Chapitre 12

itérateurs
<algorithm>
<numeric>

1. Itérateurs

Iterators, quid boni faciunt ?

- Les itérateurs fournissent un interface commun à tous les conteneurs de la STL
- Chaque conteneur fournit des méthodes `begin()` et `end()` qui retournent des `conteneur<T>::iterator` vers le premier élément et la fin du contenu.
- Ils permettent d'écrire des fonctions génériques capables de lire / modifier le contenu indépendamment du conteneur dans lequel il est stocké

```
template<typename Iterator>
void d(Iterator first, Iterator last) {
    cout << "[";
    for (Iterator it = first; it != last; ++it) {
        if (it != first) cout << ", ";
        cout << *it;
    }
    cout << "]\n";
}

int main() {
    vector<float> c1{1, 2, 3}; d(c1.begin(), c1.end());
    array<int, 3> c2{4, 5, 6}; d(c2.begin(), c2.end());
    list<double> c3{7, 8};     d(c3.begin(), c3.end());
    set<short> c4{9, 10, 11};  d(c4.begin(), c4.end());
}
```

```
[1, 2, 3]
[4, 5, 6]
[7, 8]
[9, 10, 11]
```

Itérateur minimal

- Tous les itérateurs ne sont pas créés égaux. Un `std::vector<T>::iterator` est capable de bien plus de choses qu'un `std::forward_list<T>::const_iterator`
- Cependant, tout itérateur de la STL dispose de
  - Initialisation et affectation par copie. `it1`, `it2`, et `it3` itèrent sur le même élément

```
vector<int>::iterator it1, it2;
vector<int>::iterator it3 = it1;
it2 = it1;
```

  - Accès à une (`const`) référence à l'élément itéré

```
*it1;
it1->m; // si *it1 est un objet
```

  - Passage à l'élément suivant

```
++it1;
it1++;
```

  - Test d'égalité. Les itérateurs itèrent-ils sur le même élément ?

```
it1 == it2;
it1 != it2;
```

Exemple

- Revenons à notre fonction d'affichage

```
template<typename Iterator>
void display(Iterator first, Iterator last) {
    cout << "[";
    for (Iterator it = first; it != last; ++it) {
        if (it != first) cout << ", ";
        cout << *it;
    }
    cout << "]\n";
}
```

- Elle utilise l'affectation =, le test d'égalité !=, l'incrémentation à l'élément suivant ++, et le déréférencement *
- Toutes ces opérations sont disponibles avec un itérateur minimal. Cette fonction est donc utilisable avec tout conteneur de la STL donc le contenu est affichable avec <<

Itérateurs sur un tableau (vector, array, deque)

- Les itérateurs sur les éléments d'un tableau permettent en plus de
  - parcourir les éléments en ordre inverse (*Bidirectionnal Iterator*)
  - accéder aux élément via leur indice (*Random Access Iterator*)
- Ils se comportent essentiellement comme des pointeurs. Il est possible de les
  - Décrémenter pour passer à l'élément précédent

```
--it; it--;
```

  - Décaler de n éléments en une fois

```
it + n; n + it; it - n;
it1 - it2; it += n; it -= n;
```

  - Comparer les positions relatives des éléments pointés dans le conteneur

```
it1 < it2; it1 > it2;
it1 <= it2; it1 >= it2;
```

  - Décaler et déréférencer en une fois

```
it[n]; // équivalent à *(it + n)
```

Simuler l'accès aléatoire

- Le header `<iterator>` de la STL fournit les fonctions suivantes qui simulent un accès aléatoire pour les itérateurs ne le fournissant pas
  - `next(it,n)` est équivalent à `it+n` quand l'opérateur `+` est disponible, et à appliquer `n` fois l'opérateur `++` sinon (`--` pour `n` négatif)
  - `prev(it,n)` est équivalent à `it-n` quand l'opérateur `-` est disponible, et à appliquer `n` fois l'opérateur `--` sinon (`++` pour `n` négatif)
  - `next(it)` et `prev(it)` sont équivalent à `next(it,1)` et `prev(it,1)` resp.
  - `advance(it,n)` est équivalent à `it+=n` quand l'opérateur `+=` est disponible, et à appliquer `n` fois l'opérateur `++` (`--` si `n` est négatif) sinon
  - `distance(it1, it2)` est équivalent à `it2-it1` quand l'opérateur `-` est disponible, et compter le nombre de `++` nécessaires à passer de `it1` à `it2` sinon
- Utiliser ces fonctions rend vos fonctions génériques appelables avec les itérateurs de plus de types de conteneurs. L'exécution sera plus lente pour les itérateurs non *Random Access*

begin et end

- Le header `<iterator>` fournit aussi les fonctions `begin` et `end` qui
  - prennent un conteneur `c` en paramètre et retournent `c.begin()` et `c.end()`
  - offrent une interface équivalente pour un tableau classique « à la C »

```
template<typename Iterator>
void display(Iterator first, Iterator last);

int main() {
   vector<float> c1{1, 2, 3};
   display(begin(c1), end(c1));
   display(c1.begin(), c1.end());

   int t[] = {4, 5, 6};
   display(begin(t), end(t));
   display(t,t+3);
}
```

```
[1, 2, 3]
[1, 2, 3]
[4, 5, 6]
[4, 5, 6]
```

iterator et const

- L'accès en lecture seule ou en lecture/écriture aux éléments itérés est gérée par le type d'itérateur utilisé
  - un `conteneur<T>::iterator it;` retourne une `T&` quand on déréférence `*it`. Il se comporte comme un `T*`
  - un `conteneur<T>::const_iterator it;` retourne une `const T&` quand on déréférence `*it`. Il se comporte comme un `const T*`. On ne peut pas écrire dans `*it`.
- Le type d'itérateur disponible dépend de la constance du conteneur et de la méthode appelée
  - Si le conteneur `c` est variable, `c.begin()` et `c.end()` retournent des `iterator`, `c.cbegin()` et `c.cend()` retournent des `const_iterator`
  - Si le conteneur `c` est constant, `c.begin()`, `c.end()`, `c.cbegin()` et `c.cend()` retournent tous des `const_iterator`

Parcours en ordre inverse

- Les conteneurs parcourables de droite à gauche fournissent également
  - Les méthodes `.crbegin()` et `.crend()` qui retournent des `conteneur<T>::const_reverse_iterator`
  - Les méthodes `.rbegin()` et `.rend()` qui retournent des `conteneur<T>::reverse_iterator` si le conteneur est variable, et `conteneur<T>::const_reverse_iterator` sinon
- Avec ces itérateurs inversés, ++ passe à l'élément précédent, -- au suivant

```
template<typename Iterator> void display(Iterator first, Iterator last);

int main() {
   array<float, 3 > v{1, 2, 3};
   display(v.cbegin(),v.cend());
   display(v.crbegin(),v.crend());
}
```

```
[1, 2, 3]
[3, 2, 1]
```

Parcours partiel

- En décalant les itérateurs de début et de fin, on ne parcourt qu'une partie du contenant.
- `v.begin()+i` itère sur l'élément `v[i]`
- `v.end()-i` itère sur l'élément `v[v.size()-i]`
- La plage `[first,last[` inclut `first`, mais pas `last`, le premier élément non traité

```
template<typename Iterator> void display(Iterator first, Iterator last);

int main() {
   vector<float> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   display(v.begin(), v.begin() + 4);
   display(v.begin() + 4, v.end());
   display(v.begin(), v.end() - 3);
   display(v.end() - 4, v.end());
   display(v.begin() + 3, v.end() - 3);
}
```

```
[1, 2, 3, 4]
[5, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7]
[7, 8, 9, 10]
[4, 5, 6, 7]
```

2. `std::vector<T>` avec itérateurs

Nouvel interface pour `std::vector<T>`

- La syntaxe des itérateurs nous fournit de nouvelles manières d'appeler les méthodes vues au chapitre 9, mais également de nouvelles méthodes
  - Initialisation depuis une plage d'éléments
  - Assignation depuis une plage d'éléments
  - Insertion d'un élément ou d'une plage d'éléments à un emplacement quelconque
  - Suppression d'un élément ou d'une plage d'éléments

Initialisation par une plage d'éléments

- Syntaxe :

```
vector(const_iterator first, const_iterator last);
```

- Equivalent à :

```
vector<T> v;
v.reserve(distance(first,last));
for (auto it = first; it != last; ++it)
   v.push_back(*it);
```

- Exemples :

```
array<int,4> a{1,2,3,4};
vector v1(a.begin(),a.end()-1);
// v1 de type vector<int>, contient {1,2,3}

list<double> l{5,6,7};
vector v2(next(l.begin()),l.end());
// v2 de type vector<double>, contient {6,7}
```

assign

- Syntaxe :

```
void assign(const_iterator first, const_iterator last);
```

- Equivalent à :

```
v.resize(distance(first,last));
auto ot = v.begin();
for (auto it = first; it != last; ++it, ++ot) *ot = *it;
```

- Exemple :

```
array<int,4> a{1,2,3,4};
vector<int> v;
v.assign(a.begin(),a.end()-2);
// v contient {1,2}
```

erase

- La méthode `erase` permet de supprimer des éléments en positions quelconques.

```
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
```

- Elle supprime soit un seul élément, soit la plage d'éléments `[first,last[`
- Elle retourne l'itérateur qui suit le dernier élément supprimé
- Exemples :

```
vector<int> v{2, 3, 5, 7, 11, 13, 17};
auto it = v.erase(v.begin() + 3);
display(v.begin(),v.end()); cout << *it << endl;
it = v.erase(v.begin() + 2, v.end() - 1);
display(v.begin(),v.end()); cout << *it << endl;
```

```
[2, 3, 5, 11, 13, 17]
11
[2, 3, 17]
17
```

insert

- La méthode insert permet d'insérer des éléments à une position donnée

```
iterator insert(const_iterator pos, T val);
iterator insert(const_iterator pos, size_t n, T val);
iterator insert(const_iterator pos, std::initializer_list<T> agregat);
template<typename It> iterator insert( const_iterator pos, It first, It last );
```

- Retourne un itérateur vers le premier élément inséré

```
vector<int> v{2, 3}; array<int,2> a{8, 9};
v.insert(v.begin() + 1, 1);     display(v.begin(),v.end());
v.insert(v.begin() + 2, 2, 4);  display(v.begin(),v.end());
v.insert(v.begin(), {5, 6, 7}); display(v.begin(),v.end());
auto it = v.insert(v.end(), a.begin(), a.end());
display(v.begin(),v.end()); cout << *it;
```

```
[2, 1, 3]
[2, 1, 4, 4, 3]
[5, 6, 7, 2, 1, 4, 4, 3]
[5, 6, 7, 2, 1, 4, 4, 3, 8, 9]
8
```

3. <algorithm>

La librairie `<algorithm>`

- Cette librairie fournit un catalogue de fonctions similaires à notre fonction `display` qui s'appliquent à des plages d'éléments spécifiées par deux itérateurs `first` et `last`
- On les groupes en catégories

`PRG1`

- Opérations séquentielles qui ne modifient pas le contenu
- Opérations séquentielles qui modifient le contenu
- Minimum et maximum

`ASD`

- Opérations de partition
- Opérations de tri
- Recherches dichotomiques
- Opérations sur plages d'éléments triées
- Opérations sur les ensembles
- Opérations sur les tas
- Permutations

Structure de cette section

Pour chacune des trois catégories traitées dans ce cours (et pour `<numeric>`)

- Nous présentons la liste complète des fonctions disponibles ainsi qu'une très brève description de leur effet
- Nous illustrons par quelques exemples la manière dont `<algorithm>` passe en paramètre et retourne les différents types d'information.
- Sur la base de ces principes généraux, vous devrez être capable de
  - Comprendre l'effet d'un appel à toute fonction de ces catégories
  - Utiliser ces fonctions à bon escient
  - En disposant de leur liste et de leurs interfaces dans l'aide-mémoire fourni

Opérations séquentielles non modifiantes

- `all_of`, `any_of`, `none_of` vérifient si une condition est vraie sur tout, au moins un, ou aucun élément
- `for_each`, `for_each_n` appliquent une fonction à tous les éléments
- `count`, `count_if` comptent le nombre d'éléments qui respectent un critère
- `equal` teste l'égalité entre 2 séquences , `mismatch` trouve la première position où 2 plages diffèrent, `lexicographical_compare` fournit une forme d'`operator<`
- `find`, `find_if`, `find_if_not` trouvent le premier élément qui respecte un critère
- `find_first_of` trouve le premier élément qui appartient à un ensemble
- `search`, `search_n`, `find_end` trouvent une sous-séquence dans la séquence d'éléments
- `adjacent_find` trouve les premiers éléments consécutifs égaux

count(first, last, val)

- Prenons en exemple la fonction de comptage dont l'interface est l'un des + simples

```
template<typename It, typename T >
ptrdiff_t count(It first, It last, const T& value );
```

- Une mise en œuvre possible par la STL serait

```
template<typename It, typename T >
ptrdiff_t count(It first, It last, const T& value ) {
   ptrdiff_t cnt = 0;
   for (; first != last; ++first)
      if (*first == value) ++cnt;
   return cnt;
}
```

- Il permet de traiter diverses plages d'éléments

```
vector<int> v{1, 2, 3, 2, 1, 2, 3, 3, 2};
cout << count(v.begin(), v.end(), 2) << ' '
     << count(v.cbegin() + 3, v.cend(), 3) << ' '
     << count(v.crbegin() + 1, v.crend(), 2);
```

```
4 2 3
```

count_if(first,last,predicat)

- L'autre fonction de comptage ne compte pas une valeur mais les éléments qui respectent un critère spécifié par une fonction reçue en paramètre. Sa syntaxe est

```
template<class It, class UnaryPredicate>
ptrdiff_t count_if(It first, It last, UnaryPredicate p);
```

- Une possible mise en oeuvre possible par la STL serait

```
template<class It, class UnaryPredicate>
ptrdiff_t count_if(It first, It last, UnaryPredicate p) {
   ptrdiff_t cnt = 0;
   for (; first != last; ++first)
      if (p(*first)) ++cnt;
   return cnt;
}
```

- Le troisième paramètre doit donc être le nom d'une fonction ou quelque chose de similaire, tel qu'une instanciation d'une fonction générique, un foncteur (objet d'une classe définissant l'opérateur ( )), ou une expression lambda.

count_if(first,last,p): Exemples

```cpp
bool est_pair(int e) { return e % 2 == 0; }

template<typename T, T n = 2>
bool est_multiple_generique(T e) { return e % n == 0; }

struct est_multiple_foncteur{
   int n;
   bool operator() (int e) const { return e % n == 0; }
};

int main() {
   vector<int> v{1, 2, 3, 4, 5, 6, 7};
   int n = 2;
   cout << count_if(v.begin(), v.end(), est_pair) << ' '
        << count_if(v.begin(), v.end(), est_multiple_generique<int, 2>) << ' '
        << count_if(v.begin(), v.end(), est_multiple_foncteur{n}) << ' '
        << count_if(v.begin(), v.end(), [&n](int e) { return e % n == 0; });
}
```

3 3 3 3

for_each_n(first, n, fn)

for_each(first,last,fn)

- Les fonctions dont le nom se termine par _n offrent un interface alternatif pour spécifier les séquence d'éléments à traiter : itérateur de début et nombre d'éléments
- Avec l'interface habituel, la séquence traitée serait `[first, next(first,n)[`
- Les fonctions `foreach` et `foreach_n` appliquent la fonction (ou foncteur) `fn` à tous les éléments de la séquence
- `foreach_n` retourne `next(first,n)`
- `foreach` retourne `fn`, ce qui peut être utile quand c'est un foncteur
- Mise en œuvre possible de `foreach_n` par la STL ➔

```
template<typename It, typename Size,
         typename Function>
It for_each_n(It first, Size n,
              Function f)
{
   for (Size i = 0; i < n; ++first, ++i)
      f(*first);
   return first;
}
```

for_each_n(first, n, fn) : Exemples

for_each(first,last,fn)

```
void print(int i) { cout << i << ' '; }

int main() {
   vector<int> v{1, 2, 3, 4, 5};
   for_each_n(v.begin(), 5, print);     cout << '\n';

   for_each_n(v.begin(), 3, [](int& n) { n *= n; });

   for_each(v.begin(), v.end(), print); cout << '\n';

   struct Sum {
      int sum;
      void operator()(int n) { sum += n; }
   };
   Sum s = for_each(v.cbegin(), v.cend(), Sum{0});
   cout << "Somme: " << s.sum << '\n';
}
```

```
1 2 3 4 5
```

```
1 4 9 4 5
```

```
Somme: 23
```

search(first1,last1,first2,last2)

search(first1,last1,first2,last2,pred)

- Une fonction qui prend plusieurs plages en paramètres aura typiquement un type générique par paire d'itérateurs de plage
- Cela permet par exemple que `[first1,last1[` parcoure une `const std::string` tandis que `[first2,last2[` provienne d'un `vector<char>`
- La fonction `search` recherche la suite d'éléments `[first2,last2[` dans la plage `[first1,last1[`, similairement à la méthode `.find()` de `std::string`.
- Elle retourne un itérateur sur le premier élément de la sous-plage de `[first1,last1[` si la recherche est fructueuse, et `last1` sinon
- Mise en œuvre possible par la STL ➔

```
template<class It1, class It2>
It1 search (It1 first1, It1 last1,
            It2 first2, It2 last2)
{
   if (first2==last2) return first1;
   while (first1!=last1)
   {
      It1 it1 = first1; It2 it2 = first2;
      while (*it1==*it2)
      // while(pred(*it1,*it2))
      {
         ++it1; ++it2;
         if (it2==last2) return first1;
         if (it1==last1) return last1;
      }
      ++first1;
   }
   return last1;
}
```

search(f1,l1,f2,l2,p): Exemples

```
vector<int> haystack{11, 23, 41, 53, 32, 41, 53, 23, 34, 56};

// version avec operator==
array<int, 3> needle1{41, 53, 23};
auto it = search(begin(haystack), end(haystack),
                 begin(needle1), end(needle1));

if (it != end(haystack))
   cout << "needle1 à l'index " << distance(begin(haystack), it) << '\n';
```

```
needle1 à l'indice 5
```

```
// version avec prédicat
int needle2[] = {1, 3, 2};
it = search(begin(haystack), end(haystack),
            begin(needle2), end(needle2),
            [](int a, int b) { return a % 10 == b % 10; });

if (it != end(haystack))
   cout << "needle2 à l'index " << distance(begin(haystack), it) << '\n';
```

```
needle2 à l'indice 2
```

equal(first1,last1,first2)

equal(first1,last1,first2,pred)

- Quand deux plages d'éléments ont la même longueur, celle-ci n'est spécifiée qu'une seule fois et la paramètre `last2` est omis
- La fonction `equal` vérifie si deux plages d'éléments ont un contenu identique et retourne un booléen
- Mise en œuvre possible par la STL ➔
- Exemple ↓

```
template <class It1, class It2>
bool equal (It1 first1, It1 last1,
            It2 first2)  // last2 absent
{
   while (first1!=last1) {
      if (!(*first1 == *first2))
      // if (!pred(*first1,*first2))
         return false;
      ++first1; ++first2;
   }
   return true;
}
```

```
array<int,7> a{1, 2, 3, 4, 2, 3};          // 1,2,3,4,2,3,0
vector<int> v(a.begin() + 1, a.end() - 4); // 2,3

for (size_t i = 0; i < a.size() - v.size(); ++i)
   if (equal(v.begin(), v.end(), a.begin() + i))
      cout << "v inclus dans a à l'indice " << i << '\n';
```

```
v inclus dans a à l'indice 1
v inclus dans a à l'indice 4
```

Opérations séquentielles modifiantes

- `transform` applique une fonction à une ou plusieurs plages d'éléments et stocke le résultat dans une autre plage
- `copy`, `copy_if`, et `copy_n` copient une plage d'éléments à un nouvel emplacement
- `move` déplace (voir chap. 15) une plage d'éléments à un nouvel emplacement
- `copy_backward` et `move_backward` font de même en ordre inverse
- `fill` et `fill_n` remplissent la plage d'une même valeur
- `generate` et `generate_n` font de même avec une valeur générée pour chaque élément par une fonction
- `remove`, `remove_if`, `remove_copy`, et `remove_copy_if` suppriment des éléments selon divers critères, soit en place, soit en copiant les éléments conservés ailleurs

Opérations séquentielles modifiantes (2)

- `replace`, `replace_if`, `replace_copy`, et `replace_copy_if` le remplacent par une valeur plutôt que de les supprimer
- `swap_ranges` échange le contenu de 2 séquences qui ne se chevauchent pas
- `reverse` et `reverse_copy` inverse l'ordre de la séquence
- `rotate` et `rotate_copy` effectuent une rotation circulaire des éléments
- `shift_left` et `shift_right` (C++20) déplacent les éléments vers la droite ou la gauche
- `random_shuffle` mélange les éléments aléatoirement
- `sample` extrait n éléments au hasard parmi la séquence
- `unique` et `unique_copy` suppriment les éléments consécutifs dupliqués
- `sort` et `stable_sort` trient la séquence d'éléments, par défaut par ordre croissant

copy(first, last, d_first)

- Les fonctions telles que copy qui écrivent une séquence en reçoive l'emplacement d_first du premier élément à écrire en paramètre
- Le fin ou le nombre d'éléments à écrire n'est pas passé explicitement en paramètre. Ici ce sera distance(first, last)
- L'utilisateur est responsable de fournir un emplacement avec suffisamment de place
- La fonction retourne un itérateur sur le premier emplacement d'écriture non utilisé

```
template<class InputIt, class OutputIt>
OutputIt copy(InputIt first, InputIt last,
              OutputIt d_first)
{
   for (; first != last; ++first, ++d_first)
      *d_first = *first;
   return d_first;
}
```

```
const array<int, 5> a{1, 2, 3, 4, 5};
vector<int> v(a.size());
copy(a.begin(), a.end(), v.begin());

auto print_int = [](int n) { cout << n << ' '; };
for_each(v.begin(), v.end(),print_int);

copy(a.rbegin()+1,a.rend()-1,v.begin()+1);
for_each(v.begin(), v.end(),print_int);
```

```
1 2 3 4 5
1 4 3 2 5
```

transform(first, last, d_first, op)

transform(first1, last1, first2, d_first, op)

- Rien n'empêche d'utiliser la même séquence en entrée et en sortie.
- C'est fréquemment le cas avec la fonction transform qui applique une opération unaire (binaire) à une (deux) séquence(s) et en stocke le résultat dans une séquence qui poeut être autre ou pas

```
template<typename InputIt, typename OutputIt, typename UnaryOperation>
OutputIt transform(InputIt first1, InputIt last1,
                   OutputIt d_first,
                   UnaryOperation op)
{
   while (first1 != last1)
      *d_first++ = op(*first1++);
   return d_first;
}
```

transform(first, last, d_first, op) : Exemples

transform(first1, last1, first2, d_first, op)

```
string s{"hello"};
transform(s.cbegin(), s.cend(),
          s.begin(), // même emplacement
          [](unsigned char c) { return std::toupper(c); });
cout << s << '\n';
```

```
HELLO
```

```
// alternative avec std::for_each et un passage par référence
for_each(s.begin(), s.end(),
         [](char& c) { c = std::tolower(c); });
cout << s << '\n';
```

```
hello
```

```
vector<int> a { 1, 2, 3, 4, 5}, b { -1, 1, -1, 1, -1};
vector<int> c(a.size());
transform(a.cbegin(), a.cend(), b.cbegin(), c.begin(),
          [](int e1, int e2){ return e1 * e2; });
for (int e : c) cout << e << ' ';
```

```
-1 2 -3 4 -5
```

remove(first, last, val)

remove_if(first, last, predicat)

- Pour une fonction telle que `remove` - qui supprime les éléments de valeur `val` – on ne connait pas a priori le nombre d'éléments restants
- La fonction ne peut pas changer la taille du conteneur sur lequel elle itère, n'y ayant pas accès. Elle peut seulement copier / déplacer les éléments
- Elle retourne le nombre d'éléments restants sous la forme d'un itérateur sur le premier emplacement inutilisé

```
template<typename It, typename T>
It remove(It first, It last, const T& val)
{
   first = find(first, last, val);
   if (first != last)
      for (It i = first; ++i != last;)
         if (!(*i == val))
            *first++ = *i;
   return first;
}
```

- L'utilisateur utilise par exemple cette information en appelant `vector<T>::erase()`
- Notons la mise en œuvre ci-dessus qui n'effectue qu'un seul passage sur la séquence en itérant en lecture avec `i`, et en écriture avec l'itérateur décalé `first`

remove(first, last, val) : Exemples

remove_if(first, last, predicat)

```
string s1 {"Texte avec des espaces"};

auto new_end = remove(s1.begin(), s1.end(), ' ');
cout << s1.size() << " : " << s1 << endl;
```

```
22 : Texteavecdesespacesces
```

```
s1.erase(new_end, s1.end());
cout << s1.size() << " : " << s1 << endl;
```

```
19 : Texteavecdesespaces
```

```
string s2 {"Texte Avec Des Majuscules"};
s2.erase(remove_if(s2.begin(), s2.end(),
                   [](char c) { return std::isupper(c); }),
         s2.end());
cout << s2 << '\n';
```

```
exte vec es ajuscules
```

copy_if(first, last, d_first)

- La fonction `copy_if` combine l'écriture dans une autre séquence (comme `copy`) et l'ignorance du nombre d'éléments copiés (comme `remove`)
- Elle retourne le premier élément non utilisé, i.e. le nouveau `.end()`, ce qui permet d'appliquer une technique similaire à celle de `remove`
  - créer un conteneur a priori trop grand
  - le remplir partiellement avec `copy_if`
  - redimensionner le conteneur a posteriori

```
template<typename InputIt,
         typename OutputIt,
         typename Predicate>
OutputIt copy_if(InputIt first, InputIt last,
                 OutputIt d_first,
                 Predicate pred)
{
   for (; first != last; ++first)
      if (pred(*first))
         *(d_first++) = *first;

   return d_first;
}
```

- Une approche plus efficace pour remplir un `std::vector` en sortie consiste à utiliser un `back_inserter` , défini dans `<iterator>`, qui écrit en appelant la méthode `.push_back()`

copy_if(first, last, d_first) : Exemples

```
template<int n> bool est_multiple(int i) { return i % n == 0; }

void display(span<const int> s) { for (int e : s) cout << e << ' '; cout << '\n'; }

int main() {
   vector v {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

   vector<int> v1(v.size());
   auto it = copy_if(v.cbegin(), v.cend(),
                     v1.begin(),
                     est_multiple<3>);        display(v1);
   v1.erase(it, v1.end());                    display(v1);

   vector<int> v2;
   copy_if(v.cbegin(), v.cend(),
           back_inserter(v2),
           est_multiple<3>);                  display(v2);
}
```

```
3 6 9 0 0 0 0 0 0 0
3 6 9
```

```
3 6 9
```

Minimum et maximum

- `min` et `max` retourne le minimum / maximum entre 2 valeurs ou au sein d'un agrégat
- `minmax` retourne les deux sous la forme d'une `std::pair`
- `min_element` et `max_element` retournent un itérateur sur l'élément min / max d'une séquence
- `minmax_element` retourne une paire d'itérateurs
- `clamp` ramène une valeur dans un intervalle `[lo,hi]` en montant les valeurs plus petites que `lo` à lo, `et` en descendant celles plus grandes que `hi` à `hi`

min(a,b,comp)

- `min`, comme `max`, `sort`, `binary_search`, … a un dernier paramètre optionnel `comp`
- Il permet de remplacer la comparaison par défaut avec `operator<` par une autre fonction de comparaison
- Le header `<functional>` fournit des foncteurs génériques utiles tels que `less<T>`, `greater<T>`, `equal<T>`, …

```
bool compare_unite(int a, int b) {
    return a % 10 < b % 10;
}

int main() {
    cout << min(13, 21);
      // 13
    cout << min(13, 21, compare_unite);
      // 21
    cout << min(13, 21, greater<int>());
      // 21
}
```

min({a, b, c, …})

- La fonction `min` permet de trouver le minimum de plus de 2 valeurs en prenant un agrégat en paramètre. Cet agrégat est de type `std::initializer_list<T>`
- Tous les éléments de l'agrégat doivent être du même type.

```
int main() {
   int a = 2, b = 3, c = 1;

   cout << min({a, b, c});                                    // 1

   cout << min({1.1, 2.0, 3.1, 0.8, 1.5});                    // 0.8

   cout << min({1.1, 2.0, 3.1, 0.8, 1.5}, greater<double>()); // 3.1

 // La ligne suivant ne compile pas.
 // candidate template ignored: conflicting types for parameter '_Tp'
 // ('int' vs. 'double') min(initializer_list<_Tp> __t)

   cout << min({1, 2.0, 3});
}
```

min_element(first, last, comp)

- Retourne un itérateur sur l'élément le plus petit de la plage `[first, last[`.
- Permet d'écrire un tri par sélection en deux lignes

```
template<typename Iterator, typename Comp>
void tri_selection(Iterator first, Iterator last,
                   Comp comp = less<typename Iterator::value_type>()) {
   for(Iterator it = first; it != last; ++it)
      swap(*it, *min_element(it, last, comp));
}

int main() {
   vector<int> v{3, 2, 4, 7, 5, 6, 1};
   tri_selection(v.begin(), v.end(), greater<int>());
   for(int e : v) cout << e << ' ';
}
```

```
7 6 5 4 3 2 1
```

4. <numeric>

Numeric

`<numeric>` définit des algorithmes qui utilisent les opérateurs arithmétiques +, -, *, …

- `iota` remplit une plage avec des valeurs incrémentales à partir d'une valeur donnée
- `accumulate` , `reduce`, `transform_reduce` calculent la somme de tous les éléments d'une plage d'éléments, avec une autre opération que `operator+`
- `inner_product` calcule le produit scalaire (somme des produits de chaque paire d'éléments) de 2 plages d'éléments, éventuellement avec d'autres opérations que `operator+` et `operator*`
- `adjacent_difference` calcule `w[i] = v[i]-v[i-1]`, éventuellement avec une autre opération que `operator-`
- `partial_sum`, `inclusive_scan`, `exclusive_scan`, `transform_inclusive_scan`, `transform_exclusive_scan` calculent des sommes partielles, i.e. $w[i] = \sum_{j=0}^{i} v[j]$

iota(first, last, value)

- Remplit la plage [first, last[avec value, value+1, value+2, …
- Fonctionne avec tout type qui définit operator++, y compris des pointeurs, des itérateurs, …

```
int main() {
   vector<int> v1(11);
   iota(v1.begin(),v1.end(),-5);
   for(int e : v1) cout << e << ' ';
   cout << endl;

   const vector<int> v2(v1);
   vector<vector<int>::const_iterator>
      v3(v2.size());
   iota(v3.begin(),v3.end(),v2.cbegin());
   reverse(v3.begin(),v3.end());
   for(auto it : v3) cout << *it << ' ';
   cout << endl;
}
```

```
-5 -4 -3 -2 -1 0 1 2 3 4 5
5 4 3 2 1 0 -1 -2 -3 -4 -5
```

partial_sum(first, last, d_first, binary_op)

- Avec v dans la plage [first, last[et w dans la plage [d_first, d_first + (last-first)[, calcule $w[i] = \sum_{j=0}^{i} v[j]$
- Peut utiliser une autre opération binaire que operator+.
- <functional> définit des foncteurs utiles tels que plus<T>, minus<T>, multiplies<T>, divides<T>,et modulus<T>

```
int main() {
    vector<int> v(10,1);
    partial_sum(v.begin(),v.end(),
                v.begin());

    cout << "Nombres de 1 a 10 \n";
    for(int e : v) cout << e << ' ';
    cout << endl;

    partial_sum(v.begin(),v.end(),
                v.begin(),multiplies<int>());

    cout << "Factorielles de 1 a 10 \n";
    for(int e : v) cout << e << ' ';
    cout << endl;
}
```

```
Nombres de 1 a 10
1 2 3 4 5 6 7 8 9 10
Factorielles de 1 a 10
1 2 6 24 120 720 5040 40320 362880 3628800
```