La classe vector

Par Frédéric Drouillon

Date de publication : 17 octobre 2014

N'hésitez pas à donner vos avis par rapport à ce tutoriel : **Commentez**

I - La classe vector

Les vecteurs sont des tableaux génériques et dynamiques à une dimension. Ils constituent le modèle des conteneurs séquentiels. Pour l'utiliser, il faut inclure la bibliothèque <vector> :

```
#include <vector>
```

I-A - Template, constructeurs, destructeurs

vector<type élément, type allocateur> : il définit le type des éléments et éventuellement un allocateur spécifique. Si aucun allocateur n'est spécifié, c'est l'allocateur par défaut allocator<T> qui est utilisé. Le template est :

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

Les constructeurs sont ceux évoqués plus haut.

vector() : constructeur par défaut, conteneur vide. Par exemple, voici un vecteur de double vide, zéro (0) élément :

```
vector<double> vd;
```

vector(nombre d'éléments) : crée un conteneur du nombre d'éléments indiqués, chaque élément est initialisé avec une valeur par défaut (en général 0). Par exemple, voici un vecteur de dix floats initialisés à 0 :

```
vector<float> vf(10);
```

vector(nombre d'éléments, valeur initialisation) : crée un conteneur du nombre d'éléments indiqués et initialisés avec la valeur donnée en second paramètre. Par exemple, vecteur de 15 int initialisé à 7 :

```
vector<int>vi(15, 7);
```

vector(liste en copie) : vecteur initialisé avec la copie d'une liste ou d'une portion de liste. Par exemple, voici un vecteur initialisé avec une copie d'un autre vecteur :

```
vector<int>copievi(vi);
```

Autre exemple avec la copie d'une portion d'un autre vecteur. Dans ce cas, les paramètres sont deux itérateurs, le premier donnant le début dans la liste à copier et le second la fin :

```
vector<int> portionvi(vi.begin()+5, vi.end()-5);
```

Appels explicites des destructeurs :

```
vd.~vector();
vf.~vector();
vi.~vector();
portionvi.~vector();
copievi.~vector();
```

I-B - Accès aux éléments, itérateurs

front() : retourne une référence sur le premier élément :

```
vf.front() += 11; // +11 au premier élément
```

back() : retourne une référence sur le dernier élément :

```
vf.back()+=22; // +22 au dernier élément
```

operator[ ] : l'indiçage comme pour tous les tableaux classiques, avec un accès non contrôlé (possibilité de débordement) :

```
for (unsigned i=0; i<vf.size(); i++)
    cout<<vf[i]<<"-";
```

at() : méthode pour indiçage avec accès contrôlé (possibilité de récupérer une erreur en cas de débordement) :

```
for(unsigned i=0; i<vi.size(); i++)
    vi.at(i)=rand()%100;
```

data() : retourne un pointeur sur le premier élément du tableau interne au conteneur :

```
// type redéfini de pointeur
vector<int>::pointer p1 = vi.data();
    for(unsigned i=0; i< vi.size(); i++)
        p1[i]+=1;
// ou type standard de pointeur
int*p2=vi.data();
    for(unsigned i=0; i<vi.size(); i++, p2++)
        cout<<*p2<<"-";
    cout<<endl;
```

Le parcours d'un « vector » peut aussi s'effectuer en utilisant des itérateurs (pointeurs) plutôt que le système d'indiçage. De ce fait, la classe « vector » est équipée avec toutes les méthodes les concernant :

- begin() : retourne un itérateur « iterator » qui pointe sur le premier élément ;
- end() : retourne un itérateur « iterator » qui pointe sur l'élément suivant le dernier (balise de fin, par exemple NULL) ;
- rbegin() : retourne un itérateur « reverse_iterator » qui pointe sur le premier élément de la séquence inverse (le dernier) ;
- rend() : retourne un itérateur « reverse_iterator » qui pointe sur l'élément suivant le dernier dans l'ordre inverse (balise de fin, par exemple NULL) ;
- cbegin() : retourne un itérateur « const_iterator » qui pointe sur le premier élément. L'élément pointé n'est alors accessible qu'en lecture et non en écriture. Il ne peut pas être modifié ;
- cend() : retourne un itérateur « const_iterator » qui pointe sur ce qui suit le dernier élément (balise de fin, par exemple NULL). L'élément pointé n'est alors accessible qu'en lecture et non en écriture. Il ne peut pas être modifié ;
- crbegin() : retourne un itérateur « const_reverse_iterator » qui pointe sur le premier élément de la séquence inverse (le dernier). L'élément pointé n'est accessible qu'en lecture et ne peut pas être modifié ;
- crend() : retourne un itérateur « const_reverse_iterator » qui pointe sur la fin de la séquence inverse, avant le premier élément (balise de fin sens inverse, par exemple NULL). L'élément pointé n'est accessible qu'en lecture et ne peut pas être modifié.

Exemple d'utilisation :

```
vector<int>vi(15, 7); // 15 int initialisés avec 7
vector<int>::iterator it;
    for (it=vi.begin(); it!=vi.end(); it++)
        cout<<*it<<"-";
    cout<<endl;
```

Programme de test, constructeurs, destructeurs, accès

Fichier

```
#include <iostream>
#include <vector>
using namespace std;
// TESTS CONSTRUCTEURS, DESTRUCTEURS
int main()
{
// constructeur par défaut, 0 élément
vector<double> vd;
    //size() donne le nombre d'éléments
    cout<<vd.size()<<endl;
// 10 éléments initialisés à 0
vector<float> vf(10);
    // premier et dernier élément modifiés
    vf.front()+=11;
    vf.back()+=22;
    //affichage
    for (unsigned i=0; i<vf.size(); i++)
    cout<<vf[i]<<"-";
    cout<<endl;
// 15 éléments initialisés à 7
vector<int>vi(15, 7);
// parcours avec un itérateur
vector<int>::iterator it;
    for (it=vi.begin(); it!=vi.end(); it++)
        cout<<*it<<"-";
    cout<<endl;
    // modification de vi
    // at() équivalent de [], mais avec contrôle d'erreur en plus
    for(unsigned i=0; i<vi.size(); i++)
        vi.at(i)=rand()%100;
// initialisé en copie de vi
vector<int>copievi(vi);
    for (unsigned i=0; i<copievi.size(); i++)
        cout<<copievi[i]<<"-";
    cout<<endl;
// initialisé en copie d'une portion de vi
vector<int> portionvi(vi.begin()+5, vi.end()-5);
    for (unsigned i=0; i<portionvi.size(); i++)
        cout<<portionvi.at(i)<<"-";
    cout<<endl;
// modification des données via le pointeur data
// type redéfini de pointeur
vector<int>::pointer p1=portionvi.data();
    for(unsigned i=0; i<portionvi.size(); i++)
        p1[i]+=1;
// type standard de pointeur
int*p2=portionvi.data();
    for(unsigned i=0; i<portionvi.size(); i++, p2++)
        cout<<*p2<<"-";
    cout<<endl;
    // appel explicite destructeur vector
    vd.~vector();
    vf.~vector();
    vi.~vector();
    copievi.~vector();
    portionvi.~vector();
    system("PAUSE");
    return 0;
}
```

I-C - Opérations de liste

Ce sont les opérations d'affectation, d'insertion et de suppression d'éléments n'importe où dans le tableau. À chaque insertion ou suppression, le tableau doit être entièrement reconstruit et recopié, ce qui peut s'avérer coûteux en calcul. C'est pourquoi si de nombreuses suppressions et insertions d'éléments sont prévues dans le programme, le type vecteur n'est sans doute pas le plus approprié et il est préférable alors d'utiliser un conteneur de type « list ».

Le type « vector » permet les opérations standard suivantes :

assign() : remplace le contenu d'un vecteur par un nouveau contenu en adaptant sa taille si besoin. Exemple :

```
// soit trois vecteurs
vector<int> v1;
vector<int> v2;
vector<int> v3;
// v1 remplacé par 10 int à 50
v1.assign(10, 50);
// v2 remplacé par six éléments au centre de v1
vector<int>::iterator it;
it = v1.begin() + 2;
v2.assign(it, v1.end() - 2);
// v3 remplacé par les éléments du tableau tab
int tab[] = { 10, 20, 30 };
v3.assign(tab, tab + 3);
// affichage des tailles des vecteurs
cout << int(v1.size()) << '\n';
cout << int(v2.size()) << '\n';
cout << int(v3.size()) << '\n';
```

insert(p, x) : ajoute un élément x avant l'élément désigné par l'itérateur p :

```
vector<float> v;
// ajoute 1.5 avant, au début
v.insert(v.begin(),1.5);
```

insert(p, n, x) : ajoute n copies de x avant l'élément désigné par l'itérateur p :

```
// ajoute cinq éléments initialisés 20 à
// la fin
v.insert(v.end(),5,20);
```

insert(p, premier, dernier) : ajoute avant l'élément désigné par l'itérateur p les éléments d'une liste compris entre le premier et le dernier itérateur :

```
// ajoute à partir de l'itérateur p une
// sous-liste de v2 dans V
vector<float>::iterator p;
p = v.begin()+v.size()/2;
v.insert(p, v2.begin()+3,v2.end()-3);
```

emplace(p,x) : ajoute un élément x à la position désignée par l'itérateur p et décale le reste. Les arguments passés pour x correspondent à des arguments pour le constructeur de x :

```
// ajoute 100 à la position 2
p=v.begin()+2;
v.emplace(p, 100);
```

emplace_back(x) : ajoute un élément x à la fin de la liste. Les arguments passés pour x correspondent à des arguments pour le constructeur de x :

```
// ajoute 99 puis 88 à la fin
v.emplace_back(99);
v.emplace_back(88);
```

erase(p) : supprime l'élément pointé par l'itérateur p :

```
// supprime les cinq premiers éléments
for(unsigned i=0 ; i<5; i++){
    p = v.begin();
```

```
    v.erase(p);
    affiche_vector(v);
}
```

erase(premier, dernier) : efface les éléments pointés depuis l'itérateur premier à l'itérateur dernier :

```
//supprime les éléments compris entre
//les itérateurs premier et dernier
vector<float>::iterator prem = v.begin()+1;
vector<float>::iterator dern = v.end()-1;
    v.erase(prem,dern);
    affiche_vector(v);
```

clear() : efface tous les éléments d'un conteneur. Équivalent à c.erase(c.begin(), c.end()) :

```
// efface tout le conteneur
v.clear();
```

Programme opérations liste sur vector

Fichier

```
#include <iostream>
#include <vector>
using namespace std;
void affiche_vector(vector<float> v)
{
    if(v.size()==0)
        cout<<"conteneur vide";
    else
        for (unsigned i=0; i<v.size(); i++)
            cout<<v[i]<<"-";
    cout<<endl;
}
int main()
{
vector<float> v;
    // ajoute 1.5 avant, au début
    v.insert(v.begin(),1.5);
    affiche_vector(v);
    // ajoute cinq éléments initialisés à 20 à la fin
    v.insert(v.end(),5,20);
    affiche_vector(v);
    vector<float> v2(10);
    for(unsigned i=0; i<v2.size();i++)
        v2[i]=i;
    // ajoute à partir de p une sous-liste de v2 dans v
    vector<float>::iterator p = v.begin()+v.size()/2;
    v.insert(p, v2.begin()+3,v2.end()-3);
    affiche_vector(v);
    // ajoute un élément à la position p
    p=v.begin()+2;
    v.emplace(p, 100);
    affiche_vector(v);
    // ajoute un élément à la fin
    v.emplace_back(99);
    v.emplace_back(88);
    affiche_vector(v);
    // supprime les cinq premiers éléments
    for(unsigned i=0 ; i<5; i++){
        p = v.begin();
        v.erase(p);
        affiche_vector(v);
    }
//supprime les éléments compris entre les itérateurs premier et dernier
vector<float>::iterator prem = v.begin()+1;
    vector<float>::iterator dern = v.end()-1;
    v.erase(prem,dern);
    affiche_vector(v);
```

Fichier
```
48.     // efface tout le conteneur
49.     v.clear();
50.     affiche_vector(v);
51.     system ("PAUSE");
52.     return 0;
53. }
```

I-D - Opérations de pile et file

Les opérations de pile peuvent s'implémenter avec un « vector ». En revanche, les opérations de file sont déconseillées et les fonctions push_front() et pop_front() ne sont pas implémentées. Pour avoir une file, il vaut mieux utiliser une list<>. Les opérations de pile reposent sur les deux fonctions push_back() et pop_back().

push_back(x) : empilée, elle ajoute l'élément x à la fin.

```
// Pile à partir de la fin (LIFO)
for(int i=0; i<10; i++){ // empile dix éléments
    v.push_back(rand()%100);
    affiche_vector(v);
}
```

pop_back() : dépilée, elle supprime le dernier élément.

```
for(int i=0; i<5; i++){ // dépile cinq éléments
    v.pop_back();
    affiche_vector(v);
}
```

Programme opérations pile, file sur vector

Fichier
```
1. #include <iostream>
2. #include <vector>
3. #include <ctime>
4. using namespace std;
5. void affiche_vector(vector<int> v)
6. {
7.     if(v.size()==0)
8.         cout<<"conteneur vide";
9.     else
10.         for (unsigned i=0; i<v.size(); i++)
11.             cout<<v[i]<<"-";
12.     cout<<endl;
13. }
14. int main()
15. {
16. vector<int> v;
17. int somme = 0;
18.     srand(time(NULL));
19.     // Pile à partir de la fin (LIFO)
20.     for(int i=0; i<10; i++){// empile dix éléments
21.         v.push_back(rand()%100);
22.         affiche_vector(v);
23.         somme += v.back();
24.     }
25.     cout << "Total : " << somme << endl;
26.     for(int i=0; i<5; i++){ // dépile cinq éléments
27.         somme -= v.back();
28.         v.pop_back();
29.         affiche_vector(v);
30.         cout << "Total : " << somme << endl;
31.     }
32.     system ("PAUSE");
33.     return 0;
```

Fichier
```
34. }
```

I-E - Taille et capacité

size() : retourne le nombre d'éléments du « vector ».

```
vector<int> v(5);
    cout<<v.size()<<endl; // 5
```

resize(nb), resize(nb, val) : redimensionne un « vector » avec nb éléments. Si le conteneur existe avec une taille plus petite, les éléments conservés restent inchangés et les éléments supprimés sont perdus. Avec une taille plus grande, les éléments ajoutés sont initialisés avec une valeur par défaut ou avec une valeur spécifiée en val.

```
vector<int> v(5);
for(unsigned i=0; i<v.size(); i++)
    v[i]=i;
affiche_vector(v);// 0,1,2,3,4
v.resize(7);
affiche_vector(v);// 0,1,2,3,4,0,0
v.resize(10,99);
affiche_vector(v);// 0,1,2,3,4,0,0,99,99,99
v.resize(3);
affiche_vector(v);// 0,1,2
```

Si la taille d'un « vector » est modifiée, la totalité de ses éléments peut être déplacée en mémoire. Il est donc déconseillé de conserver des pointeurs sur des éléments de « vector » dont la taille peut être modifiée.

Pour un vecteur, resize() augmente de moitié la capacité totale du conteneur et l'ajuste si la taille demandée et supérieure.

capacity() : retourne le nombre courant d'emplacements mémoire réservés. C'est-à-dire le total de mémoire allouée en nombre d'éléments pour le conteneur. Attention, à ne pas confondre avec le nombre des éléments effectivement contenus retourné par size().

```
vector<int>v(10);
cout<<"nombre elements : "<<v.size()<<endl;//10
cout<<"capacite : "<<v.capacity()<<endl; // 10
v.resize(12);
cout<<"nombre elements : "<<v.size()<<endl; //12
cout<<"capacite : "<<v.capacity()<<endl; // 15
```

empty() : retourne true si le conteneur est vide, false sinon.

```
vector<int> v ; // vide
cout<< (v.empty() ? "vide":"non vide")<<endl;
v.resize(5) ; // non vide
cout<< (v.empty() ? "vide":"non vide")<<endl;
```

max_size() : donne le nombre maximum d'éléments qu'il serait possible de stocker dans le conteneur (dépend de la RAM).

```
vector<float> v ;
cout<<"taille max : "<<v.max_size()<<endl;
```

reserve(nb) : réserve de l'espace pour un total de nb éléments. Pas d'initialisation et déclenche un length_error si > max_size(). Il n'est possible que d'augmenter la taille du vecteur. Il n'est pas possible de diminuer l'espace réservé avec la réservation d'une taille inférieure :

```
vector<int> v ;
v.reserve(100); // alloue un bloc de 100
v.reserve(50) ; // non pris en compte
v.resize (9); // n'utilise que 9
cout<<"nombre elements : "<<v.size()<<endl;//9
cout<<"capacite : "<<v.capacity()<<endl;// 100
```

shrink_to_fit() : rétrécit la capacité du vecteur au plus près du nombre d'éléments qu'il contient.

```
vector<int> v ;
v.reserve(100);
v.resize (9);
v.shrink_to_fit(); // rétrécie à 9
cout<<"nombre elements : "<<v.size()<<endl; //9
cout<<"capacite : "<<v.capacity()<<endl; // 9
```

swap(n) : le conteneur appelant et le conteneur n échangent leur contenu en ajustant chacun leur taille au nouveau contenu.

```
vector<int> v(10) ;
vector<int>v2(20);
for(unsigned i=0; i<v.size(); i++)
    v[i]=i;
for(unsigned i=0; i<v2.size(); i++)
    v2[i]=i*10;
v2.swap(v);
cout<<"nb elements v2 : "<<v2.size()<<endl ; // 10
cout<<"capacite v2 : "<<v2.capacity()<<endl; // 10
affiche_vector(v2); // copie contenu v
cout<<"nb elements v : "<<v.size()<<endl; //20
cout<<"capacite v : "<<v.capacity()<<endl; //20
affiche_vector(v); // copie contenu v2
```

get_allocator() : obtient une copie de l'allocateur du conteneur. La fonction d'un allocateur est d'offrir des méthodes standard pour allouer et désallouer de la mémoire. Mais l'utilisation d'un allocateur est nécessaire uniquement pour une gestion plus pointue et avancée de la mémoire. Les allocateurs implémentés par défaut pour les conteneurs sont suffisants.

Programme taille, capacité « vector »

Fichier

```
#include <cstdio>
#include <iostream>
#include <vector>
using namespace std;
void affiche_vector(vector<int> v);
/*************************************
*************************************/
int main()
{
vector<int> v(5);
    cout<<"------------------size, resize"<<endl;
    cout<<v.size()<<endl; // 5
    for(unsigned i=0; i<v.size(); i++)
        v[i]=i;
    affiche_vector(v);// 0,1,2,3,4
    v.resize(7);
    affiche_vector(v);// 0,1,2,3,4,0,0
    v.resize(10,99);
    affiche_vector(v);// 0,1,2,3,4,0,0,99,99,99
    v.resize(3);
    affiche_vector(v);// 0,1,2
    cout<<"--------------size, resize, capacite"<<endl;
    v.resize(10);
    cout<<"nombre elements : "<<v.size()<<endl; //10
    cout<<"capacite : "<<v.capacity()<<endl; // 10
```

Fichier

```
26.     v.resize(12);
27.     cout<<"nombre elements : "<<v.size()<<endl; //12
28.     cout<<"capacite : "<<v.capacity()<<endl; // 15
29.     cout<<"--------------empty ?, max_size"<<endl;
30.     cout<< (v.empty() ? "vide":"non vide")<<endl;
31.     cout<<"taille max : "<<v.max_size()<<endl;
32.     cout<<"--------------reserve"<<endl;
33.     v.reserve(100);
34.     v.resize (9);
35.     cout<<"nombre elements : "<<v.size()<<endl; //9
36.     cout<<"capacite : "<<v.capacity()<<endl; // 100
37.     cout<<"--------------shrink_to_fit"<<endl;
38.     v.shrink_to_fit();
39.     cout<<"nombre elements : "<<v.size()<<endl; //9
40.     cout<<"capacite : "<<v.capacity()<<endl; //9
41.     cout<<"--------------swap"<<endl;
42. vector<int>v2(20);
43.     for(unsigned i=0; i<v2.size(); i++)
44.         v2[i]=i*10;
45.     v2.swap(v);
46.     cout<<"nombre elements v2 : "<<v2.size()<<endl; //9
47.     cout<<"capacite v2 : "<<v2.capacity()<<endl; //9
48.     affiche_vector(v2); // copie contenu v
49.     cout<<"nombre elements v : "<<v.size()<<endl; //20
50. cout<<"capacite v : "<<v.capacity()<<endl; //20
51. affiche_vector(v); // copie contenu v2
52. system ("PAUSE");
53. return 0;
54. }
55. /**************************************
56. **************************************/
57. void affiche_vector(vector<int> v)
58. {
59. for (unsigned i=0; i<v.size(); i++)
60. cout<<v[i]<<"-";
61. cout<<endl;
62. }
63. /**************************************
64. **************************************/
```

I-F - Comparaisons entre vecteurs

Les opérateurs de comparaison sont surchargés pour permettre la comparaison d'un vecteur avec un autre. Seuls les types de base (int, float, string, etc.) sont pris en compte. L'algorithme consiste d'abord à regarder s'ils ont le même nombre d'éléments, et si oui, comparer chaque élément.

`==` : donne « true » si le contenu de deux conteneurs est identique, « false » sinon.

```
vector<int>v1(10,5);
vector<int>v2(10,5);
cout<<(v1==v2 ? "identiques":"différents")<<endl;
```

`!=` : donne « true » si le contenu de deux conteneurs est différent, « false » sinon.

```
cout<<(v1!=v2 ? "differents":"identiques")<<endl;
```

`a < b` : donne « true » si le conteneur a se situe avant le conteneur b d'un point de vue lexicographique. Les éléments sont comparés un par un.

```
cout<<(v1 < v2 ? "avant":"apres ou identique")<<endl;
```

Programme comparaisons entre vecteurs

Fichier

```
#include <cstdio>
#include <ctime>
#include <cstdlib>
#include <iostream>
#include <vector>
using namespace std;
int main()
{
vector<int>v1(10,5);
vector<int>v2(10,5);
     cout<<(v1==v2 ? "identiques":"différents")<<endl;
     cout<<(v1!=v2 ? "differents":"identiques")<<endl;
     cout<<(v1 < v2 ? "avant":"apres ou identique")<<endl;
     srand(time(NULL));
     for(unsigned i=0; i<v1.size();i++){
         v1[i]=rand()%10;
         cout<<v1[i]<<"-";
     }
     cout<<endl;
     for(unsigned i=0; i<v2.size();i++){
         v2[i]=rand()%10;
         cout<<v2[i]<<"-";
     }
     cout<<endl;
     cout<<(v1==v2 ? "identiques":"différents")<<endl;
     cout<<(v1!=v2 ? "differents":"identiques")<<endl;
     cout<<(v1 < v2 ? "avant":"apres ou identique")<<endl;
     system ("PAUSE");
     return 0;
}
```

I-G - Vecteurs spécialisés de booléens

Les vecteurs de booléens sont des tableaux de bits. Le fait que chaque valeur puisse être codée sur un simple bit représente une optimisation très importante en mémoire. Par exemple, un entier sur 32 bits suffit à coder un tableau de 32 booléens. Les vecteurs de booléens implémentent tous les constructeurs, destructeurs et méthodes de la classe « vector » avec les différences suivantes :

- valeurs booléennes ;
- chaque élément n'accepte que les valeurs 0, 1, false, true ;
- pas de bloc accessible ;
- il n'y a pas d'attribut data (pointeur sur le bloc mémoire alloué pour le tableau).

flip() : cette fonction inverse toutes les valeurs du tableau.

```
// 10 bool à false
vector<bool> bo(10,false);
    // premier et dernier éléments modifiés
    bo.front()=true;
    bo.back()=true;
//une copie de bo
vector<bool>copie(bo);
    // inverse les valeurs
    copie.flip();
    for (unsigned i=0; i<copie.size(); i++)
    cout<<copie.at(i)<<"-";
    cout<<endl;
```

swap(n), swap(b1,b2) : comme pour les autres vecteurs, échange le contenu du conteneur appelant avec celui donné en paramètre en ajustant les tailles si nécessaires.

```
vector<bool>b1(2,false);
vector<bool>b2(4,true);
affiche_vector(b1);// 0-0-
```

```
affiche_vector(b2);// 1-1-1-1-
b1.swap(b2);
affiche_vector(b1);// 1-1-1-1-
affiche_vector(b2);// 0-0-
```

Mais pour les booléens la fonction swap permet également de permuter des éléments passés en b1 et b2.

```
b1[0]=true;
b1[1]=false; // 1-0-1-1-
b1.swap(b1[0],b1[1]);
affiche_vector(b1); // 0-1-1-1-
```

Programme booléens sur « vector »

Fichier

```
#include <iostream>
#include <vector>
using namespace std;
void affiche_vector(vector<bool> v);
/*************************************
*************************************/
int main()
{
vector<bool> bo(10,false);
     cout<<"-----premier et dernier élément modifiés :"<<endl;
     bo.front()=true;
     bo.back()=true;
     affiche_vector(bo);
     cout<<"-----parcours avec un itérateur :"<<endl;
vector<bool>::iterator it;
     for (it=bo.begin(); it!=bo.end(); it++)
     cout<<*it<<"-";
     cout<<endl;
vector<bool>copie(bo);
     cout<<"-----copie de bo :"<<endl;
     affiche_vector(copie);
     cout<<"-----inverser les valeurs de copie :"<<endl;
     copie.flip();
     affiche_vector(copie);
     cout<<"-----echanger les valeurs entre bo et copie :"<<endl;
     bo.swap(copie);
     cout<<"bo : ";
     affiche_vector(bo);
     cout<<"copie : ";
     affiche_vector(copie);
     cout<<"---permuter des elements de bo:"<<endl;
     bo.swap(bo[0],bo[1]);
     affiche_vector(bo);
     // appel explicite destructeur vector
     bo.~vector();
     system("PAUSE");
     return 0;
}
/*************************************
*************************************/
void affiche_vector(vector<bool> v)
{
     for (unsigned i=0; i<v.size(); i++)
         cout<<v[i]<<"-";
     cout<<endl;
}
/*************************************
*************************************/
```

II - Conclusion

Ce tutoriel est extrait du livre **Du C au C++ - De la programmation procédurale à l'objet** de **Frédéric Drouillon**.

III - Remerciements

Je tiens sincèrement à remercier **Vincent VIALE** pour la mise au gabarit et **Claude LELOUP** pour la relecture orthographique.