Eléments de généricité en C++

Sources

Le langage C++ Bjarne Stroustrup - Pearson Education ed.
Cours de C/C++ Christian Casteyde - www//ChristianCasteyde
Sites : http://www.cplusplus.com/
https://cpp.developpez.com/cours/cpp/

1. LES TEMPLATES

Le terme `template` permet de définir des modèles de structures de données, des modèles de classes et des modèles de fonctions, sur des objets dont le type (la classe d'appartenance) est paramétrable.

1.1. Les modèles de fonction – ou fonctions génériques

```
// Exemple de modèle de fonction qui retourne le max. de 2 éléments
        template <class T>                  // La classe T est un paramètre formel
        T maxi (T& t1, T& t2) {             // 'maxi' est une fonction modèle
                return (t1 > t2 ? t1 : t2);
        }
```

Note La section de code qui suit la déclaration template <class T...> est considérée comme une section de code générique. Ici c'est une fonction, mais ça pourrait être une classe…

Exercice : Quel serait le code d'une fonction générique min qui retourne le min de 2 éléments ?

```
// Exemple d'utilisation de la fonction générique maxi (également dite 'modèle de fonction')
void main () {
        int i=0, j=1; char a='@', b='#'; float x=1.1, y=2.2;

        cout << maxi (i, j) << endl;        // appel de maxi pour T = int comme paramètre effectif
        cout << maxi (a, b) << endl;        // appel de maxi pour T = char comme paramètre effectif
        cout << maxi (x, y) << endl;        // appel de maxi pour T = float comme paramètre effectif
        return (0) ;
}
```

Dans cette fonction générique maxi, la variable t1 du type *formel* T utilise l'opérateur '>'. La fonction maxi n'est donc appelable qu'avec des objets *effectifs* d'un type pour lequel l'opérateur '>' est défini. Ce qui est le cas pour les types int, char et float. L'opérateur '>' est défini sur ces 3 types. C'est pourquoi les trois appels ci-dessus, à la fonction maxi, s'exécutent correctement.

Ainsi, pour une classe Produit, dont les objets auraient un libellé et un prix, le code suivant :

```
Produit prod1 ("table", 100), prod2 ("chaise", 50);
Produit prodSuperieur;
prodSuperieur = maxi (prod1, prod2); // fonctionne ssi l'opération operator> est définie dans Produit
```

ne fonctionne que si le programmeur définit dans la classe Produit l'opération '>' qui porte le nom prédéfini de operator> en C++

A savoir, par exemple :

```
        // Définition d'un opérateur (ex. operator>) dans une classe (ex. Produit)
        bool  Produit::operator> (const Produit& unProduit) const {
                bool resultat;
                resultat = (this->prix   >   unProduit.prix);
                return resultat;
        }
```

Exercice

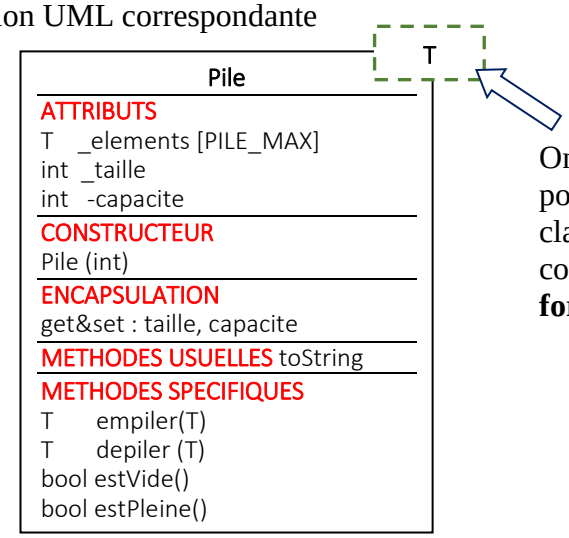
Quel serait le code de l'opérateur Produit::operator+ dont le résultat de l'addition serait un Produit dont le prix serait la somme des deux prix et le libellé serait la concaténation des deux libellé ?

1.2. Les modèles de classe

Exemple de classe template Pile qui gère une pile d'objets d'une classe paramétrable T.

```
#define PILE_MAX 100
template <class T >          // La déclaration qui suit comporte le type T comme paramètre formel
class Pile {
    private: // ATTRIBUTS
        T       _elements [PILE_MAX];
        int  _capacite;   // Maximale possible pour la pile
        int  _taille;       // Actuelle de la pile
    public: // CONSTRUCTEUR
        // Construit une pile d'une capacité de capacite éléments
        Pile (int capacite = PILE_MAX) {
            _taille = 0;    // Actuellement, aucun élément dans la pile
            // Vérification de la capacité maximale demandée lors de la construction
            _capacite = (capacite > PILE_MAX || capacite <=0 ? PILE_MAX : capacite);
        }
    public: // ENCAPSULATION : get&set capacité, taille
    public: // METHODES USUELLES : toString
    public: // METHODES SPECIFIQUES : empiler, dépiler, estVide, estPleine
        // Retourne -1 au format (T) si la pile est pleine, ou bien empile et retourne l'élément empilé
        T empiler (T & t) {
            if (! est_pleine ())
                    return (_elements [_taille++] = t);
            else    return ((T) –1);
        }
        // Retourne -1 au format (T) si la pile est vide, ou bien retourne le sommet et dépile
        T depiler () {
            if (! est_vide ())
                    return (_elements [--_taille]);
            else    return ((T) –1);
        }
        bool estVide () {     // Dit si la pile est vide
            return (_taille == 0);
        }
        bool estPleine () {          // Dit si la pile est pleine
            return (_taille == _capacite);
        }
}
```

Notation UML correspondante

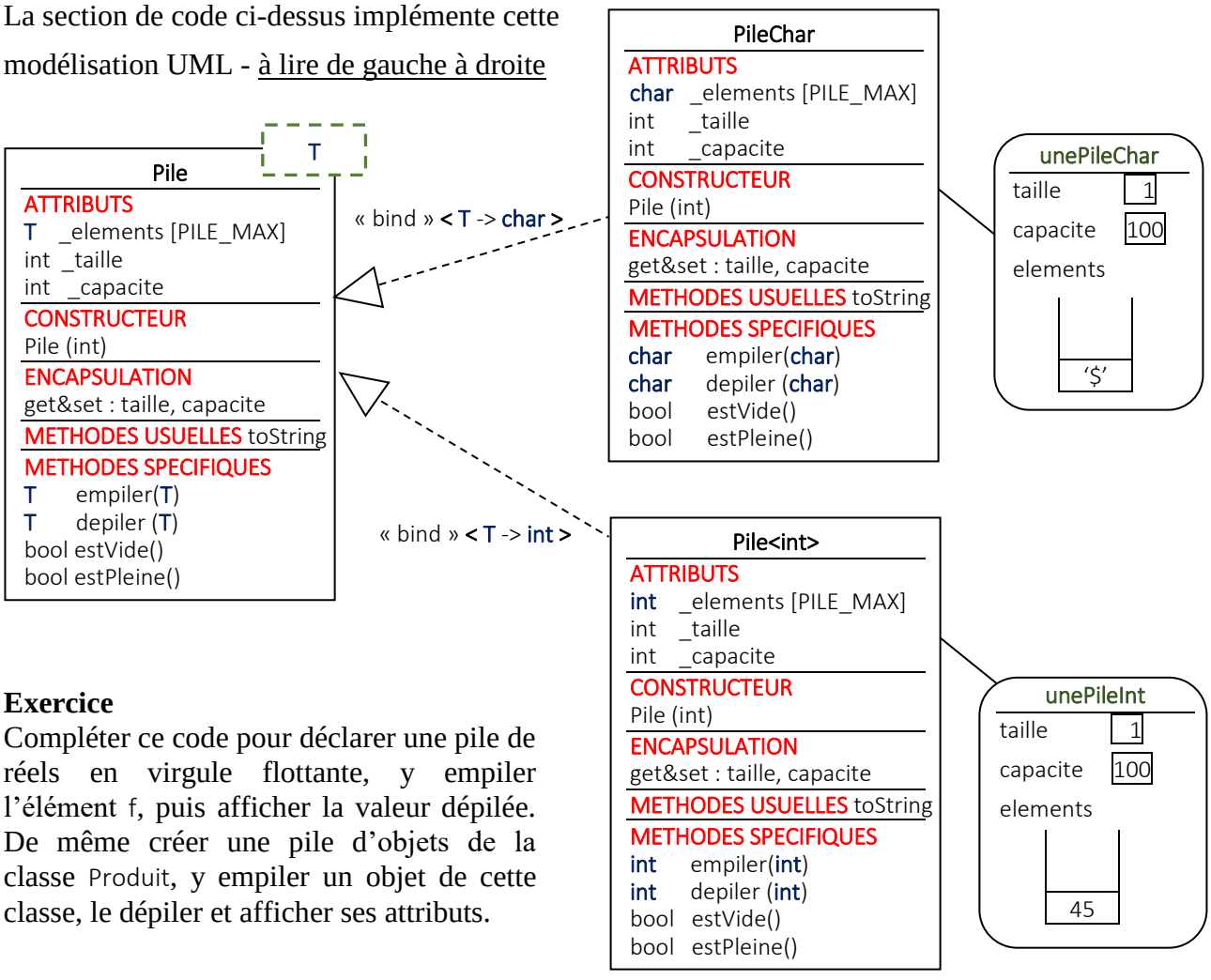


On énumère dans un cadre délimité en pointillés, et à cheval sur l'en-tête de la classe, la **liste des types** qui sont à considérer comme des **paramètres formels** dans la classe.

Exemple de section de code qui utilise cette classe générique Pile.

```
void main () {
    int i=45;   char c='$';   float f=1.1;    // Trois éléments, pour les piles déclarées ci-dessous

    typedef  Pile<char>    PileChar;  // 1. Définit la classe PileChar sur la base du modèle Pile, avec <char>
                                      //    en paramètre effectif, à la place de <T>

    PileChar    unePileChar;          // 2. Crée l'objet unePileChar de cette nouvelle classe Pile<char>

    Pile<int>   unePileInt;           // 1&2 Crée l'objet unePileInt sur la base de Pile, ou T est substitué par int
    // Les objets unePileChar et unePileInt sont des piles qui ont les méthodes et attributs qui ont été
    // définis dans la classe générique Pile
    unePileInt.empiler(i);      cout << unePileInt.depiler()  << endl;
    unePileChar.empiler(c);     cout << unePileChar.depiler() << endl;
}
```

La section de code ci-dessus implémente cette modélisation UML - à lire de gauche à droite



Exercice
Compléter ce code pour déclarer une pile de réels en virgule flottante, y empiler l'élément f, puis afficher la valeur dépilée. De même créer une pile d'objets de la classe Produit, y empiler un objet de cette classe, le dépiler et afficher ses attributs.

1.3. Les modèles de structures de données

L'approche et la syntaxe pour les structures « struct » en C++ est exactement la même que pour les classes. En effet, une structure est la même chose qu'une classe, à ceci près que par défaut les « struct » ont les membres (attributs et méthodes) qui sont public.

Note : en C++ les « struct » admettent des méthodes, ce qui n'est pas le cas en C.

2. LES CONTENEURS

Les structures de données sont appelées conteneur en raison de leur capacité à contenir d'autres objets. La bibliothèque C++ standard définit des classes de conteneurs les plus courants. Ces conteneurs sont définis par l'interface (ensemble de méthodes génériques) qu'ils proposent à l'utilisateur. Deux types de conteneurs sont disponibles, les conteneurs-séquences et les conteneurs-associations.

2.1. Les conteneurs Séquences

Ils stockent des éléments les uns à la suite des autres, leur position et ordre sont donc importants. On trouve les conteneurs :

list : souplesse d'ajout et de retrait des éléments contenus
vector : rapidité d'accès
deque : sorte de tampon dynamique circulaire

Qui plus est, les classes **adaptator** permettent de construire des conteneurs plus spécifiques : des piles, des files, files de priorité (**stack**, **queue**)

2.2. Les conteneurs Associatifs

Ils manipulent les objets au moyen de valeurs qui les identifient directement. Ces identifiants s'appellent des clefs. On trouve des conteneurs :

map : ensemble de couples (***clef***, ***objet***) sans doublon
multimap : ensemble de couples (***clef***, ***objet***) admettant des doublons pour la clef
set : ensemble d'objets (l'***objet*** lui-même sert de ***clef***) sans doublon
multiset : ensemble d'***objets*** admettant des doublons

2.3. La déclaration des conteneurs

Dans la bibliothèque C++, les classes template (i.e. modèles de classes) sont déclarées comme ci-dessous

```
template  <class T, class A = allocator <T>  > // T et A sont deux types qui sont des paramètres formels
class list {
  ...
};
```

list est un modèle de classe, et donc une classe effective dérivée de list, doit préciser :

- le premier paramètre T qui représente le type des éléments que contiendront les listes,
- le deuxième paramètre (facultatif) car affecté à allocator <T> par défaut, représente le mode d'allocation mémoire lors de l'ajout/retrait d'éléments du type T 'instancié' par le programmeur pour les listes qu'il veut gérer.

Ainsi, un exemple classique d'instanciation de la classe générique list est :

```
#include <string>
#include <list>          // Ne pas oublier d'inclure la référence à la classe générique list

using namespace std;

// Définition de la classe effective  ListeDeStrings à partir de la classe générique list
typedef     list<string>     ListeDeStrings;

// Création d'un objet uneListeDePrenoms instance de la classe ListeDeStrings
list<string>   uneListeDePrenoms;
```

Dans la bibliothèque C++, l'en-tête de déclaration des conteneurs est comme ci-dessous :

```
template          <class T, class A = allocator <T> >
class list {
public:
  typedef         T        value_type; // Rebaptise T pour manipuler les éléments en tant que value_type
  typedef value_type*  iterator; // Définit le type iterator pour pointer sur les éléments manipulés
  ...
};
```

```
template          <class T, class A = allocator <T> >
class vector {
  ...
};
```

```
template    <class Key, class T, class Cmp = less <Key>,     // ici 4 paramètres formels de types :
              class A = allocator <pair <const Key, T> > >   // Key, T, Cmp et A, dont 2 de prédéfinis
class map {
          ...
};
```

```
template          <class Key, class Cmp = less <Key>,  class A = allocator <Key> >
class set {
          ...
};
```

La classe list, ainsi que les autres conteneurs, proposent au programmeur un ensemble de méthodes et d'attributs pour manipuler leurs instances.

Par exemple :

- front() : accès au 1er élément
- back() : accès au dernier élément
- push_back() : ajoute en fin de séquence
- pop_back() : supprime le dernier élément
- push_front() : ajoute un nouveau 1er élément
- pop_front() : supprime le premier élément

Il y en a bien d'autres : insert(), erase(), size() …

Par exemple, pour alimenter uneListeDePrenoms on peut utiliser la méthode push_back()

```
// ajouter la string « Marcel » en tant qu'élément de uneListeDePrenoms
uneListeDePrenoms.push_back (« Marcel »); // ~~ça ne fonctionne pas~~ (en fait oui, avec string) !

// il faut passer en paramètre listeDeStrings::value_type(«Marcel») qui
// change le type de «Marcel» en value_type qui est le type attendu par la méthode
uneListeDePrenoms.push_back (ListeDeStrings::value_type (« Marcel »));
...
uneListeDePrenoms.push_back (listeDeStrings::value_type (« Robert »));
cout << uneListeDePrenoms.front() << endl; // affiche 1er élément pointé
```

En fait, la méthode push_back () est définie comme suit dans la classe générique list:

```
void push_back (const& value_type);  // Et  value_type  est issue de  typedef  T  value_type
```

3. LES ITERATEURS

Les conteneurs constituent une abstraction des tableaux pour lesquels on utilise généralement le * et le ++ lors de leur parcours. Pour faciliter le parcours des structures qu'ils hébergent, les conteneurs définissent et proposent au programmeur le type iterator qui permet d'accéder aux éléments contenus.

Parmi les méthodes proposées par les conteneurs, certaines retournent value_type* (cad iterator, ou encore T*), donc des pointeurs sur des éléments hébergés (hébergeables) par le conteneur. Déclaration du type iterator et de méthodes de ce type dans la classe list définie dans la bibliothèque standard de C++

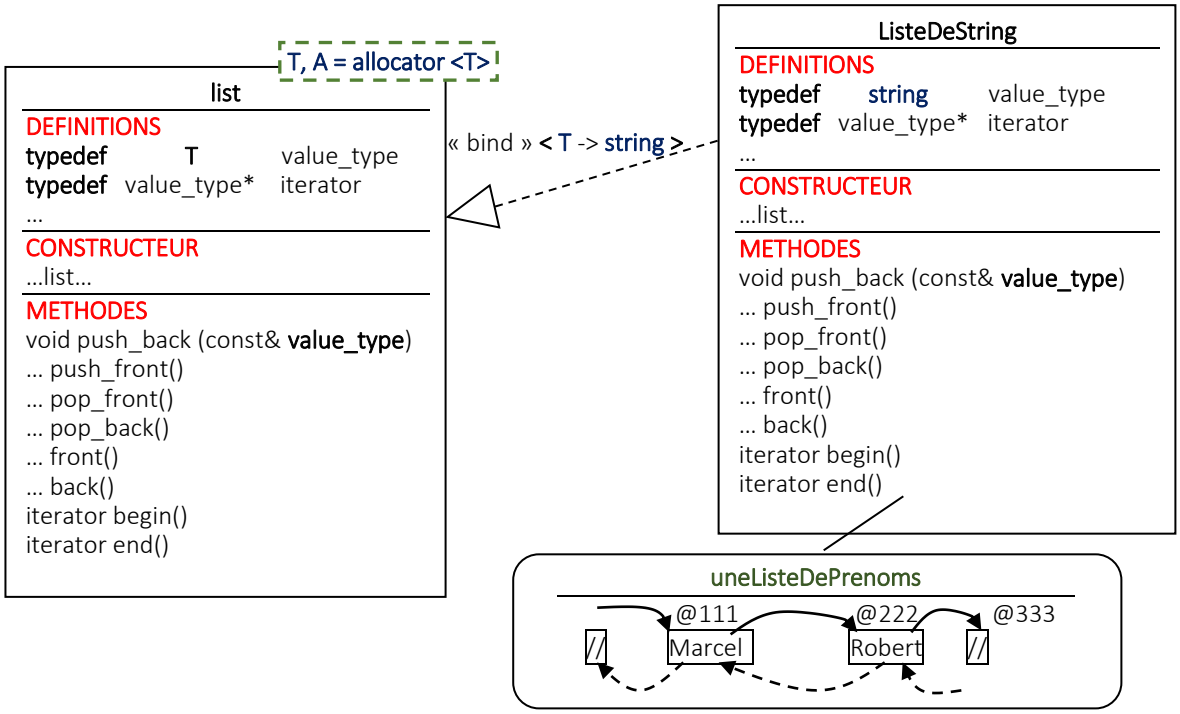
```
template <class T, class A = allocator <T>>
class list {
      public:
      typedef       T              value_type;
      typedef  value_type*  iterator;

      // Pour un parcours normal
      iterator    begin();          // retourne un pointeur sur le 1er élément
      iterator    end();            // retourne un pointeur sur la fin de la liste (attention !)

      // Pour un parcours inverse (cf. reverse)
      iterator    rbegin();         // retourne un pointeur sur le dernier élément
      iterator    rend ();          // retourne un pointeur sur le début de la liste (attention !)
      ...
      // Et bien d'autres méthodes...
};
```

Exemple d'utilisation d'un itérateur pour le parcours complet d'une liste

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

using namespace std;

main () {
        // Définit de classe effective  ListeDeStrings à partir de la classe générique list
        typedef     list<string>  ListeDeStrings;

        // Crée l'objet uneListeDePrenoms instance de la classe ListeDeStrings
        ListeDeStrings  uneListeDePrenoms;

        // Crée unPointeur pour parcourir uneListeDePrenoms du type ListeDeStrings
        ListeDeStrings::iterator   unPointeur; // Peut pointer sur un élément d'une liste

        // Initialise unPointeur avec l'adresse du 1er élément de la liste uneListePrenoms
        unPointeur = uneListePrenoms.begin();

        // Fait un parcours complet de la liste uneListePrenoms
        while (unPointeur != uneListePrenoms.end ()) {
              cout << *unPointeur << endl ;    // Accède à l'élément pointé par unPointeur
              unPointeur++;                    // Incrémente de la taille de l'objet pointé
        }
}
```

La section de code ci-dessus implémente la modélisation UML ci-dessous,

modélisation qui est à lire de gauche à droite :



unPointeur @333 // lorsque unPointeur est initialisé avec uneListeDePrenoms.begin()

Rappels :

- begin() retourne l'adresse ou est stocké le premier élément de la liste.
- end() retourne la fin de la liste, c.a.d. la paroi au fond du tiroir (cf. // dans le schéma ci-dessus) ou sont stockés les éléments.
- Les listes sont doublement chaînées (cf. begin() ⟶ end() et rbegin() - - ▸ rend()).

4. LES ALGORITHMES GENERIQUES

Dans le même esprit de généricité, C++ propose des algorithmes génériques de tri, de réordonnancement, etc. Environ 60 algorithmes classiques et spécialisables sont disponibles.

A titre d'exemples, nous verrons en TDs et TPs les algorithmes génériques :

find(), find_if(), partition() et for_each()

ANNEXES

Les modèles de conteneurs

```
template <class T, class A = allocator <T> >
class vector {
        …
};

template <class T, class A = allocator <T> >
class list {
        …
};

template <class T, class A = allocator <T> >
class deque {
        …
};

template <class T, class A = allocator <T> >
class deque {
        …
};

template <class T, class A = allocator <T> >
class queue {
        …
};

template <class Key, class T, class Cmp = less <Key>, class A = allocator <pair <const Key, T> > >
class map {
        …
};

template <class Key, class T, class Cmp = less <Key>, class A = allocator <pair <const Key, T> > >
class multimap {
        …       // diffère dans son interface (fonctions et types proposés)
};

template <class Key, class Cmp = less <Key>, class A = allocator <Key> >
class set {
        …
};

template <class Key, class Cmp = less <Key>, class A = allocator <Key> >
class multiset {
        …
};
```

Cette section fournit la liste des membres les plus courants pour les conteneurs standards. Pour plus de détails, consulter les en-têtes standards de *<vector>, <list>, <map>,* etc.

Types de membres

value_type	Type d'élément
allocator_type	Type du gestionnaire de mémoire
size_type	Type des indices, des comptes d'éléments, etc.
difference_type	Type de la différence entre deux itérateurs
iterator	Se comporte comme *value_type**
const_iterator	Se comporte comme *const value_type**
reverse_iterator	Parcourt le conteneur en ordre inverse ; comme *value_type**
const_reverse_iterator	Parcourt le conteneur en ordre inverse ; comme *const value_type**
reference	Se comporte comme *value_type&*
const_reference	Se comporte comme *const value_type&*
key_type	Type de clé (uniquement pour les constructeurs associatifs)
mapped_type	Type de *mapped_value* (uniquement pour les conteneurs associatifs)
key_compare	Type de critère de comparaison (uniquement pour les conteneurs associatifs)

Un conteneur peut être considéré comme une séquence organisée soit dans l'ordre défini par l'itérateur du conteneur, soit en ordre inverse. Dans le cas d'un conteneur associatif, l'ordre est basé sur le critère de comparaison du conteneur (< par défaut).

Itérateurs

begin ()	Pointe sur le premier élément
end()	Pointe sur l'élément suivant le dernier élément
rbegin()	Pointe sur le premier élément de la séquence inverse (le dernier)
rend()	Pointe sur l'élément suivant le dernier élément de la séquence inverse (avant le 1er)

Il est possible d'accéder directement à certains éléments :

Accès aux éléments

front()	Premier élément
back()	Dernier élément
[]	Indiçage, accès non contrôlé (pas pour une liste)
at()	Indiçage, accès contrôlé (pas pour une liste)

Les *vector* et *deque* offrent des opérations efficaces qui s'appliquent sur la fin de leur séquence d'éléments. De plus, les *list* et les *deque* fournissent les opérations qui s'appliquent au début de leurs séquences :

Opérations sur les piles et les files d'attente

push_back()	Ajoute en fin de séquence
pop_back()	Supprime le dernier élément
push_front	Ajoute un nouveau premier élément (uniquement pour *list* et *deque*)
pop_front()	Supprime le premier élément (uniquement pour *list* et *deque*)

Les conteneurs fournissent des opérations de liste :

Opérations de liste

insert(p, x)	Ajoute *x* avant *p*
insert(p, n , x)	Ajoute *n* copies de *x* avant *p*
insert(p, first, last)	Ajoute les éléments de *[first..last[* avant *p*
erase(p)	Supprime l'élément en *p*
erase(first, last)	Efface *[first..last[*
clear()	Efface tous les éléments

Tous les conteneurs fournissent des opérations liées au nombre d'éléments ainsi que quelques opérations supplémentaires :

Autres opérations

size()	Nombre d'éléments
empty()	Le conteneur est-il vide ?
max_size()	Taille maximum du conteneur
capacity()	Mémoire allouée pour *vector* (pour *vector* seulement)
reserve()	Réserve de l'espace pour une expansion future (pour *vector*)
resize()	Modifie la taille du conteneur (pour *vector*, *list* et *deque*)
swap()	Echange les éléments de deux conteneurs
get_allocator()	Obtient une copie de l'allocateur du conteneur
==	Le contenu de deux conteneurs est-il identique ?
!=	Le contenu de deux conteneurs est-il différent ?
<	Un conteneur se situe-t-il avant un autre d'un point de vue lexicographique ?

Les conteneurs fournissent divers constructeurs et opérations d'affectation :

Constructeurs, etc.

container()	Conteneur vide
container(n)	*n* éléments la valeur par défaut (pas pour les conteneurs associatifs)
container(n, x)	*n* copies de *x* (pas pour les conteneurs associatifs)
container(first, last)	Eléments initiaux dans *[first..last[*
container(x)	Constructeurs de copie ; éléments initiaux issus du conteneur *x*
~container()	Détruit le conteneur et tous ses éléments

Affectations

operator=(x)	Affectation copie ; éléments du conteneur *x*
assign(n, x)	Affecte *n* copie de *x* (pas pour les conteneurs associatifs)
assign(first, last)	Affecte à partir de *[first..last[*

Les conteneurs associatifs permettent une recherche à partir de clés :

Opérations associatives

operator[](k)	Accède à l'élément de clé *k* (pour les conteneurs avec des clés uniques)
find(k)	Cherche l'élément de clé *k*
lower_bound(k)	Cherche le premier élément de clé *k*
upper_bound(k)	Cherche le premier élément de clé supérieure à *k*
equal_range(k)	Trouve les éléments *lower_bound* et *upper_bound* de clé *k*
key_comp()	Copie de l'objet de comparaison de clé
value_comp()	Copie de l'objet de comparaison des *mapped_value()*

Pour compléter ces opérations, la plupart des conteneurs fournissent quelques opérations spécialisées.

Cette section présente les algorithmes de la bibliothèque standard, sans en donner une description poussée.

Opérations de séquences sans modification	
for_each()	Exécute l'opération sur chaque élément d'une séquence
find()	Recherche la première occurrence d'une valeur dans une séquence
find_if()	Recherche la première correspondance d'un prédicat dans une séquence
find_first_of()	Recherche une valeur provenant d'une séquence dans une autre
adjacent_find()	Recherche une paire adjacente de valeurs
count()	Compte les occurrences d'une valeur dans une séquence
count_if()	Compte les correspondances d'un prédicat dans une séquence
mismatch()	Recherche les premiers éléments pour lesquels deux séquences diffèrent
equal()	Vrai si les éléments de deux séquences sont égaux au niveau des paires
search()	Recherche la première occurrence d'une séquence en tant que sous-séquence
find_end()	Recherche la dernière occurrence d'une séquence en tant que sous-séquence
search_n()	Recherche la nième occurrence d'une valeur dans une séquence

Dans la plupart des algorithmes, l'utilisateur peut spécifier l'action réelle à exécuter pour chaque élément ou paire d'éléments.

Les opérations de séquence avec modification ont très peu de caractéristiques en commun, en dehors du fait évident qu'il leur est possible de modifier les valeurs des éléments d'une séquence.

Opérations de séquence avec modification	
tranform()	Applique une opération sur tous les éléments d'une séquence
copy()	Copie une séquence à partir de son premier élément
copy_backward()	Copie une séquence à partir de son dernier élément
swap()	Echange deux éléments
iter_swap()	Echange deux éléments pointés par des itérations
swap_ranges()	Echange des éléments de deux séquences
replace()	Remplace les éléments par une valeur donnée
replace_if()	Remplace les éléments correspondant à un prédicat
replace_copy()	Copie une séquence en remplaçant les éléments par une valeur donnée
fill()	Remplace chaque élément par une valeur donnée
fill_n()	Remplace les *n* premiers éléments par une valeur donnée
generate()	Remplace tous les éléments par le résultat d'une opération
generate_n()	Remplace les *n* premiers éléments par le résultat d'une opération
remove()	Supprime les éléments ayant une valeur donnée
remove_if()	Supprime les éléments correspondant à un prédicat
remove_copy()	Copie une séquence en suppriment les éléments d'une valeur donnée
remove_copy_if()	Copie une séquence en suppriment les éléments correspondant à un prédicat
unique()	Supprime les éléments contigus égaux
unique_copy()	Copie une séquence en supprimant les éléments contigus égaux
reverse()	Inverse l'ordre des éléments
reverse_copy()	Copie une séquence en ordre inverse
rotate()	Fait « tourner » les éléments
rotate_copy()	Copie une séquence dans une séquence dont les éléments sont « tournés »
random_shuffet()	Déplace les éléments vers une distribution uniforme

La bibliothèque standard fournit diverses opérations permettant le tri, la recherche et la manipulation de séquences à partir d'un classement :

Séquences triées

sort()	Trie avec une bonne efficacité
stable_sort()	Trie en conservant l'ordre des éléments égaux
partial-sort()	Classe la première partie d'une séquence
partial_sort_copy()	Copie en classant la première partie
nth_element()	Place le nème élément à l'emplacement approprié
lower_bound()	Recherche la première occurrence d'une valeur
upper_bound()	Recherche la dernière occurrence d'une valeur
equal_range()	Recherche une sous-séquence d'une valeur donnée
binary_search()	Une valeur donnée se trouve-t-elle dans une séquence triée
merge()	Fusionne deux séquences triées
inplace_merge()	Fusionne deux sous-séquences consécutives triées
partition()	Place en premier les éléments correspondant à un prédicat
stable_partition()	Place en premier le éléments correspondant à un prédicat, tout en préservant l'ordre relatif

Opérations ensemblistes

includes()	True si une séquence est une sous-séquence d'une autre
set_union()	Construit une union triée
set_intersection()	Construit une intersection triée
set_difference()	Construit une séquence d'éléments triés dans la première, mais pas dans la seconde séquence
set_symetric_difference()	Construit une séquence d'éléments triés dans une, mais pas dans les deux séquences

Les opérations sur des tas conservent une séquence dans un éttat qui permet d'en simplifier le tri si nécessaire :

Opérations sur le tas

make_heap()	Prépare une séquence pour être utilisée en tant que segment et mémoire ou de tas
push_heap	Ajoute un élément au tas
pop_heap	Supprime un élément du tas
sort_heap	Trie le tas

La bibliothèque fournit quelques algorithmes permettant la sélection des éléments en fonction d'une comparaison :

Minimum et maximum

min()	La plus petite des deux valeurs
max()	La plus grande des deux valeurs
min_element()	La plus petite valeur dans une séquence
max_element()	La plus grande valeur dans une séquence
lexicographical_compare()	La première des deux séquences d'une point de vue lexicographique

La bibliothèque fournit enfin différentes solutions permettant de permuter une séquence :

Les permutations

next_permutation()	Permutation suivante en fonction de l'ordre lexicographique
prev_permutation()	Permutation précédente en fonction de l'ordre lexicographique