18

Généralités sur la bibliothèque standard

Comme celle du C, la norme du C++ comprend la définition d'une bibliothèque standard. Bien entendu, on y trouve toutes les fonctions prévues dans les versions C++ d'avant la norme, qu'il s'agisse des flots décrits précédemment ou des fonctions de la bibliothèque standard du C. Mais, on y découvre surtout bon nombre de nouveautés originales. La plupart d'entre elles sont constituées de patrons de classes et de fonctions provenant en majorité d'une bibliothèque du domaine public, nommée *Standard Template Library* (en abrégé STL) et développée chez Hewlett Packard.

L'objectif de ce chapitre est de vous familiariser avec les notions de base concernant l'utilisation des principaux composants de cette bibliothèque, à savoir : les conteneurs, les itérateurs, les algorithmes, les générateurs d'opérateurs, les prédicats et l'utilisation d'une relation d'ordre.

1 Notions de conteneur, d'itérateur et d'algorithme

Ces trois notions sont étroitement liées et, la plupart du temps, elles interviennent simultanément dans un programme utilisant des conteneurs.

1.1 Notion de conteneur

La bibliothèque standard fournit un ensemble de classes dites conteneurs, permettant de représenter les structures de données les plus répandues telles que les vecteurs, les listes, les ensembles ou les tableaux associatifs. Il s'agit de patrons de classes paramétrés tout naturellement par le type de leurs éléments. Par exemple, on pourra construire une liste d'entiers, un vecteur de flottants ou une liste de points (*point* étant une classe) par les déclarations suivantes :

```
list <int>       li ;   /* liste vide d'éléments de type int      */
vector <double>  ld ;   /* vecteur vide d'éléments de type double */
list <point>     lp ;   /* liste vide d'éléments de type point    */
```

Chacune de ces classes conteneur dispose de fonctionnalités appropriées dont on pourrait penser, *a priori*, qu'elles sont très différentes d'un conteneur à l'autre. En réalité, les concepteurs de STL ont fait un gros effort d'homogénéisation et beaucoup de fonctions membres sont communes à différents conteneurs. On peut dire que, dès qu'une action donnée est réalisable avec deux conteneurs différents, elle se programme de la même manière.

Remarque

En toute rigueur, les patrons de conteneurs sont paramétrés à la fois par le type de leurs éléments et par une fonction dite allocateur utilisée pour les allocations et les libérations de mémoire. Ce second paramètre possède une valeur par défaut qui est généralement satisfaisante. Cependant, certaines implémentations n'acceptent pas encore les paramètres par défaut dans les patrons de classes et, dans ce cas, il est nécessaire de préciser l'allocateur à utiliser, même s'il s'agit de celui par défaut. Il faut alors savoir que ce dernier est une fonction patron, de nom *allocator*, paramétrée par le type des éléments concernés. Voici ce que deviendraient les déclarations précédentes dans un tel cas :

```
list <int, allocator<int> > li ;          /* ne pas oublier l'espace     */
vector <double, allocator<double> > ld ;  /* entre int> et > ; sinon, >> */
list <point, allocator<point> > lp ;      /* représentera l'opérateur >> */
```

1.2 Notion d'itérateur

C'est dans ce souci d'homogénéisation des actions sur un conteneur qu'a été introduite la notion d'itérateur. Un itérateur est un objet défini généralement par la classe conteneur concernée qui généralise la notion de pointeur :

- à un instant donné, un itérateur possède une valeur qui désigne un élément donné d'un conteneur ; on dira souvent qu'un itérateur pointe sur un élément d'un conteneur ;
- un itérateur peut être incrémenté par l'opérateur ++, de manière à pointer sur l'élément suivant du même conteneur ; on notera que ceci n'est possible que, comme on le verra plus loin, parce que les conteneurs sont toujours ordonnés suivant une certaine séquence ;

- un itérateur peut être déréférencé, comme un pointeur, en utilisant l'opérateur * ; par exemple, si *it* est un itérateur sur une liste de points, **it* désigne un point de cette liste ;
- deux itérateurs sur un même conteneur peuvent être comparés par égalité ou inégalité.

Tous les conteneurs fournissent un itérateur portant le nom *iterator* et possédant au minimum les propriétés que nous venons d'énumérer qui correspondent à ce qu'on nomme un itérateur unidirectionnel. Certains itérateurs pourront posséder des propriétés supplémentaires, en particulier :

- décrémentation par l'opérateur -- ; comme cette possibilité s'ajoute alors à celle qui est offerte par ++, l'itérateur est alors dit bidirectionnel ;
- accès direct ; dans ce cas, si *it* est un tel itérateur, l'expression *it+i* a un sens ; souvent, l'opérateur [] est alors défini, de manière que *it[i]* soit équivalent à **(it+i)* ; en outre, un tel itérateur peut être comparé par inégalité.

Remarque

Ici, nous avons évoqué trois catégories d'itérateur : unidirectionnel, bidirectionnel et accès direct. Au chapitre 21, nous verrons qu'il existe deux autres catégories (entrée et sortie) qui sont d'un usage plus limité. De même, on verra qu'il existe ce qu'on appelle des adaptateurs d'itérateur, lesquels permettent d'en modifier les propriétés ; les plus importants seront l'itérateur de flux et l'itérateur d'insertion.

1.3 Parcours d'un conteneur avec un itérateur

1.3.1 Parcours direct

Tous les conteneurs fournissent des valeurs particulières de type *iterator*, sous forme des fonctions membres *begin()* et *end()*, de sorte que, quel que soit le conteneur, le canevas suivant, présenté ici sur une liste de points, est toujours utilisable pour parcourir séquentiellement un conteneur de son début jusqu'à sa fin :

```
list<point> lp ;
   .....
list<point>::iterator il ;     /* itérateur sur une liste de points */
for (il = lp.begin() ; il != lp.end() ; il++)
{
   /* ici *il désigne l'élément courant de la liste de points lp */
}
```

On notera la particularité des valeurs des itérateurs de fin qui consiste à pointer, non pas sur le dernier élément d'un conteneur, mais juste après. D'ailleurs, lorsqu'un conteneur est vide, *begin()* possède la même valeur que *end()*, de sorte que le canevas précédent fonctionne toujours convenablement.

Remarque

Attention, on ne peut pas utiliser comme condition d'arrêt de la boucle *for*, une expression telle que *il* < *lp.end*, car l'opérateur < ne peut s'appliquer qu'à des itérateurs à accès direct.

1.3.2 Parcours inverse

Toutes les classes conteneur pour lesquelles *iterator* est au moins bidirectionnel (on peut donc lui appliquer ++ et --) disposent d'un second itérateur noté *reverse_iterator*. Construit à partir du premier, il permet d'explorer le conteneur suivant l'ordre inverse. Dans ce cas, la signification de ++ et --, appliqués à cet itérateur, est alors adaptée en conséquence ; en outre, il existe également des valeurs particulières de type *reverse_iterator* fournies par les fonctions membres *rbegin()* et *rend()* ; on peut dire que *rbegin()* pointe sur le dernier élément du conteneur, tandis que *rend()* pointe juste avant le premier. Voici comment parcourir une liste de points dans l'ordre inverse :

```
list<point> lp ;
   .....
list<point>::reverse_iterator ril ;  /* itérateur inverse sur */
                                     /* une liste de points   */
for (ril = lp.rbegin() ; ril != lp.rend() ; ril++)
{
   /* ici *ril désigne l'élément courant de la liste de points lp */
}
```

1.4 Intervalle d'itérateur

Comme nous l'avons déjà fait remarquer, tous les conteneurs sont ordonnés, de sorte qu'on peut toujours les parcourir d'un début jusqu'à une fin. Plus généralement, on peut définir ce qu'on nomme un *intervalle d'itérateur* en précisant les bornes sous forme de deux valeurs d'itérateur. Supposons que l'on ait déclaré :

```
vector<point>::iterator ip1, ip2 ;  /* ip1 et ip2 sont des itérateurs sur */
                                    /* un  vecteur de points             */
```

Supposons, de plus, que *ip1* et *ip2* possèdent des valeurs telles que *ip2* soit "accessible" depuis *ip1*, autrement dit que, après un certain nombre d'incrémentations de *ip1* par ++, on obtienne la valeur de *ip2*. Dans ces conditions, le couple de valeurs *ip1, ip2* définit un intervalle d'un conteneur du type *vector<point>* s'étendant de l'élément pointé par *ip1* jusqu'à (mais non compris) celui pointé par *ip2*. Cet intervalle se note souvent [*ip1, ip2*). On dit également que les éléments désignés par cet intervalle forment une séquence.

Cette notion d'intervalle d'itérateur sera très utilisée par les algorithmes et par certaines fonctions membres.

1.5 Notion d'algorithme

La notion d'algorithme est tout aussi originale que les deux précédentes. Elle se fonde sur le fait que, par le biais d'un itérateur, beaucoup d'opérations peuvent être appliquées à un conteneur, quels que soient sa nature et le type de ses éléments. Par exemple, on pourra trouver le premier élément ayant une valeur donnée aussi bien dans une liste, un vecteur ou ensemble ; il faudra cependant que l'égalité de deux éléments soit convenablement définie, soit par défaut, soit par surdéfinition de l'opérateur ==. De même, on pourra trier un conteneur d'objets de type *T*, pour peu que ce conteneur dispose d'un itérateur à accès direct et que l'on ait défini une relation d'ordre sur le type *T*, par exemple en surdéfinissant l'opérateur <.

Les différents algorithmes sont fournis sous forme de patrons de fonctions, paramétrés par le type des itérateurs qui leurs sont fournis en argument. Là encore, cela conduit à des programmes très homogènes puisque les mêmes fonctions pourront être appliquées à des conteneurs différents. Par exemple, pour compter le nombre d'éléments égaux à un dans un vecteur déclaré par :

```
vector<int> v ;    /* vecteur d'entiers */
```

on pourra procéder ainsi :

```
n = count (v.begin(), v.end(), 1) ; /* compte le nombre d'éléments valant 1 */
                                    /* dans la séquence [v.begin(), v.end()) */
                                    /* autrement dit, dans tout le conteneur v */
```

Pour compter le nombre d'éléments égaux à un dans une liste déclarée :

```
list<int> l ;    /* liste d'entiers */
```

on procédera de façon similaire (en se contentant de remplacer *v* par *l*) :

```
n = count (l.begin(), l.end(), 1) ; /* compte le nombre d'éléments valant 1 */
                                    /* dans la séquence [l.begin(), l.end())   */
                                    /* autrement dit, dans tout le conteneur l */
```

D'une manière générale, comme le laissent entendre ces deux exemples, les algorithmes s'appliquent, non pas à un conteneur, mais à une séquence définie par un intervalle d'itérateur ; ici, cette séquence correspondait à l'intégralité du conteneur.

Certains algorithmes permettront facilement de recopier des informations d'un conteneur d'un type donné vers un conteneur d'un autre type, pour peu que ses éléments soient du même type que ceux du premier conteneur. Voici, par exemple, comment recopier un vecteur d'entiers dans une liste d'entiers :

```
vector<int> v ;    /* vecteur d'entiers */
list<int> l ;      /* liste d'entiers   */
   .....
copy (v.begin(), v.end(), l.begin() ) ;
         /* recopie l'intervalle [v.begin(), v.end()),     */
         /* à partir de l'emplacement pointé par l.begin() */
```

Notez que, si l'on fournit l'intervalle de départ, on ne mentionne que le début de celui d'arrivée.

Remarque

On pourra parfois être gêné par le fait que l'homogénéisation évoquée n'est pas absolue. Ainsi, on verra qu'il existe un algorithme de recherche d'une valeur donnée nommé *find*, alors même qu'un conteneur comme *list* dispose d'une fonction membre comparable. La justification résidera dans des considérations d'efficacité.

1.6 Itérateurs et pointeurs

La manière dont les algorithmes ou les fonctions membres utilisent un itérateur fait que tout objet ou toute variable possédant les propriétés attendues (déréférenciation, incrémentation...) peut être utilisé à la place d'un objet tel que *iterator*.

Or, les pointeurs usuels possèdent tout naturellement les propriétés d'un itérateur à accès direct. Cela leur permet d'être employés dans bon nombre d'algorithmes. Cette possibilité est fréquemment utilisée pour la recopie des éléments d'un tableau ordinaire dans un conteneur :

```
int t[6] = { 2, 9, 1, 8, 2, 11 } ;
list<int> l ;
   .....
copy (t, t+6, l.begin()) ;   /* copie de l'intervalle [t, t+6) dans la liste l */
```

Bien entendu, ici, il n'est pas question d'utiliser une notation telle que *t.begin()* qui n'aurait aucun sens, *t* n'étant pas un objet.

Remarque

Par souci de simplicité, nous parlerons encore de séquence d'éléments (mais plus de séquence de conteneur) pour désigner les éléments ainsi définis par un intervalle de pointeurs.

2 Les différentes sortes de conteneurs

2.1 Conteneurs et structures de données classiques

On dit souvent que les conteneurs correspondent à des structures de données usuelles. Mais, à partir du moment où ces conteneurs sont des classes qui encapsulent convenablement leurs données, leurs caractéristiques doivent être indépendantes de leur implémentation. Dans ces conditions, les différents conteneurs devraient se distinguer les uns des autres uniquement par leurs fonctionnalités et en aucun cas par les structures de données sous-jacentes. Beaucoup de conteneurs posséderaient alors des fonctionnalités voisines, voire identiques.

En réalité, les différents conteneurs se caractérisent, outre leurs fonctionnalités, par l'efficacité de certaines opérations. Par exemple, on verra qu'un vecteur permet des insertions d'élé-

ments en n'importe quel point mais celles-ci sont moins efficaces qu'avec une liste. En revanche, on peut accéder plus rapidement à un élément existant dans le cas d'un vecteur que dans celui d'une liste. Ainsi, bien que la norme n'impose pas l'implémentation des conteneurs, elle introduit des contraintes d'efficacité qui la conditionneront largement.

En définitive, on peut dire que le nom choisi pour un conteneur évoque la structure de donnée classique qui en est proche sur le plan des fonctionnalités, sans pour autant coïncider avec elle. Dans ces conditions, un bon usage des différents conteneurs passe par un apprentissage de leurs possibilités, comme s'il s'agissait bel et bien de classes différentes.

2.2 Les différentes catégories de conteneurs

La norme classe les différents conteneurs en deux catégories :

- les conteneurs en séquence (ou conteneurs séquentiels),
- les conteneurs associatifs.

La notion de conteneur en séquence correspond à des éléments qui sont ordonnés comme ceux d'un vecteur ou d'une liste. On peut parcourir le conteneur suivant cet ordre. Quand on insère ou qu'on supprime un élément, on le fait en un endroit qu'on a explicitement choisi.

La notion de conteneur associatif peut être illustrée par un répertoire téléphonique. Dans ce cas, on associe une valeur (numéro de téléphone, adresse...) à ce qu'on nomme une clé (ici le nom). A partir de la clé, on accède à la valeur associée. Pour insérer un nouvel élément dans ce conteneur, il ne sera théoriquement plus utile de préciser un emplacement.

Il semble donc qu'un conteneur associatif ne soit plus ordonné. En fait, pour d'évidentes questions d'efficacité, un tel conteneur devra être ordonné mais, cette fois, de façon intrinsèque, c'est-à-dire suivant un ordre qui n'est plus défini par le programme. La principale conséquence est qu'il restera toujours possible de parcourir séquentiellement les éléments d'un tel conteneur qui disposera toujours au moins d'un itérateur nommé *iterator* et des valeurs *begin()* et *end()*. Cet aspect peut d'ailleurs prêter à confusion, dans la mesure où certaines opérations prévues pour des conteneurs séquentiels pourront s'appliquer à des conteneurs associatifs, tandis que d'autres poseront problème. Par exemple, il n'y aura aucun risque à examiner séquentiellement chacun des éléments d'un conteneur associatif ; il y en aura manifestement, en revanche, si l'on cherche à modifier séquentiellement les valeurs d'éléments existants, puisqu'alors, on risque de perturber l'ordre intrinsèque du conteneur. Nous y reviendrons le moment venu.

3 Les conteneurs dont les éléments sont des objets

Le patron de classe définissant un conteneur peut être appliqué à n'importe quel type et donc, en particulier à des éléments de type classe. Dans ce cas, il ne faut pas perdre de vue que bon

nombre de manipulations de ces éléments vont entraîner des appels automatiques de certaines fonctions membres.

3.1 Construction, copie et affectation

Toute **construction d'un conteneur**, non vide, dont les éléments sont des objets, entraîne, **pour chacun de ces éléments** :

- soit l'appel d'un constructeur ; il peut s'agir d'un constructeur par défaut lorsqu'aucun argument n'est nécessaire ;
- soit l'appel d'un constructeur par recopie.

Par exemple, on verra que la déclaration suivante (*point* étant une classe) construit un vecteur de trois éléments de type *point* :

```
vector<point> v(3) ;  /* construction d'un vecteur de 3 points */
```

Pour chacun des trois éléments, il y aura appel d'un constructeur sans argument de *point*. Si l'on construit un autre vecteur, à partir de *v* :

```
vector<point> w (v) ;  /* ou vector v = w ; */
```

il y aura appel du constructeur par recopie de la classe *vector<point>*, lequel appellera le constructeur par recopie de la classe *point* pour chacun des trois éléments de type *point* à recopier.

On pourrait s'attendre à des choses comparables avec **l'opérateur d'affectation** dans un cas tel que :

```
w = v ;      /* le vecteur v est affecté à w */
```

Cependant, ici, les choses sont un peu moins simples. En effet, généralement, si la taille de *w* est suffisante, on se contentera effectivement d'appeler l'opérateur d'affectation pour tous les éléments de *v* (on appellera le destructeur pour les éléments excédentaires de *w*). En revanche, si la taille de *w* est insuffisante, il y aura destruction de tous ses éléments et création d'un nouveau vecteur par appel du constructeur par recopie, lequel appellera tout naturellement le constructeur par recopie de la classe *point* pour tous les éléments de *v*.

Par ailleurs, il ne faudra pas perdre de vue que, par défaut, la transmission d'un conteneur en argument d'une fonction se fait par valeur, ce qui entraîne la recopie de tous ses éléments.

Les trois circonstances que nous venons d'évoquer concernent des opérations portant sur l'ensemble d'un conteneur. Mais il va de soi qu'il existe également d'autres opérations portant sur un élément d'un conteneur et qui, elles aussi, feront appel au constructeur de recopie (insertion) ou à l'affectation.

D'une manière générale, si les objets concernés ne possèdent pas de partie dynamique, les fonctions membres prévues par défaut seront satisfaisantes. Dans le cas contraire, il faudra prévoir les fonctions appropriées, ce qui sera bien sûr le cas si la classe concernée respecte le

schéma de classe canonique proposé au paragraphe 4 du chapitre 9 (et complété au paragraphe 8 du chapitre 13). Notez bien que :

> Dès qu'une classe est destinée à donner naissance à des objets susceptibles d'être introduits dans des conteneurs, il n'est plus possible d'en désactiver la recopie et/ou l'affectation.

Remarque

Dans les descriptions des différents conteneurs ou algorithmes, nous ne rappellerons pas ces différents points, dans la mesure où ils concernent systématiquement tous les objets.

3.2 Autres opérations

Il existe d'autres opérations que les constructions ou recopies de conteneur qui peuvent entraîner des appels automatiques de certaines fonctions membres.

L'un des exemples les plus évidents est celui de la recherche d'un élément de valeur donnée, comme le fait la fonction membre *find* du conteneur *list*. Dans ce cas, la classe concernée devra manifestement disposer de l'opérateur ==, lequel, cette fois, ne possède plus de version par défaut.

Un autre exemple réside dans les possibilités dites de "comparaisons lexicographiques" que nous examinerons au chapitre 19 ; nous verrons que celles-ci se fondent sur la comparaison, par l'un des opérateurs <, >, <= ou >= des différents éléments du conteneur. Manifestement, là encore, il faudra définir au moins l'opérateur < pour la classe concernée : les possibilités de génération automatique présentées ci-dessus pourront éviter les définitions des trois autres.

D'une manière générale, cette fois, compte tenu de l'aspect épisodique de ce type de besoin, nous le préciserons chaque fois que ce sera nécessaire.

4 Efficacité des opérations sur des conteneurs

Pour juger de l'efficacité d'une fonction membre d'un conteneur ou d'un algorithme appliqué à un conteneur, on choisit généralement la notation dite "de Landau" (*O(...)*) qui se définit ainsi :

> Le temps t d'une opération est dit O(x) s'il existe une constante k telle que, dans tous les cas, on ait : t <= kx.

Comme on peut s'y attendre, le nombre *N* d'éléments d'un conteneur (ou d'une séquence de conteneur) pourra intervenir. C'est ainsi qu'on rencontrera typiquement :

- des opérations en *O(1)*, c'est-à-dire pour lesquelles le temps est constant (plutôt borné par une constante, indépendante du nombre d'éléments de la séquence) ; on verra que ce sera le cas des insertions dans une liste ou des insertions en fin de vecteur ;
- des opérations en *O(N)*, c'est-à-dire pour lesquelles le temps est proportionnel au nombre d'éléments de la séquence ; on verra que ce sera le cas des insertions en un point quelconque d'un vecteur ;
- des opérations en *O(LogN)*...

D'une manière générale, on ne perdra pas de vue qu'une telle information n'a qu'un caractère relativement indicatif ; pour être précis, il faudrait indiquer s'il s'agit d'un maximum ou d'une moyenne et mentionner la nature des opérations concernées. C'est d'ailleurs ce que nous ferons dans l'annexe C décrivant l'ensemble des algorithmes standard.

5 Fonctions, prédicats et classes fonctions

5.1 Fonction unaire

Beaucoup d'algorithmes et quelques fonctions membres permettent d'appliquer une fonction donnée aux différents éléments d'une séquence (définie par un intervalle d'itérateur). Cette fonction est alors passée simplement en argument de l'algorithme, comme dans :

```
for_each(it1, it2, f) ;  /* applique la fonction f à chacun des éléments */
                         /* de  la séquence [it1, it2)                   */
```

Bien entendu, la fonction *f* doit posséder un argument du type des éléments correspondants (dans le cas contraire, on obtiendrait une erreur de compilation). Il n'est pas interdit qu'une telle fonction possède une valeur de retour mais, quoi qu'il en soit, elle ne sera pas utilisée.

Voici un exemple montrant comment utiliser cette technique pour afficher tous les éléments d'une liste :

```
main()
{ list<float> lf ;
  void affiche (float) ;
    .....
  for_each (lf.begin(), lf.end(), affiche) ; cout << "\n" ;
    .....
}
void affiche (float x) { cout << x << " " ; }
```

Bien entendu, on obtiendrait le même résultat en procédant simplement ainsi :

```
main()
{ list<float> lf ;
  void affiche (list<float>) ;
    .....
  lf.affiche() ;
    .....
}
```

```
void affiche (list<float> l)
{ list<float>::iterator il ;
  for (il=l.begin() ; il!=l.end() ; il++) cout << (*il) << " " ;
  cout << "\n" ;
}
```

5.2 Prédicats

On parle de prédicat pour caractériser une fonction qui renvoie une valeur de type *bool*. Compte tenu des conversions implicites qui sont mises en place automatiquement, cette valeur peut éventuellement être entière, sachant qu'alors 0 correspondra à *false* et que tout autre valeur correspondra à *true*.

On rencontrera des prédicats unaires, c'est-à-dire disposant d'un seul argument et des prédicats binaires, c'est-à-dire disposant de deux arguments de même type.

Là encore, certains algorithmes et certaines fonctions membres nécessiteront qu'on leur fournisse un prédicat en argument. Par exemple, l'algorithme *find_if* permet de trouver le premier élément d'une séquence vérifiant un prédicat passé en argument :

```
main()
{ list<int> l ;
  list<int>::iterator il ;
  bool impair (int) ;
    .....
  il = find_if (l.begin(), l.end(), impair) ;  /* il désigne le premier */
    .....                /* élément de  l vérifiant le prédicat impair  */
}
bool impair (int n)             /* définition du prédicat unaire impair */
{ return n%2 ; }
```

5.3 Classes et objets fonctions

5.3.1 Utilisation d'objet fonction comme fonction de rappel

Nous venons de voir que certains algorithmes ou fonctions membres nécessitaient un prédicat en argument. D'une manière générale, ils peuvent nécessiter une fonction quelconque et l'on parle souvent de "fonction de rappel" pour évoquer un tel mécanisme dans lequel une fonction est amenée à appeler une autre fonction qu'on lui a transmise en argument.

La plupart du temps, cette fonction de rappel est prévue dans la définition du patron correspondant, non pas sous forme d'une fonction, mais bel et bien sous forme d'un objet de type quelconque. Les classes et les objets fonction ont été présentés au paragraphe 6 du chapitre 9 et nous en avions alors donné un exemple simple d'utilisation. En voici un autre qui montre l'intérêt qu'ils présentent dans le cas de patrons de fonctions. Ici, le patron de fonction *essai* définit une famille de fonctions recevant en argument une fonction de rappel sous forme d'un objet fonction *f* de type quelconque. Les exemples d'appels de la fonction *essai* montrent qu'on peut lui fournir, indifféremment comme fonction de rappel, soit une fonction usuelle, soit un objet fonction.

```
#include <iostream>
using namespace std ;
class cl_fonc             /* definition d'une classe fonction */
{ int coef ;
  public :
  cl_fonc(int n) {coef = n ;}
  int operator () (int p) {return coef*p ; }
} ;
int fct (int n)           /* definition d'une fonction usuelle */
{ return 5*n ;
}
template <class T>void essai (T f)    // définition de essai qui reçoit en
{ cout << "f(1) : " << f(1) << "\n" ; // argument un objet de type quelconque
  cout << "f(4) : " << f(4) << "\n" ; // et qui l'utilise comme une fonction
}
main()
{ essai (fct) ;        // appel essai, avec une fonction de rappel usuelle
  essai (cl_fonc(3)) ; // appel essai, avec une fonction de rappel objet
  essai (cl_fonc(7)) ; // idem
}
```

```
f(1) : 5
f(4) : 20
f(1) : 3
f(4) : 12
f(1) : 7
f(4) : 28
```

Exemple d'utilisation d'objets fonctions

On voit qu'un algorithme attendant un objet fonction peut recevoir une fonction usuelle. En revanche, on notera que la réciproque est fausse. C'est pourquoi, tous les algorithmes ont prévu leurs fonctions de rappel sous forme d'objets fonction.

5.3.2 Classes fonction prédéfinies

Dans <*functional*>, il existe un certain nombre de patrons de classes fonction correspondant à des prédicats binaires de comparaison de deux éléments de même type. Par exemple, *less*<*int*> instancie une fonction patron correspondant à la comparaison par < (*less*) de deux éléments de type *int*. Comme on peut s'y attendre, *less*<*point*> instanciera une fonction patron correspondant à la comparaison de deux objets de type *point* par l'opérateur <, qui devra alors être convenablement défini dans la classe *point*.

Voici les différents noms de patrons existants et les opérateurs correspondants : *equal_to* (==), *not_equal_to* (!=), *greater* (>), *less* (<), *greater_equal* (>=), *less_equal* (<=).

Toutes ces classes fonction disposent d'un constructeur sans argument, ce qui leur permet d'être citées comme fonction de rappel. D'autre part, elles seront également utilisées comme argument de type dans la construction de certaines classes.

Remarque

Il existe également des classes fonction correspondant aux opérations binaires usuelles, par exemple *plus*<*int*> pour la somme de deux *int*. Voici les différents noms des autres patrons existants et les opérateurs correspondants : *modulus* (%), *minus* (-), *times* (*), *divides* (/). On trouve également les prédicats correspondant aux opérations logiques : *logical_and* (&&), *logical_or* (||), *logical_not* (!). Ces classes sont cependant d'un usage moins fréquent que celles qui ont été étudiées précédemment.

6 Conteneurs, algorithmes et relation d'ordre

6.1 Introduction

Un certain nombre de situations nécessiteront la connaissance d'une relation permettant d'ordonner les différents éléments d'un conteneur. Citons-en quelques exemples :

- pour des questions d'efficacité, comme il a déjà été dit, les éléments d'un conteneur associatif seront ordonnés en permanence ;
- un conteneur *list* disposera d'une fonction membre *sort*, permettant de réarranger ses éléments suivant un certain ordre ;
- il existe beaucoup d'algorithmes de tri qui, eux aussi, réorganisent les éléments d'un conteneur suivant un certain ordre.

Bien entendu, tant que les éléments du conteneur concerné sont d'un type scalaire ou *string*, pour lequel il existe une relation naturelle (<) permettant d'ordonner les éléments, on peut se permettre d'appliquer ces différentes opérations d'ordonnancement, sans trop se poser de questions.

En revanche, si les éléments concernés sont d'un type classe qui ne dispose pas par défaut de l'opérateur <, il faudra surdéfinir convenablement cet opérateur. Dans ce cas, et comme on peut s'y attendre, cet opérateur devra respecter un certain nombre de propriétés, nécessaires au bon fonctionnement de la fonction ou de l'algorithme utilisé.

Par ailleurs, et quel que soit le type des éléments (classe, type de base...), on peut choisir d'utiliser une relation autre que celle qui correspond à l'opérateur < (par défaut ou surdéfini) :

- soit en choisissant un autre opérateur (par défaut ou surdéfini),
- soit en fournissant explicitement une fonction de comparaison de deux éléments.

Là encore, cet opérateur ou cette fonction devra respecter les propriétés évoquées que nous allons examiner maintenant.

6.2 Propriétés à respecter

Pour simplifier les notations, nous noterons toujours *R*, la relation binaire en question, qu'elle soit définie par un opérateur ou par une fonction. La norme précise que R *doit être une relation d'ordre faible strict*, laquelle se définit ainsi :

- ∀a, !(a *R* a)
- *R* est transitive, c'est-à-dire que ∀a, b, c, tels que : a *R* b et b *R* c, alors a *R* c ;
- ∀a, b, c, tels que : !(a R b) et !(b R c), alors !(a R c).

On notera que l'égalité n'a pas besoin d'être définie pour que *R* respecte les propriétés requises.

Bien entendu, on peut sans problème utiliser les opérateurs < et > pour les types numériques ; on prendra garde, cependant, à ne pas utiliser <= ou >= qui ne répondent pas à la définition.

On peut montrer que ces contraintes définissent une relation d'ordre total, non pas sur l'ensemble des éléments concernés, mais simplement sur les classes d'équivalence induites par la relation *R*, une classe d'équivalence étant telle que a et b appartiennent à la même classe si l'on a à la fois !(a *R* b) et !(b *R* a). A titre d'exemple, considérons des éléments d'un type classe (*point*), possédant deux coordonnées *x* et *y* ; supposons qu'on y définisse la relation *R* par :

p1(x1, y1) *R* p2(x2, y2) si x1 < x2

On peut montrer que *R* satisfait les contraintes requises et que les classes d'équivalence sont formées des points ayant la même abscisse.

Dans ces conditions, si l'on utilise *R* pour trier un conteneur de points, ceux-ci apparaîtront ordonnés suivant la première coordonnée. Cela n'est pas très grave car, dans une telle opération de tri, tous les points seront conservés. En revanche, si l'on utilise cette même relation *R* pour ordonner intrinsèquement un conteneur associatif de type *map* (dont on verra que deux éléments ne peuvent avoir de clés équivalentes), deux points de même abscisse apparaîtront comme "identiques" et un seul sera conservé dans le conteneur.

Ainsi, lorsqu'on sera amené à définir sa propre relation d'ordre, il faudra bien être en mesure d'en prévoir correctement les conséquences au niveau des opérations qui en dépendront. Notamment, dans certains cas, il faudra savoir si l'égalité de deux éléments se fonde sur l'opérateur == (surdéfini ou non), ou sur les classes d'équivalence induites par une relation d'ordre (par défaut, il s'agira alors de <, surdéfini ou non). Par exemple, l'algorithme *find* se fonde sur ==, tandis que la fonction membre *find* d'un conteneur associatif se fonde sur l'ordre intrinsèque du conteneur. Bien entendu, aucune différence n'apparaîtra avec des éléments de type numérique ou *string*, tant qu'on se limitera à l'ordre induit par < puisqu'alors les classes d'équivalence en question seront réduites à un seul élément.

Bien entendu, nous attirerons à nouveau votre attention sur ce point au moment voulu.

7 Les générateurs d'opérateurs

N.B. Ce paragraphe peut être ignoré dans un premier temps.

Le mécanisme de surdéfinition d'opérateurs utilisé par C++ fait que l'on peut théoriquement définir, pour une classe donnée, à la fois l'opérateur == et l'opérateur !=, de manière totalement indépendante, voir incohérente. Il en va de même pour les opérateurs <, <=, > et >=.

Mais la bibliothèque standard dispose de patrons de fonctions permettant de définir :

- l'opérateur !=, à partir de l'opérateur ==
- les opérateurs >, <= et >=, à partir de l'opérateur <.

Comme on peut s'y attendre, si *a* et *b* sont d'un type classe pour laquelle on a défini ==, != sera défini par :

a != b si !(a == b)

De la même manière, les opérateurs <=, > et >= peuvent être déduits de < par les définitions suivantes :

a > b si b < a

a <= b si ! (a > b)

a >= b si ! (a < b)

Dans ces conditions, on voit qu'il suffit de munir une classe des opérateurs == et < pour qu'elle dispose automatiquement des autres.

Bien entendu, il reste toujours possible de donner sa propre définition de l'un quelconque de ces quatre opérateurs. Elle sera alors utilisée, en tant que spécialisation d'une fonction patron.

Il est très important de noter qu'il n'existe aucun lien entre la définition automatique de <= et celle de ==. Ainsi, rien n'impose, hormis le bon sens, que a==b implique a<=b, comme le montre ce petit exemple d'école, dans lequel nous définissons l'opérateur < d'une manière artificielle et incohérente avec la définition de == :

```
#include <iostream>
#include <utility>          // pour les générateurs d'opérateurs
using namespace std ;
class point
{ int x, y ;
 public :
  point(int abs=0, int ord=0) { x=abs ; y=ord ; }
  friend int operator== (point, point) ;
  friend int operator< (point, point) ;
} ;

int operator== (point a, point b)
{ return ( (a.x == b.x) && (a.y == b.y) ) ;
}
```

```
int operator<(point a, point b)
{ return ( (a.x < b.x) && (a.y < b.y) ) ;
}
main()
{ point a(1, 2), b(3, 1) ;
  cout << "a == b : " << (a==b) << "    a != b : " << (a!=b) << "\n" ;
  cout << "a < b  : " << (a<b)  << "    a <= b : " << (a<=b) << "\n" ;
  char c ; cin >> c ;
}
```

```
a == b : 0    a != b : 1
a < b  : 0    a <= b : 1
```

Exemple de génération non satisfaisante des opérateurs !=, >, <= et >=

Remarque

Le manque de cohérence entre les définitions des opérateurs == et < est ici sans conséquence. En revanche, nous avons vu que l'opérateur < pouvait intervenir, par exemple, pour ordonner un conteneur associatif ou pour trier un conteneur de type *list* lorsqu'on utilise la fonction membre *sort*. Dans ce cas, sa définition devra respecter les contraintes évoquées au paragraphe 6.