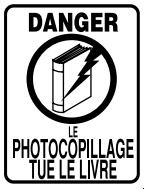
Eyrolles

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



Le code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée notamment dans les établissements d'enseignement, provoquant une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans autorisation de l'Éditeur ou du Centre Français d'exploitation du droit de Copie, 20, rue des Grands Augustins, 75006 Paris.
©Addison-Wesley Longman, a Pearson Education Company, 2000, pour l'édition en langue anglaise
© Éditions Eyrolles, 2000 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.

Télécharger la version complète
Sur http://bibliolivre.com

Télécharger la version complète
Sur http://bibliolivre.com

Sommaire

Télécharger la version complète
Sur http://bibliolivre.com

Télécharger la version complète
Sur http://bibliolivre.com

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

© copyright Éditions Eyrolles

Télécharger la version complète
Sur http://bibliolivre.com

Télécharger la version complète
Sur http://bibliolivre.com

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)

© copyright Éditions Eyrolles

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

© copyright Éditions Eyrolles

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* ».

© copyright Éditions Eyrolles

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.

© copyright Éditions Eyrolles

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";
```


© copyright Éditions Eyrolles

```
  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.

© copyright Éditions Eyrolles

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-

© copyright Éditions Eyrolles

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++.

© copyright Éditions Eyrolles

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;
   }
  };
```

© copyright Éditions Eyrolles

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*`.

© copyright Éditions Eyrolles

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>` ?*

© copyright Éditions Eyrolles

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).

© copyright Éditions Eyrolles

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; }
```

© copyright Éditions Eyrolles

```
  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>
```

© copyright Éditions Eyrolles

```
  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.

© copyright Éditions Eyrolles

```
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.

© copyright Éditions Eyrolles

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
```

© copyright Éditions Eyrolles

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
```

© copyright Éditions Eyrolles

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.

© copyright Éditions Eyrolles

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

© copyright Éditions Eyrolles

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),
```

© copyright Éditions Eyrolles

```
            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

© copyright Éditions Eyrolles

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 )
```

© copyright Éditions Eyrolles

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.

© copyright Éditions Eyrolles

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 “”
```

© copyright Éditions Eyrolles

```
  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)

© copyright Éditions Eyrolles

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.

© copyright Éditions Eyrolles

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 ?

© copyright Éditions Eyrolles

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 "";
}
```

© copyright Éditions Eyrolles

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.

© copyright Éditions Eyrolles

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();
```

© copyright Éditions Eyrolles

```
  /*...*/
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-

© copyright Éditions Eyrolles

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.

© copyright Éditions Eyrolles

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.

© copyright Éditions Eyrolles

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.

© copyright Éditions Eyrolles

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].

© copyright Éditions Eyrolles

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=()`.

Télécharger la version complète
Sur http://bibliolivre.com

© copyright Éditions Eyrolles