

Mon Premier Objet en C++

La Classe Vector

L'objectif de la classe `vector` est de fournir des objets se comportants de façon identiques aux tableaux C, mais avec l'avantage d'une gestion transparente de l'allocation mémoire. Dans le cadre de ce TME, nous travaillerons sur un tableau d'entier.

Construction

Un `Vector` devra pouvoir être construit à partir de rien (il sera vide) ou à partir d'un tableau C ordinaire. On fournira aussi une implémentation du constructeur par copie.

```
int ordered[] = { 0, 1, 2, 3, 5 };
Vector v1 ( ordered, 5 );
```

En plus des construct

Stratégie d'allocation mémoire

Plutôt que d'allouer un tableau sous-jacent de la taille exacte du nombre des éléments contenus, le `Vector` réserve à l'avance de l'espace libre. Ceci afin de limiter le nombre d'opérations de réallocation ainsi que de trop fragmenter la mémoire.

Nous sommes donc amenés à distinguer deux quantités:

- Le nombres d'éléments réellement utilisés du `Vector`, ce nombre sera indiqué par la méthode `size()`.
- Le nombres d'éléments que peut contenir le `Vector` sans avoir à être ré-alloué. Nombre indiqué par la méthode `capacity()`.

Le tableau sous-jacent du `Vector` aura une taille strictement croissante. Il se ré-allouera dans des zones mémoires de plus en plus grandes.

Attention: lors d'une ré-allocation, la zone mémoire précédemment occupée doit être libérée, ainsi qu'a la destruction de l'objet.

La taille du vecteur doublera (sur un multiple de 2) à chaque réallocation. Ce qui revient à dire qu'un `Vector` ne pourra avoir que des tailles de: 0, 1, 2, 4, 8, 16, 32, 64, ...

Cas particulier: si l'on connaît la taille *à priori* du vecteur, on dispose d'une méthode `resize(size_t)` pour le dimensionner directement à la bonne taille.

L'allocation mémoire sera gérée grâce aux deux méthodes privées suivantes:

- `_resize(size_t)`, assure la ré-allocation pour n'importe quelle capacité, ne fait rien si on demande une diminution de celle-ci. Dans ce dernier cas, on affichera le message d'avertissement suivant:

```
[WARNING] Vector::_resize() cowardly refusing to shrink
(16 to 7)
```

- `_growPolicy(size_t newcapacity)`, calcule la nouvelle capacité nécessaire pour pouvoir stocker `newcapacity`.

Ajout, retrait et accès aux éléments

Dans un premier temps, l'ajout et le retrait d'élément se fera uniquement en fin de tableau avec les méthodes suivantes:

- `void push_back(int)`, ajoute un élément.
- `int pop_back()`, renvoie et retire le dernier élément. Dans le cas où le `Vector` est vide, afficher le message suivant:

```
[ERROR] Vector::pop_back(): vector is empty."
```

L'accès aux éléments se fera par l'intermédiaire de l'opérateur d'accès indexé. Si l'index est hors borne, afficher le message:

```
[ERROR] Vector::operator[](): Index 12 is out of bound (10) "
```

Identificateur

On désire, de plus marquer chaque `Vector` avec un identificateur unique, proposer une mécanique. La méthode `{{size_t id()}}` renverra l'identificateur du vecteur.

Affichage d'un `Vector` dans un flot de sortie

Ajouter la mécanique nécessaire à l'affichage d'un vecteur dans un flot. On désire obtenir l'affichage suivant:

```
<Vector id:0 5/8 [0 1 2 3 5]>
```

Où les informations sont respectivement, l'identificateur du vecteur, le nombre d'éléments actuellement contenus, la capacité puis les valeurs des éléments.

Echange du contenu de deux `Vector`

On réalisera une fonction `void swap(Vector&)` qui échangera le contenu de deux vecteurs. Astuce: comment utiliser cette fonction pour libérer la mémoire occupée par un vecteur?

Petit Programme de test

```
#include <iostream>
using namespace std;

#include "Vector.h"

int main ( int argc, char* argv[] )
{
    int ordered[] = { 0, 1, 2, 3, 5 };
    Vector v1 ( ordered, 5 );

    cout << "v1[0] = " << v1[0] << endl;
    cout << "v1[5] = " << v1[5] << endl;
    cout << "v1 " << v1 << endl;

    Vector v2;
    v2.reserve ( 5 );
    for ( size_t i=0 ; i<5 ; ++i ) v2.push_back ( 4-i );
    cout << "v2 " << v2 << endl;
    v2.push_back ( 5 );
```

```

    cout << "v2 " << v2 << endl;
    for ( size_t i=0 ; i<8 ; ++i ) v2.pop_back ();
    cout << "v2 " << v2 << endl;

    cout << "Number of allocated vectors:" << Vector::getMaxId() << endl;

    return 0;
}

```

Iterateur

En première approximation, un *iterateur* (ou `iterator`) est une classe conçue de façon à imiter le comportement d'un pointeur.

La classe `iterator` comportera deux membres:

- `_vector`, le vecteur auquel il se rapporte. Peut être `NULL` s'il n'est apparié à aucun `Vector`
- `_index`, l'élément actuellement pointé dans le vecteur. Si l'itérateur est invalide, on renverra le message suivant:

[ERROR] Vector::iterator::operator*(): invalid iterator.

La class `Vector` s'enrichi alors de deux méthodes supplémentaires:

- `iterator begin()`, renvoie un itérateur sur le premier élément du vecteur.
- `iterator end()`, renvoie un itérateur pointant sur un élément situé *virtuellement* après le dernier élément du vecteur.

Considérant le code suivant:

```

Vector::iterator it = v1.begin ();
Vector::iterator end = v1.end ();

cout << "v1:";
for ( ; it != end ; ++it ) {
    cout << " " << *it;
}
cout << endl;

it = v1.begin();
it += 3;
cout << "iterator begin()+3 on v1: " << *it << endl;
while ( not v1.empty() ) v1.pop_back ();
cout << "iterator begin()+3 on v1: " << *it << endl;

```

Ecrire le code des itérateurs de vecteurs.

Classes imbriqués

Il est possible d'imbruquer une classe dans une autre, mise en pratique dans le cas d'un itérateur:

```

class Vector {
public:
    class Iterator {
public:
        Iterator ( Vector* v=NULL, size_t index=0 );
    };
    // ...
}

```

```
};  
  
Vector::Iterator::Iterator ( Vector* v, size_t index )  
{ }
```

Il faut simplement répéter le préfix `Vector::` devant les définitions des fonctions membres à l'extérieur de la classe.