ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

2e année – Spécialité Informatique

Année 2024 – 2025

Programmation orientée objet en C++

Notes de cours

Sébastien Fourey

Chapitre 5. Conteneurs et algorithmes de la bibliothèque standard

Version du 3 septembre 2024

Chapitre 5

Conteneurs et algorithmes de bibliothèque standard

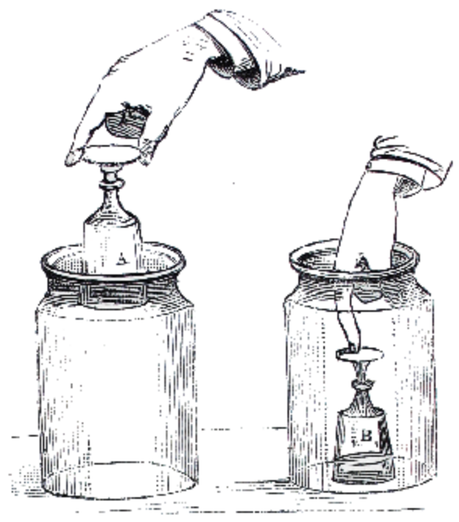


Fig. 8. — Au moment où il touche l'eau, le petit verre est plein d'air en A.

Fig. 9. — Quand le petit verre est au fond, l'air se resserre, *se comprime*, en B.

La bibliothèque standard du langage C fournit des fonctions que l'on peut qualifier de bas niveau en comparaison des possibilités offertes, par exemple, par les kits de développement du langage Java. La bibliothèque standard C++ se place à un niveau intermédiaire. Elle reste très rudimentaire, comparée à l'API [1] de Java, mais elle offre une batterie de modèles de classes et de fonctions qui dispensent le programmeur de la redéfinition de structures de données classiques, mais aussi de la récriture d'algorithmes usuels portant sur ces structures. A titre d'exemple, nous donnons ici une liste non exhaustive de structures et d'algorithmes qu'un programmeur C++ ne devrait jamais être amené à récrire :

- tableau de taille évolutive ;
- liste chaînée ;
- tas ;
- tableau associatif ;
- ensemble et opérations ensemblistes usuelles ;
- fusion de deux séquences ordonnées ;
- remplacement des éléments d'une séquence qui vérifient une condition ;
- etc.

5.1 Présentation

Comme son nom le rappelle, tous les symboles de la bibliothèque standard se trouvent dans l'espace de noms `std`. Ce préfixe est volontairement omis dans tout ce chapitre. La figure 5.1 montre la hiérarchie des modèles de classes définis par la bibliothèque. Ce sont les classes appelées *classes conteneurs*, les *adaptateurs* et les *itérateurs*. On distingue généralement deux catégories pour les classes conteneurs :

- Les *conteneurs séquentiels* dans lesquels les éléments sont classés en fonction de leur ordre d'insertion ou bien la position qui a été précisée lors de cette insertion. Les éléments ne sont pas nécessairement stockés de manière contiguë en mémoire, même si cette propriété

1. Application Programming Interface

est vraie pour le modèle `vector<>` par exemple [2].
— Les conteneurs ordonnés pour lesquels l'organisation des données en mémoire tire profit d'une relation d'ordre sur les éléments qu'ils contiennent (cf. cours d'algorithmique au sujet de la recherche dichotomique, de la structure de tas, des B-arbres, des arbres de recherche équilibrés, etc.)

Une troisième catégorie concerne les modèles qui *utilisent* des conteneurs pour offrir les services d'autres structures classiques : les piles, les files, et les files de priorité (5.1.3).

Une liste des méthodes essentielles spécifiques ou partagées par les différents types de conteneurs sera donnée dans la section 5.2. La section 5.5 dresse la liste des algorithmes génériques qui reposent en majorité sur l'utilisation des itérateurs (§ 5.4) et aussi pour certains sur la notion d'objet fonction (§ 5.6).

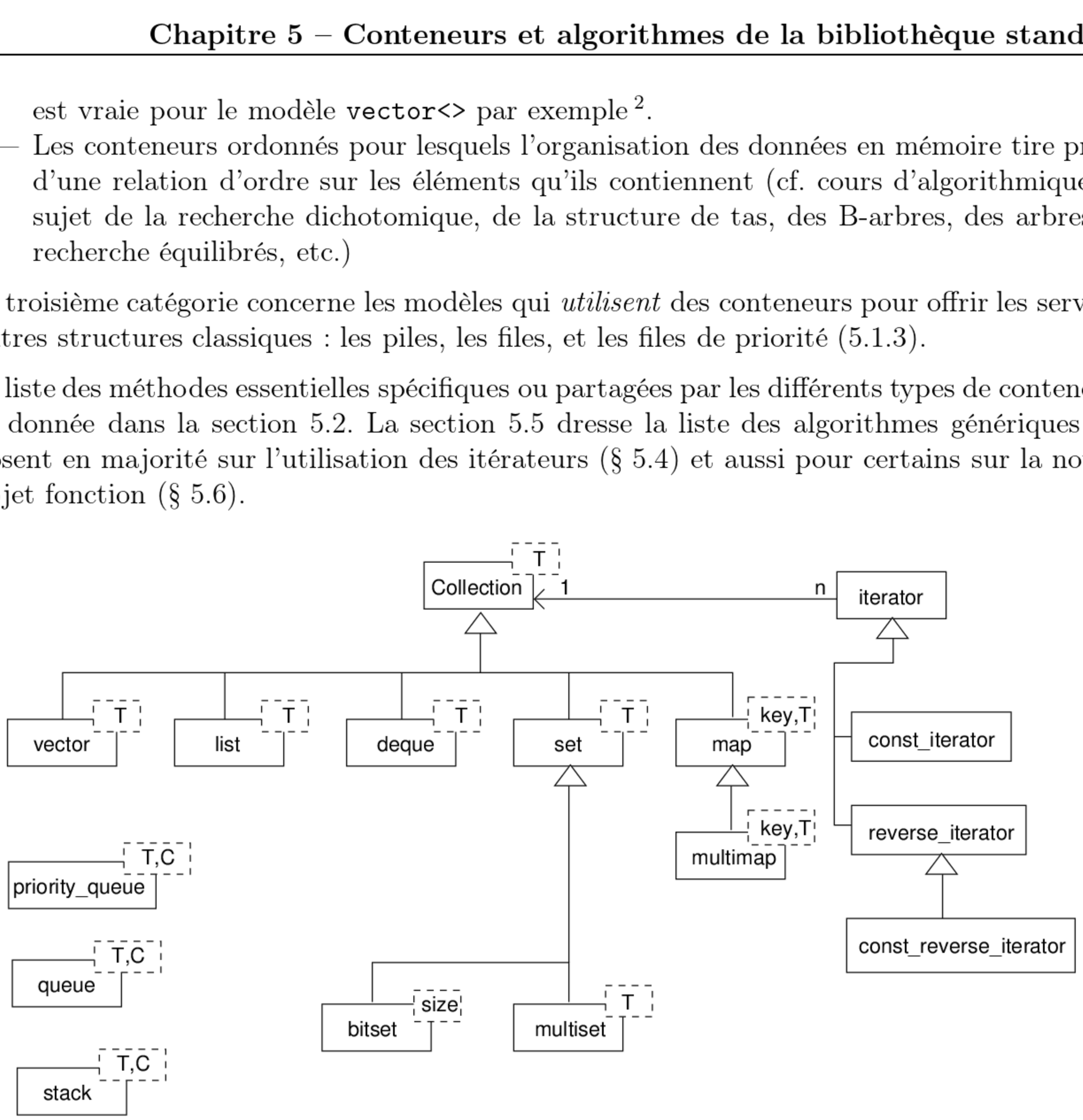


FIGURE 5.1 – Diagramme des quelques modèles de classes définis dans la bibliothèque standard (C++ 2003). Le type Collection représenté ici n'est pas un type de la bibliothèque ; il n'existe dans ce diagramme qu'à des fins d'illustration.

5.1.1 Les conteneurs séquentiels

Ils préservent l'ordre dans lequel les éléments sont insérés.

— **deque** (en-tête `<deque>`)
Modèle qui implémente une queue à double entrée. L'insertion et la suppression est faite en temps constant ($O(1)$) en début et fin de séquence. Ces deux opérations s'exécutent en $O(n)$ ailleurs, où n est le nombre d'éléments. L'accès direct à une position donnée se fait en temps constant ($O(1)$).
— **list** (en-tête `<list>`)
Modèle qui implémente une liste doublement chainée. L'insertion et la suppression est effectuée en $O(1)$ n'importe où. L'accès direct est impossible.

2. C'est d'ailleurs là une propriété, essentielle, précisée par la norme de 2003, qui était absente de la norme de 1998 !

— **vector** (en-tête `<vector>`)
Modèle des tableaux dont la taille peut varier en fonction des besoins. Les opérations d'insertion et de suppression ont un coût faible en fin de tableau, elle sont plus longues ailleurs. L'accès direct est possible. Cette classe doit être utilisée pour les tableaux là où on aurait recours en C à la fonction `realloc`. En effet, la norme stipule que les éléments sont stockés de façon contiguë en mémoire dans un vecteur. Il est donc possible de considérer l'adresse du premier élément comme un tableau C classique.

5.1.2 Les conteneurs ordonnés

Tous les conteneurs de cette catégorie stockent leurs éléments en tirant avantage d'une relation d'ordre qui doit être définie. La relation d'ordre utilisée par défaut correspond à l'opérateur `<` ([3]), éventuellement surchargé. Il est important de noter que pour les conteneurs ordonnés, deux éléments a et b sont considérés égaux si $\neg(a < b) \wedge \neg(b < a)$. Les conteneurs ordonnés sont au nombre de quatre :

— **set** et **multiset** (en-tête `<set>`)
Ces deux modèles définissent de manière générique les ensembles. La particularité d'un `multiset` est qu'il peut contenir plusieurs éléments égaux au sens de la relation d'ordre utilisée.

— **map** et **multimap** (en-tête `<map>`)
Les éléments sont des paires (clé, valeur) [4] et la rapidité d'accès aux éléments par leur clé est obtenue grâce à l'utilisation d'une relation d'ordre portant uniquement sur ces clés, et pas sur les valeurs associées. Enfin, une `multimap` peut contenir plusieurs paires ayant une même clé, au contraire d'une `map`.

Les conteneurs `set` et `map` effectuent une insertion, une suppression ou une recherche en $O(\log(n))$, si n est le nombre d'éléments présents.

5.1.3 Les adaptateurs

Ces trois modèles (file de priorité, file et pile) sont en plus paramétrés par le type de conteneur qu'ils utilisent pour stocker leurs éléments. On conçoit en effet aisément qu'un modèle de file peut être implémenté à l'aide d'une liste ou bien d'une queue à double entrée. Notez qu'il s'agit bien d'adaptateurs au sens du *design pattern* de même nom : ils n'offrent pas tous les services communs aux autres conteneurs, mais ils ont leur propres interfaces. Par exemple, l'utilisation d'un itérateur sur une pile n'a pas de sens ; le modèle n'en définit donc pas.

— **priority_queue** (en-tête `<queue>`)
Implémente un tas. (Composition avec une `deque` ou un `vector` pour l'accès direct.)

— **queue** (en-tête `<queue>`)
Implémente une file FIFO. (Composition avec une `deque` ou une `list`.)

— **stack** (en-tête `<stack>`)
Implémente une pile. (Composition avec une `deque`, une `list` ou un `vector`.)

Exercice 5.1 *Où est le haut de la pile lorsque qu'une instance de* `stack` *est composée avec un* `vector` *?*

3. Attention, $<$ n'est pas une relation d'ordre au sens mathématique. L'appellation est ici un léger abus de langage.

4. Voir section 5.2.5.

5.1.4 C++11 : conteneurs additionnels

Le C++11 apporte son lot de nouveaux conteneurs :

— **array** (en-tête `<array>`)
Définit un conteneur séquentiel, similaire à un tableau, *dont la taille est connue à la compilation*. Il peut être parcouru comme un conteneur séquentiel à l'aide d'un *bidirectional iterator* (section 5.4), gère les débordements, possède une sémantique de copie par valeur et des méthodes usuelles (`size()`, etc.). Le listing 5.1 illustre son utilisation.
— **forward_list** (en-tête `<forward_list>`)
Type similaire à `std::list` à ceci près qu'il n'est itérable qu'en avant à l'aide d'un *forward iterator* (cf. section 5.4). Implémenté par une liste simplement chainée, il est de ce fait un peu plus efficace qu'une `std::list` en terme d'occupation mémoire.
— **unordered_set** (en-tête `<unordered_set>`)
Ensemble dont les éléments ne sont pas nécessairement comparables, basé sur une table de hachage. La recherche, l'insertion et la suppression d'éléments se fait en moyenne en $O(1)$.
— **unordered_multiset** (en-tête `<unordered_set>`)
Idem que précédemment mais une même valeur peut apparaître plusieurs fois.
— **unordered_map** (en-tête `<unordered_map>`)
Tableau associatif pour lequel les clés ne sont pas nécessairement comparables (une fonction de hachage est utilisée). La recherche, l'insertion et la suppression d'éléments se fait en moyenne en $O(1)$.
— **unordered_multimap** (en-tête `<unordered_map>`)
Idem que précédemment mais une même clé peut apparaître plusieurs fois.

```
1  #include <array>
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6    array<int, 5> numbers{1, 2, 3, 4, 5};
7
8    cout << numbers[10] << endl; // Dangerous!
9    try {                        // Better
10     cout << numbers.at(10) << endl;
11   } catch (const out_of_range & e) {
12     cerr << "Out of range: " << e.what() << "\n";
13   }
14
15   for (int x : numbers) {
16     cout << x << endl;
17   }
18 }
```

Listing 5.1 – Utilisation d'un `array`.

5.2 Les conteneurs

Nous énumérons dans cette section les méthodes spécifiques à chaque type de conteneur mais aussi celles qui sont communes à plusieurs d'entre eux. Au préalable, il est nécessaire de connaître un certain nombre de synonymes de types qui sont définis *dans* les modèles de classes conteneurs.

`value_type`	Synonyme du type des éléments.
`size_type`	Type des indices et des tailles.
`difference_type`	Type des différences entre itérateurs.
`iterator`	Itérateur ($\simeq$ `value_type*`).
`const_iterator`	Itérateur ($\simeq$ `const value_type*`).
`reference`	$\simeq$ `value_type &`.
`const_reference`	$\simeq$ `const value_type &`.
`value_compare`	Type du comparateur. (Conteneurs ordonnés.)
`key_type`	Type des clés. (Conteneurs associatifs.)
`mapped_type`	Type des valeurs associées. (Conteneurs associatifs.)
`key_compare`	Type du comparateur de clés. (Conteneurs associatifs.)

5.2.1 Méthodes communes à tous les conteneurs

Elles sont données dans le tableau 5.1. Notez que si tous les conteneurs permettent la suppression d'un élément ou d'un intervalle désigné à l'aide d'itérateurs (méthodes `erase()`), mais aussi l'insertion d'un élément à une position donnée ; la complexité de ces opérations reste liée au type de conteneur. On rappelle par exemple que l'insertion est faite en temps constant n'importe où dans une liste, elle est relativement plus lente au sein d'un vecteur. (Elle peut en effet dans ce dernier cas nécessiter une réallocation, suivie de la recopie de tous les éléments situés après le point d'insertion.)

Tous les conteneurs

	conteneur`(const` *conteneur* `&);`
	`~`*conteneur*`();`
`void`	`clear();`
`size_type`	`size();`
`bool`	`empty() const;`
`iterator`	`begin()` *et* `end();`
`iterator`	`rbegin()` *et* `rend();`
`iterator`	`insert(iterator, const T &);`
`iterator`	`erase(iterator);`
`iterator`	`erase(iterator f, iterator l);`
`const` *conteneur*`&`	`operator=(const` *conteneur* `&);`

TABLE 5.1 – Méthodes des conteneurs standard.

5.2.2 Méthodes spécifiques des conteneurs séquentiels

Elle sont détaillées dans le tableau 5.2. Notez que les vecteurs et les queues à double entrée surchargent l'opérateur `[]` pour l'accès direct aux éléments d'indices 0 à `size()-1`. Une méthode

`at()` offre le même service mais diffère de l'opérateur dans le cas d'un dépassement d'indice. En effet, le comportement de l'opérateur `[]` est indéfini dans ce cas alors que la méthode `at()` lève une exception de type `std::out_of_range` (cf. chapitre 6).

La capacité d'un vecteur (`capacity()`), qu'il ne faut pas confondre avec sa taille, est le nombre d'éléments qu'il pourra contenir sans qu'il y ait réallocation de mémoire. En effet, une insertion en fin de vecteur ne provoque qu'occasionnellement une réallocation. La plupart du temps, un vecteur n'occupe qu'une partie de la zone mémoire qui lui est réellement allouée.

Tous les conteneurs séquentiels	
	conteneur`(size_type, const value_type &)`
vector	
`size_type`	`capacity()`
`void`	`reserve(size_type)`
deque vector	
`/const/ value_type&`	`at(size_type n) /const/ ; {out_of_range}`
`/const/ value_type&`	`operator[](size_type n) /const/ ;`
deque list vector	
`/const/ value_type&`	`back() /const/ ;`
`/const/ value_type&`	`front() /const/ ;`
`void`	`pop_back()`
`void`	`push_back(const T &);`
`void`	`resize(size_type n, T t = T());`
`iterator`	`insert(iterator, const T &);`
deque list	
`void`	`push_front(const T &);`
`void`	`pop_front();`

TABLE 5.2 – Méthodes des conteneurs séquentiels.

5.2.3 Méthodes spécifiques des conteneurs ordonnés

Les principales méthodes de ces conteneurs sont données dans le tableau 5.3. Mis à part le constructeur, toutes les méthodes de ce tableau sont rendues efficaces grâce à l'utilisation de la relation d'ordre définie entre les éléments. Par exemple, la recherche d'un élément dans un ensemble à partir de sa valeur, comme la recherche d'une paire dans un tableau associatif à partir d'une valeur de clé, est réalisée avec une complexité en $O(\log(n))$.

5.2.4 Les adaptateurs

Les opérations classiques portant sur les piles, les files et les tas sont présentées dans le tableau 5.4. La figure 5.2 montre que le modèle `stack` est une classe paramétrée par le type `C` de conteneur utilisé pour stocker les éléments.

5.2.5 Le modèle de classe `pair` (couples)

Il est défini dans dans `<utility>`. Le listing 5.2 donne un aperçu de sa définition.

Tous les conteneurs ordonnés

conteneur(key_compare)

map multimap set multiset

/const/_iterator	find(const key_type &) /const/;
/const/_iterator	lower_bound(const key_type &) /const/;
/const/_iterator	upper_bound(const key_type &) /const/;
pair<it.,it.>	equal_range(const key_type &);
size_type	erase(const key_type &)
size_type	count(const key_type &)
pair<it., bool>	insert(const value_type &)

map

mapped_type&	operator[](const key_type&);

TABLE 5.3 – Méthodes des conteneurs ordonnés.

Tous les adaptateurs

size_type	size();

stack

/const/ value_type&	top() /const/;
bool	empty() const;
void	pop();
void	push(const value_type &);

queue

bool	empty() const;
/const/ value_type&	back() /const/;
/const/ value_type&	front() /const/;
void	pop();
void	push(const value_type &);

priority_queue

bool	empty() const;
void	pop();
void	push(const value_type &);
const value_type&	top() const;

TABLE 5.4 – Méthodes des classes d'adaptateurs.

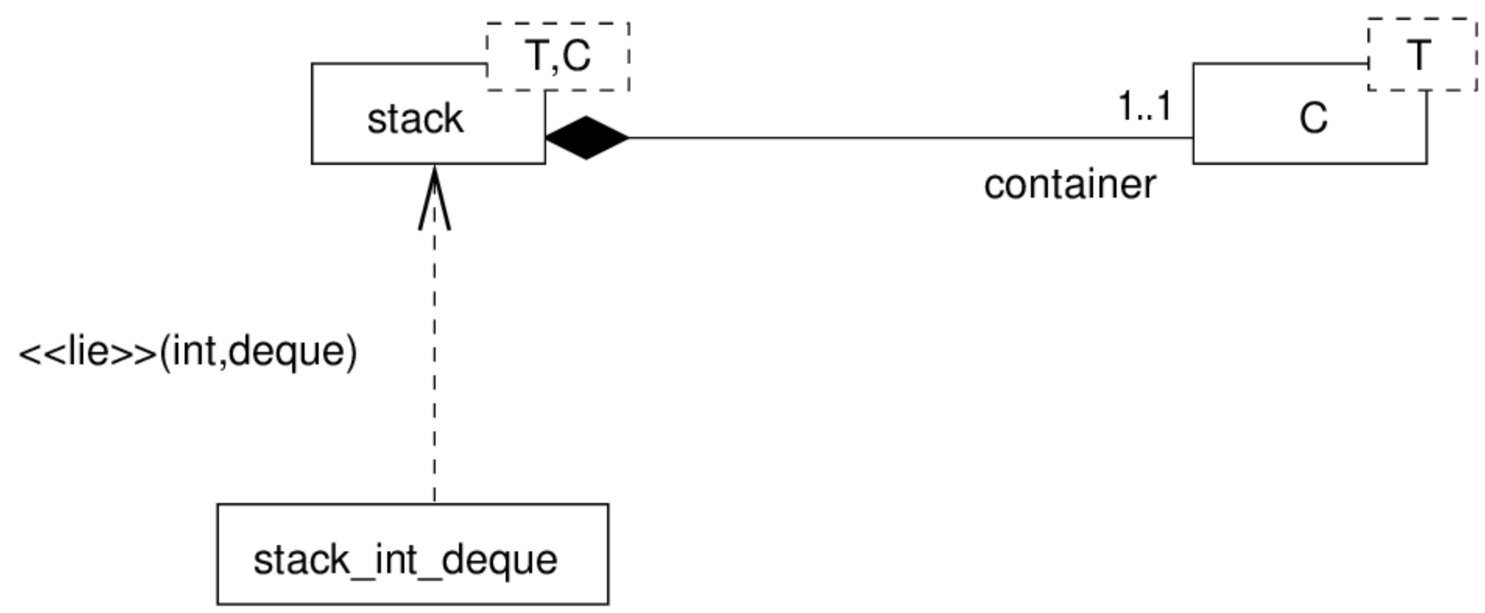


FIGURE 5.2 – Diagramme de classes du modèle `stack`.

```
1 template <typename T1, typename T2>
2 struct pair {
3   T1 first;
4   T2 second;
5   pair(const T1 & x, const T2 & y);
6   // ...
7 };
```

Listing 5.2 – Extrait de la définition du modèle `pair<>` de la bibliothèque standard.

Le modèle de fonction `make_pair` est aussi défini (cf. listing 5.3). Il permet, grâce à la syntaxe d'instanciation des modèles de fonctions en utilisant les `<>`, de créer des paires temporaires sans ambiguïté sur les types.

```
1 template <class T1, class T2>
2 pair<T1, T2> std::make_pair(const T1 &, const T2 &);
```

Listing 5.3 – Prototype du modèle de fonction `make_pair`.

5.2.6 C++11 : le modèle de classe `tuple` (n-uplet)

Le modèle de classe `tuple`, défini dans l'en-tête `<tuple>`, permet de représenter des séquences ordonnées et de taille fixe, de données *hétérogènes*. Autrement dit, il implémente la notion de n-uplet. Il peut être vu comme une structure (c.-à-d. `struct`) sans nom.

Comme pour le type `pair`, il existe un modèle de fonction `make_tuple` permettant de créer des n-uplets par déduction automatique des types utilisés comme arguments.

Un exemple d'utilisation est donné dans le listing 5.4.

Comme illustré par la ligne 21 du listing 5.4, il est possible de comparer des `tuple` pour peu que les types des éléments soient eux aussi tous comparables.

Remarque 5.1 *Le nombre maximum d'éléments dans un* `tuple` *dépend de l'implémentation.*

Pour finir, signalons que le modèle `tuple` ne peut exister que grâce à une nouveauté du C++11 : les modèles à liste de paramètres variables (cf. [12], *Variadic Templates*).

5.3 Exemples d'utilisations

5.3.1 Instanciations

Le listing ci-dessous montre quelques exemples d'instanciations de modèles de classes conteneurs.

```
1 std::stack<int> pile;
2 std::map<string, int> ages;
3 std::map<float, float> f;
4 std::map<double, pair<double, double> > courbe;
```

```cpp
#include <iostream>
#include <string>
#include <tuple>
using namespace std;

void display(const std::tuple<string, string, int> & person) {
  cout << "Firstname : " << get<0>(person) << endl;
  cout << "Lastname : " << get<1>(person) << endl;
  cout << "Age : " << get<2>(person) << endl;
}

int main() {
  std::tuple<string, string, int> author{"Sebastien", "Fourey", 20};
  int age = get<2>(author);
  display(author);
  display(std::tuple<string, string, int>{"John", "Doe", 33});
  string firstname{"John"};
  string lastname{"Doe"};
  display(make_tuple(firstname, lastname, 33));

  cout << (make_tuple(1, 2, 3) > make_tuple(1, 1, 1)) // 1
       << endl;
}
```

Listing 5.4 – Utilisation du modèle `tuple`.

5.3.2 Premier exemple

Cet exemple montre l'utilisation d'une queue à double entrée et d'un itérateur sur ce conteneur. (La notion d'itérateur fait l'objet de la section 5.4.)

```
1  #include <deque>
2  #include <iostream>
3  using std::cout;
4  using std::endl;
5
6  int main() {
7    std::deque<char> a_string;
8    std::deque<char>::iterator it, end;
9
10   cout << a_string.max_size() << endl;
11
12   a_string.push_back('@');
13   a_string.push_back('b');
14   a_string.push_back('o');
15   a_string.push_back('n');
16   a_string.push_back('j');
17   a_string.push_back('o');
18   a_string.push_back('u');
19   a_string.push_back('r');
20   a_string.pop_front();
21
22   end = a_string.end();
23   for (it = a_string.begin(); it != end; ++it) {
24     cout << *it;
25   }
26   cout << endl;
27 }
```

Listing 5.5 – Exemple d'utilisation d'un modèle de classe conteneur de la bibliothèque standard.

5.3.3 Deuxième exemple

```
1  #include <algorithm>
2  #include <iostream>
3  #include <iterator>
4  #include <list>
5  #include <vector>
6  using std::cout;
7  using std::endl;
8
9  int main() {
10   std::vector<int> v(10);
11   std::vector<int>::iterator it;
12
```

```
13    for (int i = 0; i < 10; i++) {
14      v[i] = i;
15    }
16
17    v.push_back(10);
18    v.insert(v.begin(), 5); // No push_front method
19
20    it = v.begin() + 5;
21    v.erase(it);
22
23    for (it = v.begin(); it != v.end(); ++it) {
24      cout << *i << " ";
25    }
26    cout << endl;
27    // 5 0 1 2 3 5 6 7 8 9 10
28  }
```

Listing 5.6 – Exemple d'utilisation du modèle de classe `vector`.

5.3.4 Troisième exemple

Ce dernier exemple illustre l'utilisation d'un tableau associatif `map`. La syntaxe courante d'accès en consultation comme en modification utilise l'opérateur `[]`. Dans ce listing, on montre toutefois qu'une `map` n'est autre qu'un ensemble de paires (cf. l'utilisation de la méthode `insert()` et du modèle de fonction `make_pair<>()`).

Dans une `map`, l'insertion d'une paire à l'aide la méthode `insert()` suit la règle qui est valable pour les ensembles : si la clé est déjà présente, aucune insertion ni modification n'est faite ! (Il vaut mieux, dans certains cas, utiliser les crochets.)

```
1  #include <iostream>
2  #include <map>
3  #include <string>
4  using namespace std;
5
6  int main() {
7
8    map<int, string> v;
9    map<int, string>::iterator i;
10   pair<map<int, string>::iterator, bool> p;
11
12   v.insert(make_pair(1, string("OK")));
13   v.insert(make_pair(2, string("Coral")));
14   v.insert(make_pair(3, string("Tango")));
15   p = v.insert(make_pair(3, string("Tango2"))); // void
16
17   cout << p.second << endl;
18
19   v.insert(make_pair(4, string("Charly")));
20   v[5] = string("Bravo");
```

```
21    v[5] = string("Daddy");
22
23    i = v.find(2);
24    v.erase(i);
25
26    for (i = v.begin(); i != v.end(); i++) {
27       cout << "(" << i->first << "," << i->second << ")";
28    }
29    cout << endl;
30 }
```

Listing 5.7 – Exemple d'utilisation d'un modèle de classe conteneur de la bibliothèque standard.

Le programme du listing 5.7 produit la sortie suivante :

```
0
(1,OK)(3,Tango)(4,Charly)(5,Daddy)
```

5.4 Les itérateurs

Les itérateurs sont la clé de voûte des algorithmes que la bibliothèque standard met à la dispostion du programmeur. Ils y sont définis d'une façon légèrement différente de celle qui a été vue dans le cadre des *design patterns* (cf. figure 5.3(b)). En C++ , un itérateur est plus proche d'une abstraction de la notion de pointeur [5]. Notamment, les itérateurs de la bibliothèque standard ne disposent pas de méthode `first()` ni de méthode `isDone()`. En effet, un itérateur n'a que très peu de données sur le conteneur qui lui est associé. De ce fait :

- C'est le conteneur qui fournit un itérateur *pointant* sur son premier élément via sa méthode `begin()` (car un itérateur ne sait pas se positionner de lui-même sur le début) ;
- Un conteneur peut construire un itérateur *pointant* sur un élément virtuel qui succède au dernier élément, c'est l'itérateur retourné par la méthode `end()` (car un itérateur ne sait pas signaler de lui-même qu'il a atteint la fin du conteneur).

Le second point, ajouté à la possibilité de comparer deux itérateurs à l'aide de l'opérateur `==`, offre finalement l'équivalent de la méthode `isDone()` du *design pattern*. (Voir la boucle `for` du listing 5.5).

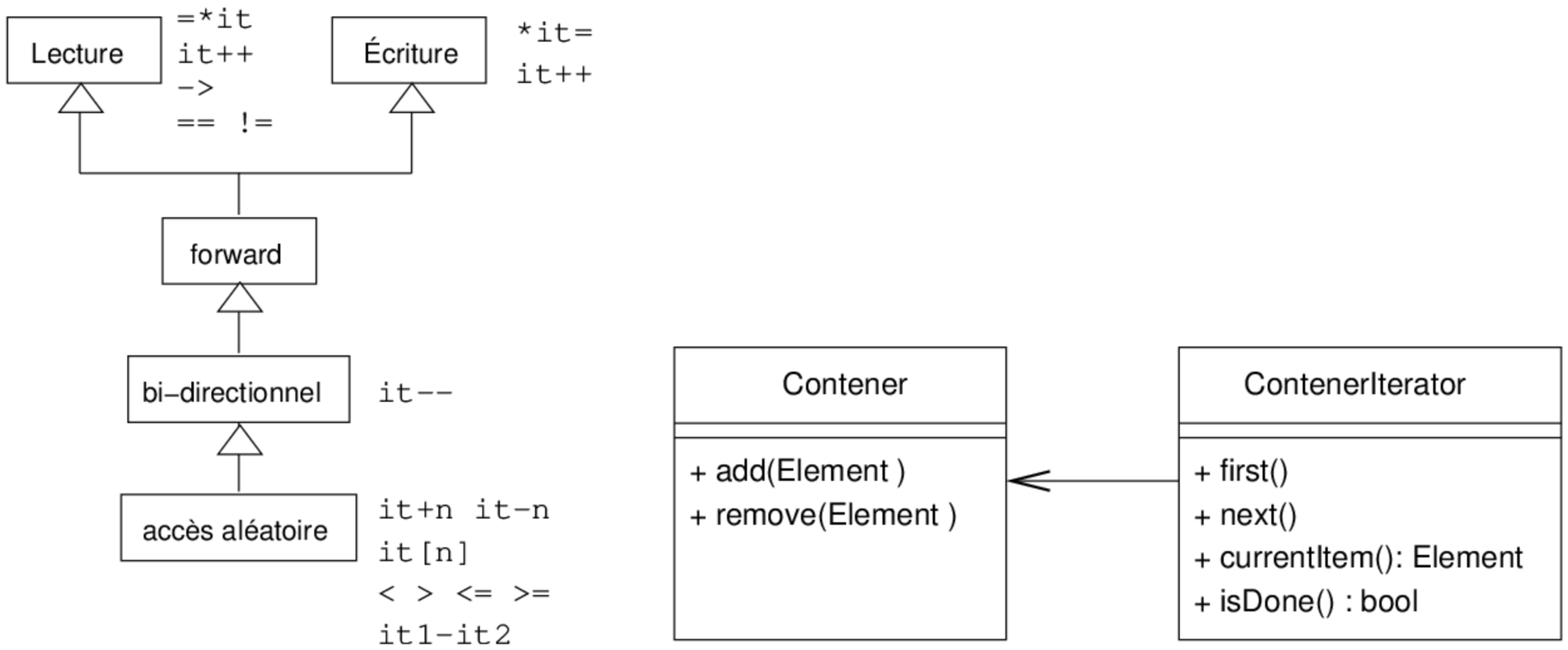
Une autre différence, qui est essentiellement syntaxique, réside dans l'utilisation des opérateurs d'incrémentation (`++`) et de déréférencement (`*`) qui remplacent respectivement les méthodes `next()` et `currentItem()`.

On peut distinguer plusieurs familles d'itérateurs (cf. figure 5.3) :

- Les `input iterator` et les `output iterator` servent de base pour les itérateurs de *flux*.
- Les `forward iterator` combinent les fonctionnalités des deux précédentes familles. Ils peuvent donc être utilisés partout où l'une des deux est attendue.
- Les `bidirectionnal iterator` servent pour les conteneurs associatifs et les listes.
- Les `random iterator` servent pour les *vecteurs* et les *queues*.

Les itérateurs n'offrent que peu de sécurité, à l'instar des pointeurs. Par exemple, rien n'empêche d'incrémenter un itérateur alors qu'il est déjà positionné à la fin d'un conteneur...

5. A tel point qu'un itérateur est parfois effectivement un simple pointeur.



(a) Les différents types d'itérateurs. (b) Aperçu du *design pattern* Itérateur.

FIGURE 5.3 – Iterateurs

5.4.1 Le modèle `iterator_traits`

Ce modèle (un *trait*) permet d'écrire des algorithmes génériques en utilisant une syntaxe valable pour des itérateurs qui sont des classes mais aussi pour ceux qui sont de simples pointeurs. (Il existe un modèle partiellement spécialisé `iterator_traits<T*>`.) Le listing 5.8 donne un aperçu des synonymes de types définis dans ces traits.

```
1 template <typename I>
2 struct iterator_traits {
3   typedef typename I::iterator_category iterator_category;
4   typedef typename I::value_type value_type;
5   typedef typename I::difference_type difference_type;
6   typedef typename I::pointer pointer;
7   typedef typename I::reference reference;
8 };
```

Listing 5.8 – Les *traits* d'un itérateur.

L'utilité de ce trait est illustrée par le modèle de fonction `distance()` donné dans la section suivante.

5.4.2 Deux modèles de fonctions sur les itérateurs

Pour illustrer le principe des algorithmes génériques définis dans la bibliothèque standard (Section 5.5), qui utilisent intensivement les itérateurs ; nous donnons ici deux exemples très simples de modèles de fonctions. (Il sont définis dans l'en-tête `<iterator>`.)

La fonction `distance()`

Le listing suivant (5.9) est une écriture possible de l'algorithme `distance`. Notez cependant qu'il peut être spécialisé pour donner une réponse en temps constant, et non pas linéaire, si le conteneur le permet (p. ex. `std::vector`).

```
1 template <typename I>
2 typename iterator_traits<I>::difference_type
3 distance(I first, I last) {
4   typename iterator_traits<I>::difference_type d = 0;
5   while (first++ != last) {
6     d++;
7   }
8   return d;
9 }
```

Listing 5.9 – Le modèle de fonction `distance<>()`.

Il est important de noter que sans le trait `iterator_traits<>` précédemment défini, il serait impossible de définir correctement et de manière générique le type de retour de cette fonction (c.-à-d. `difference_type`). Le même problème se pose, et se résoud, si on a besoin d'utiliser le type des éléments « pointés » lorsque l'on ne dispose que du paramètre de type **I** (pour itérateur).

La fonction advance()

Ci-dessous, la version « basique » de cette fonction. En réalité, elle est spécialisée selon le type d'itérateur.

```
1 template <typename Iterator, typename Distance>
2 void advance(Iterator & i, Distance n) {
3   while (n--) {
4     ++i;
5   }
6 }
```

Listing 5.10 – Le modèle de fonction advance<>().

5.5 Les algorithmes (<algorithm>)

Dans cette section, les principaux algorithmes définis dans la bibliothèque standard sont énumérés sous la forme de tableaux, avec les notations données ci-après. Tous ces algorithmes s'appliquent sur des séquences qui sont spécifiées à l'aide d'itérateurs. D'une manière générale, toute séquence d'entrée peut être précisée sous la forme de deux itérateurs : son début et sa fin. Une séquence de sortie est souvent déterminée sans ambiguïté par son début lorsqu'une séquence d'entrée a été précisée. (Les algorithmes fill_n et generate_n font parties des exceptions.)

Notez que beaucoup d'algorithmes utilisent aussi, en plus des itérateurs, des objets fonctions. Plusieurs modèles sont définis (dans <functional>) pour faciliter l'utilisation de ce type de classe. Il seront présentés dans la section 5.6.

Les notations suivantes sont utilisées dans les tableaux de cette section.

r	random access iterator	V	valeur	p	prédicat unaire
b	bidirectionnal iterator	R	référence	p2	prédicat binaire
f	forward iterator	CR	référence constante	c	fonction de comparaison
i	input iterator	P	paire	F	fonction unaire
o	output iterator	B	booléen	F2	fonction binaire
				n	compteur

5.5.1 Opérations sans modification

Ces algorithmes (tableau 5.5) ne modifient pas les séquences données comme paramètres.

5.5.2 Opérations avec modification

Ces algorithmes (tableau 5.6) modifient la séquence donnée en paramètre lorsqu'elle est unique. Si deux séquences sont données, les deux peuvent être modifiées ou bien uniquement la seconde.

Nom	Retour	Arguments	Description
for_each	F	i,i,F	Applique F sur chaque élément
find	i	i,i,V	Trouve V dans la séquence
find_if	i	i,i,p	Trouve le premier élément satisfaisant p
find_first_of	f	f,f,f,f[,p2]	Trouve le premier élément d'une séquence ayant une correspondance dans une autre
adjacent_find	f	f,f[,p2]	Recherche une paire de valeurs contigües correspondantes
count	diff. type	i,i,V	Compte le nombre d'occurence de V
count_if	diff. type	i,i,p	Compte le nombre d'éléments satisfaisant p
equal	B	i,i,i[,p2]	Test d'égalité des éléments deux à deux
mismatch	pair<i,i>	i,i,i[,p2]	Recherche la première paire d'éléments différents
search	f	f,f,f,f[,p2]	Recherche la deuxième séquence dans la première
find_end	f	f,f,f,f[,p2]	Recherche la dernière occurence de la seconde séquence dans la première
search_n	f	f,f,n,V[,p2]	Recherche une séquence de n valeurs correspondantes

TABLE 5.5 – Algorithmes en « lecture seule ».

La convention, sur l'ordre des arguments, choisie dans la bibliothèque standard pour les algorithmes qui ont une source et une destination (comme `copy()`) est l'inverse de celle utilisée dans les fonctions de la bibliothèque C (`memcpy()`, `strcpy()`, `memmove()`, etc.).

Nom	Retour	Arguments	Description
copy	o	i,i,o	Copie une séquence vers un itérateur de sortie
transform	o	i,i,o,F	Produit une tranformation de son entrée par F
transform	o	i,i,i,o,F2	Transformation de deux séquences par une fonction binaire
unique	f	f,f[,p2]	Réordonne la séquence en plaçant les éléments uniques en tête
unique_copy	o	i,i,o[,p2]	Copie en éliminant les doublons
replace	void	f,f,V,V	Remplace une valeur par une autre
replace_if	void	f,f,p,V	Remplace les valeurs satisfaisant p
replace_copy	o	i,i,o,V,V	
replace_copy_if	o	i,i,o,p,V	
remove	f	f,f,V	Supprime les éléments égaux à V (mis en fin de séquence)
remove_if	f	f,f,p	Supprime les éléments satisfaisant p
remove_copy	o	i,i,o,V	Copie tout sauf V
remove_copy_if	o	i,i,o,p	Copie ce qui ne satisfait pas p
fill	void	f,f,V	Remplit la séquence avec V
fill_n	void	o,n,V	Remplit avec n copies de V
generate	void	f,f,g()	Remplit avec le générateur g()
generate_n	void	o,n,g()	remplit avec n appels à g()
reverse	void	b,b	Renverse la séquence
reverse_copy	void	b,b,o	Crée une copie renversée
rotate	void	f,f,f	(begin,middle,last) Effectue une rotation de sorte que middle se retrouve en first
rotate_copy	void	f,f,f,o	Rotation avec recopie
random_shuffle	void	r,r[,g()]	Mélange les éléments
swap_ranges	f	f,f,f	Échange deux séquences

TABLE 5.6 – Algorithmes modifiant une ou plusieurs séquences.

5.5.3 Operations sur les séquences triées

En dehors des algorithmes dont les noms contiennent le mot « `sort` » (tableau 5.7), tous ces modèles de fonctions portent sur des séquences qui sont supposées *triées*.

Nom	Retour	Arguments	Description
sort	void	r,r[,c]	Trie une séquence
stable_sort	void	r,r[,c]	Préserve l'ordre relatif des éléments égaux
partial_sort	void	r,r,r[,c]	Trie une partie de séquence
partial_sort_copy	void	i,i,r,r[,c]	Tri avec recopie pour remplir la deuxième séquence
nth_element	void	r,r,r[,c]	(first,nth,last) Trie jusqu'à ce que l'élément nth soit le bon
binary_search	bool	f,f,V[,c]	Recherche dichotomique
lower_bound	f	f,f,V[,c]	Premier élément d'une sous-séquence d'éléments égaux
upper_bound	f	f,f,V[,c]	Fin d'une sous-séquence d'éléments égaux
equal_range	pair<f,f>	f,f,V[,c]	Intervalle d'éléments égaux
merge	o	i,i,i,i,o[c]	Fusionne deux séquences triées
inplace_merge	void	b,b,b[,c]	Fusionne deux séquences consécutives
partition	b	b,b,p	Partitionne en fonction de p
stable_partition	b	b,b,p	Partitionne et maintient l'ordre relatif

TABLE 5.7 – Algorithmes sur les séquences triées.

Exercice 5.2 *Ecrire un programme qui crée un vecteur de 500000 entiers, initialisé avec des nombres aléatoires à l'aide de l'algorithme* `generate_n<>()`*. Une copie du vecteur sera faite dans un tableau classique. Ensuite, mesurer et comparer les temps d'exécution des opérations suivantes :*

- *tri du vecteur à l'aide de l'algorithme* `sort<>()` *;*
- *tri du tableau à l'aide de la fonction* `qsort()`*.*

5.5.4 Opérations ensemblistes

Ces algorithmes (tableau 5.8) ne sont utilisables qu'avec des conteneurs ordonnés ou des séquences triées.

Nom	Retour	Arguments	Description
includes	bool	i,i,i,i[c]	Verifie l'inclusion de la deuxième séquence dans la première
set_union	o	i,i,i,i,o[,c]	Union ensembliste
set_intersection	o	i,i,i,i,o[,c]	Intersection ensembliste
set_difference	o	i,i,i,i,o[,c]	Différence entre la première et la deuxième séquence
set_symmetric_difference	o	i,i,i,i,o[,c]	Différence symétrique

TABLE 5.8 – Algorithmes ensemblistes.

5.5.5 Opérations sur les tas

Ces algorithmes (tableau 5.9) permettent de manipuler un conteneur séquentiel comme un tas.

Nom	Retour	Arguments	Description
push_heap	void	r,r[,c]	Ajoute au tas
pop_heap	void	r,r[,c]	Retire la tête du tas
make_heap	void	r,r[,c]	Transforme la séquence en tas
sort_heap	void	r,r[,c]	Transforme le tas en séquence triée

TABLE 5.9 – Algorithmes de tas.

5.5.6 Comparaisons et bornes

Nom	Retour	Arguments	Description
min	V	V,V[,c]	Minimum de deux valeurs
max	V	V,V[,c]	Maximum de deux valeurs
min_element	f	f,f[,c]	Minimum d'une séquence (première occurence)
max_element	f	f,f[,c]	Maximum d'une séquence (première coccurence)
lexicographical_compare	bool	i,i,i,i[,c]	Comparaison lexicographique

TABLE 5.10 – Algorithmes sur les bornes.

5.5.7 Permutations

Les deux modèles suivants (tableau 5.11) permettent de générer sur place les permutations lexicographiques qui précèdent ou succèdent à une séquence donnée. Elles retournent le booléen `true` s'il reste des séquences à suivre. Si la séquence donnée en argument est la dernière dans l'ordre, alors la plus petite permutation (toujours au sens lexicographique) est produite et la fonction retourne `false`.

Nom	Retour	Arguments	Description
next_permutation	bool	b,b[,c]	Prochaine permutation
prev_permutation	bool	b,b[,c]	Permutation précédente

Table 5.11 – Algorithmes de permutations.

Exercice 5.3 *Écrire un programme affichant* tous *les mots de 8 lettres qui sont formés des lettres de « ENSICAEN ».*

5.5.8 Algorithmes numériques

Ces algorithmes (tableau 5.12) sont définis dans l'en-tête `<numeric>`.

Nom	Retour	Arguments	Description
accumulate	V	i,i,V[,F2]	Somme d'une valeur et tous les éléments d'une séquence
adjacent_difference	o	i,i[,F2]	Pour la séquence <a,b,c,d>, retourne <a,b-a,c-b,d-c>
inner_product	V	i,i,i,V[,F2,F2]	Produit scalaire entre deux séquences, ou accumulation par une fonction des résultats d'une deuxième fonction appliquée terme à terme.
partial_sum	o	i,i,o[,F2]	Calcule les sommes (ou accumulations) partielles

Table 5.12 – Algorithmes numériques.

5.5.9 C++11 : nouveaux algorithmes

Les nouveaux algorithmes de la bibliothèque standard sont décrits dans le tableau 5.13.

De plus, quelques fonctions utiles basées sur les listes d'initialisation sont définies dans l'en-tête `<algorithm>`. Elles sont décrites par leur prototype dans le listing 5.11 et un exemple d'utilisation de la fonction `min` « étendue » est donné dans le listing 5.12.

5.6 Les types d'objets fonctions

Comme cela apparaît dans la section précédente, la bibliothèque standard fait un usage important des objets fonctions. Des modèles de structures, classes et fonctions sont logiquement définis pour faciliter la manipulation de ce type d'objet.

Nom	Retour	Arguments	Description
all_of	B	i,i,p	Vérifie que p est vrai pour tout élément
any_of	B	i,i,p	Vérifie que p est vrai pour un des éléments
none_of	B	i,i,p	Vérifie que p n'est vrai pour aucun des éléments
find_if_not	i	i,i,p	Trouve le premier élément qui ne satisfait pas p
copy_if	o	i,i,o,p	Copie uniquement les éléments qui vérifient p
copy_n	o	i,n,o	Copie n éléments
move	o	i,i,o	Déplace un séquence
move_backward	o	i,i,o	Déplace un séquence en l'inversant
partition_copy	pair<o,o>	i,i,o,o,p	Partitionne une séquence en deux groupes (vrai/faux) selon un prédicat
partition_point	i	i,i,p	Premier élément d'une partition qui ne verifie pas p
is_sorted	B	i,i[,c]	Vérifie qu'une séquence est triée
is_sorted_until	i	i,i[,c]	Trouve la fin de la plus grande sous-séquence triée
is_heap	B	r,r[,c]	Vérifie qu'une séquence a la structure d'un tas
is_heap_until	i	r,r[,c]	Trouve la fin de la plus grande sous-séquence correspondant à un tas
minmax_element	pair<i,i>	i,i[,c]	Recherche les éléments min et max d'une séquence
minmax	pair<CR,CR>	CR,CR[,c]	Retourne les min et max de deux valeurs
iota		f,f,V	Remplit une séquence avec les valeurs successives commençant à V

TABLE 5.13 – Nouveaux algorithmes de la bibliothèque C++11.

```
1  template <typename T>
2  T min(initializer_list<T> t);
3  template <typename T>
4  T min(initializer_list<T> t, Compare comp);
5
6  template <typename T>
7  T max(initializer_list<T> t);
8  template <typename T>
9  T max(initializer_list<T> t, Compare comp);
10
11 template <typename T>
12 pair<const T &, const T &> minmax(initializer_list<T> t);
13 template <typename T>
14 pair<const T &, const T &>
15 minmax(initializer_list<T> t, Compare comp);
```

Listing 5.11 – Min et max à partir de listes d'initialisation.

```cpp
#include <algorithm>
#include <iostream>

int main() {
   int i, j, k, l;
   std::cin >> i >> j >> k >> l;
   std::cout << std::min({i, j, k, l}) << std::endl;
}
```

Listing 5.12 – Calcul du minimum de 4 variables.

5.6.1 Structures de base pour les classes d'objets fonctions

Des modèles de structures pour des objets fonctions à un ou deux arguments sont prévus dans la bibliothèque standard. Ils servent de bases aux classes *foncteurs* (c.-à-d. classes d'objets fonctions) en définissant simplement des synonymes pour les types des arguments et valeurs de retour.

Le type unary_function

```cpp
template <class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};
```

Le type binary_function

```cpp
template <class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

5.6.2 Exemples

Le modèle de classe d'objets fonctions « `plus` ».

```cpp
template <class T>
struct plus : public binary_function<T, T, T> {
  T operator()(const T & x, const T & y) const {
     return x + y;
  }
};
```

La classe d'objets fonctions « **sinus** ».

```
1 struct sinus : public unary_function<double, double>
2 {
3   double operator()(double x) {
4     return sin(x);
5   }
6 };
```

Un objet fonction accumulateur

Le listing 5.13 donne un exemple d'objet fonction de type « accumulateur » permettant de calculer la somme des éléments sur lesquels il est appliqué.

```
1  #include <iostream>
2  #include <vector>
3
4  template <typename T>
5  class Sum {
6    T value;
7
8  public:
9    Sum(T x = 0) : value(x) {
10   }
11   void operator()(T x) {
12     value += x;
13   }
14   operator T() const {
15     return value;
16   }
17   const T & operator=(const T & x) {
18     value = x;
19   }
20 };
21
22 int main() {
23
24   std::vector<float> v;
25   v.push_back(1.0);
26   v.push_back(0.414);
27
28   Sum<float> s, r;
29   r = for_each(v.begin(), v.end(), s);
30
31   std::cout << "The sum is " << r << std::endl;
32   std::cout << "The sum is not " << s << std::endl;
33 }
```

Listing 5.13 – Calcul de la somme des éléments d'un conteneur à l'aide d'un accumulateur.

5.6.3 Les prédicats

Un certain nombre de modèles de classes d'objets fonctions correspondant à des opérateurs du langage est défini dans l'en-tête `<funtional>`. Ils sont énumérés dans le tableau 5.14.

Type	**Arité**	**Équivalent**
equal_to	Binaire	arg1 == arg2
not_equal_to	Binaire	arg1 != arg2
greater	Binaire	arg1 > arg2
less	Binaire	arg1 < arg2
greater_equal	Binaire	arg1 >= arg2
less_equal	Binaire	arg1 <= arg2
logical_and	Binaire	arg1 && arg2
logical_or	Binaire	arg1 || arg2
logical_not	Unaire	!arg

TABLE 5.14 – Les prédicats.

5.6.4 Opérations arithmétiques

Les modèles d'objets fonctions du tableau 5.15 sont définis dans l'en-tête `<funtional>`.

Type	**Arité**	**Équivalent**
plus	Binaire	arg1 + arg2
minus	Binaire	arg1 - arg2
multiplies	Binaire	arg1 * arg2
divides	Binaire	arg1 / arg2
modulus	Binaire	arg1 % arg2
negate	Unaire	-arg

TABLE 5.15 – Les classes d'objets fonctions pour les opérations arithmétiques de base.

5.6.5 L'éditeur de liaison bind2nd

Ce modèle permet d'utiliser un objet fonction à deux arguments là où on attend une fonction unaire, en précisant une valeur (constante) pour le deuxième argument. On pourra par exemple écrire :

```
1 transform( src.begin(),
2              src.end(),
3              dst.begin(),
4              bind2nd( plus<int>(), 42 ) );
```

Une définition possible de ce modèle est donnée dans le listing 5.14.

```
1  template <typename Operation>
2  class binder2nd
3    : public unary_function<typename Operation::first_argument_type,
4                            typename Operation::result_type> {
5  protected:
6    Operation op;
7    typename Operation::second_argument_type value;
8
9  public:
10   binder2nd(const Operation & x,
11             const typename Operation::second_argument_type & y)
12     : op(x), value(y) {
13   }
14
15   typename Operation::result_type
16   operator()
17   (const typename Operation::first_argument_type & x) const {
18     return op(x, value);
19   }
20 };
21
22 template <typename Operation, typename T>
23 inline binder2nd<Operation>
24 bind2nd(const Operation & op, const T & x) {
25   typedef typename Operation::second_argument_type arg2_type;
26   return binder2nd<Operation>(op, arg2_type(x));
27 }
```

Listing 5.14 – Définition du modèle de fonction `bind2nd`.

```
1 void f(std::list<Shape *> & figure) {
2   for_each(figure.begin(),
3            figure.end(),
4            std::mem_fun(&Shape::draw));
5 }
```

Listing 5.16 – Utilisation de l'appelant de méthode.

5.6.6 L'appelant de méthode mem_fun

Il est courant de devoir appeler une méthode particulière pour tous les objets d'un conteneur. L'algorithme `for_each` rend la chose aisée mais la syntaxe peut devenir fastidieuse car elle oblige à définir une fonction accessoire (cf. listing 5.15).

```
1 void draw(Shape * f) {
2   f->draw();
3 }
4
5 void f(list<Shape *> & lf) {
6   for_each(lf.begin(), lf.end(), draw);
7 }
```

Listing 5.15 – Motivation pour un appelant de méthode.

Une solution élégante consiste à créer un objet temporaire « appelant de méthode » à l'aide des modèles de fonctions `mem_fun` ou `mem_fun_ref` (cf. listing 5.16). L'objet créé par `mem_fun` à partir d'un pointeur de méthode s'applique sur les pointeurs d'objets ; alors que celui qui est produit par le modèle `mem_fun_ref` s'applique à un objet (par référence).

Exercice 5.4 *Réécrire le code du listing 5.16 en utilisant une* lambda fonction *(voir § 7.4).*

5.6.7 Le *composeur* d'objets fonctions

On rappelle le type `unary_function` :

```
1 template <typename Arg, typename Result>
2 struct unary_function {
3   typedef Arg argument_type;
4   typedef Result result_type;
5 };
```

Listing 5.17 – Le modèle de structure `unary_function`.

Le modèle de classe `composer1` est le modèle des foncteurs qui sont la composition de deux autres objets fonctions. La définition de ce modèle ainsi que celle du modèle de fonction `compose1` est donnée dans le listing suivant.

```
1 template <typename OperationF, typename OperationG>
2 class composer1
3   : public unary_function<typename OperationG::argument_type,
```

```
4                                          typename OperationF::result_type> {
5  protected:
6    OperationF f;
7    OperationG g;
8
9  public:
10   composer1(const OperationF & f,
11             const OperationG & g) : f(f), g(g) {
12     // ...
13   }
14
15   typename OperationF::result_type
16   operator()(const typename OperationG::argument_type x) const {
17     return f(g(x));
18   }
19 };
20
21 template <typename OperationF, typename OperationG>
22 inline composer1<OperationF, OperationG>
23 compose1(const OperationF & f, const OperationG & g) {
24   return composer1<OperationF, OperationG>(f, g);
25 }
```

Listing 5.18 – Modèles de classe et de fonction pour la fabrication des objets « fonction composée ».

5.6.8 Le fabriquant d'objets fonctions ptr_fun

Ce modèle permet de construire des objets fonctions à partir d'une fonction classique. Il est par exemple utile pour utiliser des fonctions avec les outils fournis par la bibliothèque standard et qui s'appliquent uniquement sur des foncteurs. Le listing 5.19 donne un exemple d'utilisation pour la composition de fonctions.

5.6.9 C++11 : Les pointeurs de fonction généralisés

Le C++11 étend largement la notion de fonction avec les *lambda functions* mais aussi le type `std::function`. Ces notions, décrites au chapitre 7, rendent potentiellement obsolètes celles qui sont décrites dans les sections 5.6.5 à 5.6.8. Pour le moins, elles en offrent une alternative intéressante.

5.6.10 Les itérateurs d'insertion

La bibliothèque standard définit des itérateurs « améliorés » qui permettent d'appliquer des algorithmes portant sur une séquence *source* et une séquence *destination* et agissant terme à terme dans essentiellement deux situations particulières (la première est en fait un cas particulier de la seconde) :

— la séquence destination est vide ;

```
1  template <typename Arg, typename Result>
2  pointer_to_unary_function<Arg, Result>
3  ptr_fun(Result (*x)(Arg));
4
5  template <typename Arg1, typename Arg2, typename Result>
6  pointer_to_binary_function<Arg1, Arg2, Result>
7  ptr_fun(Result (*x)(Arg1, Arg2));
8
9  vector<double> v(20);
10 // ...
11
12 transform(v.begin(),
13            v.end(),
14            v.begin(),
15            compose1(negate<double>, ptr_fun(fabs)));
```

Listing 5.19 – Usage du modèle de fonction `ptr_fun`

— on souhaite *ajouter* (c.-à-d. insérer) la séquence résultat à la fin d'un conteneur.

En fait, ces itérateurs ne se contentent pas d'offrir le parcours ou l'accès aux éléments des conteneurs auxquels ils sont associés, il sont capables de modifier ces conteneurs en y *insérant* des éléments.

A titre d'exemple, une utilisation possible d'un objet de type `back_insert_iterator<>` est donnée dans le listing 5.20. Ces itérateurs particuliers peuvent être créés simplement à l'aide du modèle de fonction `back_inserter<>()`.

```
1  #include <algorithm>
2  #include <iostream>
3  #include <iterator>
4  #include <list>
5  #include <vector>
6  using namespace std;
7
8  template <typename T>
9  class AddValue {
10   T value;
11
12 public:
13   AddValue(T value) {
14     AddValue<T>::value = value;
15   }
16   T operator()(T x) {
17     return x + value;
18   }
19 };
20
21 int main() {
22   vector<int> v(10);
23   list<int> res;
24   list<int>::iterator i;
25
26   for (int i = 0; i < 10; i++) {
27     v[i] = i;
28   }
29
30   AddValue<int> add_ten(10);
31
32   transform(v.begin(), v.end(), back_inserter(res), add_ten);
33
34   for (i = res.begin(); i != res.end(); i++) {
35     cout << *i << " ";
36   }
37   cout << endl;
38 }
```

Listing 5.20 – Exemple d'utilisation du modèle de classe des itérateurs d'insertion.