Chapitre VIII

Bibliothèque standard

Sommaire

Introduction

La librairie standard du C_{11}^{++} contient un ensemble de bibliothèque permettant d'enrichir le langage à travers :

- des conteneurs (tableau, liste chainée, pile, ensemble, ...),
- des algorithmes standards et d'itérateurs,
- des fonctions de calcul numérique,
- la gestion des entrée/sortie (voir TP),
- la gestion des chaines de caractères (string) et des expressions régulières,
- des outils systèmes : opération atomique, synchronisations, thread, time
- à partir du C_{17}^{++}, gestion du système de fichiers, algo. parallélisme.

Il est important de connaitre cette bibliothèque :

- évite de réécrire des TDAs classiques,
- évite de réécrire des algorithmes élémentaires,
- permet de ce concentrer sur la conception en se basant sur des objets et des algorithmes robustes (=sans bug), et souvent largement optimisé.

1 Concept de conteneur

1.1 Définition

Un conteneur est une classe qui :

- représente une collection/suite d'éléments,
- stocké sous une forme propre (chaque type de conteneur correspond à un modèle de stockage de données).

Il y a trois catégories de conteneurs proposées par la STL :

- **Les conteneurs de séquence :** collection d'éléments dans lequel chacun de ses éléments a une position qui dépend du moment et de la place d'insertion.
- **Les conteneurs associatifs :** collection d'éléments dans lequel sa position dépends de sa valeur ou d'une clef associé, et d'un critère prédéfini de tri.
- **Les adaptateurs de conteneur :** variante basée sur un conteneur de séquence ou associatif qui restreint l'interface à des fins de simplicité et/ou de clarté.

1.2 Prérequis pour les éléments d'un conteneur

Les conteneurs de la STL sont des templates.

En conséquence, les éléments stockés dans un conteneur doivent respecter les trois propriétés fondamentales suivantes :

1. un élément doit être copiable ou déplaçable à travers son constructeur.
2. un élément doit être assignable (par copie ou déplacement).
3. un élément doit être destructible.

à savoir :

1. L'élément doit explicitement ou implicitement fournir un constructeur par copie ou par déplacement. La copie générée doit être équivalente à la source (*i.e.* comparable avec ==, et le comportement de la copie est identique à la source).
2. Les conteneurs et/ou les algorithmes utilisent l'opérateur d'assignation = pour copier les nouveaux éléments sur les anciens.
3. Les conteneurs doivent détruire les copies internes des éléments quand ces éléments sont enlevés des conteneurs. Le destructeur ne doit pas lever d'exception.

Ces comportement sont généralement disponibles pour toutes les classes, sauf comportements particuliers.
Autres propriétés nécessaires pour un conteneur :

- pour certaines méthodes des conteneurs de séquence, le **constructeur par défaut** doit être disponible (exemple : création d'un conteneur non vide ou accroissement du nombre d'éléments),
- le **test d'égalité** == doit être disponible (par exemple pour les recherche),
- pour les conteneurs associatifs, un **critère de tri** doit être fourni. Par défaut, c'est l'opérateur < qui est appelé par le functor `less<>` utilisé.
- pour les conteneurs non ordonnés, une **fonction de hachage** (par défaut `std::hash`) et un critère d'équivalence doit être fourni pour les éléments (par défaut, le functor `std::equal_to`).

1.3 Sémantiques

a) Sémantique valeur

Sémantique valeur : tous les conteneurs contiennent des copies internes de leurs éléments, et retournent des copies de ces éléments.

Sens :

- Les éléments contenus dans un conteneur sont égaux, mais pas identiques aux objets qui y ont été placés.
- Si un objet du conteneur est modifié, la copie dans le conteneur est modifiée et non l'objet original.

A noter que la sémantique de déplacement apporté par le C_{11}^{++} permet de déplacer les références externes de l'objet original dans le conteneur.

Remarques :

- la copie d'un élément est simple, mais peut conduire à de mauvaises performances (voir ne pas être possible dans certains cas),
- un même objet ne peut pas être en même temps dans plusieurs conteneurs différents.

Les conteneurs de la STL ne fournissent que la sémantique valeur.

b) Sémantique référence

Sémantique référence : correspond au cas où le conteneur contient des références aux objets qu'il contient.

Remarques :

- copier une référence est toujours rapide,
- les références sont plus difficiles à gérer : une référence peut faire référence à un objet qui existe déjà, cas des références circulaires.
- un objet peut être dans plusieurs conteneur à la fois.

Les deux approches sont utiles en pratique :

- copies indépendantes des données originales,
- copies qui font toujours références aux données originales.

Solution :

- utiliser des pointeurs intelligents (`shared_ptr`),
- la classe template `std::reference_wrapper<T>` permet de construire des conteneurs contenant des références.

1.4 Itérateur

a) Définitions

Un **itérateurs** est une interface normalisée implémentant l'accès aux éléments d'un container. Il se comporte un peu comme un pointeur :

- il est utilisé pour "pointer" sur un élément du conteneur (il peut être converti en pointeur sur l'élément en cours d'itération),
- l'opérateur de déréférencement * permet d'accéder à l'objet "pointé" par l'itérateur,
- itérer l'itérateur avec ++ passe à l'élément suivant de l'ensemble (version préfixe),
- la méthode `begin` (resp. `end`) sur le conteneur retourne l'itérateur sur le premier élément (resp. l'élément qui suit le dernier élément).

Garantie multipasse : si `a` et `b` sont deux itérateurs, alors :

- `a==b` signifie que, s'ils sont tous les deux déréférençables, alors ils pointent sur le même objet, et qu'alors `++a == ++b`,
- l'assignation à travers un itérateur mutable (i.e. `*a=val`) n'invalide pas l'itérateur,
- incrémenter la copie d'un itérateur n'incrémente pas l'itérateur original.

b) Itérateur et pointeur

Remarque : sur un tableau C++ classique, un pointeur est un itérateur :

```
class tab {
private: int  n, *t;
public: tab(int p) : n(p), t(new int[p]) {};
  typedef int* iterator;
    iterator begin() { return t; };
    iterator end() { return t+n; }; // EOT
}
tab      T(10);
for(tab::iterator i=T.begin(); i != T.end(); ++i)
    std::cout << *i << std::endl;
```

Ici les opérateurs ++ et * découlent de l'algèbre des pointeurs.

Sinon, l'itérateur est spécifiquement adaptée au conteneur.

Conséquences : il sera possible d'utiliser toutes les fonctionnelles de la STL qui prennent en paramètre un itérateur avec un tableau `t` classique, en passant :

- `t` à la place de l'itérateur sur le début du conteneur,
- `t+n` (où `n` est la taille du tableau) à la place de l'itérateur sur l'élément qui suit le dernier élément).
- l'algèbre des pointeurs s'occupant des opérateurs ++ et *.
- **Attention :** opérateur < non disponible sur tous les types d'itérateur.

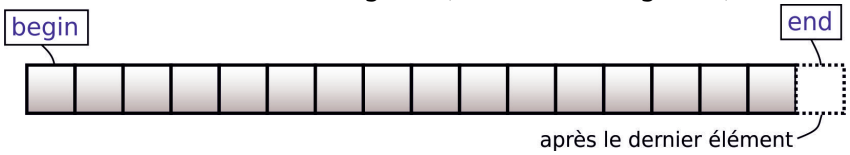
c) Classes d'itérateurs

La typologie des itérateurs est la suivante :

- `InputIterator` : comparable, itérable, déférençable
  itérateur qui peut lire l'élément pointé.
  peut seulement se déplacer vers l'avant (i.e. du début à la fin du conteneur),
  permet de traverser le conteneur une seule fois.
- `OutputIterator` : idem `InputIterator` mais en écriture, l'itérateur est alors dit *mutable*.
- `ForwardIterator` = `InputIterator` + `OutputIterator` + Garantie multipasse.
- `BidirectionalIterator` = `ForwardIterator` + déplacement dans les deux directions (= incrément + décrément),
- `RandomAccessIterator` = `BidirectionalIterator` + accès à tout élément du container en temps constant.
- `ContiguousIterator` = `RandomAccessIterator` + stockage contigu des éléments dans l'itérateur (C_{17}^{++}).

Pour tout conteneur, au moins un itérateur a été défini :

- `ForwardIterator` = itérable ++ et multi-passe.
  obtenu avec les méthodes `begin()`, `end()` ou `cbegin()`, `cend()` (sur un conteneur constant).



- `ReverseIterator` = itérable ++ (=reculer) et multi-passe.
  obtenu avec les méthodes `rbegin()`, `rend()` ou `crbegin()`, `crend()` (sur un conteneur constant).

- `BidirectionalIterator` = Forward + Reverse
- `RandomAccessIterator` = Bidirectional + décalable (`a+n`/`a-n`), directement accessible `a[n]`, position comparable (`a<b`,`b>a`), décalage calculable (`a-b`).
- `ContiguousIterator` = `RandomAccessIterator` + itérateur pointeur direct sur l'élément + éléments du conteneur stockés de façon contiguë dans la mémoire (C_{17}^{++}).

d) Opération sur un itérateur

Les fonctions suivantes permettent de :

- `operator++` : passe à l'élément suivant (Forward=avance, Reverse=recule, Bidirectionnal=avance)
- `operator--` : passe à l'élément précédent (Bidirectionel=recule)
- `operator<` : compare le position entre deux itérateurs (**RandomAccess**)
- `advance(it,p)` : déplace l'opérateur de p (peut être négatif).
- `distance(it1,it2)` : retourne la distance entre les deux itérateurs (it1 < it2 sinon résultat non défini).
- `itprev = prev(it)` : retourne le prédécesseur de l'itérateur (bidirectionnel requis),
- `itnext = next(it)` : retourne le successeur de l'itérateur.

Prérequis : inclure `<iterator>`

Applications :

- dans une liste simplement chainée, vérifier la valeur à la position suivante,

```
auto pos = coll.begin();
while (pos != coll.end() && std::next(pos) != coll.end())
    { ...  ++pos;  }
```

- **attention :** éviter les expressions du type `++coll.begin()` qui peuvent ne pas compiler (les remplacer par `std::next(coll.begin())`).

1.5 Notion d'allocateur

Définition : un allocateur est un objet :

- destiné à allouer et libérer de la mémoire,
- auquel la gestion de la mémoire est déléguée.

Or, une bonne stratégie de gestion mémoire est toujours :

- **compacité :** limiter le nombre d'allocation/desallocation.
- **localité :** allouer à proximité les objets utilisés en même temps.

En C++, un allocateur est une méthode template qui doit implémenter essentiellement les 4 méthodes suivantes :

- `allocate<T>` : alloue de la place de stockage pour n objets de type `T` (sans constructeur),
- `deallocate<T>` : désalloue une place de stockage précédemment allouée.
- `construct<T>` : construction en place d'un objet dans un espace déjà alloué.
- `destroy<T>` : destruction en place d'un objet (sans désallouer l'espace qu'il occupe).

La bibliothèque standard propose trois allocateurs :

- `std::allocator` : qui est l'allocateur standard (celui utilisé par défaut).
  il revient à effectuer de l'allocation en place et à utiliser `new`/`delete` d'une manière thread-safe.
- `std::scoped_allocator_adaptor` : allocateur multi-niveaux (vecteur d'ensemble de listes de tuples de maps, ...).
  constitué d'un allocateur externe (pour le conteneur de niveau supérieur) et d'une suite d'allocateurs internes (pour l'allocation associée à chaque niveau de stockage du conteneur).
- `std::pmr::polymorphic_allocator` (C^{++}_{17}) allocateur dont le comportement de l'allocateur dépend de la ressource mémoire avec laquelle il est utilisé.

Utilisation : inclure `<memory>`

Sauf cas particulier (par exemple, si l'on souhaite le partage de mémoire entre plusieurs conteneur différent), l'allocateur par défaut (`std::allocator`) est suffisant pour la majorité des applications.

Exemple : d'utilisation de l'allocateur (hors d'un conteneur)

```
std::allocator<std::string> a;
std::string* s = a.allocate(2);
a.construct(s, "foo");
a.construct(s + 1, "bar");
std::cout << s[0] << ' ' << s[1] << '\n';
a.destroy(s);
a.destroy(s + 1);
a.deallocate(s, 2);
```

En conséquence,

- un allocateur peut facilement être utilisée dans la conception de tout objet souhaitant faire de l'allocation en place.
- dans le cas général, on préfèrera utiliser directement le conteneur de la STL le plus approprié.
- comme indiqué dans le cours sur les allocations mémoires, les allocateurs de `boost` peuvent être utilisé comme une bonne alternative dans le cadre d'optimisation.

1.6 Utilisation des conteneurs de la STL

Pourquoi utiliser les conteneurs de la STL :

- Ces classes sont **génériques** (templates) : elles peuvent être utilisées pour stocker des éléments de tout type.
- Ils utilisent une **implémentation** parmi les plus **efficaces** connues pour stocker leurs éléments (*i.e.* vous n'écrirez probablement pas mieux),
- Ils n'ont pas besoin d'être mise au point, et ne contiennent pas de bogue.
- Ils implémentent toutes les opérations usuelles que l'on pourrait attendre pour manipuler les containers des différents types,
- Ils utilisent des interfaces normalisée (nommés **itérateurs**) qui implémentent le parcours des éléments d'un conteneur.

Vous êtes fortement encouragés à les utiliser mais vous devez :

- connaître leurs caractéristiques afin de choisir le conteneur adapté au type d'accès dont vous avez besoin,
- avoir une idée de l'implémentation interne afin d'être certain que vous les manipulez correctement.

1.7 Spécification d'un conteneur de la STL

Dans la STL, un conteneur est toujours supposé avoir les caractéristiques suivantes :

Spécifications d'un conteneur :

- **constructeurs** (par défaut, constructeur par copie et par déplacement) et **destructeur**.
- **assignation** par copie et par déplacement
- **méthode :** `empty()`, `size()`, `max_size()`
- **opérateur :** == et `!=`.
- **swap :** méthode `c1.swap(c2)` et fonction `swap(c1,c2)`
- **itérateur :** `begin()`/`end()` , `cbegin()`/`cend()`

Spécifications d'un itérateur sur un conteneur

- **opérateur *** retourne l'élément à la position courante,
- **operateur ++** passe à l'élément suivante (`--` si l'itérateur est bidirectionnel),
- **opérateur ==** (et `!=`) indique si deux itérateurs sont à la même position,
- **opérateur =** pour assigner un itérateur à la même position.

2 Conteneurs de la STL

2.1 Conteneurs de séquence

Un conteneurs de séquence stocke des objets de même type sous forme d'un arrangement linéaire :

Les conteneurs de séquence proposés par la STL sont :

- `array` : encapsule un tableau statique contigu, accès aléatoire.
- `vector` : encapsule un tableau dynamique contigu, accès aléatoire.
- `deque` : représente une liste dynamique permettant une insertion rapide au début et à la fin de la liste (stockage non contigu), accès aléatoire.
- `list` : représente une liste dynamique permettant une insertion rapide n'importe où dans la liste, accès séquentiel bidirectionnel (liste doublement chainée).
- `forward_list` : idem `list` mais la liste est simplement chainée (accès séquentiel monodirectionnel).

Critères de choix :

	`array`	`vector`	`deque`	`list`	`forward_list`
Insertion/Destruction au début	n/a	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Insertion/Destruction à la fin	n/a	$O(1)$	$O(1)$	$O(1)$	n/a
Insertion/Destruction au milieu	n/a	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Accès au premier élément	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Accès au dernier élément	$O(1)$	$O(1)$	$O(1)$	$O(1)$	n/a
Accès à un élément quelconque	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Surcoût	aucun	faible	moyen	haut	haut

Méthodes associées :

- Insertion/Destruction au début : $O(1)$=`push_front` / `pop_front`, $O(n)$=impl
- Insertion/Destruction à la fin : `push_back` / `pop_back`.
- Insertion/Destruction au milieu : $O(1)$=`insert`/`erase`, $O(n)$=impl.
- Accès au premier/dernier élément : `front` / `back`
- Accès à un élément quelconque : en $O(1)$: `[]`, en $O(n)$: itérer.
- Assignation : `assign` pour affecter le conteneur après initialisation.
- `size` : nombre d'éléments dans le conteneur.
- `emplace` : insertion/remplacement en place (arguments transmis par réf. universelle au constructeur).

a) `array`

Définition : `template<class T, std::size_t N> struct array;`

Propriétés :

- encapsule un tableau statique (= de taille constante et déterminée à la compilation) avec une taille et une efficacité équivalente,
- offre les avantages d'un conteneur standard,
- ne peut être ni redimensionné, ni réalloué.

Méthode :

- `[i]`/`at(i)` : accès aux $i^{\text{ème}}$ élément en lecture/écriture : direct / avec test des bornes (exception `std::out_of_range`).
- `data()` : retourne le pointeur vers le stockage sous-jacent.
- `fill(v)` : rempli le conteneur avec une valeur `v` unique.

Exemple :

```
std::array<int, 3>  a{ {1,2,3} };
// utilisation comme un tableau statique classique
a[0] = a[2] + 4;
std::cout << "Taille=" << a.size() << std::endl();
```

Caractéristiques : collection ordonnée (par l'indice), accès aléatoire,

Propriétés supplémentaires :

- pas de surcout par rapport à un tableau statique standard.

- pas de support d'allocateur (utilise le stack)
- stack = si le nombre d'éléments est connu, meilleure performance qu'avec un tableau dynamique.
- seul conteneur qui ne crée pas un conteneur vide :
  - nécessite l'existence d'un constructeur par défaut,
  - les éléments sont initialisés par défaut si rien n'est passé pour les initialiser.
- fournit l'interface tuple (voir cette partie, à considérer pour représenter un tuple d'éléments de même type).
- tous les algorithmes de la STL peuvent être utilisés

Notes sur les opérations :

- `swap` : comme les pointeurs ne peuvent pas être permutée, la sémantique de swap s'effectue élément par élément (temps linéaire).
- `v1=v2` : copie les éléments de v2 dans v1.
- `v1=std::move(v2)` : déplace les éléments du premier tableau dans le second.
- itérateur : accès aléatoire (constant, non constant, inverse, inverse constant)

Exception

- il n'est ni possible d'insérer ou ni d'effacer des éléments.
- les exceptions peuvent seulement avoir lieu lors d'une copie, d'un déplacement ou d'une assignation.
- `swap` peut lancer une exception car un swap entre deux éléments peut lancer une exception.

Exemples :

```
// éléments de x non initialisés
std::array<int,4> x1;
// initialisation uniforme: éléments de x initialisés à 0
std::array<int,4> x2 = {};
// C++11 initialisation partielle uniforme (x4[1..4]=0)
std::array<int,5> x3 = { 42 };
// initialisation standard sur tableau statique possible
std::array<int,5> x4 = { 42, 12, 54, 76, 88 };
```

```
// pas de constructeur avec liste d'initialisation
// std::array<int,2> x5({ 42, 88 }); // non autorisé
// std::array<int,2> x5{{ 42, 88 }}; // ok
```

```
// copie
std::array<string>  s1 = {"un","deux","trois"}, s2, s3;
s2 = s1;
// s2 = {"un","deux","trois"}
s3 = move(s1);
// s3 = {"un","deux","trois"}, s1 = {"","",""}
```

b) `vector`

Définition :

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

Propriétés :

- encapsule un tableau dynamique de mémoire séquentielle (=contigüe),

- la mémoire dans un vecteur est caractérisée par sa taille (`size()` = nombre d'éléments actuellement stocké dans le vecteur) et sa capacité (`capacity()` = nombre d'éléments pouvant être stocké dans le vecteur).
- si la capacité maximale du vecteur est atteinte, alors le stockage est réalloué, et les données copiées dans le nouvel emplacement (lent).
  Les itérateurs, les pointeurs et les références sont invalidés.
- insertion/suppression rapide ($O(1)$) en fin de vecteur avec `push_back`, `pop_back`, `emplace_back`)
  plus lent si besoin de réallouer le stockage sous-jacent.
- insertion/suppression lente ($O(n)$) au début/milieu avec `insert`, `emplace`, `erase` (déplacement de tous les éléments derrière le point d'insertion)
- la mémoire du vecteur est automatiquement libérée en fin de portée.

Prérequis : inclure <vector>
Constructeurs : d'un `vector<T>`

- `vector()` : par défaut (size=0,cap.=0) - standard.
- `vector(n)` : `n` éléments initialisés par défaut (size=`n`,cap.=`n`).
- `vector(n,val)` : `n` éléments initialisés par une `val` de type `T` (size=`n`,cap.=`n`).
- `vector(first,last)` : à partir de l'itérateur d'un conteneur de type T.
- `vector({...})` : par liste d'initialisation.
- constructeur par copie et par déplacement.

Insertion/suppression : d'un `vector<T>`

- `push_back(val)` : ajoute un élément à la fin (par copie/déplacement)
- `emplace_back(args)` : construit l'élément `T(args)` à la fin.
- `pop_back()` : détruit le dernier élément.
- `insert(pos,...)` : insertion à la position `pos` avec mêmes types d'arguments que le constructeur.
- `emplace(pos,args)` : insertion en place à la position `pos` avec `T(args)`.
- `erase(pos)`/`erase(first,last)` : efface l'élément `pos` ou l'intervalle `[first,last)` indiqué.
- `clear()` : supprime tous les éléments du vecteur, capacité inchangée.

Gestion de la capacité :

- lorsque le vecteur grandi, la réallocation a lieu : 1-4 : à chaque fois, à partir de 5 : en allouant 50% de mémoire en plus (VC14).
- lorsque la taille d'un vecteur diminue, la capacité n'est pas modifiée.

Conséquences : si la taille du vecteur est connue à sa déclaration, alors la place nécessaire pour contenir les éléments doit être réservée.

Exemple : si un vecteur est allouée par défaut, puis 10 valeurs sont insérées avec `push_back`, alors cela génère (VC14) 35 copies par déplacement/copie et autant de destruction, en plus des 10 constructeurs nécessaires.

Gestion : d'un `vector<T>`

- `reserve(n)` : augmente si nécessaire la capacité à `n`,
- `shrink_to_fit()` : réduit la taille du vecteur à ce qui est nécessaire (implique un réallocation).

- `resize(n,val)`, `resize(n)` : si la taille est inférieure, alors le vecteur est tronqué à `n` éléments, sinon ajoute le nombre d'éléments nécessaires (initialisé à `val` si fourni, par défaut sinon) pour arriver à `n`.
  réallocation seulement si nécessaire.

Exemple :

```
#include <vector>
class T {
public: int a{}, b{};
        T() {};
        T(int u, int v) : a(u), b(v) {};
        T(const T& t) = default;
};
```

```
vector<T>  v{ {1,2},{3,4} };
v.push_back(T(5, 6));
v.emplace_back(7, 8);
// ici v={ {1,2}, {3,4}, {5,6}, {7,8} }
// accès par itérateur
for (std::vector<T>::const_iterator i = v.begin();
    i != v.end(); ++i)
    std::cout << i - v.begin() << ":"
            << i->a << "/" << i->b << std::endl;
// même accès par indice
for (size_t i = 0; i < v.size(); i++)
    std::cout << i << ":" <<
          v[i].a << "/" << v[i].b << std::endl;
// suppression du dernier élément
v.pop_back();
```

Notes sur les opérations :

- si le vecteur est construit avec une taille déterminée, les éléments sont initialisés avec le constructeur par défaut.
  pour les types fondamentaux, l'initialisation à zéro est garantie.
- `reserve` alloue la mémoire nécessaire mais ne construit rien dans les éléments réservés mais non utilisés.
- si la mémoire allouée est supérieure à `reserve(p)`, la commande n'a aucun effet.
- `shrink_to_fit()` n'apporte pas la garantie `size=capacity`.
- `assign` permet d'assigner le vecteur : `assign(n,val)` pour affecter n éléments à val, `assign(ibegin, iend)` affecter l'intervalle définit par les itérateurs `{ibegin, iend}`, `assign(ilist)` affecter avec une liste d'initialisation.
- `swap` transfère aussi les itérateurs, pointeurs et références

```
std::vector<int> v1({ 1, 2, 3 }), v2({ 4, 5, 6 });
auto it1 = v1.begin();
v2.swap(v1);
std::cout << *it1 << std::endl; // it1 itérateur sur v2
```

- `move` : même remarque sur le vecteur déplacé.

Hypothèses : pour le type `T` du template

H_d le destructeur ne lance pas d'exception,

H_{cm} les opérations de copie/déplacement (par construction ou assignation) ne lancent pas d'ex-

ception,

H_{nt} les opérations de copie/déplacement (par construction/assignation) ne lance **jamais** d'exception (exemple : les PODs).

Garantie sur les exceptions :

- Toutes les garanties sur les exceptions ne tiennent que si H_d est vérifiée.
- `pop_back`, `swap` et `clear` ne lancent pas d'exception,
- si une exception est levée lors d'un `push_back`, la fonction n'a pas d'effet,
- sous l'hypothèse H_{cm} : `insert`, `erase`, `emplace`, `emplace_back`, `push_back` soit réussissent, soit ne produisent pas d'effet.
- sous l'hypothèse H_{nt}, toute opération soit réussie, soit ne produit pas de résultat.

`vector<bool>` est une version optimisée pour stocker 1 bit par élément du vecteur.

Conséquence :

- pas d'itérateur avec accés aléatoire
- les algorithmes STL peuvent ne pas fonctionner,
- les performances sont moins bonnes qu'un vecteur de bool s'il avait implémenté avec 1 octet par bool.

Autres opérations fournies :

- `flip()` : complément de tous les bits,
- `c[i]` : accès au $i^{\text{ème}}$ bit en lecture/écriture.
- `c[i].flip()` : complément du $i^{\text{ème}}$ bit.

Remarques :

- pas de garantie que les bits consécutifs soit stocké consécutivement dans la mémoire.
- si la taille du tableau est fixe, `std::bitset` est plus adapté et fourni plus d'opérateurs binaires.

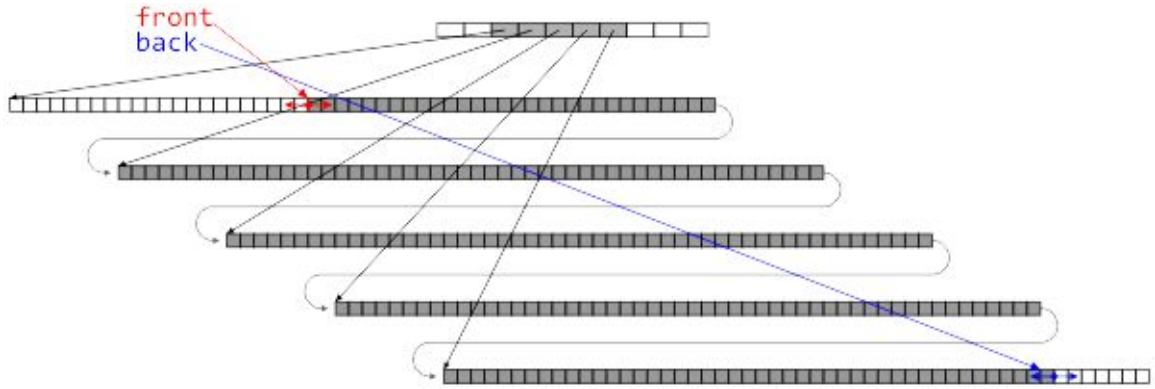
c) deque

Définition :

```
template<class T,class A=allocator<T>> class deque;
```

Propriétés :

- encapsule une file d'attente à double queue qui permet une insertion/suppression rapide en tête et en fin de la queue.
- le stockage n'est pas contigu : l'implémentation utilise en général une table de pointeurs sur des blocs de tailles identiques stockant les données. La table des pointeurs est extensible dans les deux sens.



- L'accès aux éléments est efficace (les blocs étant de tailles N identiques, un accès aléatoire est possible : l'élément i est à la position $i \mod N$ du bloc i/N ; pour VC14 : $N = 4$).
- Itérateur à accès aléatoire.

Globalement, même interface que `vector`.

Prérequis : inclure <deque>

Constructeurs : voir `vector`

Insertion/suppression : d'un `vector<T>`

- `push_front(val)` / `push_back(val)` : ajout (rapide) d'un élément au début/fin (par copie/-déplacement)
- `emplace_front(args)` / `emplace_back(args)` : construction (rapide) de l'élément `T(args)` au début/fin.
- `pop_front()`/`pop_back()` : destruction (rapide) du premier/dernier élément.
- `insert`, `emplace`, `erase` : idem `vector`.
- `clear()` : supprime tous les éléments du `deque`, capacité inchangée.

Accès : `at`, `[]`, `front`, `back`.

l'indice 0 est toujours celui du premier élément (même après `push_front`).

Accès : `size()`

Gestion mémoire : `shrink_to_fit`, `resize`

pas de `reserve` car coût d'extension faible.

Propriétés : (par rapport à `vector`)

- L'accès à un élément utilise une indirection de plus : accès aux éléments et itérateurs un peu plus lent.
- L'augmentation de taille plus efficace que `vector` (car pas de réallocation ni de copie).
- Comme pour `vector`, l'insertion/suppression au milieu est lente.
- Pas de contrôle de capacité.
- Le conteneur peut libérer les blocs mémoires qui ne sont plus utilisés.
  pas de garantie sur l'automaticité, `shrink_to_fit()` pour forcer.
- `insert` et `erase` peuvent causer une réallocation de la table des blocs.
  conséquence :
  - ◇ **en début/fin :** invalide les itérateurs
  - ◇ **au milieu :** invalide en plus pointeurs et références.

Garantie des exceptions :

- Même hypothèse que `vector` avec les modifications ci-dessous.
- `push_back`/`push_front` en cas d'exception, pas d'insertion.
- `pop_back`/`pop_front` ne levent pas d'exception.

Exemple :

```
const size_t nInsert = 10;
std::deque<int>   d;
for (size_t i = 0; i < nInsert; i++) {
    int v = ui(re);    // tirage aléatoire
    if (v % 2) d.push_back(v);
    else d.push_front(v);
}

// v={2 36 2 72 24 56 35 39 11 51}
cout << v.front(); // 2
cout << v.back();  // 51
v.pop_front();
v.pop_back();
cout << v.front(); // 36
cout << v.back();  // 11
```

d) `list`

Définition :

```
template<class T,class A=allocator<T>> class list;
```

Propriétés :

- permet de faire de l'insertion/suppression à temps constant partout dans le conteneur.
- pas d'accès à aléatoire (accès en $O(n)$).
- généralement implémentée comme une liste doublement chainée.
- itérateur invalidé seulement si l'élément courant est détruit.

Opérations : spécifiques aux listes

- `merge` : pour fusionner deux listes triées par ordre croissant.
- `splice` : insertion d'une liste dans une autre.
- `remove` : retire tous les éléments de valeur `val` de la liste,
- `remove_if` : retire tous les éléments vérifiant le prédicat `pred` de la liste,
- `reverse` : inverse l'ordre des éléments.
- `unique` : supprimer les éléments successifs identiques.
- `sort` : tri les éléments de la liste.

Prérequis : inclure `<list>`

Constructeurs : idem `vector`

Insertion/suppression : d'un `list<T>`

- `push_front(val)` / `push_back(val)` : ajout (rapide) d'un élément au début/fin (par copie/-déplacement)

- `emplace_front(args)` / `emplace_back(args)` : construction (rapide) de l'élément `T(args)` au début/fin.
- `pop_front()`/`pop_back()` : destruction (rapide) du premier/dernier élément.
- `insert`, `emplace` : insertion rapide (avant l'itérateur au point d'insertion)
- `erase` : suppression rapide (à la position de l'itérateur)
- `clear()` : supprime tous les éléments de la `list`.

Accès : `front`, `back`, itérateur pour accéder aux autres éléments (en $O(n)$).

Exemple :

```
list<int> list1, list2;
for (int i = 0; i < 6; ++i) {
    list1.push_back(i);
    list2.push_front(i);
}
// list1={0,1,2,3,4,5}
// list2={5,4,3,2,1,0}
auto pos = list1.begin(); ++pos; ++pos;
// list1={0,1,<2>,3,4,5}, <.> position de pos
list1.splice(pos, list2);
// list1={0 1 5 4 3 2 1 0 2 3 4 5}
list1.sort();
// list1={0 0 1 1 2 2 3 3 4 4 5 5}
list1.unique();
// list1={0 1 2 3 4 5}
list1.remove_if([](int v) { return (v % 2 == 0); });
// list1={1 3 5}
list1.reverse();
// list1={5 3 1}
```

```
list<int> list3 = { 1, 3, 7, 15 };
list<int> list4 = { 2, 3, 9, 20 };
list3.merge(list4);
// list1={1 2 3 3 7 9 15 20}
```

Remarques :

- L'insertion et la suppression d'élément n'invalident ni les itérateurs, ni les pointeurs, ni les références (sauf s'il font référence à l'élément supprimé).
- Pas de méthode de gestion mémoire, car pas de structure sous-jacente de type bloc.
- `at()` et `[]` non disponibles (afin de décourager les utilisations inadéquates)

Garantie des exceptions :

- `list` offre la meilleure sécurité des exceptions de la STL.
- la quasi-totalité des opérations offre une garantie forte (succès ou no-op)
- `pop_*`, `erase`, `clear`, `splice`, `reverse`, `swap` ne lancent pas d'exception.
- `push_*`, `insert`, `resize` réussissent ou n'ont pas d'effet.
- `merge`, `remove`, `remove_if`, `unique` donne une garantie forte sous la condition que l'opérateur utilisé pour comparer les éléments de lance pas d'exception.
- `sort` offre un garantie de base.

e) forward_list

Définition : `template<class T,class A=allocator<T>> class forward_list;`

idem `list` sauf que la liste est simplement chainée, et n'offre pas de surcout par rapport à l'implémentation en `C`.

Conséquences :
- l'insertion qu'en début ou après l'itérateur,
- parcours uniquement avec un forward itérator,
- `size()` n'est pas disponible (conséquence choix implémentation)
- pas de pointeur vers le dernier élément, donc par d'opération sur la fin de la liste.
- offre les mêmes garanties que `list`.

Prérequis : inclure `<forward_list>`

Opérations :
- `push_front(val)` / `emplace_front(args)` `pop_front()` (début),
- `insert_after` / `emplace_after` / `erase_after` (après l'itérateur).
- `remove`, `remove_if` : retirer les éléments qui ont une certaine valeur ou qui vérifient une condition particulière.

2.2 associatifs ordonnés

a) set/multiset

Caractéristiques :
- un conteneur associatif est un conteneur qui trie ses éléments automatiquement à partir d'un critère.
- il permet un accès direct pour stocker ou lire des éléments à travers une clef (ou clef de recherche).
- les clefs sont utilisés pour stocker les éléments de manière ordonnée dans le conteneur,

Les conteneurs associatifs de type ensemble ont deux formes associées :
- `multiset` : autorise qu'un même élément soit inséré plusieurs fois,
- `set` : chaque élément n'est présent qu'une seule fois dans l'ensemble.

Arguments du template :

`class T, class C=less<T>, class A=allocator<T>`
- `T` : type de l'élément contenu dans le conteneur.
- `C` : clef de comparaison (par défaut, functor effectuant l'opération <).
- `A` : allocateur utilisé pour l'allocation des éléments.

Utilisation : include `<set>`.

Le critère de tri doit définir un relation d'ordre R stricte faible, à savoir :
- **antisymétrie :** si $R(x, y)$ est vraie, alors $R(y, x)$ est fausse,
- **transitivité :** si $R(x, y)$ et $R(y, z)$ sont vraies, alors $R(x, z)$ est vraie.
- **irréflexivité :** $R(x, x)$ est fausse.
- **transitivité de l'équivalence :** si $R(x, y)$, $R(y, z)$, $R(z, y)$ et $R(y, x)$ sont faux, alors $R(x, z)$ et $R(z, x)$ sont faux.

Conséquences sur le critère de tri :

- $<$ est un relation d'ordre stricte faible.
- $\leq$ n'est pas une relation d'ordre stricte faible (irréflexivité), donc non utilisable pour un `set`/`multiset`.
- si $R(x, y)$ et $R(y, x)$ sont faux, alors $x = y$ (critère d'équivalence, utilisé si pour tester l'équivalence)

Conséquences sur l'ordre des éléments dans le conteneur :

- pour `set`, le critère d'équivalence est utilisé pour tester si des éléments éléments sont identiques.
- pour `multiset`, l'ordre dans lequel sont stockés les éléments identiques est aléatoire, mais stable (*i.e.* préservation de l'ordre relatif des éléments identiques, sens précisé par la suite).

Remarques :

- les set/multiset sont habituellement implémentés comme des arbres binaires équilibrés (non spécifiée dans le standard, mais conséquence de la complexité spécifiée).
- l'implémentation la plus courante utilise un arbre bicolore.
- le tri automatique impose des contraintes importantes :
  - ◇ éléments non modifiables directement sinon compromet leur ordre :
    nécessite de supprimer l'ancienne valeur et d'insérer un nouvel élément avec la nouvelle valeur.
  - ◇ pas d'accès direct aux éléments,
  - ◇ l'accès indirect s'effectue à travers un itérateur.
    du point de vue de l'itérateur : la valeur de l'élément est constante.

Définition du critère de tri : trois façons différentes :

- **par défaut :** dans ce cas, c'est `std::less<T>` qui est utilisé, et qui suppose que l'opérateur $<$ est surchargé pour le type `T` (en version externe).
- **comme paramètre du template :**
  Exemple : `std:set<int,std::greater<int>> my_set;`
  dans ce cas, le critère de recherche fait partie du type du conteneur.
  il est le type du functor, dont une instance (la fonction-objet) sera utilisée pour comparer les éléments.
  Ce type est déterminé à la compilation.
- **comme paramètre du constructeur :**
  dans ce cas, le critère de recherche est le dernier paramètre passé après les valeurs utilisées pour initialiser le conteneur.
  il est de type fonction-objet (c'est l'instance d'un functor). Il remplace celui qui est défini dans le type.
  Ce fonction-object peut être changé à l'exécution.

Une fois l'objet construit, le critère de tri ne peut plus être changé.

Note : même fonctionnement pour l'allocateur.

Exemple :

```
class A {
protected: int x,y;
public:
  struct LessA {
    bool operator<(const A& u, const A& v)
         { return(u.x < v.x); }
  };
  struct GreaterA {
    bool operator<(const A& u, const A& v)
         { return(u.x > v.x); }
  };
  friend bool operator<(const A& u, const A& v)
    { return u.y < v.y; }
};
```

```
// critère de tri par défaut: < du type A
std::set<A>           a1;
// critère de tri dans le type de l'ensemble (type functor)
std::set<A,A::LessA>  a2;
// critère de tri dans le constructor (object-function)
A::GreaterA   plusgrand;
set::set<A>           a3(plusgrand);
```

Constructeurs : (critère et allocateur peuvent être ajouté après)

- par défaut : construit un ensemble vide
- par copie et par déplacement à partir d'un autre set/multiset du même type.
- à partir de l'intervalle `[ibegin,iend)` issu d'un autre conteneur (possiblement de nature différente, mais de type identique) ou de tout objet pouvant engendrer un itérateur (entrée standard, fichier, ...).
- à partir d'une liste d'initialisation.

Comparaisons : de deux set/multiset de même type T.

- `c1==c2` signifie que les deux ensembles `c1` et `c2` ont les mêmes éléments (multiset avec le même nombre de répétitions).
  L'opérateur == pour le type T doit obligatoirement être défini afin de pouvoir comparer deux éléments (différent de la clef de tri).
  `c1!=c2` est évalué comme `!(c1==c2)`.
- `c1<c2` utilise l'ordre lexicographique.
  L'opérateur < pour le type T doit être défini (différent de la clef de tri).
  `c1>c2` est évalué comme `(c2<c1)`, `c1<=c2` comme `!(c2<c1)`, et `c1>=c2` comme `!(c1<c2)`.

Itérateurs : bidirectionnel (begin/end), bidirectionnel constant, inverse, inverse constant.

Opérations de recherche :

- `count(v)` : compte le nombre d'éléments de l'ensemble avec la valeur `v`.
- `find(v)` : retourne un itérateur sur le premier élément égal à `v` trouvé (ou `end()` sinon)
- `lower_bound(v)` (resp. `upper_bound(v)`) : retourne la première (resp. dernière) position où `v` serait inséré, *i.e.* premier élément $\geq$ `v` (resp. $>$ `v`).
- `equal_range(v)` retourne une `std::pair` d'itérateurs correspondant à l'intervalle sur lequel les éléments sont égaux à `v`.

Note : Ces fonctions sont à privilégier lors des recherches, plutôt que les fonctions génériques.
Exemple :

```
set<int> c={1,2,4,5,6};
// *c.lower_bound(3) = 4,       *c.lower_bound(5) = 5,
// *c.upper_bound(3) = 4,       *c.upper_bound(5) = 6,
// *c.equal_range(3).first = 4  *c.equal_range(5).first = 5
// *c.equal_range(3).second = 4 *c.equal_range(5).second = 6
```

Assignation : en plus des assignations habituelles, et du `swap` ont été ajoutés l'assignation avec une liste d'initialisation.
Exemple : `c = {2,4,1,3};` est valide avec un set/multiset.

Insertion et suppression d'éléments

- `insert(v)` : insertion de la valeur `v` (version alternative avec un itérateur en plus positionné si possible juste avant la position d'insertion).
  version `emplace`/`emplace_int` où sont passés les arguments du constructeur de l'élément à construire.
- `insert(ibegin,iend)` : insertion à partir d'un intervalle défini par un couple d'itérateur (autre conteneur, fichier, ...)
- `insert(initlist)` : insertion à partir d'une liste d'initialisation.
- `erase(v)` : efface la valeur `v` ; retourne le nombre de valeurs effacées.
- `erase(pos)` : efface l'élément à la position définie par l'itérateur `pos`.
- `erase(ibegin,iend)` : efface l'intervalle défini par le couple d'itérateur `(ibegin,iend)` sur le conteneur courant.
- `clear()` : vide le conteneur.

Attention : un effacement rend invalide tout itérateur sur l'élément effacé (runtime error si un tel itérateur est utilisé).

Exemple :

```
std::multiset<int> a = { 8, 2, 6, 4, 4, 2, 8, 10 };
// génère un runtime-error
for (auto i = a.begin(); i != a.end(); ++i)
    if (*i == 2) a.erase(i);
// solution C++11: erase retourne l'itérateur itéré
for (auto i = a.begin(); i != a.end();)
    if (*i == 2) i = a.erase(i); else ++i;
```

Valeur de retour : lors de l'insertion d'un élément dans un `set`, retourne un `pair<it,bool>` où `it` est la position de l'insertion, et `bool` est vrai si l'élément a été inséré.

Remarque : (C_{11}^{++})

Pour les multisets, toutes les fonctions d'insertion et de suppression préservent l'ordre relatif des éléments.

En particulier, une insertion, s'effectue toujours à la fin des valeurs équivalentes déjà existante.

Un conteneur associatif est basé sur une structure sous-jacente basée sur des nœuds.

Gestion des exceptions :

- tout échec de construction laisse le conteneur tel qu'il était avant la construction.
- comme les destructeurs ne lancent pas d'exception, la suppression d'un nœuds à partir d'itérateurs ne lance pas d'exception.
- pour une insertion multi-éléments, la récupération après le lancement d'une exception n'est pas praticable.
  conséquence : à implémenter comme une succession d'insertions simples, et stocker les itérators renvoyés sur la position d'insertion, afin de pouvoir au besoin effacer les éléments insérés avec succès avant la levée de l'exception.
- la suppression par valeur ne lance pas d'exception, sous l'hypothèse que le critère de recherche ne lance pas d'exception.
- l'assignation, la copie ou `swap` peuvent lancer des exceptions.

Exemple :

```
set<int> s = { 2, 1, 4, 2, 5, 6 };
// s={1 2 4 5 6}
int n1 = s.count(3);  // n1 = 0
int n2 = s.count(2);  // n2 = 1
auto p1 = s.insert(3);
// s={1 2 <3> 4 5 6}, <.> position p.second
// p.first = true
auto p2 = s.insert(2);
// s={1 <2> 3 4 5 6}, <.> position p.second
// p.first = false
int n3 = s.erase(2);  // n3 = 1
```

```
vector<int>   v = { 4, 8, 12, 16 };
s.insert(v.begin(), v.end());
// s={1 3 4 5 6 8 12 16}
```

b) `map/multimap`

Caractéristiques d'un `map/multimap` :

- les capacités et les opérations sur un `map/multimap` sont essentiellement les mêmes que sur `set/multiset`.
- les éléments d'un `map/multimap` sont des paires clef/valeur.
  - ◇ le tri dans le conteneur se fait toujours en fonction de la clef,
  - ◇ la valeur n'a aucun rôle dans le conteneur que pour but d'être associée à la clef et stockée.
- la clef est externe à la valeur et associée à la valeur dans le conteneur (`set/multiset` : le tri s'effectue à partir d'une fonction de comparaison sur le conteneur).
- les `map/multimap` peuvent être utilisées de manières associatives.

Les conteneurs associatifs de type application (map) ont deux formes associées :

- `multimap` : autorise qu'une clef soit insérée plusieurs fois,
- `map` : chaque clef n'est présente qu'une seule fois dans l'ensemble (indépendamment de la valeur associée).

Arguments du template :

`class K, class T, class C=less<K>, class A=allocator<T>`

- `K` : type de la clef.
- `T` : type de la valeur contenue dans le conteneur.
- `C` : critère de tri (par défaut, functor effectuant l'opération <).
- `A` : allocateur utilisé pour l'allocation des éléments.

Conditions nécessaires :

- les types `K` et `T` doivent être copiable et déplaçable.
- le critère de tri `C` doit pouvoir comparer des éléments de type `T`, et doit définir un ordre faible strict.
  même type de définition que sur un `set`/`multiset`.

Utilisation : include <map>.

Remarques : Le type des paires d'éléments est `std::pair<const K,C>`

Conséquences sur le fonctionnement du conteneur :

- une clef n'est plus modifiable après création,
- une valeur est modifiable après insertion dans le conteneur.

Conséquence sur la manipulation des éléments du conteneur :

- `std::pair` est l'encapsulation dans une structure d'une paire générique.
  `first` (resp. `second`) est la valeur du premier (resp. second) élément de la paire ; `first_type` (resp. `second_type`) idem avec le type instancié pour la paire manipulée.
- une paire de ce type se crée de deux manières :
  - ◇ avec le constructeur `std::pair<K,T>(k,v)`
  - ◇ avec la fonction `make_pair(k,v)` qui construit la paire à partir du type déduit de `k` et `v`.

  Note : syntaxe possible pour liste d'initialisation `{{k1,v1},{k2,v2}}`.
- `std::pair<K,T>(k,v)::value_type` retourne aussi le type de la paire.
- pour les (λ-) fonctions, un argument de type `std::pair<const K,C>&` permet de passer un élément du conteneur et de me modifier sa valeur.

Exemple :

```
map<float, int> map1;
map1[4.2f] = 69;
map1[8.5f] = 199;
map1[0.4f] = 50;
// map1 = {(0.4,50) (4.2,69) (8.5,199)}
auto p = map1.find(8.5f);
// map1 = {(0.4,50) (4.2,69) <(8.5,199)>} <.> position p
```

Opérations constantes :
`empty()`, `size()` : idem autres conteneurs.

Opérations de comparaison : idem `set`/`multiset`
exige la surcharge de == et < sur le type `T`.

Itérateur :
idem `set`/`multiset`

Opérations de recherche :
idem `set`/`multiset`

Assignation :
idem `set`/`multiset`

Insertion et suppression :
idem `set`/`multiset`

Gestion d'exceptions :
idem `set`/`multiset`

Utilisation de `map` comme un tableau associatif :

Dans le cas de `map`, la clef est unique et externe à la valeur stockée.

Elle peut donc être utilisée afin d'accéder à la valeur à partir de clef.

Deux méthodes permettent ce comportement sur un conteneur `c` :

- `c.at(k)` retourne la valeur `v` associé à la clef `k` (i.e. la paire `(k,v)` existe dans le conteneur). si la clef `k` n'existe pas, l'exception `out_of_range` est lancée.
- `c[k]` a deux actions possibles :
  - ◇ si la clef `k` existe dans le conteneur, alors retourne une référence vers la valeur `v` associée (lecture/écriture).
  - ◇ sinon, créé l'entrée <`k,T{}`>, et retourne une référence vers la valeur `v` créée (lecture/écriture).

L'insertion conditionnelle devient également possible (C_{17}^{++}) :

`try_emplace(K,args)` = si la clef existe déjà, ne fait rien, sinon effectue un `emplace`.

Attention : fonctionnalité non disponible sur un `multimap` (pas de valeur unique associée à une clef).

2.3 associatifs non ordonnés

Mettre dans cette partie :

- `unordered_set`/`unordered_multiset`
- `unordered_map`/`unordered_multimap`

Implémentation :

- utilise une fonction de hashage pour un accès direct aux éléments.
- structure interne dépendant du conteneur.

voir la documentation.

2.4 adaptateurs de conteneur

Les adapteurs de conteneur sont des interfaces spécialisées utilisant un conteneur standard pour stocker les données :

- `stack` : implémentation d'une pile FILO
  interface spécialisée : top(), pop() et push(val)
  conteneur sous-jacent :
  - ◇ par défaut : `deque`
  - ◇ doit implémenter : `back`, `push_back`, `pop_back`
- `queue` : implémentation d'une pile FIFO
  interface spécialisée : top(), pop() et push(val)
  conteneur sous-jacent :
  - ◇ par défaut : `deque`
  - ◇ doit implémenter : `back`, `front`, `push_back`, `pop_back`)
- `priority_queue` : implémentation d'une file de priorité
  interface spécialisée : top(), pop() et push(val)
  conteneur sous-jacent :
  - ◇ par défaut : `vector`
  - ◇ doit implémenter : `front`, `push_back`, `pop_back`

2.5 Remarques générales

Assignation et swap :

- **assignation par copie :** l'assignation entre deux conteneurs retire tous les anciens éléments du conteneur de destination, et y copie tous les élements du conteneur source,
  ⇒ l'opération est relativement couteuse (complexité linéaire)
- **assignation par déplacement :** depuis C_{11}^{++}, tous les conteneurs implémentent l'assignation par déplacement, qui swappe les pointeurs en mémoire plutôt que de copier les valeurs.
  ceci n'existe *a priori* que pour les rvalues, mais il est possible de transformer une lvalue en xvalue grace à l'opérateur std : :move. Attention, dans ce cas, le contenu de conteneur déplacé est indéfini dans la norme (*i.e.* dépend de l'implémentation : le contenu peut être vide, être le contenu du conteneur vers lequel on a déplacé, ...)
- **swap :** permet de permuter le contenu de deux conteneurs (implémentés pour tout conteneur). compléxité constante (excepté pour `array<>`).

Exemple :

```
std::vector<int> v1 = {1,4,5,7,8}, v2, v3;
v2 = v1;                // assignation par copie
v3 = std::move(v1);     // déplacement v1->v3, v1 indéfini
v3 = std::swap(v2,v1); // échange contenu de v2 et v1
```

Comparaisons :

- **cas des conteneurs non ordonnés** (= unordered set, multiset, map et multimap)
  seules les comparaisons == et != sont définies comme : si tout élément d'un conteneur est également inclus dans l'autre conteneur, indépendamment de l'ordre.
- **cas des conteneurs ordonnés** : les comparaisons ==, !=, <, <=, >, >= sont définies.
  Les règles permettant de comparer deux conteneurs sont les suivantes :

- ◇ les deux conteneurs doivent avoir le même type,
- ◇ deux conteneurs sont égaux si leurs éléments sont égaux et qu'ils sont dans le même ordre (l'opérateur == est utilisé entre les éléments),
- ◇ l'opérateur < entre deux conteneurs correspond à l'ordre lexicographique.

Exemple :

```
std::vector<int> v1 = {1,2,3}, v2 = v1, v3 = {1,3,4};
// ici: (v1 == v2) vrai
// ici: (v1 < v3) vrai
```

Opérations par bloc sur un conteneur :

- toujours préférer les méthodes qui travaillent sur les intervalles plutôt que les méthodes sur des éléments. Soit `itFirst` et `itLast`, le premier et le dernier élément d'un intervalle d'un conteneur c.
  - ◇ **construction par intervalle :** `list<int> data(itFirst,itLast)`
  - ◇ **insertion par intervalle :** `c2.insert(position,itFirst,itLast)`
  - ◇ **effacement par intervalle :** `c.erase(itFirst,itLast)` (note : `itFirst` et `itLast` sont des itérateurs de `c`).
  - ◇ **assignation par intervalle :** `c2.assign(itFirst,itLast)`

  est toujours plus rapide que d'effectuer la même opération en la répétant sur chaque élément de l'intervalle (s'il faut réallouer, la réallocation sera déjà faite à la taille de la totalité de l'intervalle ; s'il faut déplacer, un seul déplacement est toujours meilleurs que n).
- à partir du C_{11}^{++}, on peut utiliser des `intializer_list` à la place d'un intervalle pour obtenir le même résultat pour les constructions, les insertions et les assignations.
  Exemple : `std::set<int> myset({2,4,8,16,7,12});`

Itération sur un conteneur :

- toujours préférer les itérateurs préfixes `++i` et `--i`,
- en C_{11}^{++}, profiter de `auto` pour écrire plus facilement le parcours d'un conteneur :

```
for(auto pos=cell.begin(); pos!=cell.end(); ++pos) {
    // utiliser *pos pour accéder à l'élément
}
```

  si `cell` est de type `list<char>`, il aurait fallu écrire `list<char>::const_iterator`.
- en C_{11}^{++}, si la totalité du conteneur doit être parcouru, utiliser plutôt la version "range-based" de for, à savoir :

```
std::list<int>  cell({4,7,5,3});
for(const int &v : cell) { /* v = élément de cell */ }
```

  On remarquera que l'on obtient directement ici une référence constante vers un élément de `cell`.
  On peut modifier le mode d'accès à l'élément de manière intuitive en modifiant le type de la variable de boucle :
  - ◇ `const T&` : accès par référence constante (conteneur non modifié).
  - ◇ `T&` : accès par référence (valeur dans le conteneur modifiable).
  - ◇ `T` : accès par valeur (la variable de boucle est une copie par valeur).

3 Adaptateurs d'itérateur

Tout ce qui se comporte comme un itérateur est un itérateur (rappel : c'est une abstraction pure).

Il est donc possible d'écrire des classes qui ont l'interface d'un itérateur mais qui font quelque chose de complètement différent. C'est le cas des adaptateurs d'itérateur.

Il y a quatre types d'adaptateur d'itérateurs :

- les **itérateurs d'insertion** : itérateur qui fonctionne en mode insertion (à la place de la réécriture).
- les **itérateurs de flux** : itérateur permettant de lire ou d'écrire dans un flux,
- les **itérateurs inverse** (reverse iterator ; déjà traité),
- les **itérateurs de déplacement** : itérateur permettant de convertir l'accès à un élément en un déplacement.

3.1 Itérateur d'insertion

Par défaut, l'itérateur parcourt les éléments déjà existant dans le conteneur.

Problème : si l'itérateur est utilisé alors qu'il est en train d'écrire de nouveaux éléments, et que la taille du conteneur est insuffisante, provoque une erreur.

Trois types de conteneur d'insertion sont définis :

- `front_insert_iterator` : itérateur effectuant l'insertion au début du conteneur (utilise `push_front`),
- `insert_iterator` : itérateur effectuant insertion à l'endroit où pointe l'itérateur (utilise `insert`),
- `back_insert_iterator` : itérateur effectuant insertion à la fin du conteneur (utilise `push_back`) (construit avec un `back_inserter`.

Un itérateur d'insertion redéfinit l'interface de la manière suivante :

- si une valeur est affecté à l'itérateur (*i.e.* `*i = v`), alors elle insère l'élément à la position associé au type de l'itérateur.
- les opérateurs * et ++ ne modifient pas l'itérateur (et retour l'itérateur).

Utilisation : inclure <iterator>

Création des itérateurs : pour un conteneur ContainerObject, l'itérateur sur ce conteneur est obtenu avec :

- `front_inserter(ContainerObject)` pour obtenir un itérateur d'insertion avant.
- `inserter(ContainerObject,iPos)` pour un itérateur d'insertion après l'endroit où se trouve l'itérateur `iPos`.
- `back_inserter(ContainerObject)` pour obtenir un itérateur d'insertion arrière.

Exemple :

```
std::vector<int>  v = {5,8,12,3,16,24};
std::fill_n(v.begin(), 3, 1);
// v = {1,1,1,3,16,24}
std::fill_n(back_inserter(v), 3, 2);
// v = {1,1,1,3,16,24,2,2,2}
```

3.2 Itérateur de déplacement

Un **itérateurs de déplacement** permet de convertir l'accès à un élément en un déplacement (i.e. le déréférencement de l'itérateur provoque la conversion de l'objet retourné en référence sur une `rvalue`).

Utilisation : permet le déplacement d'un élément d'un conteneur dans un autre lors d'une opération de construction ou lors d'un algorithme (l'opérateur d'assignation copie = devient une assignation par déplacement). Le type utilisé doit avoir son assignation/constructeur par déplacement défini (sinon copie).

Déclaration :

- `move_iterator<T>(iPos)` retourne l'itérateur de déplacement à partir d'un itérateur standard sur un itérateur de type `T`.
  exemple : `move_iterator<vector<int>::iterator>(v.begin())`
- `make_move_iterator(iPos)` retourne l'itérateur de déplacement directement construit à partir du type de l'itérateur.
  exemple : `make_move_iterator(v.begin())`

Exemple :

```
vector<string>   words({ "patin", "arbre", "chat" }),dico;
auto iStart = make_move_iterator(words.begin()),
     iEnd = make_move_iterator(words.end());
for (auto iPos = iStart; iPos != iEnd; ++iPos)
    dico.push_back(*iPos);
```

3.3 Itérateur de flux

Un itérateur de flux a pour but de transformer :

- un flux d'entrée (lecture) en un itérateur permettant de lire à chaque appel à l'opérateur ++ l'élément suivant du flux,
- un flux de sortie (écriture) en un itérateur permettant d'écrire à chaque affectation de l'itérateur (déréférencé ou non) par un élément l'écrit dans le flux.

Il existe 2 types d'itérateurs de flux chacun avec les versions d'entrée et de sortie :

- les itérateurs sur un `basic_stream` (= tous les streams d'E/S standard)
  - ◇ `istream_iterator` : itérateur sur un flux d'entrée standard.
  - ◇ `ostream_iterator` : itérateur sur un flux de sortie standard.

  typiquement contruit à partir d'un `iostream` (i.e. surcharge « ou »)
- les itérateurs sur un `basic_streambuf` (= tous les streams de caractères basé sur un buffer)
  - ◇ `istreambuf_iterator` : itérateur sur un buffer d'entrée.
  - ◇ `ostreambuf_iterator` : itérateur sur un buffer de sortie.

  typiquement construit à partir d'un string, mais également compatible avec les `iostream`s. Cette interface est la sous-couche de `iostream` (plus bas niveau pour manipulation de `char`),

Utilisation :

- dans tous les cas, le paramètre du template doit être spécifié le type d'objet lu/envoyé à l'itérateur

- pour les `stream_iterator`, l'itérateur est initialisé avec un stream classique (cin, cout, ifstream, ofstream, ... = le flux d'entrée/sortie dans lequel on lit/écrit).
- pour les `streambuf_iterator`, l'itérateur est le plus souvent initialisé avec un objet de type `stringstream` qui permet de transformer une chaine de caractères en stream.
  par nature, un streambuf est simple passe (à savoir, même si plusieurs itérateurs utilisent le même `istreambuf`, alors chaque caractère lu dans le stream par un itérateur est supprimé du flux pour tous les autres.
- lorsqu'un itérateur en entrée est précisé dans une fonctionnelle, l'itérateur de fin est obtenu avec le même constructeur que l'itérateur de début mais sans paramètre.

Prototypes :

- le constructeur prend en paramètre le flux sur lequel on veut construire l'itérateur.
- pour `ostream`, le constructeur prend un paramètre supplémentaire représentant le séparateur envoi dans le flux.

Exemple :

```
// ostream iterator
vector<int>  u({ 1,2,3,4 });
std::copy(u.begin(), u.end(), ostream_iterator<int>(cout, ","));
```

```
// istream iterator
std::istringstream str("1 2 3 4");
std::copy(
  istream_iterator<int>(str), // it. de début sur str
  istream_iterator<int>(),    // it. de fin (=end of stream)
  ostream_iterator<int>(cout, ",")); // sortie standard
```

```
// ostringstream iterator
vector<char> v({'a','b','c','d'});
ostringstream sout;
copy(v.begin(), v.end(), ostreambuf_iterator<char>(sout));
cout << sout.str() << endl;
```

```
// istringstream iterator
std::istringstream in("Hello");
std::vector<char> w(
  (std::istreambuf_iterator<char>(in)), // double parenthèses
  std::istreambuf_iterator<char>(in),
  std::istreambuf_iterator<char>());
std::copy(w.begin(), w.end(), ostream_iterator<char>(cout, ","));
```

4 Algorithmes

Les fonctionnelles fournissent un ensemble de méthodes standards pour traiter les éléments d'une collection permettant de chercher, trier, copier, réordonner, modifier des ensembles d'éléments.

Ce ne sont pas des fonctions membres de conteneur, mais des méthodes globales génériques prenant en paramètre :

- des itérateurs (ou tout objet partageant la même interface),
  à savoir implémentant ++ (préfixe), !=, * (déréférencement)
- des fonctionnelles (opérateurs ou fonctionnelles suivant les cas).

Elles ont donc vocations à être utilisées partout où cela est possible, à savoir :

- sur les itérateurs standards,
- sur les tableaux,
- etc ... partout où le parcours par itérateur est raisonnable.

Ces algorithmes ont comme entrée un intervalle d'itérateurs `[ibegin,iend)`

- cet intervalle n'est valide que si :
  - ◇ ces itérateurs sont issus du même conteneur,
  - ◇ `ibegin` précède `iend` (= `iend` est atteignable depuis `ibegin` en itérant `ibegin` de manière répétée),

  Si ces conditions ne sont pas vérifiées, alors le comportement est indéterminée (boucle infinie, accès mémoire invalide, ...), donc pas plus sécurisé que des pointeurs.
- si cet intervalle est qualifié constant (i.e. par l'utilisation d'itérateur constant), alors :
  - ◇ il ne sera pas possible de modifier le conteneur à partir d'un itérateur issu de cet intervalle,
  - ◇ si la sortie est un itérateur dans cet intervalle, alors la sortie est aussi qualifiée constante.

L'intervalle est semi-ouvert (à savoir la position iend est exclue de l'intervalle)

- pour un intervalle vide ibegin=iend,
- si un itérator ipos est dans l'intervalle `[ibegin,iend)`, alors :
  - ◇ `[ibegin,ipos)` et `[ipos,iend)` sont disjoints,
  - ◇ `ipos` appartient au second intervalle.
  - ◇ précédé de ++`ipos`, `ipos` appartient alors au premier intervalle.

  `[ibegin,++ipos)` est l'intervalle qui contient tous les éléments de `ibegin` à `ipos`.

Lorsqu'une fonctionnelle utilise des intervalles multiples,

- typiquement, le premier intervalle `[ibegin,iend)` définit aussi le nombre d'éléments qui sera traité (n = distance(ibegin,iend))
- pour les intervalles suivants, seuls obegin est spécifié, mais tous les éléments de l'intervalle [obegin,obegin+n) doivent être définis.
- alternativement, un itérateur d'insertion peut être utilisé pour les intervalles suivants (insère au lieu de passer à l'élément suivant).

Rappel : lorsqu'un prédicat ou une fonctionnelle est attendue, alors il est possible de passer :

- soit une fonction ou +un pointeur de fonction
- soit un functor,
- soit une lambda-expression,
- soit une `std::function`/`std::bind`

à savoir, n'importe quoi qui peut être appelé sous une forme fonctionnelle.

Exemple : les arguments suivant peuvent être passés à la place d'un prédicat

- ```
  [](int i){ return i % 2 == 0; }
  ```
- ```
  bind(modulus<int>(),placeholders::_1, 2))
  ```
- ```
  DivisibleBy(2)
  ```

  où :
  ```
  struct DivisibleBy {
      const int d;
      DivisibleBy(int n) : d(n) {}
      bool operator()(int n) const { return n % d == 0; }
  };
  ```
  ```

Classification des algorithmes :
Deux suffixes spéciaux sont utilisés dans les noms d'algorithmes :

- `_if` : la test effectué n'utilise pas une valeur mais un prédicat.
  l'utilisation possible d'un prédicat supplémentaire n'est pas marquée par `_if` (exemple : `min_element`).
- `_copy` : indique que le résultat est copié dans l'intervalle résultant.
  `reverse` inverse l'ordre des éléments dans le conteneur, `reverse_copy` inverse l'ordre dans le conteneur résultant.

Les algorithmes peuvent être classifiés ainsi :

- algorithmes sans modification : ne change ni l'ordre, ni la valeur des éléments qu'il traite,
- algorithmes de modification : peuvent modifier les éléments directement,
- algorithmes de suppression : peuvent supprimer les éléments,
- algorithmes de mutation : peuvent changer l'ordre des éléments mais pas leurs valeurs,
- algorithmes de tri : trie les éléments.
- algorithmes sur des intervalles triés : nécessite que l'intervalle sur lequel il s'exécute soit trié.
- algorithmes numériques : combine les éléments avec des opérateurs numériques.

4.1 Algorithmes sans modification

a) Tests de prédicat

- `all_of(ibegin,iend,uPred)` : prédicat `uPred` vrai pour tous les éléments de l'intervalle (vrai si vide),
- `any_of(ibegin,iend,uPred)` : prédicat `uPred` vrai pour au moins un élément de l'intervalle (faux si vide),
- `none_of(ibegin,iend,uPred)` : prédicat `uPred` vrai pour aucun élément de l'intervalle (vrai si vide).

où `bool uPred(const T&)` est un prédicat unaire (reçoit le type `T` de l'élément pointé par l'itérateur et retourne un booléen).

Inclure : `<algorithm>`

Exemple :

```
vector<int> v(10, 2);
if (all_of(v.cbegin(), v.cend(), [](int i){ return i == 2; }))
    cout << "tous les nombres sont égaux à 2" << endl;
v[4] = 1;
if (any_of(v.cbegin(), v.cend(), [](int i){ return i == 1; }))
    cout << "au moins un nombre est égal à 1" << endl;
if (none_of(v.cbegin(), v.cend(), [](int i){ return i == 0; }))
    cout << "aucun nombre n'est égal à 0" << endl;
```

b) Parcours

Parcours : `for_each(ibegin,iend,uFunc)`

- applique la fonction unaire `uFunc` sur chaque élément de l'intervalle.
- `uFunc` a pour prototype `void uFunc(const Type &x)` (`const` et `&` optionnels).
- retourne `move(uFunc)`

Exemple :

```
struct Sum {
    int sum{};
    void operator()(int n) { sum += n; }
};
```

```
vector<int> nums = {3, 4, 2, 8, 15, 267};
// incrémente chaque élément de 1.
for_each(nums.begin(), nums.end(), [](int &n){ ++n; });
// alternative
for(auto &x: nums) ++x;
// somme de tous les éléments
Sum s = for_each(nums.begin(), nums.end(), Sum());
```

Note : pour l'appliquer sur un conteneur `c` complet, utiliser plutôt les boucles `for(const auto &x : c) { ... }`. Qualificateurs `&` et `const` optionnels.

c) Comptage

- `count(ibegin,iend,val)` : compte le nombre d'éléments dans l'intervalle tels que `*i = val`.
- `count_if(ibegin,iend,uPred)` : compte le nombre d'éléments dans l'intervalle tels que `uPred(*i)`.

Notes :

- `uPred` est un prédicat unaire (`bool pred(const Type &a)`)
- la variable retournée est assimilable à un entier (`ptrdiff_t`).

Exemple :

```
vector<int> v = { 1, 2, 3, 4, 4, 3, 7, 8, 9, 10 };
// nombre d'éléments égal à 4
int n4 = count(v.begin(), v.end(), 4);
// nombre d'éléments divisible par 3
int d3 = count_if(v.begin(), v.end(),
                  [](int i) {return i % 3 == 0;});
```

d) Comparaison de deux ranges

- `equal(ibegin1,iend1,ibegin2[,end2][,uPred])` : retourne vrai si tous les éléments aux mêmes positions dans chacun des intervalles sont égaux.
- `mismatch(ibegin1,iend1,ibegin2[,end2][,uPred])` : retourne une paire d'itérateurs (`std::pair`, un par intervalle) correspondant à la première position où les intervalles ont des valeurs différentes.
- `lexicographical_compare(ibegin1,iend1,ibegin2,end2[,uPred])` : retourne l'ordre lexicographique entre les deux intervalles (=ordre du premier élément différent des deux intervalles).

Notes :

- sans prédicat, utilise la surcharge de == pour comparer les éléments.
- `uPred` est un prédicat binaire (`bool pred(const Type1 &a1, const Type2 &a2)`).

Exemple :

```
string   s1 = "radar", s2 = "radin;
// teste si s1 est un palindrome
bool isP = equal(s1.begin(),s1.begin()+s1.size()/2,s1.rbegin());
// partie où s1 et s2 diffèrent
auto pos = mismatch(s.begin(), s.end(), s2.begin());
copy(pos.first,s.end(),ostream_iterator<char>(cout,""));   //"ar"
copy(pos.second,s2.end(),ostream_iterator<char>(cout,""));  //"in"
```

e) Recherche d'élément

- `find(ibegin,iend,val)` : retourne l'itérateur ipos sur le premier élément tel que *ipos=val.
- `find_if(ibegin,iend,uPred)` : retourne l'itérateur `ipos` sur le premier élément tel que uPred(*ipos) est vrai.
- `find_not(ibegin,iend,uPred)` : retourne l'itérateur `ipos` sur le premier élément tel que uPred(*ipos) est faux.
- `adjacent_find(ibegin,iend[,uPred])` : retourne l'itérateur `ipos` sur le premier élément tel que *ipos = *(ipos+1) (ou `uPred(*ipos,*(ipos+1))`).
- `search_n(ibegin,iend,n,val[,uPred])` : retourne l'itérateur `ipos` sur la position où l'on trouve `n` fois la valeur `val` consécutivement.

Exemple 1 :

```
vector<int> nums = {3, 4, 2, 2, 8, 15, 267};
// recherche par valeur
auto p = find(nums.begin(),nums.end(),15);
if (p != nums.end()) // p sur l'élément 15
// recherche par prédicat: premier nombre>10
auto q = find(nums.begin(),nums.end(),
                    [](int n){return (n>100);});
if (q != nums.end()) // p sur l'élément 267
```

Notes :

- les éléments sont comparés avec == si `bPred` n'est pas fourni.
- type du prédicat binaire : `bool bPred(const T1 &a, const T2 &b)`.

Exemple 2 :

```
vector<int> nums = {3, 4, 2, 2, 8, 2, 2, 2, 7 };
// recherche du premier élément en double
auto p = adjacent_find(nums.begin(),nums.end());
if (p != nums.end()) // p sur le premier 2
else // pas d'élément en double dans la suite
// recherche d'une répétition de trois 2
auto q = search_n(nums.begin(),nums.end(),3,2);
if (p != nums.end()) // p sur le premier paquet de trois 2
else // aucune succession de trois 2 trouvée
```

f) Recherche d'intervalles

Recherche d'intervalles dans un intervalle : (non modifiant)

- `search(ibegin1,iend1,ibegin2,iend2[,bPred])` retourne l'itérateur `ipos` à la position de le première occurence où l'intervalle `[ibegin2,iend2)` est trouvé dans `[ibegin1,iend1)`.
- `find_end(ibegin1,iend1,ibegin2,iend2[,bPred])` retourne l'itérateur `ipos` à la position de la dernière occurence où l'intervalle `[ibegin2,iend2)` est trouvé dans `[ibegin1,iend1)`.

Notes :

- les éléments sont comparés avec == si `bPred` n'est pas fourni.
- type du prédicat binaire : `bool bPred(const T1 &a, const T2 &b)`.

Exemple :

```
vector<int>  v = { 3, 5, 6, 2, 5, 6, 3 };
vector<int>  s = { 5, 6 };
// recherche de la première occurence de s dans v
auto p1 = search(v.begin(), v.end(), s.begin(), s.end());
if (p1 != v.end()) // s trouvé,  ici s = v + 1
// recherche de la dernière occurence de s dans v
auto p2 = find_end(v.begin(), v.end(), s2.begin(), s2.end());
if (p2 != v.end()) // s trouvé, ici s = v + 4
```

Recherche d'éléments d'intervalle dans des éléments d'intervalle :

`find_first_of(ibegin1,iend1,ibegin2,iend2[,bpred])`

retourne l'itérateur où le premier élément de l'intervalle `[ibegin2, iend2)` est trouvé dans `[ibegin1, iend1)`.

Notes :

- les éléments sont comparés avec == si `bPred` n'est pas fourni.
- type du prédicat binaire : `bool bPred(const T1 &a, const T2 &b)`.

Exemple :

```
vector<int>  v = { 3, 5, 6, 2, 5, 6, 3 };
vector<int>  s = { 4, 6 }, t = {1,8};

// recherche la position dans v du premier élément de s qui
// y est trouvé: ici trouve 6 dans v (4 non présent)
auto p1 = find_first_of(v.begin(), v.end(), s.begin(), s.end());
if (p1 != v.end()) // s trouvé,  ici s = v + 2 (*s=6)

// recherche la position dans v du premier élément de s
// ici: ne trouve rien (aucun élément de t n'existe dans v)
auto p2 = find_end(v.begin(), v.end(), t.begin(), t.end());
if (p2 == v.end()) // aucune occurrence trouvée
```

4.2 Algorithmes avec modification

a) transformation

Transformations fonctionnelles d'ensemble d'éléments :

- `T transform(ibegin,iend,obegin,uFunc)` : applique la fonction unaire `uFunc` sur l'ensemble des éléments de `[ibegin, iend)` et place le résultat dans `obegin`.
- `T transform(ibegin1,iend1,ibegin2,obegin,uFunc)` : applique la fonction binaire `bFunc` sur l'ensemble des éléments de `[ibegin1, iend1)` (1er paramètre) et de `[ibegin2, iend2)` (1er paramètre), et place le résultat dans `obegin`.

Remarques :

- `uFunc` est une fonctionnelle prenant un seul paramètre du type associé à `ibegin` et retourne une valeur de type associé à `obegin`.
- `bFunc` est une fonctionnelle prenant deux paramètres, le 1er du type associé à `ibegin1`, le 2nd du type associé à `ibegin2` et retourne une valeur de type associé à `obegin`.

Exemple :

```
// transformation en majuscule
std::string s("hello");
std::transform(s.begin(), s.end(), s.begin(),
    [](unsigned char c) { return std::toupper(c); });
// ajouter d'autres exemples
```

b) Copie d'éléments

- `copy(ibegin1,iend1,ibegin2)` copie les éléments de l'intervalle source `[ibegin1,iend1)` vers `ibegin2`.
- `copy_if(ibegin1,iend1,ibegin2,bPred)` copie les éléments de l'intervalle source `[ibegin1, iend1)` pour lequel le prédicat est vrai vers `ibegin2`.
- `copy_n(ibegin1,cnt,ibegin2)` copie `cnt` élément de l'itérateur source vers l'itérateur cible.
- `copy_backward(ibegin1,iend1,iend2)` copie les éléments de l'intervalle `[ibegin1,iend1)` dans l'ordre inverse à partir de `iend2` (suppose que les itérateurs soient bidirectionnels), utilise l'opérateur `--`.

Notes :

- il doit y avoir suffisamment de place dans l'itérateur cible, sauf si un itérateur d'insertion est utilisé.
- retourne l'itérateur vers le dernier élément copié dans l'itérateur de destination.

Exemple :

```
vector<int>  src = { 1, 2, 3, 4, 5, 6 };
vector<int>  dst1 = { 7, 8 }, dst2(4), dst3,
             dst4(src.size()), dst5(10);
```

```
// copie dans un tableau de même taille
dst1.resize(src.size());  // sinon dst1 pas assez grand
copy(src.begin(), src.end(), dst1.begin());
// dst1 = {1,2,3,4,5,6}
```

```
// copie les 4 premiers éléments dans dst2
copy_n(src.begin(), 4, dst2.begin());
// dst2 = {1,2,3,4}
```

```
// copie inversée dans le tableau de destination
copy(src.rbegin(), src.rend(), dst3.begin());
// dst3 = {6,5,4,3,2,1}
```

```
// copie des éléments pair à la fin de dst4
copy_if(src.begin(), src.end(), back_inserter(dst4),
                [](int n) { return n % 2 == 0;  });
// dst4 = {2,4,6}
```

```
// copie backward
copy_backward(src.begin(), src.end(), dst5.end());
// dst5 = {0,0,0,0,1,2,3,4,5,6}
```

c) Déplacement d'éléments

- `move(ibegin1,iend1,ibegin2)` : déplace les éléments de l'intervalle source `[ibegin1,iend1)` vers `ibegin2`.
- `move_backward(ibegin1,iend1,iend2)` déplace les éléments de l'intervalle `[ibegin1,iend1)` dans l'ordre inverse à partir de `iend2` (suppose que les itérateurs soient bidirectionnels), utilise l'opérateur `--`.

Swap d'éléments : (modifiant)

- `swap(e1,e2)` : échange les éléments `e1` et `e2`.
- `swap(t1,t2)` : échange des tableaux statiques `t1` et `t2`.
- `iter_swap(i1,i2)` : échange les valeurs pointées par les itérateurs `i1` et `i2`.
- `swap_ranges(ibegin1,iend1,ibegin2)` : échange les éléments de l'intervalle `[ibegin1,iend1)` avec ceux à partir de `ibegin2`.

Exemple :

```
vector<string>  s1 = { "arc", "car" }, s2 = { "dog", "cat" };
swap_ranges(s1.begin(), s1.end() , s2.begin());
// s1 = { "dog", "cat" }, s2 = { "arc", "car" }
move(s1.begin(), s1.end(), s2.begin());
// s1 = { "", "" }, s2 = { "dog", "cat" }
```

d) Remplissage

- `fill(ibegin,iend,val)` : remplit l'intervalle `[ibegin,iend)` avec `val`.
- `fill(ibegin,cnt,val)` : remplit `cnt` éléments avec la valeur `val` à partir de la position pointée par `ibegin`. Retourne un itérateur sur l'élément suivant la dernière valeur remplie.
- `generate(ibegin,iend,gen)` : idem `fill(ibegin,iend,gen)` mais avec des appels consécutif de `gen`.

- `generate_n(ibegin,cnt,gen)` : idem `fill(ibegin,iend,gen)` mais avec `cnt`+1 appels consécutif de `gen`.

Note : `gen()` est une fonction qui retourne le type de valeur pointée par l'itérateur.

Exemple :

```
vector<int>  v1(6);            // v1={0,0,0,0,0,0}
fill(v1.begin(),v1.end(),3);   // v1={3,3,3,3,3,3}
fill_n(v1.begin(),2,4);        // v1={4,4,3,3,3,3}
int n{};
generate(v1.begin(),v1.end(),[&n]() { return ++n; });
// v1={1,2,3,4,5,6}
generate_n(v1.begin(),5, []() { return rand()%3; });
// v1={2,0,0,1,2,1}
```

e) Suppressions

- `remove(ibegin,iend,val)` : supprime la valeur `val` dans l'intervalle `[ibegin,iend)`.
- `remove(ibegin,iend,uPred)` : supprime les valeurs de l'intervalle `[ibegin,iend)` qui vérifient le prédicat `uPred`.
- `remove_copy(ibegin1,iend1,ibegin2,val)` : copie les valeurs de l'intervalle `[ibegin,iend)` différentes de val dans `ibegin2`.
- `remove_copy_if(ibegin1,iend1,ibegin2,uPred)` : copie les valeurs dans l'intervalle `[ibegin,iend)` qui ne vérifie pas `uPred` dans `iBegin2`.

Notes :

- **attention**, pour les fonctions travaillant directement sur l'intervalle, **le conteneur n'est pas modifié par l'opération** : les valeurs de l'intervalle sont juste déplacées. La modification du conteneur est laissée à la charge de l'utilisateur.
- ces fonctions retournent un itérateur sur l'élément suivant la dernière valeur remplie.

Exemple :

```
vector<int>  v1(10), v2, v3;
int n{};
generate(v1.begin(), v1.end(), [&n]() { return ++n; });
// v1={1,2,3,4,5,6,7,8,9,10}
```

```
// retire l'élément 3
remove(v1.begin(), v1.end(), 3);
// v1={1,2,4,5,6,7,8,9,10,<10>} <= décalage
```

```
// retire les éléments pair de v1
remove_if(v1.begin(), v1.end(),
    [](int n) { return n % 2 == 0; });
// v1={1,5,7,9,<6>,7,8,9,10,10} <= décalage
```

```
// retire 9 de v1
remove_copy(v1.begin(), v1.end(), back_inserter(v2), 9);
// v1={1,5,7,6,7,8,10,10,<>}
```

```
// retire les éléments pair
remove_copy_if(v1.begin(), v1.end(), back_inserter(v3),
    [](int n) { return n % 2 == 0; });
// v1={1,5,7,9,7,9,<>}
```

f) Remplacement

- `replace(ibegin,iend,val,nval)` : remplace la valeur `val` de l'intervalle `[ibegin,iend)` par la valeur `nval`.
- `replace(ibegin,iend,uPred,nval)` : remplace les valeurs de l'intervalle `[ibegin,iend)` qui vérifient le prédicat `uPred` par la valeur `nval`.
- `replace_copy(ibegin1,iend1,ibegin2,val,nval)` : copie les valeurs de l'intervalle `[ibegin,iend)` dans `ibegin2`, en remplaçant les valeurs égales à `val` par `nval`.
- `replace_copy_if(ibegin1,iend1,ibegin2,uPred,nval)` : copie les valeurs de l'intervalle `[ibegin,iend)` dans `ibegin2`, en remplaçant les valeurs égales qui vérifie `uPred` par `nval`.

Note : les deux dernières fonctions retournent un itérateur sur l'élément qui suit la dernière valeur remplie.

Exemple :

```
vector<int>  v1(10), v2, v3;
int n{};
generate(v1.begin(), v1.end(), [&n]() { return ++n; });
// v1={1,2,3,4,5,6,7,8,9,10}
```

```
// retire l'élément 3
replace(v1.begin(), v1.end(), 3, 0);
// v1=1,2,0,4,5,6,7,8,9,10}
```

```
// retire les éléments pair de v1
replace_if(v1.begin(), v1.end(),
    [](int n) { return n % 2 == 0; }, 0);
// v1={1,0,0,0,5,0,7,0,9,0} <= décalage
```

```
// retire 9 de v1
replace_copy(v1.begin(), v1.end(), back_inserter(v2), 0, 2);
// v2={1,2,2,2,5,2,7,2,9,2}
```

```
// retire les éléments pair
replace_copy_if(v1.begin(), v1.end(), back_inserter(v3),
    [](int n) { return n % 2 == 0; }, 1);
// v3={1,1,1,1,5,1,7,1,9,1}
```

g) Inversion, rotation

- `reverse(ibegin,iend)` : inverse l'ordre des éléments dans l'intervalle `[ibegin,iend)` (itérateur bidirectionnel).
- `reverse_copy(ibegin1,iend1,ibegin2)` : inverse l'ordre des éléments de l'intervalle `[ibegin,iend)` (itérateur bidirectionnel) dans une copie en `ibegin2`.

- `rotate(ibegin1,ibegin2,iend)` : effectue une rotation des éléments dans l'intervalle `[ibegin1, iend)`, de façon à ce que `ibegin2` $\in$ `[ibegin1,iend)` soit la nouvelle position du premier élément.
- `rotate_copy(ibegin1,ibegin2,iend,oend)` : idem mais le résultat est construit à partir de `oend`.

Exemple :

```
vector<int>  v={1,2,3,4,5,6};
reverse(v.begin(), find(v.begin(), v.end(), 3));
// v={2,1,3,4,5,6}
rotate(v.begin(), find(v.begin(), v.end(), 4), v.end());
// v={4,5,6,2,1,3}
```

Note : les fonctions de copie retournent un itérateur sur l'élément qui suit la dernière valeur remplie.

h) Permutation aléatoire

- `random_shuffle(ibegin,iend[,uGen])` : effectue une permutation aléatoire de l'intervalle `[ibegin,iend)`, en utilisant le générateur de nombre aléatoire uniforme `uGen` si fourni.
- `shuffle(ibegin,iend,uGen)` : idem (nouvelle version utilisant `std::random`).

Exemple :

```
vector<int>  v={1,2,3,4,5,6};
random_shuffle(v.begin(), v.end());
// v={5,2,4,3,1,6}
std::random_device rd;
std::mt19937 g(rd());
std::shuffle(v.begin(), v.end(), g);
// v={4,2,6,5,3,1}
```

i) Suppression des doublons consécutifs

- `unique(ibegin,iend[,uPred])` : enlève les éléments consécutifs identiques dans l'intervalle `[ibegin,iend)` ; si précisé, en utilisant le prédicat `uPred` pour comparer les éléments.
- `unique_copy(ibegin1,iend1,ibegin2[,bPred])` : idem en construisant le résultat dans la copie en `ibegin2`.

Notes :

- les éléments sont comparés avec == si `bPred` n'est pas fourni.
- type du prédicat binaire : `bool bPred(const T1 &a, const T2 &b)`.
- retourne un itérateur sur le dernier élément écrit.
- la taille du conteneur ne change : la taille doit être ajusté en utilisant les méthodes appropriées associées au conteneur.

Exemple :

```
vector<int>  v = { 1,1,2,2,2,1,2,2,1,1,2 }, v2;
auto p = unique(v.begin(), v.end());
// v={1,2,1,2,1,2,<2>,2,1,1,2}
// v.erase(p,v.end()) pour supprimer la fin
unique_copy(v.begin(), v.end(), back_inserter(v2));
// v2={1,2,1,2,1,2,<1>,2}
```

4.3 Partitionnement, tri et fonctionnelles pour ensembles triés

a) Partitionnement

- `partition(ibegin,iend,uPred)` : réordonne les éléments de l'intervalle `[ibegin,iend)` de façon les éléments pour lesquels `uPred` est vrai précèdent les éléments pour lesquels `uPred` est faux. Retourne un itérateur placé sur le premier élément faux. L'ordre relatif des éléments n'est pas conservé.
- `stable_partition(ibegin,iend,uPred)` : idem en préservant l'ordre relatif des éléments.
- `partition_copy(ibegin,iend,obegin1,obegin2,uPred)` : tri les éléments de l'intervalle `[ibegin,iend)`, en plaçant en `obegin1` les éléments pour lesquels `uPred` est vrai, et en `obegin2` ceux pour lesquels `uPred` est faux. Retourne une paire d'itérateur placé à la fin des intervalles `obegin1` et `obegin2`.
- `partition_point(ibegin,iend,uPred)` : retourne un itérateur sur le premier point de partitionnement trouvé dans l'intervalle `[ibegin,iend)`, à savoir le premier élément tel que `uPred` est faux.
- `is_partitioned(ibegin,iend,uPred)` : retourne vrai si tous les éléments de l'intervalle `[ibegin,iend)` pour lesquels `uPred` est vrai précèdent ceux pour lesquels `uPred` est faux.

Exemple :

```
vector<int>  v = { 0, 0, 3, 0, 2, 4, 5, 0, 7 };
vector<int>  vc1 = v, vc2 = v, vc3 = v, vTrue, vFalse;
auto pred = [](int n) { return n == 0; };
```

```
auto p1 = partition(v.begin(), v.end(), pred);
// v={0,0,0,0,<2>,4,5,3,7}, p1 marqué par <.>
```

```
auto p2 = stable_partition(vc1.begin(), vc1.end(), pred);
// vc1={0,0,0,0,<3>,2,4,5,7,}, p2 marqué par <.>
```

```
auto p3 = partition_copy(vc2.begin(), vc2.end(),
    back_inserter(vTrue), back_inserter(vFalse), pred);
// vTrue={0,0,0,0,<>}, p3.first marqué par <.>
// vFalse={3,2,4,5,7,<>}, p3.second marqué par <.>
```

```
auto p4 = partition_point(vc3.begin(), vc3.end(), pred);
// v = { 0, 0, <3>, 0, 2, 4, 5, 0, 7 }, p4 marqué par <.>
```

```
bool b1 = (is_partitioned(v.begin(), v.end(), pred));
bool b2 = (is_partitioned(vc3.begin(), vc3.end(), pred));
// b1 = vrai, b2 = faux
```

b) Tri

- `sort(ibegin,iend[,bCmp])` : tri les éléments de l'intervalle `[ibegin, iend)` (RandomIt),
- `stable_sort(ibegin,iend[,bCmp])` : idem sort, l'ordre des éléments égaux est conservé.
- `partial_sort(ibegin,imid,iend[,bCmp])` : tri partiel de l'intervalle `[ibegin, iend)`, tel que `[ibegin, imid)` contient les plus petits éléments de l'intervalle `[ibegin, iend)` trié, l'ordre du reste des éléments `[imid, iend)` n'est pas spécifié (RandomIt),
- `partial_sort_copy(ibegin,iend,obegin,oend[,bCmp])` : en notant `n = distance(obegin, oend)`, place le tri des `n` plus petits éléments de l'intervalle `[ibegin, iend)` dans `[obegin, oend)`.
- `nth_element(ibegin,ipos,iend[,bCmp])` : effectue un tri partiel de l'intervalle `[ibegin, iend)`, en garantissant que l'élément à la position `ipos` est le même qui si l'intervalle avait été trié (exemple d'application : le médian).
- `is_sort(ibegin,iend[,bCmp])` : retourne vrai si l'intervalle `[ibegin,iend)` est trié (ForwardIt).
- `is_sorted_until(ibegin,iend[,bCmp])` : retourne l'itérateur placé à la position à laquelle l'intervalle `[ibegin, iend)` n'est plus trié (ForwardIt).

Exemple :

```
vector<int>   v = { 9, 5, 3, 1, 2, 4, 6, 8, 7 };
vector<int>   v1 = v, v2 = v, v3 = v;
```

```
sort(v1.begin(), v1.end());    // v1={1,2,3,4,5,6,7,8,9}
```

```
partial_sort(v2.begin(), v2.begin()+4, v2.end());
// v2={1,2,3,4,9,5,6,8,7} : les 4 plus petits triés
```

```
vector<int>   u(5);
partial_sort_copy(v.begin(), v.end(), u.begin(), u.end());
// u={1,2,3,4,5} : les 5 plus petits triés
```

```
using iDuo = pair<int,int>;
vector<iDuo>  w = { { 4,7 }, {4,2}, { 1,5 }, {1,2} };
vector<iDuo>  w1 = w, w2 = w;
auto cmp = [](const iDuo &a, const iDuo &b)
    { return (a.first < b.first); }; // .second ignoré
sort(w1.begin(), w1.end(), cmp);
stable_sort(w2.begin(), w2.end(), cmp);
// w2: garantie ordre {4,7},{4,2} et {1,5},{1,2} conservé
// w1: aucun garantie autre que tri suivant le critère
```

```
auto pos3 = v3.begin() + 4;
nth_element(v3.begin(), pos3, v3.end());
// *pos3 en position pos3 garanti être le même qui si v3
// avait été trié (v3 possiblement partiellement trié)
```

c) Recherche sur des ensembles triés

- `binary_search(ibegin,iend,val[,comp])` : recherche par dichotomie de la valeur `val` dans l'intervalle `[ibegin, iend)`. Retourne vrai si l'élément est trouvé.

- `lower_bound(ibegin,iend,val[,comp])` : retourne un itérateur `ipos` vers le plus petit élément de l'intervalle `[ibegin, iend)` qui n'est pas inférieur à `val` (à savoir, *ipos=val ou *ipos>val, plus petit élément supérieur ou égal).
- `upper_bound(ibegin,iend,val[,comp])` : retourne un itérateur `ipos` vers le plus petit élément de l'intervalle `[ibegin, iend)` strictement supérieur à `val` (à savoir *ipos>val = élément qui suit val).
- `equal_range(ibegin,iend,val[,comp])` : retourne une paire d'itérateur (`std::pair`) représentant un intervalle pour lequel tous les éléments sont équivalents à `val`.

Notes :

- utilise l'opérateur de comparaison < si la fonction de comparaison n'est pas fournie.
- l'équivalence est utilisé pour décider que l'élément a été trouvé (a<val) et (val<a) sont tous les deux faux.

Exemple :

```
vector<int>   v = { 2, 2, 3, 1, 2, 4, 3, 1, 2 };
sort(v.begin(), v.end());
// v={1,1,2,2,2,2,3,3,4}
```

```
bool b1 = binary_search(v.begin(), v.end(), 3); // true
bool b2 = binary_search(v.begin(), v.end(), 5); // false
```

```
auto lpos = lower_bound(v.begin(), v.end(), 2);
// v={1,1,<2>,2,2,2,3,3,4}; <.> = position de lpos
auto upos = upper_bound(v.begin(), v.end(), 2);
// v={1,1,2,2,2,2,<3>,3,4}; <.> = position de upos
```

```
auto pos = equal_range(v.begin(), v.end(), 2);
// v={1,1,<2>,2,2,2,3,3,4}; <.> = position de pos.first
// v={1,1,2,2,2,2,<3>,3,4}; <.> = position de pos.second
```

Rappel :

si applicable, toujours tester si un itérateur renvoyé est différent du `end()` du conteneur (à savoir ci-dessus, lpos et upos peuvent être égaux à v.end() si la valeur cherchée est plus grande que la valeur maximale).

d) Opération ensembliste sur des ensembles triés

- `merge(ibegin1,iend1,ibegin2,iend2,obegin[,comp])` : place en `obegin` le résultat de la fusion les valeurs issus des intervalles `[ibegin1,iend1)` et `[ibegin2,iend2)`. Toutes les valeurs équivalentes sont conservées.
- `inplace_merge(ibegin,imiddle,iend[,comp])` : idem `merge` avec `[ibegin,imiddle)` et `[imiddle,iend)`, où chaque intervalle est trié et le résultat est écrit en place. L'ordre relatif des éléments équivalent est conservé.
- `set_union(ibegin1,iend1,ibegin2,iend2,obegin[,comp])` : comme `merge` sauf qui si une valeur équivalente est présente `n` fois dans `[ibegin1, iend1)` et `m` fois dans `[ibegin2, iend2)`, alors elle apparait `max(n,m)` dans l'intervalle résultat.
- `set_difference(ibegin1,iend1,ibegin2,iend2,obegin[,comp])` : place en `obegin` l'ensemble des valeurs de `[ibegin1, iend1)` qui ne sont pas dans `[ibegin2, iend2)`.

- `set_intersection(ibegin1,iend1,ibegin2,iend2,obegin [,comp])` : place en `obegin` l'ensemble des valeurs qui sont à la fois dans `[ibegin1, iend1)` et dans `[ibegin2, iend2)`.
- `set_symetric_difference(ibegin1,iend1,ibegin2,iend2,obegin [,comp])` : place en `obegin` l'ensemble des valeurs qui sont, soit dans `[ibegin1, iend1)`, soit dans `[ibegin2, iend2)`, mais pas dans les deux intervalles à la fois.
- `includes(ibegin1,iend1,ibegin2,iend2,obegin[,comp])` : retourne vrai si tous les éléments de `[ibegin2, iend2)` se trouvent dans `[ibegin1, iend1)`, et faux sinon.

Remarques :

- Les éléments sont comparés avec l'opérateur < ou la fonction de comparaison `comp` si elle est définie.
- Ces fonctions retournent un itérateur sur le dernier élément inséré.

Exemple :

```
vector<int>  e1 = { 2, 2, 2, 4, 8, 10 };
vector<int>  e2 = { 1, 1, 2, 4, 5, 6 , r1, r2, r3}, r4, r5;
```

```
merge(e1.begin(),e1.end(),
      e2.begin(),e2.end(), back_inserter(r1));
// r1={1,1,2,2,2,4,4,5,6,8,10}
set_union(e1.begin(),e1.end(),
          e2.begin(),e2.end(), back_inserter(r2));
// r2={1,1,2,2,2,4,5,6,8,10}
```

```
vector<int>   e3 = { 1, 4, 7, 3, 4, 8 };
inplace_merge(e3.begin(), e3.begin() + 3, e3.end());
// e3={1,3,4,4,7,8}
```

```
set_difference(e1.begin(),e1.end(),
               e2.begin(),e2.end(), back_inserter(r3));
// r3={2,2,8,10}
```

```
set_intersection(e1.begin(),e1.end(),
                 e2.begin(),e2.end(), back_inserter(r4));
// r4={2,4}
```

```
set_symmetric_difference(e1.begin(),e1.end(),
                   e2.begin(),e2.end(), back_inserter(r5));
// r5={1,1,2,2,5,6,8,10}
```

```
vector<int>   sub = { 2, 6, 8 };
bool b = includes(e3.begin(),e3.end(),sub.begin(),sub.end());
```

4.4 Tas-max

Définition : un tas-max (max-heap) est un tas pour lequel le maximum du tas se trouve placé en première position, l'arrangement des autres éléments n'est pas spécifié (= dépend de l'implémentation).

Les algorithmes tas-max permettent de manipuler un intervalle définit pour se comporter comme un tas-max (RandIt) :

- `make_heap(ibegin,iend[,comp])` : construit un tas-max à partir de l'intervalle `[ibegin,iend)`. Après cette fonction, `*ibegin` contient la valeur du maximum.

- `push_heap(ibegin,iend[,comp])` : pousse l'élément `iend-1` dans le tas-max contenu dans l'intervalle `[ibegin,iend-1)`.
- `pop_heap(ibegin,iend[,comp])` : dépile l'élément au sommet du tas-max contenu dans l'intervalle `[ibegin,iend)` (à savoir `*ibegin`) et le place en `iend-1`. Après cette fonction, `[ibegin,iend-1)` est le tas-max mis à jour, et `*(iend-1)` contient la valeur dépilée.
- `is_heap(ibegin,iend[,comp])` : retourne un booléen vrai si l'intervalle `[ibegin,iend)` est un tas-max.
- `is_heap_until(ibegin,iend[,comp])` : retourne un itérateur `ipos` sur `[ibegin,iend)` pour lequel `[ibegin,ipos)` représente un tas-max.
- `sort_heap(ibegin,iend[,comp])` : convertit le tas-max contenu dans l'intervalle `[ibegin,iend)` en la liste triée des éléments (note : le résultat n'est pas un tas-max).

Exemple :

```
vector<int>   v = {5,3,1,2,9,4,6,8,7};
make_heap(v.begin(),v.end());   // v={9,8,6,7,3,4,1,2,5}
```

```
v.push_back(12);                // v={9,8,6,7,3,4,1,2,5,12}
push_heap(v.begin(),v.end());   // v={12,9,6,7,8,4,1,2,5,3}
v.push_back(7);                 // v={12,9,6,7,8,4,1,2,5,3,7}
push_heap(v.begin(),v.end());   // v={12,9,6,7,8,4,1,2,5,3,7}
```

```
pop_heap(v.begin(),v.end());    // v={9,8,6,7,7,4,1,2,5,3,12}
int vmax = v.back();
v.pop_back();                   // v={9,8,6,7,7,4,1,2,5,3}
pop_heap(v.begin(),v.end());    // v={8,7,6,7,3,4,1,2,5,9}
```

```
bool b1 = is_heap(v.begin(),v.end());     // faux
auto p = is_heap_until(v.begin(),v.end());
// v={8,7,6,7,3,4,1,2,5,<9>}, <.> = position de p
v.pop_back();                   // v={8,7,6,7,3,4,1,2,5}
bool b2 = is_heap(v.begin(),v.end());     // vrai
```

```
sort_heap(v.begin(),v.end());   // v={1,2,3,4,5,6,7,7,8}
```

4.5 Opération Min/Max

Algorithme min-max : (ForwardIt)

- `min(v1,v2[,comp])` : retourne le min de deux éléments.
- `min(iList[,comp])` : idem `min` sur la liste d'éléments passés dans la liste d'initialisation `iList`.
- `min_element(ibegin,iend[,comp])` : retourne un itérateur vers le minimum des éléments dans l'intervalle `[ibegin,iend)`.
- `max(v1,v2[,comp])` : idem `min` avec calcul du `max`.
- `max(iList[,comp])` : idem `min` avec calcul du `max`.
- `max_element(ibegin,iend[,comp])` : idem `min_element` avec calcul du `max`.
- `minmax(v1,v2[,comp])` : retourne une paire (std : :pair) contenant le min (first) et le max (second) des deux éléments.
- `minmax(iList[,comp])` : idem `minmax` mais sur une liste d'initialisation.
- `minmax_element(ibegin,iend[,comp])` : retourne une paire (std : :pair) d'itérateur vers le min (first) et le max (second) des éléments dans l'intervalle `[ibegin,iend)`.

Notes :

- la fonction `comp` est utilisée si elle est fournie, sinon utilise l'opérateur <.
- sur un intervalle, s'il y a plusieurs éléments équivalents au min/max, l'itérateur vers le premier est retourné.
- les éléments sont retournés par valeur constante (min/max/minmax).

Exemple :

```
int imin1 = min(2, 4);          // imin1=2
int imin2 = min({ 4,7,9,1,2,3 });    // imin2=1
const int &rmin = min(imin1,imin2); // rmin=1
```

```
vector<int> v = { 1,2,3,4,5,6,7,7,8 };
auto pmin = min_element(v.begin(), v.end());
// v={ <1>,2,3,4,5,6,7,7,8 }; <.> position pmin
// *pmin = 1
```

```
auto Mm1 = minmax({ 4,7,9,1,2,3 });
// Mm1.first=1 Mm1.second=9
auto Mm2 = minmax_element(v.begin(), v.end());
// Mm2.first=1 Mm2.second=8
```

```
auto comp = [](int u, int v) { return u > v; };
auto Mm3 = minmax({ 4,7,9,1,2,3 },comp);
// Mm3.first=9 Mm1.second=1
```

4.6 Permutation

Les permutations considérées se basent sur un intervalle ordonné :

- utilise la génération de permutation par ordre lexicographique (permet le calcul de la permutation suivante à partir de précédente).
- l'intervalle ordonné par ordre croissant (avec `sort`) est la première de la permutation (toutes les suivantes obtenues avec `next`).
- l'intervalle ordonné par ordre décroissant est la dernière permutation (toutes les précédentes obtenues avec `prev`).

Algorithme de permutation : (BidirIt)

- `next_permutation(ibegin,iend[,comp])` : calcule la permutation suivante des éléments de l'intervalle `[ibegin,iend)`, et retourne vrai tant que l'on n'est pas revenu à la première permutation.
- `prev_permutation(ibegin,iend[,comp])` : calcule la permutation précédente des éléments de l'intervalle `[ibegin,iend)`, et retourne vrai si la permutation obtenue est différente de la dernière permutation.
- `is_permutation(ibegin1,iend1,ibegin2[,iend1][,comp])` : retourne vrai si l'intervalle `[ibegin1,iend1)` et une permutation de l'intervalle `[ibegin2,iend2)`.

Note : utilise l'opérateur == pour comparer les éléments sauf si la fonction de comparaison `comp` est fournie (les éléments égaux sont identiques au regard des permutations)

Exemple :

```
vector<int> v = {1,2,3}, r = v;
while (next_permutation(v.begin(),v.end())) { /* v ici */ }
// v={1,3,2} {2,1,3} {2,3,1} {3,1,2} {3,2,1}
reverse(r.begin(), r.end());
// r={3,2,1}
while (prev_permutation(r.begin(),r.end())) { /* r ici */ }
// r={3,1,2} {2,3,1} {2,1,3} {1,3,2} {1,2,3}
```

```
vector<int> v2 = {1,2,2}, r2 = v2;
while (next_permutation(v2.begin(),v2.end())) { /* v2 ici */ }
// v2={2,1,2} {2,2,1}
reverse(v2.begin(), v2.end());
// r2={2,2,1}
while (prev_permutation(r2.begin(),r2.end())) { /* r2 ici */ }
// r2={2,1,2} {1,2,2}
```

```
bool b1=is_permutation(v.begin(),v.end(),r.begin());  // vrai
bool b2=is_permutation(v.begin(),v.end(),v2.begin()); // faux
```

4.7 Opération numérique

Les fonctionnelles numériques ont pour but d'effectuer des calculs numériques sur les intervalles considérés.

Utilisation : inclure <numeric>

5 fonctions sont proposées :

- `iota` : U[i] = ++val
- `accumulate` : s = s +U[i]
- `inner_product` : s = s + U[i] * V[i]
- `partial_sum` : V[i] = U[i] + V[i-1]
- `adjacent_difference` : V[i] = U[i] - V[i-1]

Les opérateurs (+, -, *) peuvent être suchargées par des fonctionnelles lors de l'utilisation.

A noter que toutes ces fonctions numériques sont à la base des spécialisations de `transform`, mais proposeront des versions optimisées multithreadées en C_{17}^{++} (voir plus loin).

a) `iota`

`iota(ibegin,iend,val)`

initialisation de l'intervalle `(ibegin,iend]` par incrémentation consécutive d'une objet (valeur numérique, itérateur, qui implémente ++ et convertible dans le type de la cible).

Algorithme équivalent sur un tableau T de taille n :

void `iota(U,val)`
  for $i = 0$ **to** $n - 1$ **do**
    U[i] = *val*;
    ++val;

Exemple :

```
std::list<int> l(10);   // liste de taille 10
// initialisée de -4 à 5
std::iota(l.begin(), l.end(), -4);
// vecteur d'itérateurs sur cette liste
std::vector<std::list<int>::iterator> v(l.size());
std::iota(v.begin(), v.end(), l.begin());
// mélange du vecteur
std::shuffle(v.begin(), v.end(),
    std::mt19937{std::random_device{}()});
// affichage de liste mélangée
for(auto i: v) std::cout << *i << ' ';
```

b) accumulation

`T accumulate(ibegin,iend,init[,op])`

calcule la somme (défaut : `op=plus`) des éléments de l'intervalle `[ibegin,iend)`

Les algorithmes équivalents sont les suivants :

T **accumulate**(*U,val*)
  s = val;
  for $i = 0$ **to** $n - 1$ **do**
    s = s + U[i];
  return *s*;

T **accumulate**(*U,val,***OpSum**)
  s = val;
  for $i = 0$ **to** $n - 1$ **do**
    s = **OpSum**(*s,U[i]*) ;
  return *s*;

Exemple :

```
std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
// somme des éléments du vecteur
int sum = std::accumulate(v.begin(), v.end(), 0);
// produit des éléments du vecteur
int product = std::accumulate(v.begin(), v.end(),
    1, std::multiplies<int>());
// ajoute un - entre chaque caractère d'une chaine
std::string s = std::accumulate(std::next(v.begin()), v.end(),
    std::to_string(v[0]), // start with first element
    [](std::string a, int b) {
        return a + '-' + std::to_string(b);
});
```

c) produit scalaire

`T inner_product(ibegin1,iend1,ibegin2,init[,opSum,opProd]])`

calcule le produit scalaire entre les éléments de `[ibegin1, iend1)` et `[ibegin2, iend2)` où `opSum` (par défaut `plus`) est l'opérateur entre deux éléments de chaque ensemble, et `opProd` (par défaut `multiplies`) est l'opérateur qui contatène les résultats (initialisé avec `init`). `end2` est déduit de la taille de `[ibegin1, iend1)`.

Les algorithmes équivalents sont les suivants :

```
inner_product(U, V, val)
    s = val;
    for i = 0 to n − 1 do
        s = s + U[i]*V[i];
    return s;
```

```
inner_product(U,V,val,OpSum,OpProd)
    s = val;
    for i = 0 to n − 1 do
        s = OpSum( s ,
            OpProd(U[i],V[i]));
    return s;
```

Exemple :

```
std::vector<int> a{0, 1, 2, 3, 4};
std::vector<int> b{5, 4, 2, 3, 1};
// produit scalaire entre a et b
int r1 = std::inner_product(a.begin(), a.end(), b.begin(), 0);
// nombre de d'éléments identiques dans les deux vecteurs
int r2 = std::inner_product(a.begin(), a.end(), b.begin(), 0,
                        std::plus<int>(), std::equal_to<int>());
```

d) somme partielle

`Output partial_sum(ibegin1,iend1,ibegin2,OpSum)`

Calcule la somme cumulée des éléments de `[ibegin1, iend1)` et place le résultat dans `[ibegin2, iend2)` où `OpSum` est par défaut `plus`. Retourne le dernier élément écrit (donc `iend2`).

```
partial_sum(U,V)
    V[0] = U[0];
    for i = 1 to n − 1 do
        V[i] = U[i] + V[i-1];
    return V+n;
```

```
partial_sum(U, V, OpSum)
    V[0] = U[0];
    for i = 1 to n − 1 do
        V[i] = OpSum(U[i],V[i-1]);
    return V+n;
```

Exemple :

```
std::vector<int>v(10, 2); // v contient 10 fois la valeur 2
// affiche la 10 premiers nombres pairs (n additions de 2)
std::partial_sum(v.begin(), v.end(),
        std::ostream_iterator<int>(std::cout, "␣"));
// calcule les 10 premières puissances de 2
std::partial_sum(v.begin(), v.end(), v.begin(),
        std::multiplies<int>());
```

e) différence adjacente

`Output adjacent_difference(ibegin1,iend1,ibegin2,OpDiff)`

Calcule la différence des éléments adjacents de `[ibegin1, iend1)` et place le résultat dans `[ibegin2, iend2)` où `OpDiff` est par défaut `minus`. Retourne le dernier élément écrit (donc `iend2`).

adjacent_difference(*U,V*)
- V[0] = U[0];
- **for** *i* = *1* **to** *n* − 1 **do**
  - V[i] = U[i] - U[i-1];
- **return** *V*+*n*;

adjacent_difference(*U,V,OpDiff*)
- V[0] = U[0];
- **for** *i* = *1* **to** *n* − 1 **do**
  - V[i] = **OpDiff**(*U[i],U[i-1]*);
- **return** *V*+*n*;

Exemple :

```
std::vector<int> v{2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
std::adjacent_difference(v.begin(), v.end(), v.begin());
// ici v = { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 };
std::adjacent_difference(v.begin(), v.end() - 1,
                         v.begin() + 1, std::plus<int>());
// ici v = { 2, 2, 4, 6, 10, 16, 26, 42, 68, 110 }
```

f) politique d'exécution

A partir du C_{17}^{++}, toutes les fonctionnelles numériques vont offrir la possibilité d'être exécutée en multi-threading.

Ceci s'effectuera par la définition d'une politique d'exécution à utiliser sur les fonctions qui le supportent :

- `sequential` (`std::sequential_execution_policy`)
- `par` (`std::parallel_execution_policy`)
- `par_vec` (`std::parallel_vector_execution_policy`)

De nouvelles fonctions numériques seront également introduites utilisant ces politiques :

`reduce`, `transform_reduce`, `inclusive_scan`, `exclusive_scan`, `transform_inclusive_scan`, `transform_exclusive_scan`.

Exemple : les appels suivants retournent le résultat d'une réduction (somme généralisée décomposable en sous-intervalle permutable)

- `std::reduce(v.begin(), v.end())` : (défaut) reduction séquentielle.
- `std::reduce(std::sequential, v.begin(), v.end())` : idem, politique explicite.
- `std::reduce(std::par, v.begin(), v.end())` : réduction parallèle.

5 Autres composants de la STL

La STL est constituée d'autres composants :

- la bibliothèque d'entrée/sortie (cin, cout, ifstream, ofstream, ...)

- les chaines de caractères (string)
- les expressions régulières (regex), depuis $\mathtt{C}_{11}^{++}$
- la génération de nombres aléatoires (congruence linéaire, mersenne twister 32/64 bits,lagged fibonacci 24/48 bits), depuis $\mathtt{C}_{11}^{++}$.
- la bibliothèque numérique (ratio, complex, valarray, gestion statut float, ...), à partir de $\mathtt{C}_{17}^{++}$ seront ajoutés les fonctions mathématiques spéciales.
- l'ensemble des fonctions mathématiques standard a été complété depuis $\mathtt{C}_{11}^{++}$ (voir http ://en.cppreference.com/w/cpp/numeric/math).
- différent type d'horloges, dont une horloge haute résolution pour les mesures de temps ($\mathtt{C}_{11}^{++}$)
- `pair`, `tuple` ($\mathtt{C}_{11}^{++}$)
- opération atomique, thread, mutex, future ($\mathtt{C}_{11}^{++}$)
- système de fichiers ($\mathtt{C}_{17}^{++}$)

Voir la documentation pour plus de détails.

6 Extension STL

Extension de la STL : ajout de types supplémentaires (des itérateurs, des conteneurs, des algorithmes).

Principes à respecter :

- tout ce qui se comporte comme un itérateur est un itérateur (*i.e.* respecte les spécification d'un itérateur)
- tout ce qui se comporte comme un conteneur est un conteneur (*i.e.* respecte les spécification d'un conteneur)

Limitations : dans la mesure où le respect de ces spécifications ne nuit pas à l'objectif du conteneur.

exemple : conteneur avec un stockage circulaire ⇒ difficile d'utiliser un itérateur standard.

Dérivation d'un type de la STL :

Idée : étendre le comportement d'un type de la STL en dérivant de ce type et en ajoutant de nouveaux comportements/propriétés.

Problème : pour des raisons de performance, les classes de la STL n'ont pas de méthodes virtuelles.

Conséquences : ne sont pas adaptées aux polymorphismes à travers l'héritage public (car pas de destructeur virtuel).

Solutions :

- hériter de la classe de manière privée.
- utiliser la STL de manière interne (i.e. comme un membre)

Conclusion

Nous avons vu dans cette leçon que :

- les conteneurs STL implémentent les TDAs classiques :
  - ◇ sous forme de template (utilisable sur tout objet),
  - ◇ sans besoin de mise au point,
  - ◇ gère les copie et les déplacements, pour les lvalues et rvalues,
  - ◇ avec des garanties sur les exceptions
- un itérateur est un moyen standard de parcourir un conteneur,
- les algorithmes de la STL sont des implémentations d'algorithmes standards sur tout conteneur implémentant un itérateur standard.
- il est peu probable que votre implémentation soit plus efficace et plus robuste (en terme de bugs ou d'exception) que celle de la STL si vous implémentez des objets de base correctement conçus et des itérateurs adaptés à vos conteneurs.
- la connaissance et la maîtrise de la STL permet de gagner un temps considérable en terme de conception et de codage.