Le langage C++ est un langage intéressant puisqu'il offre avec une grande souplesse d'écriture. Par rapport au langage C dont il hérite, il propose beaucoup plus de choses comme, bien entendu, toute la philosophie objet, mais également la possibilité de redéfinir les opérateurs ainsi que la possibilité de fabriquer des modèles. Malheureusement, il conserve aussi quelques tares issues de son prédécesseur, notamment les tableaux et les chaînes de caractères.

N'oubliez pas que les tableaux ne sont pas de véritables tableaux, mais des pointeurs vers des mémoires consécutives. Même si elle traite de caractères, la chaîne n'est pas mieux lotie puisqu'elle est également considérée comme un tableau (et donc comme un pointeur).

On peut se demander légitimement pourquoi avoir conservé cette ancienne façon de faire alors que le langage C++ est un langage évolué ? La réponse est simple : c'est la performance en terme de vitesse qui est privilégiée. Le programmeur C++ doit savoir se qu'il fait, même si les tableaux et les chaînes sont sources de nombreuses erreurs.

Dans les précédentes études, nous avons justement appris à fabriquer de véritables tableaux de type quelconque grâce à la notion de modèle « *template* », ainsi qu'une classe « *Chaîne* » qui représente les fonctionnalités attendues d'une véritable chaîne de caractères.

Dans cette étude, nous allons apprendre à utiliser des classes de haut niveau déjà toutes faites et développées par les ingénieurs de Hewlett Packard, comme justement les chaînes de caractères (appelées *string*) ainsi que les tableaux (appelés *vector*). Les précédentes études nous ont permis de comprendre les mécanismes mis en jeu, mais dès aujourd'hui, il sera préférable d'utiliser des composants déjà construits et surtout standardisés.

Les ingénieurs de Hewlett Packard ont développé beaucoup d'autres classes comme les nombres complexes, les nombres binaires, des classes conteneurs comme les listes, les piles, les ensembles, etc. Tous ces éléments sont rassemblés dans une bibliothèque standard et sont construits sous forme de modèles. Cette librairie s'appelle STL (Standard Template Library).

Contenu de la STL

Les ingénieurs de Hewlett Packard ont développé beaucoup de classes génériques qui sont utiles pour presque tous les programmes. Plutôt que de se casser la tête à systématiquement tout reconstruire, il est préférable d'utiliser ces classes qui sont suffisamment générique pour s'adapter à toutes les situations. Il existe plusieurs grands thèmes. Dans ce chapitre, nous recensons juste l'ensemble des classes. Dans les chapitres suivants, nous nous intéresserons plus particulièrement à leurs utilisations.

Les conteneurs séquentiels :

Il est très fréquent d'avoir besoin de stocker dans une même entité mémoire un ensemble d'objets de même type. En plus, il peut être intéressant d'utiliser un système qui fonctionne quelque soit l'objet que nous développons. Les conteneurs séquentiels permettent effectivement de stocker des objets en séquence, c'est-à-dire les uns à la suite des autres, ce qui permettra ensuite de parcourir le conteneur dans un ordre particulier. Il existe plusieurs conteneurs séquentiels :

⇒ ***vector*** : cette classe est un vecteur qui représente un tableau de haut niveau (les cases sont consécutives). Avec cette classe, il est possible d'atteindre n'importe quelle élément du tableau facilement grâce à l'indexation « [ ] ». Nous pouvons insérer de nouveaux éléments, en supprimer, etc.

⇒ ***list*** : cette classe implémente une liste doublement chaînée. Avec cette classe, il est plus facile de supprimer un élément particulier par rapport au vecteur (en effet pour un vecteur, si nous supprimons une case, il est nécessaire de décaler les cases suivantes vers le bas). Par contre, cette liste utilise systématiquement deux pointeurs pour parcourir la séquence ce qui prend plus de place en mémoire.

⇒ ***deque*** : cette classe est très peu utilisée et offre le même comportement mais plus spécialisé que la classe vecteur. Sa spécialisation consiste à pouvoir facilement ajouter ou retirer le premier élément. Cette classe est une abstraction d'une file pour laquelle le premier élément est retiré chaque fois.

Adaptateurs de conteneurs séquentiels :

La bibliothèque standard dispose de trois patrons particuliers qui s'ajoutent aux conteneurs séquentiels en modifiant leurs comportements classiques. Généralement, il s'agit d'une restriction et d'une adaptation à des fonctionnalités données :

⇒ ***stack*** : ce patron est destiné à la gestion des piles LIFO (Last In, First Out).

⇒ ***queue*** : ce patron est destiné à la gestion des piles de type FIFO (First In, First Out).

⇒ ***priority_queue*** : un tel conteneur ressemble à une file d'attente, dans laquelle on introduit toujours des éléments en fin.

Les conteneurs associatifs :

Les éléments d'un conteneur associatif ne sont plus placés dans un ordre particulier. Pour retrouver une entité, nous ferons appel dans ce cas là à une clé qui nous orientera vers la valeur recherchée.

⇒ ***map*** : ce conteneur représente une correspondance entre deux entités sous la forme d'une paire ***clé/valeur***, la ***clé*** étant utilisée pour la recherche et la ***valeur*** contenant les données que l'on souhaite utiliser. Par exemple, un répertoire téléphonique est représenté par une correspondance entre le nom de l'individu (la ***clé***) et son numéro de téléphone (la ***valeur***).

⇒ ***multimap*** : ce conteneur représente une multicorrespondance, il peut donc stocker plusieurs occurrences d'une même clé. Par exemple, une même personne peut posséder plusieurs numéros de téléphones.

⇒ ***set*** : ce conteneur représente la théorie des ensembles et contient une valeur de clé unique et supporte les requêtes concernant sa présence ou non. En effet, grâce à ce conteneur, nous pourrons indiquer si un élément fait parti de l'ensemble ou pas.

⇒ ***mulitset*** : représente également la théorie des ensembles avec en plus la possibilité de comptabiliser le nombre de fois qu'un même élément apparaît dans l'ensemble. Ce conteneur autorise donc la présence de plusieurs éléments identiques, ce qui n'est pas le cas pour le conteneur ***set***.

Les algorithmes génériques :

De même que la STL est composée de classes génériques, elle est également composée de fonctions génériques qui fournissent des opérations supplémentaires bien utiles sur les différents conteneurs étudiés précédemment. Ces fonctions permettent d'effectuer un certain nombre de traitements différents, comme des insertions, des copies, des recherches, etc., dans une suite d'éléments d'un des conteneurs utilisé. L'intérêt de ces fonctions génériques, que l'on appelle aussi algorithmes génériques, c'est qu'elles sont opérationnelles pour tous les types de conteneur, comme vector, list, map, etc.

La liste ci-dessous vous donnera une idée de quelques fonctions génériques intéressantes :

⇒ ***copy*** : copie d'une séquence dans une autre,
⇒ ***count*** : comptabilise le nombre de fois qu'un élément est présent dans une suite,
⇒ ***find*** : recherche d'une valeur particulière,
⇒ ***max_element*** : recherche du maximum,
⇒ ***min_element*** : recherche du minimum,
⇒ ***replace*** : remplacement de valeurs,
⇒ ***rotate*** : permutation de valeurs,
⇒ ***remove*** : suppression de valeurs,
⇒ ***unique*** : suppression de doublons,
⇒ ***sort*** : tri d'une séquence,
⇒ ***merge*** : fusion de deux conteneurs,
⇒ ***reverse*** : inverse l'ordre des éléments dans un conteneur.

Quelques classes bien utiles :

Nous avons commencé cette étude en indiquant que le langage C++ ne possédait pas certains éléments qui sont indispensables à la programmation de haut niveau, comme les chaînes de caractères. Par ailleurs, dans nos différentes études, nous avons en œuvre de toute pièce une classe qui représente les nombres complexes. Il faut savoir qu'une telle classe fait partie de cette bibliothèque. Voici une liste non exhaustive de classes qui me paraissent intéressante :

⇒ ***string*** : cette classe représente une chaîne de caractères et possède beaucoup de méthodes qui permettent tous les traitements possibles sur ses caractères. Dorénavant, lorsque nous aurons besoin d'une chaîne de caractères, ce sera systématiquement cette classe que nous prendrons.
⇒ ***complex*** : cette classe représente les nombres complexes,
⇒ ***bitset*** : cette classe représente les nombres binaires ou plus précisément un ensemble de bits et possèdent des méthodes associées à leurs traitements.

La classe « *string* » - (inclure <string>)

Un objet de type « *string* » contient a un instant donné, une suite formée d'un nombre quelconque de caractères. Sa taille peut évoluer dynamiquement au fil de l'exécution du programme. En fait, cette chaîne réserve un bloc mémoire suffisant pour stocker un certain nombre de caractères. Si la chaîne désirée est plus grand que cette zone réservée, la classe augmente automatiquement ce bloc en proposant une nouvelle allocation mémoire et en prenant la précaution d'avoir un bloc plus grand que nécessaire afin de répondre rapidement à une petite augmentation de la taille de la chaîne. La notion de caractère de fin de chaîne n'existe plus pour cette classe, et ce caractère de code nul peut apparaître au sein de la chaîne, éventuellement à plusieurs reprises.

---Constructions ----------

La classe « *string* » dispose de plusieurs constructeurs :

```
1  #include <string>  // nécessaire pour utiliser la classe "string"
2  using namespace std;
3  //-------------------------------------------------------------------
4  int main()
5  {
6    string ch1;               // construction d'une chaîne vide
7    string ch2(10, '*');      // chaîne de 10 caractères égaux à '*'
8    string mess1("bonjour");  // chaîne "bonjour" composée de 7 caractères
9    string mess2 = "bonjour"; // même chose
10   char *adr = "salut";
11   string mess3(adr);        // chaîne composée à partir d'un (char *)
12   string ch3 = mess1;       // construction par copie
13   ;
14   return 0;
15 }
16 //-------------------------------------------------------------------
```

--- Les autres méthodes ----------

La classe « *string* » dispose de beaucoup de méthodes (que l'on retrouvera dans d'autres classes génériques) bien utiles :

⇒ *append()* : ajoute une chaîne, à la fin d'une autre. Il s'agit d'une concaténation qui peut être également traitée par l'opérateur +=.
⇒ *assign()* : affecte à l'objet une nouvelle chaîne de caractères.
⇒ *at()* : permet de lire ou de récupérer un caractère à la position indiquée. La première position est 0. Il est nécessaire de donner une position compatible et inférieure à la taille de la chaîne sinon une exception est levée. Cette méthode est similaire à la redéfinition de l'opérateur « [ ] ».
⇒ *capacity()* : retourne la dimension du bloc mémoire réservé. Cette méthode fournit donc le nombre maximal de caractères qu'on pourra introduire, sans qu'il soit besoin de procéder à une nouvelle allocation mémoire. Les méthodes *reserve()* et *resize()* pouront être utilisée pour agir directement sur la capacité de ce bloc mémoire. La valeur retournée est toujours plus grande ou égale à la valeur que retourne la méthode *size()*.
⇒ *clear()* : vide entièrement la chaîne de caractères.

⇒ *compare()* : cette méthode gère l'ordre alphabétique et retourne une valeur numérique négative ou positive suivant le placement de la chaîne par rapport à celle qui est passée en argument. Une valeur négative indique que la chaîne se trouve avant celle qui est passée en argument. Une valeur positive dans le cas contraire, et une valeur nulle dans le cas où les deux chaînes sont rigoureusement identiques. La plupart du temps, il sera préférable d'utiliser les opérateurs relationnels « <, <=, ==, !=, >, >= » pour gérer ce genre de problème.

⇒ *c_str()* : cette méthode permet de passer d'une chaîne de type « *string* » vers une chaîne classique du C++ (*const char **). Attention, il est possible de récupérer cette chaîne sans toutefois pouvoir la modifier puisque une constante est déclarée.

⇒ *empty()* : retourne *true* si la chaîne est vide (sans aucun caractère), sinon retourne *false*.

⇒ *erase()* : efface une partie de la chaîne ou un caractère spécifié en argument.

⇒ *find(), rfind(), find_first_of(), find_last_of(), find_first_not_of(), find_last_not_of()* : effectuent des recherches sur une partie de la chaîne ou sur un caractère spécifié en argument.

⇒ *insert()* : permet d'insérer une autre chaîne ou bien un ou plusieurs caractères donnés.

⇒ *length()* : retourne la longueur de la chaîne de caractères. Similaire à la méthode *size()*.

⇒ *replace()* : remplace une partie de chaîne.

⇒ *reserve()* : réserve un bloc mémoire dont la taille est fixé par l'argument. Cette méthode doit être rarement utilisé, juste dans le cas où la performance en terme de rapidité est primordiale ou alors, éventuellement, dans le cas où nous sommes très limité dans la capacité de la mémoire.

⇒ *resize()* : donne une nouvelle dimension à votre chaîne de caractères. Attention ! à utiliser avec beaucoup de précaution.

⇒ *size()* : retourne la longueur d'une chaîne de caractères. Similaire à *length()*.

⇒ *substr()* : retourne une partie de chaîne.

⇒ *swap()* : assure la permutation de deux chaînes de caractères.

⇒ *begin() et end()* : ces opérations retournent des itérateurs au début et à la fin de la chaîne. Un itérateur est une abstraction d'un pointeur de classe générique, fourni par la bibliothèque standard. Ce sujet sera traité ultérieurement lorsque nous utiliserons la classe « *vector* ».

--- Redéfinition des opérateurs ----------

Un certain nombre d'opérateurs ont été redéfinis afin de permettre une écriture plus concise est plus intuitive que les méthodes que nous venons de voir. Certains opérateurs n'ont d'ailleurs pas de méthodes équivalentes.

⇒ = : affecte une chaîne à une autre. Cet opérateur est équivalent à la méthode *assign()* vue plus haut.

⇒ [ ] : joue le même rôle que pour une chaîne de caractères classique du C++. Est similaire à la méthode *at()*

⇒ += : ajoute une chaîne à la fin d'une autre. Cet opérateur joue le même rôle que la méthode *append()*.

⇒ + : cet opérateur assure la concaténation de deux chaînes de caractères pour former une troisième chaîne.

⇒ ==, !=, <, >, <=, >= : ces opérateurs renvoient *true* ou *false* suivant la comparaison qui est faite sur deux chaînes de caractères. Les différentes comparaisons évaluent en fait l'ordre alphabétique.

--- Opérateurs et fonction supplémentaires ----------

⇒ >>, << : il est bien entendu possible d'afficher ou de saisir des chaînes de caractères de type « string » en utilisant les opérateurs des classes « *iostream* ».

```
1 #include <string>
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6   string message;
7   cout << "Saisissez votre message : ";
8   cin >> message;
9   cout << "Votre message est : " << message;
10  return 0;
11 }
```

```
Saisissez votre message : Bonjour
Votre message est : Bonjour
```

```
Saisissez votre message : Bonjour à tous
Votre message est : Bonjour
```

⇒ *getline(istream&, string, char délimiteur)*: cette fonction est très utile lorsque nous devons saisir tout un texte à partir du clavier. En effet, lorsque nous réalisons une saisie classique à l'aide de « *cin* », les caractères de séparation (les espaces, les tabulations, aller à la ligne, etc.) ne sont pas tolérés. Cette fonction permet justement de saisir n'importe quel texte, avec même la possibilité de tolérer le retour à la ligne, et donc d'avoir un texte sur plusieurs lignes. Pour cela, Il est alors nécessaire de choisir un caractère qui servira de caractère de fin de texte.

```
1 #include <string>
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6   string message;
7   cout << "Saisissez votre message : ";
8   getline(cin, message, '\n');
9   cout << "Votre message est : " << message;
10  return 0;
11 }
```

Saisie d'un texte sur une seule ligne. Le retour à la ligne provoque la fin de la saisie

```
Saisissez votre message : Bonjour à tous
Votre message est : Bonjour à tous
```

--- Concaténation ----------

L'opérateur « **+** » a été redéfini de manière à permettre la concaténation,

⇒ de deux objets de type « string »,

⇒ d'un objet de type « string » avec une chaîne usuelle ou avec un caractère, et ceci dans n'importe quel ordre.

L'opérateur « **+=** » a également été redéfini pour la concaténation.

```cpp
#include <string>
using namespace std;
//-----------------------------------------
int main()
{
  string ch1 = "bon";   // ch1 <-- "bon"
  string ch2 = "jour"; // ch2 <-- "jour"
  string ch3;

  ch3 = ch1 + ch2;      // ch3 <-- "bonjour"
  ch3 = ch1 + ' ';      // ch3 <-- "bon "
  ch3 += ch2;           // ch3 <-- "bon jour"
  ch3 += " monsieur";   // ch3 <-- "bon jour monsieur"
  return 0;
}
//-----------------------------------------
```

--- Recherche d'une chaîne ----------

Ces méthodes permettent de retrouver la première ou la dernière occurrence d'une chaîne ou d'un caractère donnés, d'un caractère appartenant à une suite de caractères donnés, d'un caractère n'appartenant pas à une suite de caractères donnés.

Lorsqu'une telle chaîne ou un tel caractère a été localisé, on obtient en retour l'indice correspondant au premier caractère concerné ; si la recherche n'aboutit pas, on obtient une valeur d'indice en dehors des limites permises pour la chaîne, ce qui rend quelque peu difficile l'examen de sa valeur.

Heureusement, dans la classe « *string* », il existe un attribut constant public et statique appelé « *npos* » (qui veut dire « no position ») qui généralement est initialisé à la valeur -1. Lorsque vous utilisez une des méthodes de recherche, il serait souhaitable de tester la valeur retournée avec cette constante statique « *npos* » de « *string* » afin de savoir si votre recherche a aboutie..

```cpp
#include <string>
#include <iostream>
using namespace std;
//-----------------------------------------
int main(  )
{
  string message = "Bonjour";
  int position = message.find('n');
  if (position != string::npos)
    cout << "La position est : " << position;
  return 0;
}
```

Résultat --> La position est : 2

```cpp
#include <string>
using namespace std;
//-----------------------------------------
int main()
{
  string ch = "anticonstitutionnellement";
  string mot = "on";
  char *ad = "ti";
  int i;

  i = ch.find("elle");  // i == 17
  i = ch.find("elles"); // i<0 ou i>ch.size()
  i = ch.find(mot);     // i == 5
  i = ch.find(ad);      // i == 2
  i = ch.find('n');     // i == 1
  i = ch.find('n', 5);  // i == 6
  i = ch.find('p');     // i<0 ou i>ch.size()
  return 0;
}
//-----------------------------------------
```

Recherche avec la méthode « *find()* » :

La méthode *find()* permet de rechercher, dans une chaîne donnée, la première occurrence :

⇒ d'une autre chaîne (on parle alors de sous-chaîne) fournie en argument,

⇒ d'une autre chaîne usuelle, soit d'un caractère donné.

Par défaut, la recherche commence au début de la chaîne, mais on peut la faire débuter à un caractère de rang donné.

Recherche avec la méthode « *rfind()* » :

De manière semblable, la méthode *rfind()* permet de rechercher la dernière occurrence d'une autre chaîne ou d'un caractère.

```cpp
#include <string>
using namespace std;
//-----------------------------------------
int main()
{
  string ch = "anticonstitutionnellement";
  string mot = "on";
  char *ad = "ti";
  int i;

  i = ch.rfind("elle");  // i == 17
  i = ch.rfind("elles"); // i<0 ou i>ch.size()
  i = ch.rfind(mot);     // i == 14
  i = ch.rfind(ad);      // i == 12
  i = ch.rfind('n');     // i == 23
  i = ch.rfind('n', 18); // i == 16
  return 0;
}
//-----------------------------------------
```

```cpp
#include <string>
using namespace std;
//-----------------------------------------
int main()
{
  string ch = "anticonstitutionnellement";
  char *ad = "oie";
  int i;

  i = ch.find_first_of("aeiou");         // i == 0
  i = ch.find_first_not_of("aeiou");     // i == 1
  i = ch.find_first_of("aeiou", 6);      // i == 9
  i = ch.find_first_not_of("aeiou", 6); // i == 6
  i = ch.find_first_of(ad);              // i == 3
  i = ch.find_last_of("aeiou");          // i == 22
  i = ch.find_last_not_of("aeiou");      // i == 24
  i = ch.find_last_of("aeiou", 6);       // i == 5
  i = ch.find_last_not_of("aeiou", 6);  // i == 6
  i = ch.find_last_of(ad);               // i == 22
  return 0;
}
//-----------------------------------------
```

Autres méthodes de recherche :

La méthode *find_first_of()* recherche la première occurrence de l'un des caractères d'une autre chaîne (string ou usuelle), tandis que *find_last_of()* en recherche la dernière occurrence. La méthode *find_first_not_of()* recherche la première occurrence d'un caractère n'appartenant pas à une autre chaîne, tandis que *find_last_not_of()* en recherche la dernière.

--- Insertions, suppressions et remplacements ----------

Ces possibilités sont relativement classiques, mais elles se recoupent partiellement, dans la mesure où l'on peut :

⇒ d'une part utiliser, non seulement les objets de type « *string* », mais aussi des chaînes usuelles « *char** » ou des caractères,

⇒ d'autre part définir une sous-chaîne, soit par indice, soit par itérateur, cette dernière n'étant cependant pas offerte systématiquement.

Insertions :

La fonction *insert()* permet d'insérer, à une position donnée, définie par un indice :

⇒ une autre chaîne (objet de type *string*) ou une partie de chaîne définie par un indice de début et une éventuelle longueur,

⇒ une chaîne usuelle (type *char **) ou une partie de chaîne usuelle définie par une longueur,

⇒ un certain nombre de fois caractère donné ;

La fonction *insert()* permet d'insérer, à une position donnée, définie par un itérateur :

⇒ une séquence d'élément de type char, définie par un itérateur de début et un itérateur de fin,

⇒ une ou plusieurs fois un caractère donné.

Suppressions :

La fonction *erase()* permet de supprimer :

⇒ une partie d'une chaîne, définie soit par un itérateur de début et un itérateur de fin, soit par un indice de début et une longueur ;

⇒ un caractère donné défini par un itérateur de début.

Remplacements :

La fonction *replace()* permet de remplacer une partie d'une chaîne définie, soit par un indice et une longueur, soit par un intervalle d'itérateur, par :

⇒ une autre chaîne (objet de type *string*) ;

⇒ une partie d'une autre chaîne définie par un indice de début et, éventuellement, une longueur,

⇒ une chaîne usuelle (type *char **) ou une partie de longueur donnée,

⇒ un certain nombre de fois un caractère donné.

En outre, on peut remplacer une partie d'une chaîne définie par un intervalle par une autre séquence d'éléments de type char, définie par un itérateur de début et un itérateur de fin.

--- Méthodes et fonction supplémentaires ----------

```
taille : 0
capacité : 7
la chaine est vide !
taille : 9
capacité : 15
b-i-e-n-v-e-n-u-e-enve
message : bienvenue
chaine : salut
-----------------
Première ligne
deuxième ligne@
-----------------
Première ligne
deuxième ligne
```

```cpp
#include <string>
#include <list>
using namespace std;
//-------------------------------------------------------------------
int main()
{
  string ch = "0123456";
  string voy = "aeiou";
  char t[] = "778899";

  ch.insert(ch.begin()+1, 'a');     // "0a123456"
  ch.insert(4, 1, 'b');             // "0a12b3456"
  ch.insert(ch.end(), 3, 'x');      // "0a12b3456xxx"
  ch.insert(6, 3, 'x');             // "0a12b3xxx456xxx"
  ch.insert(0, voy);                // "aeiou0a12b3xxx456xxx"
  ch.insert(0, voy, 1, 3);          // "eioaeiou0a12b3xxx456xxx"
  ch.insert(ch.begin()+2, t, t+6);  // "ei778899oaeiou0a12b3xxx456xxx"
  return 0;
}
//-------------------------------------------------------------------
```

```cpp
#include <string>
#include <list>
using namespace std;
//-------------------------------------------------------------------
int main()
{
  string ch = "0123456789";
  string ch_bis = ch;

  ch.erase(3, 2);                          // "01256789"
  ch = ch_bis;
  ch.erase(ch.begin()+3, ch.begin()+6);    // "0126789"
  ch.erase(3);                             // "012"
  ch = ch_bis;
  ch.erase(ch.begin()+4);                  // "012356789"
  return 0;
}
```

```cpp
#include <string>
#include <list>
using namespace std;
//-------------------------------------------------------------------
int main()
{
  string ch = "0123456";
  string voy = "aeiou";
  char t[ ] = "+*-/=<>";
  char *message = "hello";

  ch.replace(2, 3, voy);                  // 01aeiou56
  ch.replace(0, 1, voy, 1, 2);            // ei1aeiou56
  ch.replace(1, 2, 8 ,'*');               // e********aeiou56
  ch.replace(1, 2, 8 ,'#');               // e##########aeiou56
  ch.replace(2, 4 ,"xxxxxx");             // e#xxxxxx********aeiou56
  ch.replace(ch.size()-7, ch.size(), message, 3); // e#xxxxxx********hel
  ch.replace(ch.begin(), ch.begin()+ch.size()-1, t, t+5);    // +*-/=1
  return 0;
}
//-
```

```cpp
#include <string>
#include <iostream>
using namespace std;
//-------------------------------------------------------------------
int main( )
{
  string chaine;
  cout << "taille : " << chaine.size() << endl;
  cout << "capacité : " << chaine.capacity() << endl;
  if (chaine.empty()) cout << "la chaine est vide !" << endl;
  chaine = "bienvenue";
  chaine.reserve(15);
  cout << "taille : " << chaine.size() << endl;
  cout << "capacité : " << chaine.capacity() << endl;
  for (int i=0; i<chaine.size(); i++)
    cout << chaine[i] << '-';
  cout << chaine.substr(2, 4) << endl;
  string message = "salut";
  message.swap(chaine);
  cout << "message : " << message << endl;
  cout << "chaine : " << chaine << endl;
  cout << "-----------------" << endl;
  getline(cin, chaine, '@');
  cout << "-----------------" << endl << chaine;
  const char *ancien = chaine.c_str();
  return 0;
}
//-------------------------------------------------------------------
```

La classe « *bitset* »

Dans le dernier TP, nous avons mis en œuvre une classe générique « *Binaire* » qui offre un certain nombre de manipulations sur les nombres binaires. Dans la STL, une telle classe existe, elle est représentée par la classe générique « *bitset* » et permet également de manipuler efficacement des suites de bits dont la taille est spécifiée en paramètre du modèle. L'affectation n'est donc possible qu'entre suites de même taille.

```
1  template <unsigned long bits>
2  class bitset
3  {
4  public:
5    bitset();                          // constructeur par défaut
6    bitset(unsigned long);
7    explicit bitset(const string&);    ATTENTION ! Construction explicite
8    bitset(const bitset<bits>&);       // constructeur de copie
9    ...
10 };
```

--- Construction ----------

Il existe quatre constructeurs :

⇒ sans argument : on obtient une suite de bits nuls,

⇒ à partir d'un *unsigned long* : on obtient la suite correspondant au motif binaire contenu dans l'argument,

⇒ à partir d'une chaîne de caractères « *string* » (attention toutefois, il s'agit d'une construction de type « *explicit* »).

⇒ la construction par copie est implémentée et donc possible.

```
1  #include <bitset>
2  using namespace std;
3  int main()
4  {
5    bitset<8> bits0;              // "0000 0000"
6    bitset<16> bits1(0xFFFF);     // "1111 1111 1111 1111"
7    string binaire = "10101010";
8    bitset<8> bits2(binaire);     // "1010 1010"
9    ...
10   bitset<8> bits3 = binaire;    Interdit - Construction implicite
11   ...
12   bitset<8> bits3("10101010");  Interdit - Construction implicite d'un char * vers un string
13   ...
14   bits0 = "10101010";           Interdit - Constructions implicites d'un char * vers un string vers un bitset<8>
15   ...
16   bits0 = bitset<8>(string("10101010"));
17   return 0;                     Bonne écriture
18 }
```

explicit

Par défaut, les constructeurs permettent de réaliser des conversions implicites lorsque cela est nécessaire. Il peut arriver que dans certaines situations cette conversion automatique soit gênante. Nous pouvons alors bloquer le comportement par défaut du constructeur en demandant que l'argument passer au constructeur au moment de la création de l'objet soit rigoureusement du type attendu. Pour offrir cette alternative, il est nécessaire de préfixer le constructeur du mot réservé « *explicit* ». Lorsque nous avons un constructeur avec un seul paramètre, il est possible d'utiliser l'opérateur « = ». Si vous déclarez un constructeur de type « *explicit* », cette opportunité n'est plus possible (« = » ➔ création implicite de l'objet). Il est alors nécessaire d'utiliser systématiquement les parenthèses.

--- Redéfinition des opérateurs ----------

Nous disposons des opérateurs classiques de manipulation globale des bits « ***&, |, ~, ^, <<, >>, &=, |=, ~=, ^=, <<=, >>=, ==, !=*** » qui fonctionnent de la même façon que les mêmes opérateurs appliqués à des entiers.

Nous pouvons accéder à un bit de la suite à l'aide de l'opérateur « ***[ ]*** » ; il déclenche une exception « *out_of_range* » si son opérande n'est pas dans les limites permises (les exceptions seront traitées ultérieurement).

>>, << : il est également possible d'afficher ou de saisir des « *bitset* » en utilisant les opérateurs des classes « *iostream* ».

--- Les autres méthodes ----------

La classe « *bitset* » dispose de méthodes supplémentaires qui, associées à l'opérateur « ***[ ]*** », permettent de satisfaire les programmeurs lorsqu'il s'agit de manipuler efficacement une information binaire.

⇒ *any()* : existe-t-il au moins un bit dans le nombre binaire qui est à 1 ?
⇒ *none()* : inverse de la précédente, tous les bits sont-ils à 0 ?
⇒ *count()* : détermine le nombre de bits mis à 1.
⇒ *size()* : indique la capacité (nombre de bits) du nombre binaire.
⇒ *set()* : tous les bits du nombre sont mis à 1.
⇒ *set(position)* : le bit désigné par position est mis à 1.
⇒ *reset()* **:** tous les bits sont mis à 0.
⇒ *reset(position)* **:** Le bit désigné par position est mis à 0.
⇒ *flip()* : inverse tous les bits du nombre.
⇒ *flip(position)* : inverse le bit désigné par position.
⇒ *test(position)* : teste si le bit désigné par position est à 1. La méthode renvoie *true*, et *false* suivant le résultat du test.
⇒ *to_string()* : retourne la valeur binaire sous forme de « *string* ».
⇒ *to_ulong()* : retourne la valeur binaire sous forme de valeur entière de type « *unsigned long* ».

La classe « *vector* » en tant que tableau - (inclure <vector>)

Comme les deux précédentes, nous avons besoin d'une classe qui palie au manque de compétence du langage C++. La classe « ***vector*** » sera presque systématiquement utilisée pour implémenter les tableaux. Notre première approche sera justement dans ce sens. Toutefois, la classe « *vector* » représente bien plus que cela. Elle fait également partie de l'ensemble

des conteneurs, et notamment des conteneurs de type séquentiel. Notre deuxième approche sera donc liée à cette notion, et nous en profiterons pour généraliser le concept de conteneur. Pour finir, nous utiliserons les algorithmes génériques afin de résoudre un certain nombre de critères qui ne sont pas spécialement intégrés dans cette classe.

--- Le tableau « *vector* » ----------

La classe « *vector* » remplace aisément les tableaux classiques en offrant des manipulations simples et intuitives. Ainsi, il est possible de construire des tableaux de type quelconque, en indiquant le nombre de cases requis. Il est également possible d'initialiser un tableau avec une valeur particulière pour toutes les cases du tableau ou même de spécifier une valeur d'initialisation différente pour chacune des cases du tableau.

Avec ce tableau, un certain nombre d'opérations peuvent être réalisées simplement, comme :

⇒ = : l'affectation est possible entre deux tableaux de même type (Attention, il faut aussi qu'ils comportent le même nombre de cases).

⇒ [ ] : l'opérateur d'indexation à bien évidemment été redéfini pour supporter le comportement classique d'un tableau, puisque cet opérateur à été spécialement créé pour les tableaux.

⇒ ==, !=, <, <=, >, >= : Il est de plus possible de comparer le contenu de deux tableaux entre eux en utilisant les opérateurs classiques de comparaison.

Vu la simplicité d'utilisation, il est impératif d'utiliser cette classe pour implémenter les tableaux.

```
#include <vector>
using namespace std;
int main()
{
  vector<double> tableau1(10);  // tableau de 10 réels
  vector<int> tableau2(5, -6);  // tableau de 5 entiers initialisés à -6
  int tab[] = {3, 4, -2, 18};
  vector<int> tableau3(tab, tab+4); // tableau de 4 entiers tous
                                    // initialisés à une valeur précise
  vector<int> tableau4 = tableau3;  // construction par copie
  tableau4[2] = -15;                // indexation possible
  tableau3 = tableau4;              // affectation entre tableaux

  return 0;
}
```

--- Le conteneur « *vector* » ----------

Les ingénieurs ont continués à développer cette classe afin d'élargir ces compétences pour en faire quelque chose de plus générique. Des méthodes ont donc été rajoutées afin d'offrir des fonctionnalités supplémentaires et pour que le tableau devienne un conteneur performant.

Dans le chapitre qui suit, se trouve la liste de ces méthodes et vous allez toute de suite remarquer que le nom de la plupart d'entre elles vous paraîtra familier puisque nous les avons déjà utilisées dans la classe « *string* ». Nous avons donc une certaine homogénéité dans le nom des méthodes utilisées, ce qui facilite grandement l'apprentissage. Nous allons d'ailleurs nous intéresser dans un premier temps aux méthodes communes à tous les conteneurs de type séquentiels, à la suite de quoi, nous ferons une étude plus spécifique sur chacun d'entre eux en montrant bien leurs particularités.

Propriétés communes aux conteneurs séquentiels « *vector, list, deque* »

J'ai déjà donné une brève description de ces trois types de conteneurs, je ne vais donc pas m'y étendre. Ce qui m'intéresse dans ce chapitre, c'est de voir le comportement commun qui donne une certaine homogénéité dans la STL.

--- Directive de compilation « *typedef* » ----------

Le compilateur offre des mécanismes fort intéressants pour simplifier le travail du programmeur. Le préprocesseur, grâce à la directive « *typedef* » permet de donner un synonyme à un type de donnée standard ou défini par l'utilisateur. C'est comme si nous définissions un nouveau type alors qu'il s'agit en fait d'un simple changement de texte durant la phase de précompilation.

Une définition par « *typedef* » commence avec le mot clé « *typedef* », suivi du type de donnée et ensuite de l'identificateur. L'identificateur, ou le nom « *typedef* », n'introduit pas un nouveau type mais plutôt un synonyme pour le type de donnée existant. Un nom « *typedef* » peut apparaître n'importe où dans le programme, là ou un nom de type peut apparaître.

```
#include <vector>
using namespace std;
//-------------------- Types primitifs ou définis par l'utilisateur
int main( )
{
  typedef  unsigned char  Octet;
  typedef  vector<int>    VecteurEntier;

  Octet mot = 0xFA;          // Nouveaux noms - Alias ou Synonymes

  VecteurEntier tableau(10);

  return 0;                  // Tableau de 10 entiers
}
```

--- Itérateur et parcours d'un conteneur ----------

Un itérateur fournit un processus général pour accéder successivement à chaque élément à l'intérieur de n'importe quel type de conteneur. Un itérateur correspond à un pointeur qui, comme tous les pointeurs, permet d'utiliser l'incrémentation ou la décrémentation. Ainsi, il est possible de consulter dans le sens direct ou en sens inverse une suite d'éléments faisant partie de la séquence.

--- Parcours d'un conteneur dans le sens direct ----------

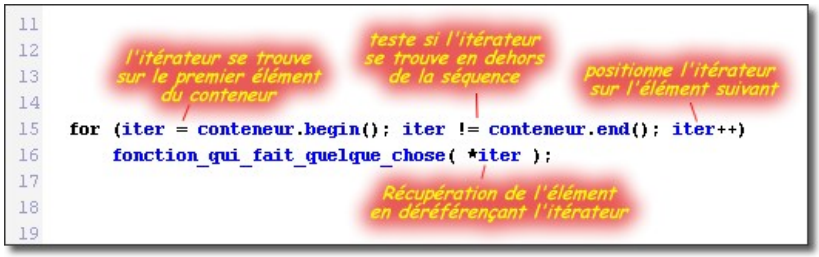
Nous avons donc une homogénéisation du parcours d'une séquence avec un itérateur, savoir :

⇒ Un itérateur peut être incrémenté par l'opérateur « **++** », de manière à pointer sur l'élément suivant du même conteneur.

⇒ Un itérateur peut être déréférencé par l'opérateur « ***** » ; nous pouvons donc récupérer la valeur de l'élément de la séquence.

⇒ Chaque type de conteneur fournit deux méthodes :

- *begin()* : retourne un itérateur qui adresse le premier élément du conteneur,
- *end()* : retourne un itérateur qui adresse un élément après le dernier élément du conteneur.



--- Parcours d'un conteneur dans le sens inverse ---

Il est également possible de parcourir un conteneur en sens inverse. Attention, il faut pouvoir démarrer sur le dernier élément, ce que ne fait pas la méthode *end()*, puisqu'elle se trouve après le dernier élément. D'autres méthodes ont donc été implémentées pour résoudre cette situation :

⇒ *rbegin()* : retourne un itérateur qui adresse le dernier élément du conteneur,

⇒ *rend()* : retourne un itérateur qui adresse un élément juste avant le premier élément du conteneur.


```
//-----------------------------------
template <class Type>
class vector
{
  Type *contenu;
  unsigned taille, nombreElement;
public:
  typedef Type *iterator;

};
//-----------------------------------
int main( )
{
  vector<int>::iterator iterateur;

  return 0;
}
```

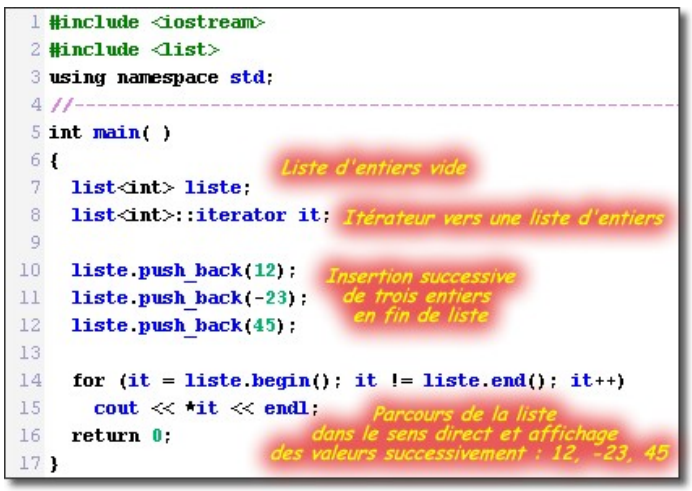
--- Déclaration d'un itérateur ---

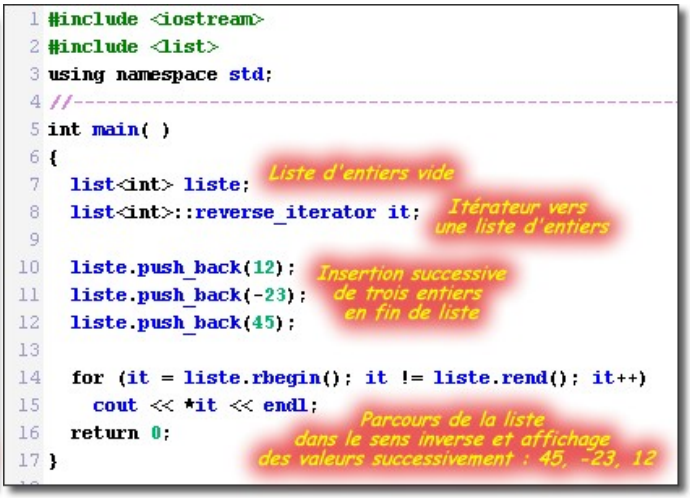
L'intérêt de ce mécanisme, c'est qu'il fonctionne quelque soit le type de conteneur utiliser alors que la structure interne de chacun d'entre eux peut être totalement différente. Pour les vecteurs, par exemple, les éléments sont stockés sur des cases mémoires contiguës, alors que pour la liste ce n'est pas du tout le cas. La seule solution pour résoudre ces difficultés est effectivement de prendre le mécanisme des pointeurs. Il faut bien comprendre également que ces différents conteneurs stockent des données de type quelconque (*int*, *double*, *string*, etc.).

Ceci dit pour réaliser ces différents parcours, il est bien évidemment nécessaire de déclarer la variable qui représente l'itérateur. Souvenez-vous que, pour que l'incrémentation d'un pointeur se fasse dans les bonnes conditions, il est impératif de connaître le type de l'élément faisant parti du conteneur.

La solution retenue a donc été de proposer un attribut public, présent sur tous les conteneurs, qui s'appelle « *iterator* ». Cet attribut est en fait un pointeur sur le type passé en paramètre du modèle de la classe conteneur. Cette démarche est possible grâce à la directive de compilation « *typedef* ». Cette astuce est géniale malgré la petite lourdeur de la déclaration. Vous avez ci-dessous un exemple d'un parcours dans le sens direct d'une liste d'entiers.

Il existe également un deuxième itérateur spécialisé pour parcourir le conteneur en sens inverse, qui s'appelle « *reverse_iterator* ». Par ailleurs, bien que rarement utilisé, la décrémentation peut être utilisée par les deux types d'itérateurs.





--- Constructions ---

Ces trois classes disposent de plusieurs constructeurs qui permettent de résoudre la plupart des situations envisagées :

⇒ Construction d'un conteneur vide : L'appel d'un constructeur sans argument construit un conteneur vide, c'est-à-dire ne comportant aucun élément.

⇒ Construction avec un nombre donné d'éléments : De façon comparable à ce qui se passe avec la déclaration d'un tableau classique, l'appel d'un constructeur avec un seul argument entier « *n* » construit un conteneur comprenant « *n* » éléments. L'initialisation de ces éléments n'est correctement gérée que dans le cas ou les éléments sont des objets. En effet, ces derniers disposent d'un constructeur par défaut. Dans le cas des types primitifs, les valeurs sont aléatoires.

⇒ Construction avec un nombre d'éléments initialisés avec une valeur précise : Le premier argument fourni le nombre d'éléments alors que le second fixe la valeur d'initialisation.

⇒ Construction à partir d'une séquence : Nous pouvons construire un conteneur à partir d'une séquence d'éléments de même type. Dans ce cas, nous fournissons simplement au constructeur deux arguments représentant les bornes de l'intervalle correspondant.

⇒ Construction par recopie : Chaque type de conteneur dispose de son propre constructeur de copie. Attention, il est nécessaire d'utiliser des conteneurs rigoureusement identiques (même conteneur et même type d'éléments).

```cpp
#include <vector>
#include <list>
#include <string>
using namespace std;
//-----------------------------------------------------------------------
int main()
{
  list<float> lf;                // liste de float vide
  list<float> lfi(5);            // liste de 5 éléments de type float
  vector<string> vs(5);          // tableau de 5 chaînes vides
  list<int> lii(5, 999);         // liste de 5 entiers ayant tous la valeur 999
  string message = "Bonjour";
  list<string> lsi(10, message); // liste de 10 chaînes "Bonjour"
  list<string> ls;
  vector<string> vss(lsi.begin(), lsi.end()); // construit un tableau de
                                              // chaînes à partir de la séquence donnée par la liste
  list<string> lsinv(lsi.rbegin(), lsi.rend()); // construit une liste de
                                                // chaînes à partir de la séquence donnée par la liste
                                                // en sens inverse.
  return 0;
}
```

-- Affectation et comparaisons ----------

Ce que nous avons découvert sur la classe « *vector* » en tant que tableau s'applique également sur tous les conteneurs séquentiels.

Ainsi, il est possible d'affecter un conteneur d'un type donné à un autre conteneur de même type, c'est-à-dire ayant le même nom de patron et le même type d'éléments. Bien entendu, il n'est nullement nécessaire que le nombre d'éléments de chacun des conteneurs soit identique.

De la même façon, les opérateurs relationnels « ==, !=, <, <=, >, >= » ont été redéfinis pour supporter tous les types de comparaison, quelque soit le conteneur utilisé.

```cpp
list<int> liste;
vector<int> vecteur, vi;
...
liste.assign(vecteur.begin(), vecteur.end());
vecteur.assign(10, -23); // vecteur <-- 10 fois -23
char t[] = "Salut";
list<char> lc;      // lc <-- ( )
lc.assign(t, t+5); // lc <-- (S, a, l, u, t)
lc.assign(3, 'z'); // lc <-- (z, z, z)
```

Copie du contenu de vecteur vers la liste

--- Les autres méthodes ----------

⇒ *assign(début, fin)* : alors que l'affectation n'est possible qu'entre conteneurs de même type, la méthode « *assign* » permet d'affecter, à un conteneur existant, les éléments d'une autre séquence définie par un intervalle (*début*, *fin*), à condition que les éléments des deux séquences soient de même type.

⇒ *assign(nombreDeFois, valeur)* : il existe également une version permettant d'affecter à un conteneur, un nombre donné d'éléments ayant une valeur imposée.

⇒ *clear()* : vide le conteneur de son contenu.

⇒ *empty()* : teste si le conteneur est vide et renvoie *true* si c'est le cas, et *false* ans le cas contraire.

⇒ *swap()* : permet d'échanger le contenu de deux conteneurs de même type.

⇒ *insert(position, valeur)* : insère une valeur avant l'élément pointé par la position.

⇒ *insert(position, nombreDeFois, valeur)* : insère un certain nombre de fois une valeur avant l'élément pointé par la position.

⇒ *insert(début, fin, position)* : insère les valeurs de l'intervalle (*début*, *fin*) avant l'élément pointé par la position.

⇒ *push_back(valeur)* : cette méthode est spécialisée pour insérer une valeur en fin de conteneur à la manière d'une pile.

⇒ *erase(position)* : supprime l'élément désigné par la position

⇒ *erase(début, fin)* : supprime les valeurs de l'intervalle « *début (compris)*, *fin (non compris)* ».

⇒ *pop_back()* : cette méthode est spécialisée pour supprimer la dernière valeur du conteneur à la manière d'une pile.

⇒ *size()* : détermine le nombre d'éléments que contient le conteneur.

```cpp
vecteur.clear();          // vide le vecteur
if (vecteur.empty()) ...; // teste si vecteur est vide
vi.swap(vecteur);    // inverse le contenu de vi
                     // avec le contenu de vecteur
```

```cpp
list<double> ld;
list<double>::iterator il;
vector<double> vd;
...
ld.insert(ld.begin(),  6.7); // insère 6.7 au début de ld
ld.insert(ld.end(), 3.2); // insère 3.2 en fin de ld
ld.push_back(3.2);   // fait la même chose que la ligne précédente
ld.insert(il, 2.5); // insère 2.5 avant l'élément pointé par il
ld.insert(il, 10, -1); // insère 10 fois -1 avant l'élément pointé par il
ld.insert(ld.begin(), vd.begin(), vd.end());
                // insère tous les éléments de vd en début de la liste ld
```

```cpp
list<double> ld;
list<double>::iterator il1, il2;

ld.erase(ld.begin()); // supprime le premier élément
double valeur = ld.pop_back(); // récupère et supprime la dernière valeur
ld.erase(il1, il2); // supprime les éléments entre il1 (compris)
                    // et il2 (non compris)
ld.erase(ld.begin(), ld.end()); // efface toute la liste
ld.clear(); // même fonctionnalité
```

Les algorithmes génériques

Les conteneurs ont en commun beaucoup de méthodes. Chaque conteneur dispose également de méthodes supplémentaires qui font leur spécificité. Cependant, une fois que nous avons choisi un conteneur, il peut être intéressant de rajouter d'autres fonctionnalités non intégrées par le conteneur. Dans ce cas là, nous avons besoin des fonctions génériques. Rappelons que les fonctions génériques sont opérationnelles quelque soit le conteneur utilisé.

Nous remarquons, par exemple, que la recherche d'un élément au sein d'un conteneur ne fait pas parti des méthodes communes, alors que c'est un comportement qui est souvent souhaitable. Nous allons d'ailleurs détailler un certain nombre de fonctions qui sont généralement assez utiles. Il faudra quand même vérifier que votre conteneur ne dispose pas déjà de telles fonctions internes (méthodes).
Voici quelques fonctions génériques intéressantes :

⇒ ***copy****(conteneur1début, conteneur1fin, conteneur2début)* : copie d'une séquence dans une autre. Il suffit de préciser l'intervalle désiré en donnant les itérateurs du premier conteneur. Cet intervalle est ensuite copié à partir de l'itérateur donné par le second conteneur.

⇒ ***count****(iterateurdébut, iterateurfin, valeurRecherchée)* : comptabilise le nombre de fois qu'un élément est présent dans un conteneur,

⇒ *iterateur* ***find****(iterateurdébut, iterateurfin, valeurRecherchée)* : recherche d'une valeur particulière par rapport à l'intervalle d'une séquence. La fonction retourne l'itérateur correspondant à l'endroit où se situe la valeur recherchée. Si la recherche n'a pas aboutie, la fonction renvoie un itérateur sur la fin de la séquence - *conteneur.end()*

⇒ *iterateur* ***max_element****(iterateurdébut, iterateurfin)* : recherche la valeur maximale d'une séquence. La fonction retourne l'itérateur correspondant à l'endroit où se situe la valeur recherchée.

⇒ *iterateur* ***min_element****(iterateurdébut, iterateurfin)* : recherche la valeur minimale d'une séquence. La fonction retourne l'itérateur correspondant à l'endroit où se situe la valeur recherchée.

⇒ ***replace****(iterateurdébut, iterateurfin, ancienneValeur, nouvelleValeur)* : remplace toutes les instances d'une valeur particulière par une nouvelle valeur,

⇒ ***remove****(iterateurdébut, iterateurfin, valeurASupprimer)* : supprime toutes les instances d'une valeur particulière par rapport à l'intervalle proposé.

⇒ ***sort****(iterateurdébut, iterateurfin)* : tri de la séquence. Reclasse les éléments de l'intervalle proposé dans l'ordre croissant.

⇒ ***reverse****(iterateurdébut, iterateurfin)* : inverse l'ordre des éléments dans un conteneur.

```cpp
#include <algorithm>  // indispensable pour les fonctions génériques
#include <iostream>
#include <vector>
using namespace std;
//------------------------------------------------------------------------
int main()
{
  int t[7] = {2, 3, 1, 8, 2, 11, 9};
  vector<int> vecteur(t, t+7);
  vector<int>::iterator it;
  copy(vecteur.begin()+2, vecteur.end(), vecteur.begin());
  cout << "Vecteur = ";
  for (it = vecteur.begin(); it!=vecteur.end(); it++)
    cout << *it << ' ';
  cout << "\nNombre de 9 : " << count(vecteur.begin(), vecteur.end(), 9);
  vecteur.erase(find(vecteur.begin(), vecteur.end(), 8));
  it = min_element(vecteur.begin(), vecteur.end());
  cout << "\nPlus petit : " << *it << endl;
  cout << "Plus grand : " << *max_element(vecteur.begin(), vecteur.end());
  replace(vecteur.begin(), vecteur.end(), 9, 5);
  remove(vecteur.begin(), vecteur.end(), 11);
  sort(vecteur.begin(), vecteur.end());
  reverse(vecteur.begin(), vecteur.end());
  cout << "\nNombre d'éléments : " << vecteur.size() << endl;
  cout << "Vecteur = ";
  for (it = vecteur.begin(); it!=vecteur.end(); it++)
    cout << *it << ' ';
  return 0;
}
```

```
Vecteur = 1 8 2 11 9 11 9
Nombre de 9 : 2
Plus petit : 1
Plus grand : 11
Nombre d'éléments : 6
Vecteur = 11 5 5 5 2 1
```

Spécificités du conteneur « *vector* » - (inclure <vector>)

Nous connaissons déjà la classe « *vector* » en tant que tableau et par ailleurs nous connaissons un certain nombre de méthodes qui sont communes à tous les conteneurs. Nous allons découvrir dans ce chapitre d'autres méthodes qui sont propre à la classe « *vector* ». Nous en profiterons pour déterminer les avantages et les inconvénients de ce type de conteneur.

Ce conteneur représente bien un tableau, c'est-à-dire que les éléments qui constituent la séquence, sont placés dans des cases mémoires contiguës. Ce conteneur « *vector* » est relativement performant, puisqu'il s'agit en fait d'un tableau dynamique, c'est-à-dire, que suivant le besoin, le nombre de cases qui composent le tableau peut augmenter en cours d'utilisation. Si le tableau est plein, lorsque nous essayons d'introduire une nouvelle valeur, la gestion interne du vecteur prévoit de réserver un nouvel emplacement mémoire dont la capacité est le double de la précédente, ensuite une copie des anciennes valeurs est effectuée vers ce nouvel emplacement. La copie d'anciennes valeurs prend du temps, c'est pour cette

raison que la nouvelle capacité est doublée. Dans cette situation, nous avons un bon compromis entre la capacité du tableau (utiliser le moins de place possible) et le temps de réponse en général (répondre le plus rapidement possible à une requête).

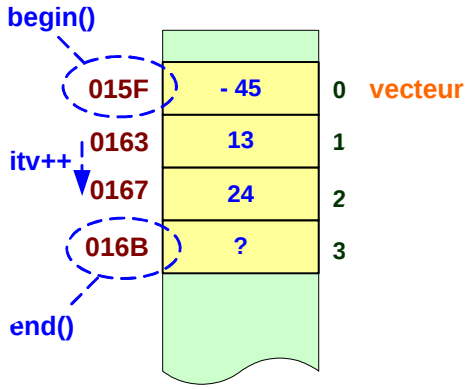
Du coup, il faut noter que la capacité du tableau peut être plus grande que le nombre d'éléments déjà introduits dans le conteneur. Cette classe propose un certain nombre de méthodes qui permet de contrôler cette situation. Ci-dessous, se trouve l'ensemble des méthodes spécifiques à la classe « *vector* » :

⇒ *back()* : récupère la valeur du dernier élément sans toutefois l'enlever du conteneur comme c'est le cas avec la méthode *pop_back()*.
⇒ *front()* : récupère la valeur du premier élément sans l'enlever du conteneur.
⇒ *size()* : cette méthode n'est pas spécifique à « *vector* », toutefois, je rappelle que cette méthode retourne le nombre d'éléments présents dans le conteneur.
⇒ *capacity()* : retourne la capacité du tableau (nombre de cases allouées) au moment de la consultation. *« capacity() >= size() »*.
⇒ *reserve(taille)* : permet d'imposer une capacité.

--- Intérêt d'utiliser le conteneur « *vector* » ----------

Le conteneur « *vector* » présente deux caractéristiques essentielles :

1. Comme les cases mémoires relatives aux éléments sont contiguës, nous savons que l'élément qui suit un autre se trouve sur la case d'après, ainsi, il n'est pas nécessaire de rajouter de pointeurs supplémentaires pour parcourir un conteneur, comme c'est le cas avec une liste. Ainsi, la taille mémoire que prend ce type de conteneur est relativement réduite.
2. Il s'agit en fait d'un tableau, et grâce à l'opérateur d'indexation « ***[ ]*** », nous pouvons accéder directement à un élément du conteneur sans être obligé de parcourir toute la séquence. L'accès aléatoire est donc très rapide. Nous pouvons même nous passer de l'écriture explicite d'un « *iterator* ». Généralement, nous utilisons plutôt la syntaxe des itérateurs pour conserver une certaine homogénéité, plus tard, il sera alors plus facile de changer de type de conteneur si le besoin s'en fait sentir. (Comme le vecteur est un tableau, l'itérateur correspond, dans ce cas, là à l'adresse d'une des cases du tableau).




```
#include <vector>
#include <iostream>
using namespace std;
//-----------------------------------------------------------------------
int main( )
{
  vector<int> vecteur;       // size() = 0, capacity() = 0
  vecteur.push_back(-45);    // size() = 1, capacity() = 1
  vecteur.push_back(13);     // size() = 2, capacity() = 2
  vecteur.push_back(24);     // size() = 3, capacity() = 4

  vector<int>::iterator itv;
  for (itv = vecteur.begin(); itv != vecteur.end(); itv++)
    cout << *itv << ' ';         // affiche successivement -45 13 24
  cout << endl;
  for (int i=0; i<vecteur.size(); i++)
    cout << vecteur[i] << ' ';  // affiche successivement -45 13 24
  return 0;
}
```

Ce type de conteneur est également très bien adapté à l'insertion de nouveaux éléments en fin de conteneur, ce que fait très bien la méthode *push_back()*.

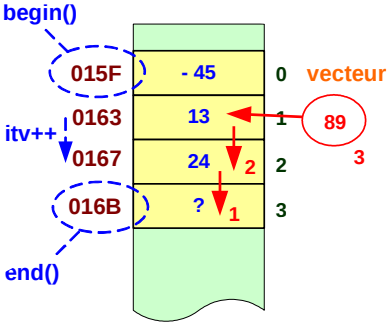
Conteneur « deque »

Le conteneur qui lui ressemble beaucoup, c'est le conteneur « *deque* » qui offre la possibilité d'insérer « *push_front()* » et de supprimer « *pop_front()* » de nouveau éléments en début de conteneur, ce que ne permet pas le conteneur « *vector* ». « *deque* », par contre, ne possède pas la gestion de la capacité du tableau.

--- Limites du conteneur « *vector* » ----------

Ce conteneur est vraiment intéressant à plus d'un titre, pourtant, il existe deux domaines où ces performances se dégradent fortement ; c'est lorsque nous devons insérer ou supprimer un élément à un endroit quelconque du conteneur (pas le dernier). Effectivement, dans ce cas là, toutes les cases qui se trouvent après l'insertion ou la suppression vont être décalées, ce qui prend, bien évidemment, beaucoup de temps.

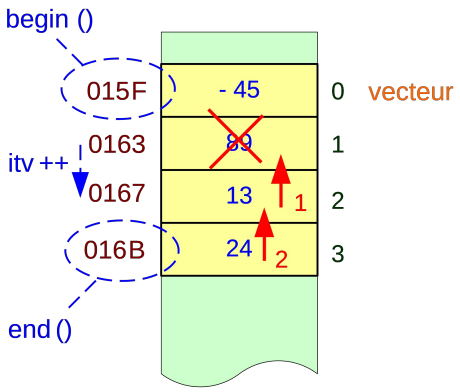
Insertion :




```
#include <vector>
#include <iostream>
using namespace std;
//-----------------------------------------------------------------------
int main( )
{
  vector<int> vecteur;       // size() = 0, capacity() = 0
  vecteur.push_back(-45);    // size() = 1, capacity() = 1
  vecteur.push_back(13);     // size() = 2, capacity() = 2
  vecteur.push_back(24);     // size() = 3, capacity() = 4

  vecteur.insert(find(vecteur.begin(), vecteur.end(), 13), 89);
  for (int i=0; i<vecteur.size(); i++)
    cout << vecteur[i] << ' ';
  return 0;                  // affiche successivement -45 89 13 24
}

```

```cpp
#include <vector>
#include <iostream>
using namespace std;
//-------------------------------------------------------
int main( )
{
  vector<int> vecteur;       // size() = 0, capacity() = 0
  vecteur.push_back(-45);    // size() = 1, capacity() = 1
  vecteur.push_back(13);     // size() = 2, capacity() = 2
  vecteur.push_back(24);     // size() = 3, capacity() = 4

  vecteur.insert(find(vecteur.begin(), vecteur.end(), 13), 89);
  vecteur.erase(find(vecteur.begin(), vecteur.end(), 89));
  for (int i=0; i<vecteur.size(); i++)
    cout << vecteur[i] << ' ';
  return 0;              // affiche successivement -45 13 24
}
```

Suppression :



Spécificités du conteneur « *list* » - (inclure <list>)

Si vous avez justement besoin de gérer beaucoup d'insertions et de suppressions, la liste est totalement adaptée à ce genre de situation. Elle dispose également de méthodes fort intéressantes qui évitent d'utiliser les fonctions génériques. Elle dispose, par exemple, de sa propre méthode de tri, ce que ne permet pas le conteneur « *vector* ».

Ce conteneur est implémenté par une liste en double chaînage, ce qui fait que chaque élément est constitué de deux pointeurs supplémentaires. Du coup, la taille mémoire est plus conséquente que pour le conteneur « *vector* ». Surtout, le conteneur « *list* » ne gère pas l'accès direct (accès aléatoire). Pour atteindre un élément, vous êtes obligé de parcourir jusqu'à l'élément désiré afin de pouvoir l'atteindre. Le temps de réponse pour accéder à un élément peut donc être très important. Dans ces conditions, à vous de choisir le conteneur qui offre le plus de souplesse possible pour votre application, et ceci sans trop de contraintes.

Ci-dessous, se trouve l'ensemble des méthodes spécifiques à la classe « *list* » :

⇒ *remove(valeur)* : supprime tous les éléments égaux à valeur. Cette méthode remplace essaiment la fonction générique équivalente, ainsi que la méthode *erase()* commune à tous les conteneurs.

⇒ *sort()* : tri la liste. Reclasse les éléments dans l'ordre croissant. Dans le cas d'un tri d'une liste, c'est cette méthode qu'il faut utiliser. Il ne faut pas prendre la fonction générique algorithmique équivalente.

⇒ *unique()* : permet d'éliminer les éléments en double, à condition de la faire porter sur une liste préalablement triée.

⇒ *merge(liste)* : fusionne *liste* avec la liste concernée. Attention, à la fin de cette opération, *liste* est vide.

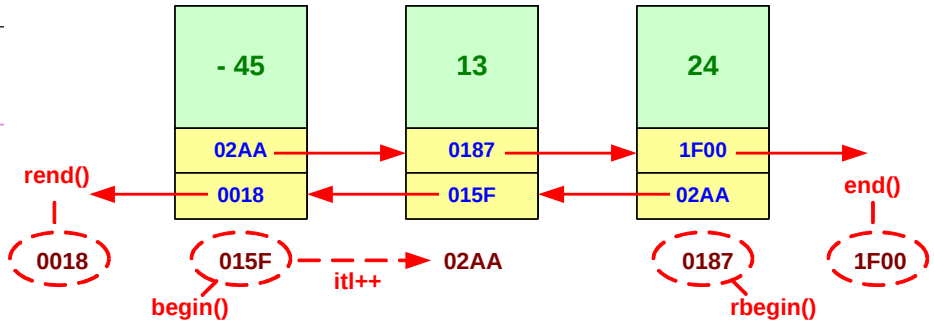
⇒ *splice(position, liste_origine)* :

⇒ *splice(position, liste_origine, position_origine)* :

⇒ *splice(position, liste_origine, début_origine, fin_origine)* : permet de déplacer des éléments d'une autre liste dans la liste concernée. Attention, comme avec *merge()*, les éléments déplacés sont supprimés de la liste d'origine et pas seulement copiés.

--- Organisation de la mémoire ----------

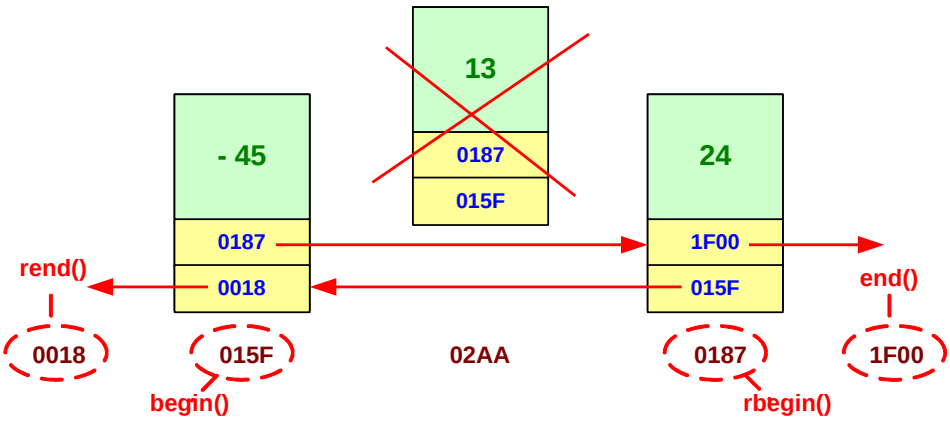
```cpp
#include <list>  // indispensable
#include <iostream>
using namespace std;
//-------------------------------
int main()
{
  list<int> liste;
  liste.push_back(-45);
  liste.push_back(13);
  liste.push_back(24);

  list<int>::iterator itl;
  for (itl = liste.begin(); itl != liste.end(); itl++)
    cout << *itl << ' ';  // affiche successivement -45 13 24
  return 0;
}
```



Dans le scénario ci-dessus, vous remarquez que chaque élément se trouve à un emplacement mémoire qui est totalement indépendant des autres éléments. La localisation de l'un par rapport à l'autre s'effectue au travers des pointeurs intégrés avec l'élément qui sont là justement pour permettre de parcourir la liste et ainsi de passer, soit à l'élément suivant, ou soit à l'élément précédent. Lorsque vous désirez supprimer une valeur de la liste, il suffit alors de redéfinir un des pointeurs de l'élément suivant ainsi qu'un des pointeurs de l'élément précédent.

Suppression :

```
#include <list>
#include <iostream>
using namespace std;
//-----------------------------
int main( )
{
  list<int> liste;
  liste.push_back(-45);
  liste.push_back(13);
  liste.push_back(24);

  liste.remove(13);
  list<int>::iterator itl;
  for (itl = liste.begin(); itl != liste.end(); itl++)
    cout << *itl << ' '; // affiche successivement -45 24
  return 0;
}
```



Les adaptateurs de conteneur « *queue* » et « *stack* »

La bibliothèque standard dispose de deux patrons particuliers « *stack* » et « *queue* » dits adaptateurs de conteneurs. Il s'agit de classes génériques construites sur un conteneur séquentiel qui en modifie l'interface, à la fois en la restreignant et en l'adaptant à leurs propres fonctionnalités. Ils disposent tous d'un seul constructeur qui est le constructeur par défaut.

L'adaptateur « *stack* » :
La classe générique « *stack* » est destinée à la gestion des piles de type LIFO (Last In, First Out) ; il peut être construit à partir de l'un des trois conteneurs séquentiels « *vector*, *deque*, *list* ».

```
stack<int, vector<int> > pile; // pile de int, utilisant un conteneur vector
stack<int, list<int> > pile;   // pile de int, utilisant un conteneur list
stack<int, deque<int> > pile;  // pile de int, utilisant un conteneur deque
```

Dans un tel conteneur, on ne peut qu'introduire « *push()* » des informations qui s'empile les unes sur les autres et que l'on recueille, à raison d'une seule à la fois, en extrayant la dernière introduite. Nous trouvons uniquement les méthodes suivantes :

⇒ *empty()* : renvoie *true* si la pile est vide.
⇒ *size()* : fournit le nombre d'éléments de la pile.
⇒ *top()* : accès à l'information située au sommet de la pile que nous pouvons consulter ou même modifier (sans la supprimer).
⇒ *push(valeur)* : place *valeur* sur le sommet de la pile.
⇒ *pop()* : suppression de l'élément situé au sommet de la pile (ne retourne aucune valeur).

```
#include <stack>
#include <vector>
#include <iostream>
using namespace std;
//-----------------------------------------------------
int main(  )
{
  stack<int, vector<int> > pile;
  cout << "Taille initiale : " << pile.size() << endl;
  for (int i=0; i<10; i++) pile.push(i*i);
  cout << "Taille après for : " << pile.size() << endl;
  cout << "Sommet de la pile : " << pile.top() << endl;
  pile.top() = 99; // modification de sommet de la pile
  cout << "On dépile : ";
  for (int i=0; i<10; i++) {
    cout << pile.top() << ' ';
    pile.pop();
  }
  return 0;
}
```

```
Taille initiale : 0
Taille après for : 10
Sommet de la pile : 81
On dépile : 99 64 49 36 25 16 9 4 1 0
```

L'adaptateur « *queue* » :
La classe générique « *queue* » est destinée à la gestion de piles de files d'attentes de type FIFO (First In, First Out). Dans ce cas là, les informations sont placées à la fin du conteneur et peuvent être ensuite récupérées au début. Un tel conteneur peut

être construit à partir de l'un des deux conteneurs séquentiels « *deque*, *list* ». Le conteneur « *vector* » n'est pas approprié puisqu'il ne dispose pas d'insertions efficaces au début.

⇒ *empty()* : renvoie *true* si la pile est vide.
⇒ *size()* : fournit le nombre d'éléments de la pile.
⇒ *front()* : accès à l'information située en tête de la file que nous pouvons consulter ou même modifier (sans la supprimer).
⇒ *back()* : accès à l'information située à la fin de la file que nous pouvons consulter ou même modifier (sans la supprimer).
⇒ *push(valeur)* : place *valeur* dans la file.
⇒ *pop()* : suppression de l'élément situé en tête de la file (ne retourne aucune valeur).

```cpp
#include <queue>
#include <deque>
#include <iostream>
using namespace std;
//-------------------------------------------------------------
int main( )
{
  queue<int, deque<int> > file;
  for (int i=0; i<10; i++) file.push(i*i);
  cout << "Tête de la file : " << file.front() << endl;
  cout << "Queue de la file : " << file.back() << endl;
  file.front() = 99;  // modification de la tête de la file
  file.back() = -99;  // modification de la queue de la file
  cout << "Vidage de la file : ";
  for (int i=0; i<10; i++) {
    cout << file.front() << ' ';
    file.pop();
  }
  return 0;
}
```

```
Tête de la file : 0
Queue de la file : 81
Vidage de la file : 99 1 4 9 16 25 36 49 64 -99
```

Conteneurs associatifs

Les conteneurs se classent en deux catégories : les conteneurs séquentiels et les conteneurs associatifs. Nous venons de le voir, les conteneurs séquentiels sont ordonnés suivant un ordre imposé explicitement par le programme lui-même ; nous accédons à un des éléments en tenant compte cet ordre, que nous utilisions un indice ou un itérateur.

Les conteneurs associatifs ont pour principale vocation de retrouver une information, non plus en fonction de sa place dans le conteneur, mais en fonction de sa valeur nommée « clé ». Nous avons déjà cité l'exemple du répertoire téléphonique, dans lequel on retrouve le numéro de téléphone à partir de la clé formée du nom de la personne concernée. Malgré tout, pour de simple questions d'efficacité, un conteneur associatif se trouve ordonné intrinsèquement en permanence, en se fondant sur une relation (par défaut « **<** ») choisie à la construction.

Les deux conteneurs associatifs les plus importants sont « *map* » et « *multimap* ». Ils correspondent pleinement au concept de conteneur associatif, en associant une clé et une valeur. Mais, alors que « *map* » impose l'unicité des clés, autrement dit l'absence de deux éléments ayant la même clé, « *multimap* » ne l'impose pas et nous pourrons trouver plusieurs éléments d'une même clé qui apparaîtront alors consécutivement. Si nous reprenons l'exemple du répertoire téléphonique, « *multimap* » autorise la présence de plusieurs numéros pour une même personne, tandis que « *map* » ne l'autorise pas.

----- Conteneur « *map* » ----------

Ce conteneur offre une grande souplesse d'emploi. Il permet effectivement d'intégrer ou de rechercher des éléments à l'aide de l'opérateur d'indexation « ***[ ]*** » sans se préoccuper spécialement de l'endroit où le conteneur stocke sa donnée. L'exemple ci-contre illustre cette simplicité de programmation.

```cpp
#include <map>
#include <iostream>
using namespace std;
//-------------------------------------------------------------
int main( )
{                                   map<clé, valeur>
  map<char, int> stockage;
  stockage['c'] = 10;
  stockage['f'] = 20;   Insertion des éléments en indiquant
  stockage['x'] = 30;   la clé entre les crochés et la valeur
  stockage['p'] = 40;   après l'opérateur d'affectation
  cout << stockage['f'] << ' ' << stockage.size();
  return 0;                                          Résultat : 20 4
}                   Recherche d'un élément
//-------------   en fournissant la clé
                    entre les crochets
```

Bien entendu, un certain nombre de méthodes sont proposés pour faciliter l'utilisation de ce type de conteneur.

⇒ *empty()* : renvoie *true* si le conteneur « *map* » est vide.
⇒ *size()* : fournit le nombre d'éléments du conteneur.
⇒ *clear()* : vide le conteneur de tout élément.
⇒ *Objetmap[clé] = valeur* : place le couple (*clé*, *valeur*) dans le conteneur *Objetmap*.
⇒ *Valeur = Objetmap[clé]* : recherche la valeur associée à *clé* dans le conteneur *Objetmap* et la retourne à *Valeur*.
⇒ *erase(clé)* : supprime un élément du conteneur référencé par *clé*.
⇒ *begin(), end(), rbebin(), rend()*: il est possible de parcourir le conteneur afin, par exemple, de recenser l'ensemble des éléments. Chacune de ces méthodes retourne un itérateur.
⇒ *iterator, reverse_iterator* : Pour permettre ce parcours, comme tous les autres conteneurs, nous nous servons donc itérateurs.
⇒ *first, second* : il existe deux attributs qui représentent respectivement la *clé* et la *valeur* d'un élément du conteneur.
⇒ *insert(élément)* : bien que nous ayons pour ce conteneur l'opérateur d'indexation qui permet d'introduire de nouveaux éléments de façon intuitive, nous pouvons également utiliser cette méthode. En paramètre, il est nécessaire de passer l'élément en entier, c'est-à-dire le couple (*clé*, *valeur*). Pour cela, « *élément* » sera fabriqué à l'aide de la fonction « *make_pair* » définie ci-dessous.

⇒ *make_pair(clé, valeur)* : cette fonction générique permet de fabriquer un élément du conteneur « *map* » constitué d'une *clé* et d'une *valeur*.

⇒ *Itérateur find(clé)* : bien que nous ayons pour ce conteneur l'opérateur d'indexation qui permet de rechercher une *valeur*, il est également possible d'utiliser cette méthode qui retourne un itérateur qui identifie l'emplacement de l'élément référencé par la *clé* passée en paramètre. Si aucun élément n'est retrouvé, la méthode renvoie l'itérateur relatif à « *end()* ».

----- Conteneur « *multimap* » -----

Le conteneur « *multimap* » est un conteneur « *map* » qui accepte en plus d'avoir plusieurs valeurs pour une même clé.

Nous retrouverons donc les mêmes méthodes, sauf pour l'opérateur d'indexation puisque, dans ce cas là, l'utilisation d'un tel opérateur ne conviendrait pas. Il faut donc impérativement utiliser la méthode « *insert(élément)* » associée à la fonction générique « *make_pair(clé, valeur)* » pour palier ce manque.

La méthode « *erase(clé)* » cette fois ci permet d'effacer toutes les valeurs associées à une même clé.

Par contre, il faut pouvoir parcourir l'ensemble des valeurs associées à une même clé. D'autres méthodes qui existées déjà dans le conteneur « *map* » prennent maintenant toute leur utilité.

```cpp
#include <map>
#include <iostream>
using namespace std;
//------------------------------------------------------------------------
int main( )
{
  map<char, int> stock;
  map<char, int>::iterator it;
  stock['a'] = 28;
  stock['x'] = -89;
  stock.insert(make_pair('w', 45));
  for (it = stock.begin(); it != stock.end(); it++)
    cout << '(' << it->first << ", " << it->second << ')' << endl;
  cout << stock['a'] << " -- " << stock.find('a')->second << endl;
  stock.erase('x');
  it = stock.find('x');
  if (it != stock.end()) cout << it->second << endl;
  if (!stock.empty()) cout << stock.size();
  return 0;
}
```

```
(a, 28)
(w, 45)
(x, -89)
28 -- 28
2
```

⇒ *count(clé)* : retourne le nombre d'éléments associés à une clé.

⇒ *Itérateur lower_bound(clé)* : retourne un itérateur sur le premier élément associé à *clé*.

⇒ *Itérateur upper_bound(clé)* : retourne un itérateur juste après le dernier élément associé à *clé*.

```cpp
#include <map>
#include <iostream>
using namespace std;
//------------------------------------------------------------------------
int main(  )
{
  multimap<char, int> stockage;
  multimap<char, int>::iterator it;

  stockage.insert(make_pair('c', 10));
  stockage.insert(make_pair('f', 20));
  stockage.insert(make_pair('f', 30));
  stockage.insert(make_pair('p', 10));

  cout << "Nombre d'éléments :" << stockage.size() << endl;
  cout << "Nombre de f : " << stockage.count('f') << endl;

  cout << "Ensemble de tous les éléments" << endl;
  for (it = stockage.begin(); it != stockage.end(); it++)
    cout << '(' << it->first << ", " << it->second << ')' << endl;

  cout << "Ensemble de tous les f" << endl;
  for (it = stockage.lower_bound('f'); it != stockage.upper_bound('f'); it++)
    cout << '(' << it->first << ", " << it->second << ')' << endl;

  return 0;
}
```

```
Nombre d'éléments :4
Nombre de f : 2
Ensemble de tous les éléments
(c, 10)
(f, 20)
(f, 30)
(p, 10)
Ensemble de tous les f
(f, 20)
(f, 30)
```