# TP POO C++ : Les conteneurs STL



## Sommaire



**Les objectifs de ce tp sont de découvrir la STL (et ses conteneurs) en C++.**

# La librairie standard C++

Le C++ possède une **bibliothèque standard** (SL pour *Standard Library*) qui est composée, entre autre, d'une bibliothèque de flux, de la bibliothèque standard du C, de la gestion des exceptions, ..., et de la **STL** (*Standard Template Library* : **bibliothèque de modèles standard**). La STL implémente de nombreux types de données de manière efficace et générique.

En fait, STL est une appellation historique communément acceptée et comprise. Dans la norme, on ne parle que de SL.

Dans un programme C++, on privilégiera toujours l'utilisation de la STL par rapport à une implémentation manuelle : gain en efficacité, robustesse, facilité, lisibilité car standard.

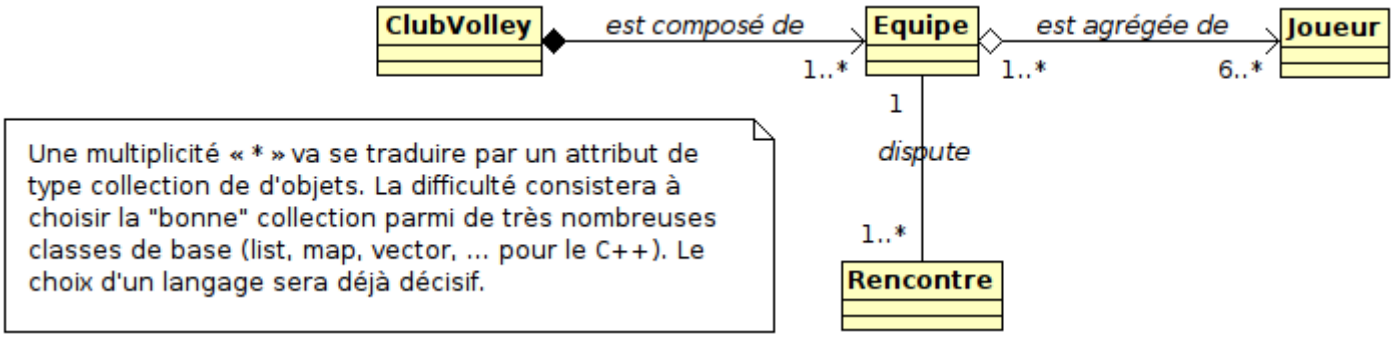
L'objectif de ce document n'est pas de faire un inventaire exhaustif des possibilités offertes par la STL, mais de donner quelques exemples courants d'utilisation. On pourra trouver un aperçu très détaillé des classes de la STL ici : www.sgi.com/tech/stl/ et www.cplusplus.com/reference/stl/.

## Notion de conteneurs

La **STL** fournit un certain nombre de **conteneurs** pour gérer des **collections d'objets** : les **tableaux** (***vector***), les **listes** (*list*), les **ensembles** (*set*), les **piles** (*stack*), et beaucoup d'autres ...

Un **conteneur** (*container*) est un **objet qui contient d'autres objets**. Il fournit un moyen de gérer les objets contenus (au minimum ajout, suppression, parfois insertion, tri, recherche, ...) ainsi qu'un accès à ces objets qui dans le cas de la STL consiste très souvent en un **itérateur**.

*Exemples :* une liste d'articles à commander, une flotte de bateaux, les équipes d'un club, ...




Bien qu'il soit possible de créer des tableaux d'objets, ce ne sera pas forcément la bonne solution. On préfère recourir à des collections parmi lesquelles les plus utilisées sont :

- Java : ArrayList et HashMap
- C# : ArrayList, SortedList et HashTable
- C++ (STL) : vector, list, map et set

## Notion de complexité

Il est particulièrement important de choisir une classe fournie par la STL **cohérente avec son besoin**. Certaines structures sont effectivement plus ou moins efficaces pour accéder à une mémoire ou en terme

de réallocation mémoire lorsqu'elles deviennent importantes. L'enjeu de cette partie consiste à présenter les avantages et les inconvénients de chacune d'elle.

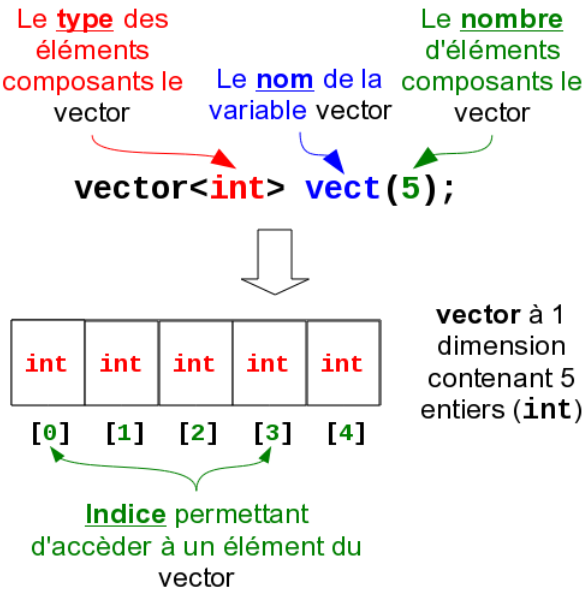
Il est au préalable nécessaire d'avoir quelques notions de **complexité**. Soit `n` la taille d'un conteneur : un algorithme est dit linéaire (en `O(n)`) si son temps de calcul est proportionnel à `n`. De la même façon, un algorithme peut être instantané (`O(1)`), logarithmique `O(log(n))`, linéaire (en `O(n)`), polynomial $O(n^k)$, exponentiel `O(e(n))`, etc... dans l'ordre croissant des proportions de temps de calcul.

Dans ce document, on s'intéressera principalement à la complexité pour l'accès (recherche) à une donnée stockée dans un conteneur et pour insérer une donnée.

## Tableau dynamique en C++ (`vector`)

Un **vector** est un **tableau dynamique** où il est particulièrement aisé d'**accéder directement** aux divers éléments par un **index**, et d'en ajouter ou en retirer à la fin.

A la manière des tableaux de type C, l'espace mémoire alloué pour un objet de type *vector* est toujours continu, ce qui permet des algorithmes rapides d'accès aux divers éléments.




```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
  vector<int> vect(5); // un vecteur de 5 entiers

  vect[0] = 1; // accès direct à l'indice 0 pour affecter la valeur 1
  vect[1] = 2; // accès direct à l'indice 1 pour affecter la valeur 2

  // augmente et diminue la taille du vector
  vect.push_back( 6 ); // ajoute l'entier 6 à la fin
  vect.push_back( 7 ); // ajoute l'entier 7 à la fin
  vect.push_back( 8 ); // ajoute l'entier 8 à la fin
```

```
  vect.pop_back(); // enleve le dernier élément et supprime l'entier 8

  cout << "Le vecteur vect contient " << vect.size() << " entiers : \n";

  // utilisation d'un indice pour parcourir le vecteur vect
  for(int i=0;i<vect.size();i++)
      cout << "vect[" << i << "] = " << vect[i] << '\n';
  cout << '\n';

  return 0;
}
```

*Exemple d'utilisation d'un `vector`*

On obtient à l'exécution :

```
Le vecteur vect contient 7 entiers :
vect[0] = 1
vect[1] = 2
vect[2] = 0
vect[3] = 0
vect[4] = 0
vect[5] = 6
vect[6] = 7
```

Une "case" n'est accessible par l'opérateur `[]` que si elle a été allouée au préalable (sinon une erreur de segmentation est déclenchée).

Il ne faut pas perdre de vue qu'une réallocation mémoire est coûteuse en terme de performances. Ainsi si la taille d'un `vector` est connue par avance, il faut autant que possible le créer directement à cette taille (voir méthodes `resize` et `reserve`).

**Complexité**

**- Accès : `O(1)`**

**- Insertion : `O(1)` en fin de `vector` (`push_back`) sinon `O(n)`. Dans les deux cas des réallocations peuvent survenir.**

## Notion d'itérateurs (`iterator`)

Les **itérateurs** (*iterator*) sont une généralisation des **pointeurs** : ce sont des **objets qui pointent sur d'autres objets**.

Comme son nom l'indique, les itérateurs sont utilisés pour **parcourir une série d'objets** de telle façon que si on incrémente l'itérateur, il désignera l'objet suivant de la série.

```
#include <iostream>
#include <vector>

using namespace std;
```

```
int main()
{
  vector<int> v2(4, 100); // un vecteur de 4 entiers initialisés avec la valeur 100

  cout << "Le vecteur v2 contient " << v2.size() << " entiers : ";
  // utilisation d'un itérateur pour parcourir le vecteur v2
  for (vector<int>::iterator it = v2.begin(); it != v2.end(); ++it)
    cout << ' ' << *it;
  cout << '\n';

  return 0;
}
```

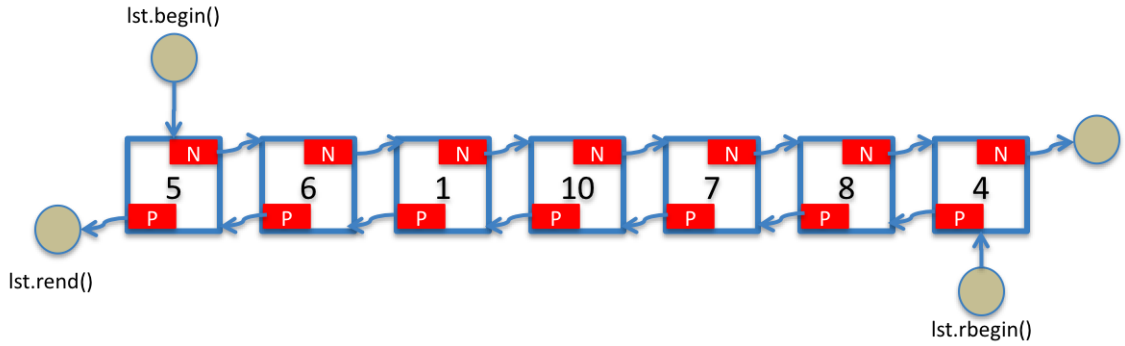
*Exemple d'utilisation d'un `iterator`*

On obtient à l'exécution :

```
Le vecteur v2 contient 4 entiers : 100 100 100 100
```

## Le conteneur liste (`list`)

La classe `list` fournit une structure générique de **listes doublement chaînées** (cad que l'on peut parcourir dans les deux sens) pouvant éventuellement contenir des doublons.




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

using namespace std;

int main()
{
  list<int> lst; // une liste vide

  lst.push_back( 5 );
  lst.push_back( 6 );
  lst.push_back( 1 );
  lst.push_back( 10 );
  lst.push_back( 7 );
  lst.push_back( 8 );
  lst.push_back( 4 );
  lst.push_back( 5 );
```

```
  lst.pop_back(); // enleve le dernier élément et supprime l'entier 5

  cout << "La liste lst contient " << lst.size() << " entiers : \n";

  // utilisation d'un itérateur pour parcourir la liste lst
  for (list<int>::iterator it = lst.begin(); it != lst.end(); ++it)
    cout << ' ' << *it;
  cout << '\n';

  // afficher le premier élément
  cout << "Premier element : " << lst.front() << '\n';

  // afficher le dernier élément
  cout << "Dernier element : " << lst.back() << '\n';

  // parcours avec un itérateur en inverse
  for ( list<int>::reverse_iterator rit = lst.rbegin(); rit != lst.rend(); ++rit )
  {
      cout << ' ' << *rit;
  }
  cout << '\n';

  return 0;
}
```

*Exemple d'utilisation d'une* `list`

On obtient à l'exécution :

```
La liste lst contient 7 entiers :
 5 6 1 10 7 8 4
Premier element : 5
Dernier element : 4
 4 8 7 10 1 6 5
```

**Complexité**

**- Insertion (en début ou fin de liste) : `O(1)`**

**- Recherche : `O(n)` en général, `O(1)` pour le premier et le dernier maillon**

## La table associative (`map`)

Une **table associative** `map` permet d'**associer une clé à une donnée**.

La `map` prend au moins deux paramètres :
– le **type de la clé** (dans l'exemple ci-dessous, une chaîne de caractères `string`)
– le **type de la donnée** (dans l'exemple ci-dessous, un entier non signé `unsigned int`)

```
#include <iostream>
#include <iomanip>
#include <map>
```

```cpp
#include <string>

using namespace std;

int main()
{
  map<string,unsigned int> nbJoursMois;

  nbJoursMois["janvier"] = 31;
  nbJoursMois["février"] = 28;
  nbJoursMois["mars"] = 31;
  nbJoursMois["avril"] = 30;
  //...

  cout << "La map contient " << nbJoursMois.size() << " elements : \n";
  for (map<string,unsigned int>::iterator it=nbJoursMois.begin(); it!=nbJoursMois.end(); ++
     it)
  {
    cout << it->first << " -> \t" << it->second << endl;
  }

  cout << "Nombre de jours du mois de janvier : " << nbJoursMois.find("janvier")->second <<
     '\n';

  // affichage du mois de janvier
  cout << "Janvier : \n" ;
  for (int i=1; i <= nbJoursMois["janvier"]; i++)
  {
     cout << setw(3) << i;
     if(i%7 == 0)
        cout << endl;
  }
  cout << endl;

  return 0;
}
```

*Exemple d'utilisation d'une* `map`

On obtient à l'exécution :

```
La map contient 4 elements :
avril ->       30
février ->     28
janvier ->     31
mars ->        31
Nombre de jours du mois de janvier : 31
Janvier :
  1  2  3  4  5  6  7
  8  9 10 11 12 13 14
 15 16 17 18 19 20 21
 22 23 24 25 26 27 28
 29 30 31
```

**Complexité**

- **Insertion : O(log(n))**

- **Recherche : O(log(n))**

Le fait d'accéder à une clé via l'opérateur [] insère cette clé dans la map (avec le constructeur par défaut pour la donnée). Ainsi l'opérateur [] n'est pas adapté pour vérifier si une clé est présente dans la map, il faut utiliser la méthode find.

## Une paire d'éléments (pair)

Une **paire** est une **structure contenant deux éléments éventuellement de types différents**.

Certains algorithmes de la STL (find par exemple) retournent des paires (position de l'élément trouvé et un booléen indiquant s'il a été trouvé).

En pratique, il faut voir les classes conteneurs de la STL comme des "legos" (briques logicielles) pouvant être imbriqués les uns dans les autres. Ainsi, on pourra tout à fait manipuler un vector de pair, etc ...

```
#include <iostream>
#include <utility>
#include <vector>
#include <list>
#include <map>
#include <set>

using namespace std;

int main()
{
  pair<char,int> c1 = make_pair('B', 2); // coordonnées type "bataille navale"
  pair<char,int> c2 = make_pair('J', 1);

  cout << "Un coup en " << c1.first << '.' << c1.second << endl;

  pair<int,int> p; // position de type "morpion"

  p.first = 1;
  p.second = 2;

  cout << "Un coup en " << p.first << ',' << p.second << endl;
  cout << endl;

  vector<pair<char,int> > tableauCoups(2); // ou par exemple : list<pair<char,int> >
     listeCoups;

  tableauCoups[0] = c1;
  tableauCoups[1] = c2;
  cout << "Le vector contient " << tableauCoups.size() << " coups : \n";
```

```
  for (vector<pair<char,int> >::iterator it=tableauCoups.begin(); it!=tableauCoups.end(); ++
      it)
  {
    cout << (*it).first << "." << (*it).second << endl;
  }
  cout << endl;

  map<pair<char,int>,bool> mapCoups;

  mapCoups[c1] = true; // ce coup a fait mouche
  mapCoups[c2] = false; // ce coup a fait plouf

  cout << "La map contient " << mapCoups.size() << " coups : \n";
  for (map<pair<char,int>,bool>::iterator it=mapCoups.begin(); it!=mapCoups.end(); ++it)
  {
    cout << it->first.first << "." << it->first.second << " -> \t" << it->second << endl;
  }
  cout << endl;

  set<pair<char,int> > ensembleCoups;

  ensembleCoups.insert(c1);
  ensembleCoups.insert(c2);

  cout << "L'ensemble set contient " << ensembleCoups.size() << " coups : \n";
  for (set<pair<char,int> >::iterator it=ensembleCoups.begin(); it!=ensembleCoups.end(); ++
      it)
  {
    cout << (*it).first << "." << (*it).second << endl;
  }
  cout << endl;

  return 0;
}
```

*Exemple d'utilisation d'une* ***pair***

On obtient à l'exécution :

```
Un coup en B.2
Un coup en 1,2

Le vector contient 2 coups :
B.2
J.1

La map contient 2 coups :
B.2 -> 1
J.1 -> 0

L'ensemble set contient 2 coups :
B.2
J.1
```

**Complexité**

- **Insertion : `O(1)`**

- **Recherche : `O(1)`**

## Un ensemble d'éléments (`set`)

Dans l'exemple ci-dessus, on utilise un autre conteneur qui se nomme `set`. La classe `set` est un conteneur qui stocke des éléments uniques suivants un ordre spécifique (c'est-à-dire un ensemble ordonné et sans doublons d'éléments). La complexité est `O(log(n))` pour la recherche et l'insertion.

Les ensembles `set` sont généralement mises en oeuvre dans les arbres binaires de recherche.

```cpp
#include <iostream>
#include <set>

using namespace std;

int main()
{
  int desEntiers1[] = {75, 23, 65, 42, 13, 100}; // non ordonné
  int desEntiers2[] = {75, 23, 75, 23, 13}; // non ordonné avec des doublons
  set<int> ensemble1 (desEntiers1, desEntiers1+6); // the range (first,last)
  set<int> ensemble2 (desEntiers2, desEntiers2+5); // the range (first,last)

  cout << "L'ensemble set 1 contient :";
  for (set<int>::iterator it=ensemble1.begin(); it!=ensemble1.end(); ++it)
  {
    cout << ' ' << *it;
  }

  cout << endl;

  cout << "L'ensemble set 2 contient :";
  for (set<int>::iterator it=ensemble2.begin(); it!=ensemble2.end(); ++it)
  {
    cout << ' ' << *it;
  }

  cout << endl;

  return 0;
}
```

*Exemple d'utilisation d'un conteneur `set`*

À l'exécution, on obtient deux ensembles ordonnés sans doublons :

```
L'ensemble set 1 contient : 13 23 42 65 75 100
L'ensemble set 2 contient : 13 23 75
```

## Suppresion des données contenues dans un conteneur

On distingue deux situations qui dépendent de la nature de ce qui est stocké dans le conteneur :

– s'il s'agit d'**un objet**, il n'est pas utile de le détruire, il le sera lorsqu'il est retiré du vecteur, ou lorsque le vecteur est détruit.
– s'il s'agit d'**un pointeur sur un objet**, il faut le détruire car un pointeur n'est pas un objet. Si cette destruction n'est pas faite, le programme présentera une fuite de mémoire.

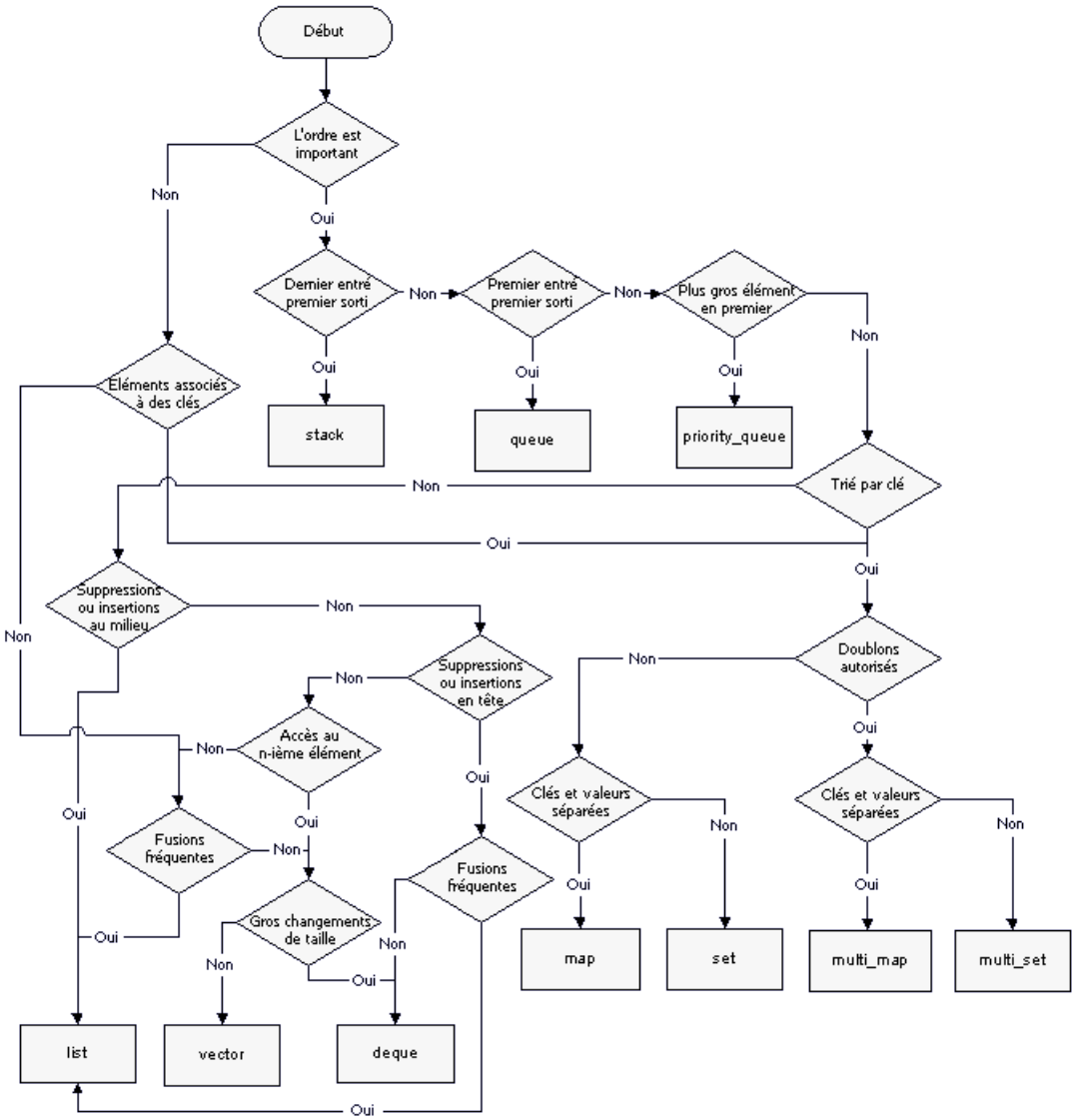
Certains conteneurs proposent la méthode `erase`. Lire la FAQ C++ : cpp.developpez.com/faq/cpp/

## Choix d'un conteneur

La panoplie de conteneurs proposée par la STL est conséquente : conteneurs ordonnés, associatifs, listes chaînées ...

Le choix du "bon" conteneur dépend principalement des opérations que l'on va effectuer sur les données : suppression, ajout, recherche ...

Voici un schéma récapitulatif qui aidera à faire son choix :



## Exemple détaillée : utilisation d'un vector

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <stdexcept>

using namespace std;

int main()
{
    // créer un tableau d'entiers vide
    std::vector<int> v;

    // ajouter l'entier 10 à la fin
    v.push_back( 10 );

    // afficher le premier élément (10)
    cout << "Premier element : " << v.front() << '\n';

    // afficher le dernier élément (10)
    cout << "Dernier element : " << v.back() << '\n';

    // enlever le dernier élément
    v.pop_back(); // supprime '10'

    // le tableau est vide
    if ( v.empty() )
    {
        cout << "Tout est normal : tableau vide\n";
    }

    // redimensionner le tableau
    // resize() initialise tous les nouveaux entiers à 0
    v.resize( 10 );

    // quelle est sa nouvelle taille ?
    cout << "Nouvelle taille : " << v.size() << '\n'; // affiche 10

    // sa taille est de 10 : on peut accéder directement aux
    // 10 premiers éléments
    v[ 9 ] = 5;

    // intitialiser tous les éléments à 100
    std::fill( v.begin(), v.end(), 100 );

    // vider le tableau
    v.clear(); // size() == 0

    // on va insérer 50 éléments
    // réserver (allouer) de la place pour au moins 50 éléments
    v.reserve( 50 );
```

```
    // vérifier que la taille n'a pas bougé (vide)
    cout << "Taille : " << v.size() << '\n';

    // capacité du tableau = nombre d'éléments qu'il peut stocker
    // sans devoir réallouer (modifié grâce à reserve())
    cout << "Capacite : " << v.capacity() << '\n'; // au moins 50, sûrement plus

    for ( int i = 0; i < 50; ++i )
    {
        // grâce à reserve() on économise de multiples réallocations
        // du fait que le tableau grossit au fur et à mesure
        v.push_back( i );
    }

    // afficher la nouvelle taille
    cout << "Nouvelle taille : " << v.size() << '\n'; // affiche 50

    // rechercher l'élément le plus grand (doit être 49)
    cout << "Max : " << *std::max_element( v.begin(), v.end() ) << '\n';

    // tronquer le tableau à 5 éléments
    v.resize( 5 );

    // les trier par ordre croissant
    std::sort( v.begin(), v.end() );

    // parcourir le tableau
    for ( size_t i = 0, size = v.size(); i < size; ++i )
    {
        // attention : utilisation de l'opérateur []
        // les accès ne sont pas vérifiés, on peut déborder !
        cout << "v[" << i << "] = " << v[ i ] << '\t';
    }
    cout << '\n';

    // utilisation de at() : les accès sont vérifiés
    try
    {
        v.at( v.size() ) = 10; // accès en dehors des limites !
    }
    catch ( const std::out_of_range &e )
    {
        cerr << "at() a levé une exception std::out_of_range : " << e.what() << endl;
    }

    // parcours avec un itérateur en inverse
    for ( std::vector<int>::reverse_iterator i = v.rbegin(); i != v.rend(); ++i )
    {
        cout << *i << '\t';
    }
    cout << '\n';

    // on crée un tableau v2 de taille 10
```

```
    std::vector<int> v2( 10 );
    v2.at( 9 ) = 5; // correct, le tableau est de taille 10

    // on crée un tableau v3 de 10 éléments initialisés à 20
    std::vector<int> v3( 10, 20 );

    // faire la somme de tous les éléments de v3
    // on doit obtenir 200 (10 * 20)
    cout << "Somme : " << std::accumulate( v3.begin(), v3.end(), 0 ) << '\n';

    // on recopie v3 dans v
    v = v3;

    // on vérifie la recopie
    if ( std::equal( v.begin(), v.end(), v3.begin() ) )
    {
        cout << "v est bien une copie conforme de v3\n";
    }

    return 0;
}
```

On obtient à l'exécution :

```
Premier element : 10
Dernier element : 10
Tout est normal : tableau vide
Nouvelle taille : 10
Taille : 0
Capacite : 50
Nouvelle taille : 50
Max : 49
v[0] = 0      v[1] = 1      v[2] = 2      v[3] = 3      v[4] = 4
at() a levé une exception std::out_of_range : vector::_M_range_check
4      3      2      1      0
Somme : 200
v est bien une copie conforme de v3
```

# Travail demandé

## Les tableaux dynamiques (`vector`)

Dans cet exercice, on va réaliser un programme `vecteur.cpp` qui permet de stocker des `std::string` et manipuler ceux-ci de manière simple.

**Question 1.** Vous devez déclarer un vecteur de nom `monVecteur` stockant des `std::string`.

**Question 2.** Ajoutez 5 `std::string` dans le vecteur qui vaudront respectivement "bonjour", "comment", "allez", "vous", " ?".

**Question 3.** Affichez la taille de votre vecteur. Afficher sa capacité (`capacity`). Quel est la différence avec sa taille ?

**Question 4.** Afficher le contenu du vecteur en utilisant la notation indexée de tableau en accédant à la valeur avec une syntaxe du type : `monVecteur[k]`.

**Question 5.** Afficher le contenu du vecteur en utilisant les itérateurs (`iterator`) sur votre vecteur.

**Question 6.** Réaliser un échange entre le contenu de la case d'indice 1 et le contenu de la case d'indice 3 de votre vecteur (vérifiez votre résultat en affichant le vecteur). Notez l'existence de `std::swap`.

**Question 7.** Trier le vecteur en utilisant un algorithme de la STL (inclure l'en-tête `algorithm`). L'ordre de tri par défaut est celui de la comparaison sur des `std::string`. Afficher le résultat obtenu.

**Question 8.** Créer une fonction `affiche` qui affiche le contenu du vecteur passé en paramètre. Chaque élément sera espacé d'un 'espace' à l'affichage. Notez qu'ici, on passera le vecteur sous forme de référence constante car il n'a pas à être modifié, ni copié.

**Question 9.** Créer une fonction `concatene` qui concatène l'ensemble des éléments du vecteur dans une seule variable de type `std::string`. Chaque élément sera espacé d'un 'espace' dans la `std::string`. Réfléchir au prototype de votre fonction, sous quelle forme vous passez le paramètre d'entrée, sous quelle forme retournez-vous le `std::string` ?

**Question 10.** Insérer la valeur "a tous" après le premier élément dans votre vecteur. Vérifier votre résultat.

**Question 11.** Changer le point d'interrogation " ?" par " !". Vérifier votre résultat.

## Les listes (`list`)

Éditer un fichier source `liste.cpp` qui permettra de :

**Question 12.** Créer une liste de 30 entiers (les 30 premiers entiers impairs).

**Question 13.** Afficher votre liste.

**Question 14.** Supprimer le troisième élément.

**Question 15.** Afficher à nouveau votre liste.

**Question 16.** Insérer les 20 entiers impairs suivants.

**Question 17.** Afficher à nouveau votre liste.

## Les tables associatives (`map`)

Éditer un fichier source `map.cpp` qui permettra de construire une `std::map` dont la clé est une chaîne de caractère, et la valeur est un nombre flottant.

**Question 18.** Remplir cette `map` avec un exemple de type "facture de restaurant" (nom du plat et prix du plat). Afficher le contenu de la `map`.

**Question 19.** Afficher le contenu de la `map`.

**Question 20.** Calculer et afficher le prix total des plats présents dans la `map`.