# Chapitre 5. Les conteneurs, itérateurs et foncteurs

La plupart des programmes informatiques doivent, à un moment donné ou à un autre, conserver un nombre arbitraire de données en mémoire, généralement pour y accéder ultérieurement et leur appliquer des traitements spécifiques. En général, les structures de données utilisées sont toujours manipulées par des algorithmes classiques, que l'on retrouve donc souvent, si ce n'est plusieurs fois, dans chaque programme. Ces structures de données sont communément appelées des conteneurs en raison de leur capacité à contenir d'autres objets.

Afin d'éviter aux programmeurs de réinventer systématiquement la roue et de reprogrammer les structures de données et leurs algorithmes associés les plus classiques, la bibliothèque standard définit un certain nombre de classes template pour les conteneurs les plus courants. Ces classes sont paramétrées par le type des données des conteneurs et peuvent donc être utilisées virtuellement pour toutes les situations qui se présentent.

Les conteneurs de la bibliothèque standard ne sont pas définis par les algorithmes qu'ils utilisent, mais plutôt par l'interface qui peut être utilisée par les programmes clients. La bibliothèque standard impose également des contraintes de performances sur ces interfaces en termes de complexité. En réalité, ces contraintes sont tout simplement les plus fortes qui soient, ce qui garantit aux programmes qui les utilisent qu'ils auront les meilleures performances possibles.

La bibliothèque classifie les conteneurs en deux grandes catégories selon leurs fonctionnalités : les séquences et les conteneurs associatifs. Une séquence est un conteneur capable de stocker ses éléments de manière séquentielle, les uns à la suite des autres. Les éléments sont donc parfaitement identifiés par leur position dans la séquence, et leur ordre relatif est donc important. Les conteneurs associatifs, en revanche, manipulent leurs données au moyen de valeurs qui les identifient indirectement. Ces identifiants sont appelées des clefs par analogie avec la terminologie utilisée dans les bases de données. L'ordre relatif des éléments dans le conteneur est laissé dans ce cas à la libre discrétion de ce dernier et leur recherche se fait donc, généralement, par l'intermédiaire de leurs clefs.

La bibliothèque fournit plusieurs conteneurs de chaque type. Chacun a ses avantages et ses inconvénients. Comme il n'existe pas de structure de données parfaite qui permette d'obtenir les meilleures performances sur l'ensemble des opérations réalisables, l'utilisateur des conteneurs de la bibliothèque standard devra effectuer son choix en fonction de l'utilisation qu'il désire en faire. Par exemple, certains conteneurs sont plus adaptés à la recherche d'éléments mais sont relativement coûteux pour les opérations d'insertion ou de suppression, alors que pour d'autres conteneurs, c'est exactement l'inverse. Le choix des conteneurs à utiliser sera donc déterminant quant aux performances finales des programmes.

## 1. Fonctionnalités générales des conteneurs

Au niveau de leurs interfaces, tous les conteneurs de la bibliothèque standard présentent des similitudes. Cet état de fait n'est pas dû au hasard, mais bel et bien à la volonté de simplifier la vie des programmeurs en évitant de définir une multitude de méthodes ayant la même signification pour chaque conteneur. Cependant, malgré cette volonté d'uniformisation, il existe des différences entre les différents types de conteneurs (séquences ou conteneurs

associatifs). Ces différences proviennent essentiellement de la présence d'une clef dans ces derniers, qui permet de manipuler les objets contenus plus facilement.

Quelle que soit leur nature, les conteneurs fournissent un certain nombre de services de base que le programmeur peut utiliser. Ces services comprennent la définition des itérateurs, de quelques types complémentaires, des opérateurs et de fonctions standards. Les sections suivantes vous présentent ces fonctionnalités générales. Toutefois, les descriptions données ici ne seront pas détaillées outre mesure car elles seront reprises en détail dans la description de chaque conteneur.

### 1.1.Définition des itérateurs

Pour commencer, il va de soi que tous les conteneurs de la bibliothèque standard disposent d'itérateurs. Comme on l'a vu dans la Section 13.4, les itérateurs constituent une abstraction de la notion de pointeur pour les tableaux. Ils permettent donc de parcourir tous les éléments d'un conteneur séquentiellement à l'aide de l'opérateur de déréférencement * et de l'opérateur d'incrémentation ++. Les conteneurs définissent donc tous un type iterator et un type const_iterator, qui sont les types des itérateurs sur les éléments du conteneur. Le type d'itérateur const_iterator est défini pour accéder aux éléments d'un conteneur en les considérant comme des constantes. Ainsi, si le type des éléments stockés dans le conteneur est T, le déréférencement d'un const_iterator renverra un objet de type const T. Les conteneurs définissent également les types de données difference_type et size_type que l'on peut utiliser pour effectuer des calculs d'arithmétique des pointeurs avec leurs itérateurs. Le type difference_type se distingue du type size_type par le fait qu'il peut contenir toute valeur issue de la différence entre deux itérateurs, et accepte donc les valeurs négatives. Le type size_type quant à lui est utilisé plus spécialement pour compter un nombre d'éléments, et ne peut prendre que des valeurs positives. Afin de permettre l'initialisation de leurs itérateurs, les conteneurs fournissent deux méthodes begin et end, qui renvoient respectivement un itérateur référençant le premier élément du conteneur et la valeur de fin de l'itérateur, lorsqu'il a passé le dernier élément du conteneur. Ainsi, le parcours d'un conteneur se fait typiquement de la manière suivante :

```
// Obtient un itérateur sur le premier élément :
Conteneur::iterateur i = instance.begin();
// Boucle sur toutes les valeurs de l'itérateur
// jusqu'à la dernière :
while (i != instance.end())
{
    // Travaille sur l'élément référencé par i :
    f(*i);
    // Passe à l'élément suivant :
    ++i;
}
```

où Conteneur est la classe de du conteneur et instance en est une instance.

**Note :** Pour des raisons de performances et de portabilité, la bibliothèque standard ne fournit absolument aucun support du multithreading sur ses structures de données. En fait, la gestion du multithreading est laissée à la discrétion de chaque implémentation. Généralement, seul le code généré par le compilateur est sûr vis-à-vis des threads (en particulier, les opérateurs

d'allocation mémoire new et new[], ainsi que les opérateurs delete et delete[] peuvent être appelés simultanément par plusieurs threads pour des objets différents). Il n'en est pas de même pour les implémentations des conteneurs et des algorithmes de la bibliothèque standard.

Par conséquent, si vous voulez accéder à un conteneur à partir de plusieurs threads, vous devez prendre en charge vous-même la gestion des sections critiques afin de vous assurer que ce conteneur sera toujours dans un état cohérent. En fait, il est recommandé de le faire même si l'implémentation de la bibliothèque standard se protège elle-même contre les accès concurrents à partir de plusieurs threads, afin de rendre vos programmes portables vers d'autres environnements.

Les itérateurs utilisés par les conteneurs sont tous au moins du type ForwardIterator. En pratique, cela signifie que l'on peut parcourir les itérateurs du premier au dernier élément, séquentiellement. Cependant, la plupart des conteneurs disposent d'itérateurs au moins bidirectionnels, et peuvent donc être parcourus dans les deux sens. Les conteneurs qui disposent de ces propriétés sont appelés des conteneurs réversibles.

Les conteneurs réversibles disposent, en plus des itérateurs directs, d'itérateurs inverses. Ces itérateurs sont repectivement de type reverse_iterator et const_reverse_iterator. Leur initialisation peut être réalisée à l'aide de la fonction rbegin, et leur valeur de fin peut être récupérée à l'aide de la fonction rend.

### 1.2.Définition des types de données relatifs aux objets contenus

Outre les types d'itérateurs, les conteneurs définissent également des types spécifiques aux données qu'ils contiennent. Ces types de données permettent de manipuler les données des conteneurs de manière générique, sans avoir de connaissance précises sur la nature réelle des objets qu'ils stockent. Ils sont donc couramment utilisés par les algorithmes de la bibliothèque standard.

Le type réellement utilisé pour stocker les objets dans un conteneur n'est pas toujours le type template utilisé pour instancier ce conteneur. En effet, certains conteneurs associatifs stockent les clefs des objets avec la valeur des objets eux-mêmes. Ils utilisent pour cela la classe pair, qui permet de stocker, comme on l'a vu en Section 14.2.2, des couples de valeurs. Le type des données stockées par ces conteneurs est donc plus complexe que le simple type template par lequel ils sont paramétrés.

Afin de permettre l'uniformisation des algorithmes travaillant sur ces types de données, les conteneurs définissent tous le type value_type dans leur classe template. C'est en particulier ce type qu'il faut utiliser lors des insertions d'éléments dans les conteneurs. Bien entendu, pour la plupart des conteneurs, et pour toutes les séquences, le type value_type est effectivement le même type que le type template par lequel les conteneurs sont paramétrés.

Les conteneurs définissent également d'autres types permettant de manipuler les données qu'ils stockent. En particulier, le type reference est le type des références sur les données, et le type const_reference est le type des références constantes sur les données. Ces types sont utilisés par les méthodes des conteneurs qui permettent d'accéder à leurs données.

### 1.3.Spécification de l'allocateur mémoire à utiliser

Toutes les classes template des conteneurs de la bibliothèque standard utilisent la notion

d'allocateur pour réaliser les opérations de manipulation de la mémoire qu'elles doivent effectuer lors du stockage de leurs éléments ou lors de l'application d'algorithmes spécifiques au conteneur. Le type des allocateurs peut être spécifié dans la liste des paramètres template des conteneurs, en marge du type des données contenues. Les constructeurs des conteneurs prennent tous un paramètre de ce type, qui sera l'allocateur mémoire utilisé pour ce conteneur. Ainsi, il est possible de spécifier un allocateur spécifique pour chaque conteneur, qui peut être particulièrement optimisé pour le type des données gérées par ce conteneur.
Toutefois, le paramètre template spécifiant la classe de l'allocateur mémoire à utiliser dispose d'une valeur par défaut, qui représente l'allocateur standard de la bibliothèque allocator<T>. Il n'est donc pas nécessaire de spécifier cet allocateur lors de l'instanciation d'un conteneur. Cela rend plus simple l'utilisation de la bibliothèque standard C++ pour ceux qui ne désirent pas développer eux-même un allocateur mémoire. Par exemple, la déclaration template du conteneur list est la suivante :

```
template <class T, class Allocator = allocator<T> >
```

Il est donc possible d'instancier une liste d'entiers simplement en ne spécifiant que le type des objets contenus, en l'occurrence, des entiers :

```
typedef list<int> liste_entier;
```

De même, le paramètre des constructeurs permettant de spécifier l'allocateur à utiliser pour les conteneurs dispose systématiquement d'une valeur par défaut, qui est l'instance vide du type d'allocateur spécifié dans la liste des paramètres template. Par exemple, la déclaration du constructeur le plus simple de la classe list est la suivante :

```
template <class T, class Allocator>
list<T, Allocator>::list(const Allocator & = Allocator());
```

Il est donc parfaitement légal de déclarer une liste d'entier simplement de la manière suivante :

```
liste_entier li;
```

**Note :** Il est peut-être bon de rappeler que toutes les instances d'un allocateur accèdent à la même mémoire. Ainsi, il n'est pas nécessaire, en général, de préciser l'instance de l'allocateur dans le constructeur des conteneurs. En effet, le paramètre par défaut fourni par la bibliothèque standard n'est qu'une instance parmi d'autres qui permet d'accéder à la mémoire gérée par la classe de l'allocateur fournie dans la liste des paramètres template.
Si vous désirez spécifier une classe d'allocateur différente de celle de l'allocateur standard, vous devrez faire en sorte que cette classe implémente toutes les méthodes des allocateurs de la bibliothèque standard.

### 1.4.Opérateurs de comparaison des conteneurs

Les conteneurs disposent d'opérateurs de comparaison permettant d'établir des relations d'équivalence ou des relations d'ordre entre eux. Les conteneurs peuvent tous être comparés directement avec les opérateurs == et !=. La relation d'égalité entre deux conteneurs est définie par le respect des deux propriétés suivantes : les deux conteneurs doivent avoir la même taille ;leurs éléments doivent être identiques deux à deux.
Si le type des objets contenus dispose des opérateurs d'infériorité et de supériorités strictes, les mêmes opérateurs seront également définis pour le conteneur. Ces opérateurs utilisent l'ordre lexicographique pour déterminer le classement entre deux conteneurs. Autrement dit, l'opérateur d'infériorité compare les éléments des deux conteneurs un à un, et fixe son verdict dès la première différence constatée. Si un conteneur est un sous-ensemble du deuxième, le

conteneur le plus petit est celui qui est inclus dans l'autre.

**Note :** Remarquez que la définition des opérateurs de comparaison d'infériorité et de supériorité existe quel que soit le type des données que le conteneur peut stocker. Cependant, comme les conteneurs sont définis sous la forme de classes template, ces méthodes ne sont instanciées que si elles sont effectivement utilisées dans les programmes. Ainsi, il est possible d'utiliser les conteneurs même sur des types de données pour lesquels les opérateurs d'infériorité et de supériorité ne sont pas définis. Cependant, cette utilisation provoquera une erreur de compilation, car le compilateur cherchera à instancier les opérateurs à ce moment.

### 1.5. Méthodes d'intérêt général

Enfin, les conteneurs disposent de méthodes générales permettant d'obtenir des informations sur leurs propriétés. En particulier, le nombre d'éléments qu'ils contiennent peut être déterminé grâce à la méthode size. La méthode empty permet de déterminer si un conteneur est vide ou non. La taille maximale que peut prendre un conteneur est indiquée quant à elle par la méthode max_size. Pour finir, tous les conteneurs disposent d'une méthode swap, qui prend en paramètre un autre conteneur du même type et qui réalise l'échange des données des deux conteneurs. On utilisera de préférence cette méthode à toute autre technique d'échange car seules les références sur les structures de données des conteneurs sont échangées avec cette fonction, ce qui garantit une complexité indépendante de la taille des conteneurs.

## 2. Les séquences

Les séquences sont des conteneurs qui ont principalement pour but de stocker des objets afin de les traiter dans un ordre bien défini. Du fait de l'absence de clef permettant d'identifier les objets qu'elles contiennent, elles ne disposent d'aucune fonction de recherche des objets. Les séquences disposent donc généralement que des méthodes permettant de réaliser l'insertion et la suppression d'éléments, ainsi que le parcours des éléments dans l'ordre qu'elles utilisent pour les classer.

### 2.1. Fonctionnalités communes

Il existe un grand nombre de classes template de séquences dans la bibliothèque standard qui permettent de couvrir la majorité des besoins des programmeurs. Ces classes sont relativement variées tant dans leurs implémentations que dans leurs interfaces. Cependant, un certain nombre de fonctionnalités communes sont gérées par la plupart des séquences. Ce sont ces fonctionnalités que cette section se propose de vous décrire.

Les exemples fournis dans cette section se baseront sur le conteneur list, qui est le type de séquence le plus simple de la bibliothèque standard. Cependant, ils sont parfaitement utilisables avec les autres types de séquences de la bibliothèque standard, avec des niveaux de performances éventuellement différents en fonction des séquences choisies bien entendu.

### 2.2. Construction et initialisation

La construction et l'initialisation d'une séquence peuvent se faire de multiples manières. Les séquences disposent en effet de plusieurs constructeurs et de deux surcharges de la méthode assign qui permet de leur affecter un certain nombre d'éléments. Le constructeur le plus simple ne prend aucun paramètre, hormis un allocateur standard à utiliser pour la gestion de la séquence, et permet de construire une séquence vide. Le deuxième constructeur prend en paramètre le nombre d'éléments initial de la séquence et la valeur de ces éléments. Ce constructeur permet donc de créer une séquence contenant déjà un certain nombre de copies

d'un objet donné. Enfin, le troisième constructeur prend deux itérateurs sur une autre séquence d'objets qui devront être copiés dans la séquence en cours de construction. Ce constructeur peut être utilisé pour initialiser une séquence à partir d'une autre séquence ou d'un sous-ensemble de séquence.

Les surcharges de la méthode assign se comportent un peu comme les deux derniers constructeurs, à ceci près qu'elles ne prennent pas d'allocateur en paramètre. La première méthode permet donc de réinitialiser la liste et de la remplir avec un certain nombre de copies d'un objet donné, et la deuxième permet de réinitialiser la liste et de la remplir avec une séquence d'objets définie par deux itérateurs.

Exemple 1. Construction et initialisation d'une liste

```
#include <iostream>
#include <list>

using namespace std;

typedef list<int> li;

void print(li &l)
{
    li::iterator i = l.begin();
    while (i != l.end())
    {
        cout << *i << " " ;
        ++i;
    }
    cout << endl;
}

int main(void)
{
    // Initialise une liste avec trois éléments valant 5 :
    li l1(3, 5);
    print(l1);
    // Initialise une autre liste à partir de la première
    // (en fait on devrait appeler le constructeur de copie) :
    li l2(l1.begin(), l1.end());
    print(l2);
    // Affecte 4 éléments valant 2 à l1 :
    l1.assign(4, 2);
    print(l1);
    // Affecte l1 à l2 (de même, on devrait normalement
    // utiliser l'opérateur d'affectation) :
    l2.assign(l1.begin(), l1.end());
    print(l2);
```

Bien entendu, il existe également un constructeur et un opérateur de copie capables d'initialiser une séquence à partir d'une autre séquence du même type. Ainsi, il n'est pas nécessaire d'utiliser les constructeurs vus précédemment ni les méthodes assign pour

initialiser une séquence à partir d'une autre séquence de même type.

#### 2.2.1.1. Ajout et suppression d'éléments

L'insertion de nouveaux éléments dans une séquence se fait normalement à l'aide de l'une des surcharges de la méthode insert. Bien entendu, il existe d'autres méthodes spécifiques à chaque conteneur de type séquence et qui leur sont plus appropriées, mais ces méthodes ne seront décrites que dans les sections consacrées à ces conteneurs. Les différentes versions de la méthode insert sont récapitulées ci-dessous :

```
iterator insert(iterator i, value_type valeur)
```

Permet d'insérer une copie de la valeur spécifiée en deuxième paramètre dans le conteneur. Le premier paramètre est un itérateur indiquant l'endroit où le nouvel élément doit être inséré. L'insertion se fait immédiatement avant l'élément référencé par cet itérateur. Cette méthode renvoie un itérateur sur le dernier élément inséré dans la séquence.

```
void insert(iterator i, size_type n, value_type valeur)
```

Permet d'insérer n copies de l'élément spécifié en troisième paramètre avant l'élément référencé par l'itérateur i donné en premier paramètre.

```
void insert(iterator i, iterator premier, iterator dernier)
```

Permet d'insérer tous les éléments de l'intervalle défini par les itérateurs premier et dernier

avant l'élément référencé par l'itérateur

Exemple -2. Insertion d'éléments dans une liste

```
#include <iostream>
#include <list>

using namespace std;

typedef list<int> li;

void print(li &l)
{
    li::iterator i = l.begin();
    while (i != l.end())
    {
        cout << *i << " " ;
        ++i;
    }
    cout << endl;
    return ;
}

int main(void)
{
    li l1;
    // Ajoute 5 à la liste :
    li::iterator i = l1.insert(l1.begin(), 5);
    print(l1);
    // Ajoute deux 3 à la liste :
    l1.insert(i, 2, 3);
    print(l1);
    // Insère le contenu de l1 dans une autre liste :
    li l2;
    l2.insert(l2.begin(), l1.begin(), l1.end());
    print(l2);
```

```
#include <iostream>
#include <list>

using namespace std;

typedef list<int> li;

void print(li &l)
{
    li::iterator i = l.begin();
    while (i != l.end())
    {
        cout << *i << " " ;
        ++i;
    }
    cout << endl;
    return ;
}

int main(void)
{
    li l1;
    // Ajoute 5 à la liste :
    li::iterator i = l1.insert(l1.begin(), 5);
    print(l1);
    // Ajoute deux 3 à la liste :
    l1.insert(i, 2, 3);
    print(l1);
    // Insère le contenu de l1 dans une autre liste :
    li l2;
    l2.insert(l2.begin(), l1.begin(), l1.end());
    print(l2);
```

De manière similaire, il existe deux surcharges de la méthode erase qui permettent de spécifier de différentes manières les éléments qui doivent être supprimés d'une séquence. La première méthode prend en paramètre un itérateur sur l'élément à supprimer, et la deuxième un couple d'itérateurs donnant l'intervalle des éléments de la séquence qui doivent être supprimés. Ces deux méthodes retournent un itérateur sur l'élément suivant le dernier élément supprimé ou l'itérateur de fin de séquence s'il n'existe pas de tel élément. Par exemple, la suppression de tous les éléments d'une liste peut être réalisée de la manière suivante :

```
// Récupère un itérateur sur le premier
// élément de la liste :
list<int>::iterator i = instance.begin();
while (i != instance.end())
{
    i = instance.erase(i);
}
```

où instance est une instance de la séquence Sequence.

Vous noterez que la suppression d'un élément dans une séquence rend invalide tous les itérateurs sur cet élément. Il est à la charge du programmeur de s'assurer qu'il n'utilisera plus les itérateurs ainsi invalidés. La bibliothèque standard ne fournit aucun support pour le diagnostic de ce genre d'erreur.

**Note :** En réalité, l'insertion d'un élément peut également invalider des itérateurs existants pour certaines séquences. Les effets de bord des méthodes d'insertion et de suppression des séquences seront détaillés pour chacune d'elle dans les sections qui leur sont dédiées.

Il existe une méthode clear dont le rôle est de vider complètement un conteneur. On utilisera donc cette méthode dans la pratique, le code donné ci-dessous ne l'était qu'à titre d'exemple

La complexité de toutes ces méthodes dépend directement du type de séquence sur lequel elles sont appliquées.

### 2.3.Les différents types de séquences

La bibliothèque standard fournit trois classes fondamentales de séquence. Ces trois classes sont respectivement la classe list, la classe vector et la classe deque. Chacune de ces classes possède ses spécificités en fonction desquelles le choix du programmeur devra se faire. De plus, la bibliothèque standard fournit également des classes adaptatrices permettant de construire des conteneurs équivalents, mais disposant d'une interface plus standard et plus habituelle aux notions couramment utilisées en informatique. Toutes ces classes sont décrites dans cette section, les adaptateurs étant abordés en dernière partie.

### 2.4.Les listes

La classe template list est certainement l'une des plus importantes car, comme son nom l'indique, elle implémente une structure de liste chaînée d'éléments, ce qui est sans doute l'une des structures les plus utilisées en informatique. Cette structure est particulièrement adaptée pour les algorithmes qui parcourent les données dans un ordre séquentiel.

Les propriétés fondamentales des listes sont les suivantes :

- elles implémentent des itérateurs bidirectionnels. Cela signifie qu'il est facile de passer d'un élément au suivant ou au précédent, mais qu'il n'est pas possible d'accéder aux éléments de la liste de manière aléatoire ;
- elles permettent l'insertion et la suppression d'un élément avec un coût constant, et sans invalider les itérateurs ou les références sur les éléments de la liste existants. Dans le cas d'une suppression, seuls les itérateurs et les références sur les éléments supprimés sont invalidés.

Les listes offrent donc la plus grande souplesse possible sur les opérations d'insertion et de suppression des éléments, en contrepartie de quoi les accès sont restreints à un accès séquentiel.

Comme l'insertion et la suppression des éléments en tête et en queue de liste peuvent se faire sans recherche, ce sont évidemment les opérations les plus courantes. Par conséquent, la classe template list propose des méthodes spécifiques permettant de manipuler les éléments qui se trouvent en ces positions. L'insertion d'un élément peut donc être réalisée respectivement en tête et en queue de liste avec les méthodes push_front et push_back. Inversement, la suppression des éléments situés en ces emplacements est réalisée avec les méthodes pop_front et pop_back. Toutes ces méthodes ne renvoient aucune valeur, aussi l'accès aux deux éléments situés en tête et en queue de liste peut-il être réalisé respectivement par l'intermédiaire des accesseurs front et back, qui renvoient tous deux une référence (éventuellement constante si la liste est elle-même constante) sur ces éléments.

Exemple -3. Accès à la tête et à la queue d'une liste

```
#include <iostream>
#include <list>

using namespace std;

typedef list<int> li;

int main(void)
{
    li l1;
    l1.push_back(2);
    l1.push_back(5);
    cout << "Tête : " << l1.front() << endl;
    cout << "Queue : " << l1.back() << endl;
    l1.push_front(7);
    cout << "Tête : " << l1.front() << endl;
    cout << "Queue : " << l1.back() << endl;
    l1.pop_back();
    cout << "Tête : " << l1.front() << endl;
    cout << "Queue : " << l1.back() << endl;
    return 0;
}
```

Les listes disposent également de méthodes spécifiques qui permettent de leur appliquer des traitements qui leur sont propres. Ces méthodes sont décrites dans le tableau ci-dessous :

**Tableau -1. Méthodes spécifiques aux listes**

| Méthode | Fonction |
|---|---|
| remove(const T &) | Permet d'éliminer tous les éléments d'une liste dont la valeur est égale à la valeur passée en paramètre. L'ordre relatif des éléments qui ne sont pas supprimés est inchangé. La complexité de cette méthode est linéaire en fonction du nombre d'éléments de la liste. |
| remove_if(Predicat) | Permet d'éliminer tous les éléments d'une liste qui vérifient le prédicat unaire passé en paramètre. L'ordre relatif des éléments qui ne sont pas supprimés est inchangé. La complexité de cette méthode est linéaire en fonction du nombre d'éléments de la liste. |
| unique(Predicat) | Permet d'éliminer tous les éléments pour lesquels le prédicat binaire passé en paramètre est vérifié avec comme valeur l'élément courant et son prédécesseur. Cette méthode permet d'éliminer les doublons successifs dans une liste selon un critère défini par le prédicat. Par souci de simplicité, il existe une surcharge de cette méthode qui ne prend pas de paramètres, et qui utilise un simple test d'égalité pour éliminer les doublons.<br>L'ordre relatif des éléments qui ne sont pas supprimés est inchangé, et le nombre d'applications du prédicat est exactement le nombre d'éléments de la liste moins un si la liste n'est pas vide. |

| | |
|---|---|
| splice(iterator position, list<T, Allocator> liste, iterator premier, iterateur dernier) | Injecte le contenu de la liste fournie en deuxième paramètre dans la liste courante à partir de la position fournie en premier paramètre. Les éléments injectés sont les éléments de la liste source identifiés par les itérateurs premier et dernier. Ils sont supprimés de la liste source à la volée. Cette méthode dispose de deux autres surcharges, l'une ne fournissant pas d'itérateur de dernier élément et qui insère uniquement le premier élément, et l'autre ne fournissant aucun itérateur pour référencer les éléments à injecter. Cette dernière surcharge ne prend donc en paramètre que la position à laquelle les éléments doivent être insérés et la liste source elle-même. Dans ce cas, la totalité de la liste source est insérée en cet emplacement. Généralement, la complexité des méthodes splice est proportionnelle au nombre d'éléments injectés, sauf dans le cas de la dernière surcharge, qui s'exécute avec une complexité constante. |
| sort(Predicat) | Trie les éléments de la liste dans l'ordre défini par le prédicat binaire de comparaison passé en paramètre. Encore une fois, il existe une surcharge de cette méthode qui ne prend pas de paramètre et qui utilise l'opérateur d'infériorité pour comparer les éléments de la liste entre eux. L'ordre relatif des éléments équivalents (c'est-à-dire des éléments pour lesquels le prédicat de comparaison n'a pas pu statuer d'ordre bien défini) est inchangé à l'issue de l'opération de tri. On indique souvent cette propriété en disant que cette méthode est stable. La méthode sort s'applique avec une complexité égale à N×ln(N), où N est le nombre d'éléments de la liste. |

## 2.5. Les vecteurs

La classe template vector de la bibliothèque standard fournit une structure de données dont la sémantique est proche de celle des tableaux de données classiques du langage C/C++. L'accès aux données de manière aléatoire est donc réalisable en un coût constant, mais l'insertion et la suppression des éléments dans un vecteur ont des conséquences nettement plus lourdes que dans le cas des listes.

Les propriétés des vecteurs sont les suivantes :

- les itérateurs permettent les accès aléatoires aux éléments du vecteur ;
- l'insertion ou la suppression d'un élément à la fin du vecteur se fait avec une complexité constante, mais l'insertion ou la suppression en tout autre point du vecteur se fait avec une complexité linéaire. Autrement dit, les opérations d'insertion ou de suppression nécessitent a priori de déplacer tous les éléments suivants, sauf si l'élément inséré ou supprimé se trouve en dernière position ;
- dans tous les cas, l'insertion d'un élément peut nécessiter une réallocation de mémoire. Cela a pour conséquence qu'en général, les données du vecteur peuvent être déplacées en mémoire et que les itérateurs et les références sur les éléments d'un vecteur sont a priori invalidés à la suite d'une insertion. Cependant, si aucune réallocation n'a lieu, les itérateurs et les références ne sont pas invalidés pour tous les éléments situés avant l'élément inséré ;
- la suppression d'un élément ne provoquant pas de réallocation, seuls les itérateurs et les références sur les éléments suivant l'élément supprimé sont invalidés.

**Note :** Notez bien que les vecteurs peuvent effectuer une réallocation même lorsque l'insertion se fait en dernière position. Dans ce cas, le coût de l'insertion est bien entendu très élevé. Toutefois, l'algorithme de réallocation utilisé est suffisament évolué pour garantir que ce coût est constant en moyenne (donc de complexité constante). Autrement dit, les réallocations ne se font que très rarement.
Tout comme la classe list, la classe template vector dispose de méthodes front et back qui permettent d'accéder respectivement au premier et au dernier élément des vecteurs. Cependant, contrairement aux listes, seule les méthodes push_back et pop_back sont définies, car les vecteurs ne permettent pas d'insérer et de supprimer leurs premiers éléments de manière rapide.

En revanche, comme nous l'avons déjà dit, les vecteurs ont la même sémantique que les tableaux et permettent donc un accès rapide à tous leurs éléments. La classe vector définit donc une méthode at qui prend en paramètre l'indice d'un élément dans le vecteur et qui renvoie une référence, éventuellement constante si le vecteur l'est lui-même, sur cet élément. Si l'indice fourni en paramètre référence un élément situé en dehors du vecteur, la méthode at lance une exception out_of_range. De même, il est possible d'appliquer l'opérateur [] utilisé habituellement pour accéder aux éléments des tableaux. Cet opérateur se comporte exactement comme la méthode at, et est donc susceptible de lancer une exception out_of_range.

```
#include <iostream>
#include <vector>

using namespace std;

int main(void)
{
    typedef vector<int> vi;
    // Crée un vecteur de 10 éléments :
    vi v(10);
    // Modifie quelques éléments :
    v.at(2) = 2;
    v.at(5) = 7;
    // Redimensionne le vecteur :
    v.resize(11);
    v.at(10) = 5;
    // Ajoute un élément à la fin du vecteur :
    v.push_back(13);
    // Affiche le vecteur en utilisant l'opérateur [] :
    for (int i=0; i<v.size(); ++i)
    {
        cout << v[i] << endl;
    }
    return 0;
```

Par ailleurs, la bibliothèque standard définit une spécialisation de la classe template vector pour le type bool. Cette spécialisation a essentiellement pour but de réduire la consommation mémoire des vecteurs de booléens, en codant ceux-ci à raison d'un bit par booléen seulement. Les références des éléments des vecteurs de booléens ne sont donc pas réellement des booléens, mais plutôt une classe spéciale qui simule ces booléens tout en manipulant les bits

réellement stockés dans ces vecteurs. Ce mécanisme est donc complètement transparent pour l'utilisateur, et les vecteurs de booléens se manipulent exactement comme les vecteurs classiques.

**Note :** La classe de référence des vecteurs de booléens disposent toutefois d'une méthode flip dont le rôle est d'inverser la valeur du bit correspondant au booléen que la référence représente. Cette méthode peut être pratique à utiliser lorsqu'on désire inverser rapidement la valeur d'un des éléments du vecteur.

2.5.1.1. Les deques

Pour ceux à qui les listes et les vecteurs ne conviennent pas, la bibliothèque standard fournit un conteneur plus évolué qui offre un autre compromis entre la rapidité d'accès aux éléments et la souplesse dans les opérations d'ajout ou de suppression. Il s'agit de la classe template deque, qui implémente une forme de tampon circulaire dynamique.

Les propriétés des deques sont les suivantes :

- les itérateurs des deques permettent les accès aléatoires à leurs éléments ;
- l'insertion et la suppression des éléments en première et en dernière position se fait avec un coût constant. Notez ici que ce coût est toujours le même, et que, contrairement aux vecteurs, il ne s'agit pas d'un coût amorti (autrement dit, ce n'est pas une moyenne). En revanche, tout comme pour les vecteurs, l'insertion et la suppression aux autres positions se fait avec une complexité linéaire ;
- contrairement aux vecteurs, tous les itérateurs et toutes les références sur les éléments de la deque deviennent systématiquement invalides lors d'une insertion ou d'une suppression d'élément aux autres positions que la première et la dernière ;
- de même, l'insertion d'un élément en première et dernière position invalide tous les itérateurs sur les éléments de la deque. En revanche, les références sur les éléments restent valides. Remarquez que la suppression d'un élément en première et en dernière position n'a aucun impact sur les itérateurs et les références des éléments autres que ceux qui sont supprimés.

Comme vous pouvez le constater, les deques sont donc extrêmement bien adaptés aux opérations d'insertion et de suppression en première et en dernière position, tout en fournissant un accès rapide à leurs éléments. En revanche, les itérateurs existants sont systématiquement invalidés, quel que soit le type d'opération effectuée, hormis la suppression en tête et en fin de deque.

Comme elle permet un accès rapide à tous ses éléments, la classe template deque dispose de toutes les méthodes d'insertion et de suppression d'éléments des listes et des vecteurs. Outre les méthodes push_front, pop_front, push_back, pop_back et les accesseurs front et back, la classe deque définit donc la méthode at, ainsi que l'opérateur d'accès aux éléments de tableaux []. L'utilisation de ces méthodes est strictement identique à celle des méthodes homonymes des classes list et vector et ne devrait donc pas poser de problème particulier.

## 2.6.Les adaptateurs de séquences

Les classes des séquences de base list, vector et deque sont supposées satisfaire à la plupart des besoins courants des programmeurs. Cependant, la bibliothèque standard fournit des adaptateurs pour transformer ces classes en d'autres structures de données plus classiques. Ces adaptateurs permettent de construire des piles, des files et des files de priorité.

## 2.7.Les piles

Les piles sont des structures de données qui se comportent, comme leur nom l'indique, comme un empilement d'objets. Elles ne permettent donc d'accéder qu'aux éléments situés en haut de la pile, et la récupération des éléments se fait dans l'ordre inverse de leur empilement. En raison de cette propriété, on les appelle également couramment LIFO, acronyme de l'anglais « Last In First Out » (dernier entré, premier sorti).

La classe adaptatrice définie par la bibliothèque standard C++ pour implémenter les piles est la classe template stack. Cette classe utilise deux paramètres template : le type des données lui-même et le type d'une classe de séquence implémentant au moins les méthodes back, push_back et pop_back. Il est donc parfaitement possible d'utiliser les listes, deques et vecteurs pour implémenter une pile à l'aide de cet adaptateur. Par défaut, la classe stack utilise une deque, et il n'est donc généralement pas nécessaire de spécifier le type du conteneur à utiliser pour réaliser la pile.

L'interface des piles se réduit au strict minimum, puisqu'elles ne permettent de manipuler que leur sommet. La méthode push permet d'empiler un élément sur la pile, et la méthode pop de l'en retirer. Ces deux méthodes ne renvoient rien, l'accès à l'élément situé au sommet de la pile se fait donc par l'intermédiaire de la méthode top.

Exemple -5. Utilisation d'une pile

```
#include <iostream>
#include <stack>

using namespace std;

int main(void)
{
    typedef stack<int> si;
    // Crée une pile :
    si s;
    // Empile quelques éléments :
    s.push(2);
    s.push(5);
    s.push(8);
    // Affiche les éléments en ordre inverse :
    while (!s.empty())
    {
        cout << s.top() << endl;
        s.pop();
    }
    return 0;
```

## 2.8.Les files

Les files sont des structures de données similaires aux piles, à la différence près que les éléments sont mis les uns à la suite des autres au lieu d'être empilés. Leur comportement est donc celui d'une file d'attente où tout le monde serait honnête (c'est-à-dire que personne ne doublerait les autres). Les derniers entrés sont donc ceux qui sortent également en dernier, d'où leur dénomination de FIFO (de l'anglais « First In First Out »).

Les files sont implémentées par la classe template queue. Cette classe utilise comme paramètre template le type des éléments stockés ainsi que le type d'un conteneur de type séquence pour lequel les méthodes front, back, push_back et pop_front sont implémentées. En pratique, il est possible d'utiliser les listes et les deques, la classe queue utilisant d'ailleurs ce type de séquence par défaut comme conteneur sous-jacent.

**Note :** Ne confondez pas la classe queue et la classe deque. La première n'est qu'un simple adaptateur pour les files d'éléments, alors que la deuxième est un conteneur très évolué et beaucoup plus complexe.

Les méthodes fournies par les files sont les méthodes front et back, qui permettent d'accéder respectivement au premier et au dernier élément de la file d'attente, ainsi que les méthodes push et pop, qui permettent respectivement d'ajouter un élément à la fin de la file et de supprimer l'élément qui se trouve en tête de file.

```
#include <iostream>
#include <queue>

using namespace std;

int main(void)
{
    typedef queue<int> qi;
    // Crée une file :
    qi q;
    // Ajoute quelques éléments :
    q.push(2);
    q.push(5);
    q.push(8);
    // Affiche récupère et affiche les éléments :
    while (!q.empty())
    {
        cout << q.front() << endl;
        q.pop();
    }
    return 0;
```

## 2.9.Les files de priorités

Enfin, la bibliothèque standard fournit un adaptateur permettant d'implémenter les files de priorités. Les files de priorités ressemblent aux files classiques, mais ne fonctionnent pas de la même manière. En effet, contrairement aux files normales, l'élément qui se trouve en première position n'est pas toujours le premier élément qui a été placé dans la file, mais celui qui dispose de la plus grande valeur. C'est cette propriété qui a donné son nom aux files de priorités, car la priorité d'un élément est ici donnée par sa valeur. Bien entendu, la bibliothèque standard permet à l'utilisateur de définir son propre opérateur de comparaison, afin de lui laisser spécifier l'ordre qu'il veut utiliser pour définir la priorité des éléments.

**Note :** On prendra garde au fait que la bibliothèque standard n'impose pas aux files de priorités de se comporter comme des files classiques avec les éléments de priorités égales. Cela signifie que si plusieurs éléments de priorité égale sont insérés dans une file de priorité, ils n'en sortiront pas forcément dans l'ordre d'insertion. On dit généralement que les algorithmes utilisés par les files de priorités ne sont pas stables pour traduire cette propriété.

La classe template fournie par la bibliothèque standard pour faciliter l'implémentation des files de priorité est la classe priority_queue. Cette classe prend trois paramètres template : le type des éléments stockés, le type d'un conteneur de type séquence permettant un accès direct à ses éléments et implémentant les méthodes front, push_back et pop_back, et le type d'un prédicat binaire à utiliser pour la comparaison des priorités des éléments. On peut donc implémenter une file de priorité à partir d'un vecteur ou d'une deque, sachant que, par défaut, la classe priority_queue utilise un vecteur. Le prédicat de comparaison utilisé par défaut est le foncteur less<T>, qui effectue une comparaison à l'aide de l'opérateur d'infériorité des éléments stockés dans la file.

Comme les files de priorités se réorganisent à chaque fois qu'un nouvel élément est ajouté en fin de file, et que cet élément ne se retrouve par conséquent pas forcément en dernière position s'il est de priorité élevée, accéder au dernier élément des files de priorité n'a pas de sens. Il n'existe donc qu'une seule méthode permettant d'accéder à l'élément le plus important de la pile : la méthode top. En revanche, les files de priorité implémentent effectivement les méthodes push et pop, qui permettent respectivement d'ajouter un élément dans la file de priorité et de supprimer l'élément le plus important de cette file.

```
#include <iostream>
#include <queue>

using namespace std;

// Type des données stockées dans la file :
struct A
{
    int k;         // Priorité
    const char *t; // Valeur
    A() : k(0), t(0) {}
    A(int k, const char *t) : k(k), t(t) {}
};

// Foncteur de comparaison selon les priorités :
class C
{
public:
    bool operator()(const A &a1, const A &a2)
    {
        return a1.k < a2.k ;
    }
};

int main(void)
{
    // Construit quelques objets :
    A a1(1, "Priorité faible");
    A a2(2, "Priorité moyenne 1");
    A a3(2, "Priorité moyenne 2");
    A a4(3, "Priorité haute 1");
    A a5(3, "Priorité haute 2");
    // Construit une file de priorité :
    priority_queue<A, vector<A>, C> pq;
    // Ajoute les éléments :
    pq.push(a5);
    pq.push(a3);
    pq.push(a1);
    pq.push(a2);
    pq.push(a4);
    // Récupère les éléments par ordre de priorité :
    while (!pq.empty())
    {
        cout << pq.top().t << endl;
        pq.pop();
```

**Note :** En raison de la nécessité de réorganiser l'ordre du conteneur sous-jacent à chaque ajout ou suppression d'un élément, les méthodes push et pop s'exécutent avec une complexité en ln(N), où N est le nombre d'éléments présents dans la file de priorité.

## 2.10. Les conteneurs associatifs

Contrairement aux séquences, les conteneurs associatifs sont capables d'identifier leurs éléments à l'aide de la valeur de leur clef. Grâce à ces clefs, les conteneurs associatifs sont capables d'effectuer des recherches d'éléments de manière extrêmement performante. En effet,

les opérations de recherche se font généralement avec un coût logarithmique seulement, ce qui reste généralement raisonnable même lorsque le nombre d'éléments stockés devient grand. Les conteneurs associatifs sont donc particulièrement adaptés lorsqu'on a besoin de réaliser un grand nombre d'opération de recherche.

La bibliothèque standard distingue deux types de conteneurs associatifs : les conteneurs qui différencient la valeur de la clef de la valeur de l'objet lui-même et les conteneurs qui considèrent que les objets sont leur propre clef. Les conteneurs de la première catégorie constituent ce que l'on appelle des associations car ils permettent d'associer des clefs aux valeurs des objets. Les conteneurs associatifs de la deuxième catégorie sont appelés quant à eux des ensembles, en raison du fait qu'ils servent généralement à indiquer si un objet fait partie ou non d'un ensemble d'objets. On ne s'intéresse dans ce cas pas à la valeur de l'objet, puisqu'on la connaît déjà si on dispose de sa clef, mais plutôt à son appartenance ou non à un ensemble donné.

Si tous les conteneurs associatifs utilisent la notion de clef, tous ne se comportent pas de manière identique quant à l'utilisation qu'ils en font. Pour certains conteneurs, que l'on qualifie de conteneurs « à clefs uniques », chaque élément contenu doit avoir une clef qui lui est propre. Il est donc impossible d'insérer plusieurs éléments distincts avec la même clef dans ces conteneurs. En revanche, les conteneurs associatif dits « à clefs multiples » permettent l'utilisation d'une même valeur de clef pour plusieurs objets distincts. L'opération de recherche d'un objet à partir de sa clef peut donc, dans ce cas, renvoyer plus d'un seul objet.

La bibliothèque standard fournit donc quatre types de conteneurs au total, selon que ce sont des associations ou des ensembles, et selon que ce sont des conteneurs associatifs à clefs multiples ou non. Les associations à clefs uniques et à clefs multiple sont implémentées respectivement par les classes template map et multimap, et les ensembles à clefs uniques et à clefs multiples par les classes template set et multiset. Cependant, bien que ces classes se comportent de manière profondément différentes, elles fournissent les mêmes méthodes permettant de les manipuler. Les conteneurs associatifs sont donc moins hétéroclites que les séquences, et leur manipulation en est de beaucoup facilitée.

Les sections suivantes présentent les différentes fonctionnalités des conteneurs associatifs dans leur ensemble. Les exemples seront donnés en utilisant la plupart du temps la classe template map, car c'est certainement la classe la plus utilisée en pratique en raison de sa capacité à stocker et à retrouver rapidement des objets identifiés de manière unique par un identifiant. Cependant, certains exemples utiliseront des conteneurs à clefs multiples afin de bien montrer les rares différences qui existent entre les conteneurs à clefs uniques et les conteneurs à clefs multiples.

### 2.11. Généralités et propriétés de base des clef

La contrainte fondamentale que les algorithmes des conteneurs associatifs imposent est qu'il existe une relation d'ordre pour le type de donnée utilisé pour les clefs des objets. Cette relation peut être définie soit implicitement par un opérateur d'infériorité, soit par un foncteur que l'on peut spécifier en tant que paramètre template des classes des conteneurs.

lors que l'ordre de la suite des éléments stockés dans les séquences est très important, ce n'est pas le cas avec les conteneurs associatifs, car ceux-ci se basent exclusivement sur l'ordre des clefs des objets. En revanche, la bibliothèque standard C++ garantit que le sens de parcours utilisé par les itérateurs des conteneurs associatifs est non décroissant sur les clefs des objets

itérés. Cela signifie que le test d'infériorité strict entre la clef de l'élément suivant et la clef de l'élément courant est toujours faux, ou, autrement dit, l'élément suivant n'est pas plus petit que l'élément courant.

**Note :** Attention, cela ne signifie aucunement que les éléments sont classés dans l'ordre croissant des clefs. En effet, l'existence d'un opérateur d'infériorité n'implique pas forcément celle d'un opérateur de supériorité d'une part, et deux valeurs comparables par cet opérateur ne le sont pas forcément par l'opérateur de supériorité. L'élément suivant n'est donc pas forcément plus grand que l'élément courant. En particulier, pour les conteneurs à clefs multiples, les clefs de deux éléments successifs peuvent être égales.
Comme pour tous les conteneurs, le type des éléments stockés par les conteneurs associatifs est le type value_type. Cependant, contrairement aux séquences, ce type n'est pas toujours le type template par lequel le conteneur est paramétré. En effet, ce type est une paire contenant le couple de valeurs formé par la clef et par l'objet lui- même pour toutes les associations (c'est-à-dire pour les map et les multimap). Dans ce cas, les méthodes du conteneur qui doivent effectuer des comparaisons sur les objets se basent uniquement sur le champ first de la paire encapsulant le couple (clef, valeur) de chaque objet. Autrement dit, les comparaisons d'objets sont toujours définies sur les clefs, et jamais sur les objets eux-mêmes. Bien entendu, pour les ensembles, le type value_type est strictement équivalent au type template par lequel ils sont paramétrés.

Pour simplifier l'utilisation de leurs clefs, les conteneurs associatifs définissent quelques types complémentaires de ceux que l'on a déjà présentés dans la Section 17.1.2. Le plus important de ces types est sans doute le type key_type qui, comme son nom l'indique, représente le type des clefs utilisées par ce conteneur. Ce type constitue donc, avec le type value_type, l'essentiel des informations de typage des conteneurs associatifs. Enfin, les conteneurs définissent également des types de prédicats permettant d'effectuer des comparaisons entre deux clefs et entre deux objets de type value_type. Il s'agit des types key_compare et value_compare.

### 2.12. Construction et initialisation

Les conteneurs associatifs disposent de plusieurs surcharges de leurs constructeurs qui permettent de les créer et de les initialiser directement. De manière générale, ces constructeurs prennent tous deux paramètres afin de laisser au programmeur la possibilité de définir la valeur du foncteur qu'ils doivent utiliser pour comparer les clefs, ainsi qu'une instance de l'allocateur à utiliser pour les opérations mémoire. Comme pour les séquences, ces paramètres disposent de valeurs par défaut, si bien qu'en général il n'est pas nécessaire de les préciser.
Hormis le constructeur de copie et le constructeur par défaut, les conteneurs associatifs fournissent un troisième constructeur permettant de les initialiser à partir d'une série d'objets. Ces objets sont spécifiés par deux itérateurs, le premier indiquant le premier objet à insérer dans le conteneur et le deuxième l'itérateur référençant l'élément suivant le dernier élément à insérer. L'utilisation de ce constructeur est semblable au constructeur du même type défini pour les séquences et ne devrait donc pas poser de problème particulier.

```
#include <iostream>
#include <map>
#include <list>

using namespace std;

int main(void)
{
    typedef map<int, char *> Int2String;
    // Remplit une liste d'éléments pour ces maps :
    typedef list<pair<int, char *> > lv;
    lv l;
    l.push_back(lv::value_type(1, "Un"));
    l.push_back(lv::value_type(2, "Deux"));
    l.push_back(lv::value_type(5, "Trois"));
    l.push_back(lv::value_type(6, "Quatre"));
    // Construit une map et l'initialise avec la liste :
    Int2String i2s(l.begin(), l.end());
    // Affiche le contenu de la map :
    Int2String::iterator i = i2s.begin();
    while (i != i2s.end())
    {
        cout << i->second << endl;
        ++i;
    }
    return 0;
```

**Note :** Contrairement aux séquences, les conteneurs associatifs ne disposent pas de méthode assign permettant d'initialiser un conteneur avec des objets provenant d'une séquence ou d'un autre conteneur associatif. En revanche, ils disposent d'un constructeur et d'un opérateur de copie.

### 2.13. Ajout et suppression d'éléments

Du fait de l'existence des clefs, les méthodes d'insertion et de suppression des conteneurs associatifs sont légèrement différentes de celles des séquences. De plus, elles n'ont pas tout à fait la même signification. En effet, les méthodes d'insertion des conteneurs associatifs ne permettent pas, contrairement à celles des séquences, de spécifier l'emplacement où un élément doit être inséré puisque l'ordre des éléments est imposé par la valeur de leur clef. Les méthodes d'insertion des conteneurs associatifs sont présentées ci-dessous :

```
iterator insert(iterator i, const value_type &;valeur)
```

Insère la valeur valeur dans le conteneur. L'itérateur i indique l'emplacement probable dans le conteneur où l'insertion doit être faite. Cette méthode peut donc être utilisée pour les algorithmes qui connaissent déjà plus ou moins l'ordre des éléments qu'ils insèrent dans le conteneur afin d'optimiser les performances du programme. En général, l'insertion se fait avec une complexité de ln(N) (où N est le nombre d'éléments déjà présents dans le conteneur). Toutefois, si l'élément est inséré après l'itérateur i dans le conteneur, la complexité est constante. L'insertion se fait systématiquement pour les conteneurs à clefs multiples, mais peut ne pas avoir lieu si un élément de même clef que celui que l'on veut insérer est déjà présent pour les conteneurs à clefs uniques. Dans tous les cas, la valeur retournée est un itérateur référençant l'élément inséré ou l'élément ayant la même clef que l'élément à insérer.

```
void insert(iterator premier, iterator dernier)
```

Insère les éléments de l'intervalle défini par les itérateurs premier et dernier dans le conteneur. La complexité de cette méthode est n×ln(n+N) en général, où N est le nombre d'éléments déjà présents dans le conteneur et n est le nombre d'éléments à insérer. Toutefois, si les éléments à insérer sont classés dans l'ordre de l'opérateur de comparaison utilisé par le conteneur, l'insertion se fait avec un coût proportionnel au nombre d'éléments à insérer.

```
pair<iterator, bool> insert(const value_type &;valeur)
```

Insère ou tente d'insérer un nouvel élément dans un conteneur à clefs uniques. Cette méthode renvoie une paire contenant l'itérateur référençant cet élément dans le conteneur et un booléen indiquant si l'insertion a effectivement eu lieu. Cette méthode n'est définie que pour les conteneurs associatifs à clefs uniques (c'est-à-dire les map et les set). Si aucun élément du conteneur ne correspond à la clef de l'élément passé en paramètre, cet élément est inséré dans le conteneur et la valeur renvoyée dans le deuxième champ de la paire vaut true. En revanche, si un autre élément utilisant cette clef existe déjà dans le conteneur, aucune insertion n'a lieu et le deuxième champ de la paire renvoyée vaut alors false. Dans tous les cas, l'itérateur stocké dans le premier champ de la valeur de retour référence l'élément inséré ou trouvé dans le conteneur. La complexité de cette méthode est logarithmique.

```
iterator insert(const value_type &;valeur)
```

Insère un nouvel élément dans un conteneur à clefs multiples. Cette insertion se produit qu'il y ait déjà ou non un autre élément utilisant la même clef dans le conteneur. La valeur retournée est un itérateur référençant le nouvel élément inséré. Vous ne trouverez cette méthode que sur les conteneurs associatifs à clefs multiples, c'est-a- dire sur les multimap et les multiset. La complexité de cette méthode est logarithmique.

Comme pour les séquences, la suppression des éléments des conteneurs associatifs se fait à l'aide des surcharges de la méthode erase. Les différentes versions de cette méthode sont indiquées ci-dessous :

```
void erase(iterator i)
```

Permet de supprimer l'élément référencé par l'itérateur i. Cette opération a un coût amorti constant car aucune recherche n'est nécessaire pour localiser l'élément.

```
void erase(iterator premier, iterator dernier)
```

Supprime tous les éléments de l'intervalle défini par les deux itérateurs premier et dernier. La complexité de cette opération est ln(N)+n, où N est le nombre d'éléments du conteneur avant suppression et n est le nombre d'éléments qui seront supprimés.

```
size_type erase(key_type clef)
```

Supprime tous les éléments dont la clef est égale à la valeur passée en paramètre. Cette

opération a pour complexité ln(N)+n, où N est le nombre d'éléments du conteneur avant suppression et n est le nombre d'éléments qui seront supprimés. Cette fonction retourne le nombre d'éléments effectivement supprimés. Ce nombre peut être nul si aucun élément ne correspond à la clef fournie en paramètre, ou valoir 1 pour les conteneurs à clefs uniques, ou être supérieur à 1 pour les conteneurs à clefs multiples.

Les conteneurs associatifs disposent également, tout comme les séquences, d'une méthode clear permettant de vider complètement un conteneur. Cette opération est réalisée avec un coût proportionnel au nombre d'éléments se trouvant dans le conteneur.

Exemple -10. Insertion et suppression d'éléments d'une association

```
#include <iostream>
#include <map>

using namespace std;

typedef map<int, char *> Int2String;

void print(Int2String &m)
{
    Int2String::iterator i = m.begin();
    while (i != m.end())
    {
        cout << i->second << endl;
        ++i;
    }
    return ;
}

int main(void)
{
    // Construit une association Entier -> Chaîne :
    Int2String m;
    // Ajoute quelques éléments :
    m.insert(Int2String::value_type(2, "Deux"));
    pair<Int2String::iterator, bool> res =
        m.insert(Int2String::value_type(3, "Trois"));
    // On peut aussi spécifier un indice sur
    // l'emplacement où l'insertion aura lieu :
    m.insert(res.first,
        Int2String::value_type(5, "Cinq"));
    // Affiche le contenu de l'association :
    print(m);
    // Supprime l'élément de clef 2 :
    m.erase(2);
    // Supprime l'élément "Trois" par son itérateur :
    m.erase(res.first);
    print(m);
```

## 2.14. Fonctions de recherche

Les fonctions de recherche des conteneurs associatifs sont puissantes et nombreuses. Ces méthodes sont décrites ci-dessous :

```
iterator find(key_type clef)
```

Renvoie un itérateur référençant un élément du conteneur dont la clef est égale à la valeur passée en paramètre. Dans le cas des conteneurs à clefs multiples, l'itérateur renvoyé référence un des éléments dont la clef est égale à la valeur passée en paramètre. Attention, ce n'est pas forcément le premier élément du conteneur vérifiant cette propriété. Si aucun élément ne correspond à la clef, l'itérateur de fin du conteneur est renvoyé.

```
iterator lower_bound(key_type clef)
```

Renvoie un itérateur sur le premier élément du conteneur dont la clef est égale à la valeur passée en paramètre. Les valeurs suivantes de l'itérateur référenceront les éléments suivants dont la clef est supérieure ou égale à la clef de cet élément.

```
iterator upper_bound(key_type clef)
```

Renvoie un itérateur sur l'élément suivant le dernier élément dont la clef est égale à la valeur passée en paramètre. S'il n'y a pas de tel élément, c'est-à-dire si le dernier élément du

```
pair<iterator, iterator> equal_range(key_type clef)
```

conteneur utilise cette valeur de clef, renvoie l'itérateur de fin du conteneur.

Renvoie une paire d'itérateurs égaux respectivement aux itérateurs renvoyés par les méthodes lower_bound et upper_bound. Cette paire d'itérateurs référence donc tous les éléments du conteneur dont la clef est égale à la valeur passée en paramètre.

```
#include <iostream>
#include <map>

using namespace std;

int main(void)
{
    // Déclare une map à clefs multiples :
    typedef multimap<int, char *> Int2String;
    Int2String m;
    // Remplit la map :
    m.insert(Int2String::value_type(2,  "Deux"));
    m.insert(Int2String::value_type(3,  "Drei"));
    m.insert(Int2String::value_type(1, "Un"));
    m.insert(Int2String::value_type(3, "Three"));
    m.insert(Int2String::value_type(4, "Quatre"));
    m.insert(Int2String::value_type(3, "Trois"));
    // Recherche un élément de clef 4 et l'affiche :
    Int2String::iterator i = m.find(4);
    cout << i->first << " : " << i->second << endl;
    // Recherche le premier élément de clef 3 :
    i = m.lower_bound(3);
    // Affiche tous les éléments dont la clef vaut 3 :
    while (i != m.upper_bound(3))
    {
        cout << i->first << " : " << i->second << endl;
        ++i;
    }
    // Effectue la même opération, mais de manière plus efficace
    // (upper_bound n'est pas appelée à chaque itération) :
    pair<Int2String::iterator, Int2String::iterator> p =
        m.equal_range(3);
    for (i = p.first; i != p.second; ++i)
    {
        cout << i->first << " : " << i->second << endl;
    }
```

**Note :** Il existe également des surcharges const pour ces quatre méthodes de recherche afin de pouvoir les utiliser sur des conteneurs constants. Ces méthodes retournent des valeurs de

type const_iterator au lieu des itérateurs classiques, car il est interdit de modifier les valeurs stockées dans un conteneur de type const.

La classe template map fournit également une surcharge pour l'opérateur d'accès aux membres de tableau []. Cet opérateur renvoie la valeur de l'élément référencé par sa clef et permet d'obtenir directement cette valeur sans passer par la méthode find et un déréférencement de l'itérateur ainsi obtenu. Cet opérateur insère automatiquement un nouvel élément construit avec la valeur par défaut du type des éléments stockés dans la map si aucun élément ne correspond à la clef fournie en paramètre. Contrairement à l'opérateur [] des classes vector et deque, cet opérateur ne renvoie donc jamais l'exception out_of_range.

Les recherches dans les conteneurs associatifs s'appuient sur le fait que les objets disposent d'une relation d'ordre induite par le foncteur less appliqué sur le type des données qu'ils manipulent. Ce comportement est généralement celui qui est souhaité, mais il existe des situations où ce foncteur ne convient pas. Par exemple, on peut désirer que le classement des objets se fasse sur une de leur donnée membre seulement, ou que la fonction de comparaison utilisée pour classer les objets soit différente de celle induite par le foncteur less. La bibliothèque standard fournit donc la possibilité de spécifier un foncteur de comparaison pour chaque conteneur associatif, en tant que paramètre template complémentaire au type de données des objets contenus. Ce foncteur doit, s'il est spécifié, être précisé avant le type de l'allocateur mémoire à utiliser. Il pourra être construit à partir des facilités fournies par la bibliothèque standard pour la création et la manipulation des foncteurs.

```
#include <iostream>
#include <map>
#include <string>
#include <functional>
#include <cstring>
using namespace std;
// Fonction de comparaison de chaînes de caractères
// non sensible à la casse des lettres :
bool stringless_nocase(const string &s1, const string &s2)
{

    return (strcasecmp(s1.c_str(), s2.c_str()) < 0);
}

int main(void)
{

    // Définit le type des associations chaînes -> entiers
    // dont la clef est indexée sans tenir compte
    // de la casse des lettres :
    typedef map<string, int,
        pointer_to_binary_function<const string &,
            const string &, bool> > String2Int;
    String2Int m(ptr_fun(stringless_nocase));
    // Insère quelques éléments dans la map :
    m.insert(String2Int::value_type("a. Un", 1));
    m.insert(String2Int::value_type("B. Deux", 2));
    m.insert(String2Int::value_type("c. Trois", 3));
    // Affiche le contenu de la map :
    String2Int::iterator i = m.begin();
    while (i != m.end())
    {

        cout << i->first << " : " << i->second << endl;
        ++i;
    }

    return 0;
}
```

Dans cet exemple, le type du foncteur est spécifié en troisième paramètre de la classe template map. Ce type est une instance de la classe template pointer_to_binary_function pour les types string et bool. Comme on l'a vu dans la Section 13.5, cette classe permet d'encapsuler toute fonction binaire dans un foncteur binaire. Il ne reste donc qu'à spécifier l'instance du foncteur que la classe template map doit utiliser, en la lui fournissant dans son constructeur. L'exemple précédent utilise la fonction utilitaire ptr_fun de la bibliothèque standard pour construire ce foncteur à partir de la fonction stringless_nocase.

En fait, il est possible de passer des foncteurs beaucoup plus évolués à la classe map, qui peuvent éventuellement être paramétrés par d'autres paramètres que la fonction de comparaison à utiliser pour comparer deux clefs. Cependant, il est rare d'avoir à écrire de tels foncteurs et même, en général, il est courant que la fonction binaire utilisée soit toujours la même. Dans ce cas, il est plus simple de définir directement le foncteur et de laisser le constructeur de la classe map prendre sa valeur par défaut. Ainsi, seul le paramètre template donnant le type du foncteur doit être spécifié, et l'utilisation des conteneurs associatif en est d'autant facilitée. L'exemple suivant montre comment la comparaison de chaînes de caractères non sensible à la casse peut être implémentée de manière simplifiée.

```
#include <iostream>
#include <string>
#include <map>
#include <functional>
#include <cstring>
using namespace std;
// Classe de comparaison de chaînes de caractères :
class StringLessNoCase : public binary_function<string, string, bool>
{

public:
    bool operator()(const string &s1, const string &s2)
    {

        return (strcasecmp(s1.c_str(), s2.c_str()) < 0);
    }

};

int main(void)
{

    // Définition du type des associations chaînes -> entiers
    // en spécifiant directement le type de foncteur à utiliser
    // pour les comparaisons de clefs :
    typedef map<string, int, StringLessNoCase> String2Int;
    // Instanciation d'une association en utilisant
    // la valeur par défaut du foncteur de comparaison :
    String2Int m;
    // Utilisation de la map :
    m.insert(String2Int::value_type("a. Un", 1));
    m.insert(String2Int::value_type("B. Deux", 2));
    m.insert(String2Int::value_type("c. Trois", 3));
    String2Int::iterator i = m.begin();
    while (i != m.end())
    {

        cout << i->first << " : " << i->second << endl;
        ++i;
    }

    return 0;
}
```

**Note :** Les deux exemples précédents utilisent la fonction strcasecmp de la bibliothèque C standard pour effectuer des comparaisons de chaînes qui ne tiennent pas compte de la casse

des caractères. Cette fonction s'utilise comme la fonction strcmp, qui compare deux chaînes et renvoie un entier dont le signe indique si la première chaîne est plus petite ou plus grande que la deuxième. Ces fonctions renvoient 0 si les deux chaînes sont strictement égales. Si vous désirez en savoir plus sur les fonctions de manipulation de chaînes de la bibliothèque C, veuillez vous référer à la bibliographie.

Pour finir, sachez que les conteneurs associatifs disposent d'une méthode count qui renvoie le nombre d'éléments du conteneur dont la clef est égale à la valeur passée en premier paramètre. Cette méthode retourne donc une valeur du type size_type du conteneur, valeur qui peut valoir 0 ou 1 pour les conteneurs à clefs uniques et n'importe quelle valeur pour les conteneurs à clefs multiples. La complexité de cette méthode est ln(N)+n, où N est le nombre d'éléments stockés dans le conteneur et n est le nombre d'éléments dont la clef est égale à la valeur passée en paramètre. Le premier terme provient en effet de la recherche du premier élément disposant de cette propriété, et le deuxième des comparaisons qui suivent pour compter les éléments désignés par la clef.

**Note :** Les implémentations de la bibliothèque standard utilisent généralement la structure de données des arbres rouges et noirs pour implémenter les conteneurs associatifs. Cette structure algorithmique est une forme d'arbre binaire équilibré, dont la hauteur est au plus le logarithme binaire du nombre d'éléments contenus. Ceci explique les performances des conteneurs associatifs sur les opérations de recherche.