IN202 - Programmation Orientée Objet pour l'embarqué
TP2

1 Préambule

1.1 Outils de développement

L'utilisation d'un IDE tel que `code::blocks` présente de nombreux avantages mais peut aussi rendre les choses plus confuses aux néophites. Je recommande donc à tous ceux qui ne sont pas à l'aise avec code : :blocks de passer sur un éditeur de texte simple avec un terminal pour compiler.

1.1.1 Récupération des corrections

Les corrections peuvent être récupérées à l'adresse `https://github.com/MeatBoy106/IN202`. Les explications sont sur la page.

2 LIFO

Cet exercice vise à implémenter une structure de type Last In First Out aussi appelée pile (stack). Une pile est une structure de donnée supportant 2 opérations :

— push : ajout d'un élément sur le sommet de la pile

— pop : lecture et retrait de l'élément au sommet de la pile

Vous prêterez attention au principe d'encapsulation en ne laissant accessible que l'interface de la classe implémentée. Rappel de vocabulaire : la *déclaration* prend place dans les headers (.h ou .hpp) et la *définition* dans les fichiers source (.cpp). Certaines questions demandent de définir des choses, d'autres de les déclarer. Ne mélangez pas ces notions si vous voulez respecter la progression de l'exercice.

2.1 Pile de taille fixe

1. Déclarez dans un fichier `stack.hpp` une classe Stack contenant uniquement un tableau statique [1] d'entiers de taille 1024
2. Déclarez les méthodes `push`, `pop` et `size`.
3. Déclarez une variable de type Stack dans la fonction `main`. Cela compile-t-il ? Est-ce normal ?
4. Vérifiez si la pile créée est bien vide en affichant la valeur retournée par `Stack::size()`. Cela compile-t-il ? Est-ce normal ? Faites le nécessaire pour corriger le problème.
5. Définissez maintenant les méthodes `push` et `pop` de Stack. Testez-les.
6. Que se passe-t-il si `pop` est appelé sur une pile vide ou que `push` est appelé sur une pile pleine ? [2]
7. Définissez un attribut de type `const int` qui contient la taille de la pile. Cela compile-t-il ? Utiliser un attribut statique pour corriger le problème.

1. Ne pas confondre tableau statique et attribut statique. Il est ici demandé un tableau statique, c'est à dire un tableau de taille fixe. Il n'est pas demandé que ce tableau statique soit précédé du mot clef `static`, ce qui n'aurait aucun sens pour notre Stack.

2. Une fonctionnalité du C++ appelée les *exceptions* permet de gérer ce genre de situation. Elle ne sera pas abordée dans ce cours car rare dans le contexte de la programmation embarquée.

2.2 Pile de taille variable

1. Utilisez le constructeur de `Stack` pour définir dynamiquement la taille du tableau. Utilisez pour cela l'opérateur `new[]`.
2. Apportez les modifications supplémentaires nécessaires et testez votre pile dynamique.
3. Au fait, qu'avez-vous fait de la mémoire allouée par `new[]` ? Utilisez le déstructeur de `Stack` pour faire le ménage.

2.3 Bonus difficile

Ajustez dynamiquement la taille de la pile lorsqu'un élément est ajouté alors qu'elle est pleine.

3 À la rencontre de la Standard Template Library (STL)

La STL est la bibliothèque standard du C++. Tout langage digne de ce nom dispose d'une bibliothèque standard, c'est à dire d'une bibliothèque fournissant tout un ensemble de fonctionnalités de base utiles dans à peu près tous les programmes.

3.1 std : :vector, le conteneur de référence

Le conteneur le plus utilisé et le plus polivalent de la STL est `std::vector`. Il s'agit d'une classe représentant un tableau dynamique, c'est à dire une tableau dont la taille peut changer au cours de l'exécution d'un programme. Afin de créeer un vecteur, une syntaxe particulière doit être utilisée.

```
#include <vector>
//Create an empty vector of int
std::vector<int> vi;
//Create a vector of 10 doubles filled with 42
std::vector<double> vd(10, 42.);
```

Il est alors possible d'ajouter des éléments dans un vecteur ou d'en retirer.

```
#include <vector>
std::vector<int> vi; //vi = []
vi.push_back(1);     //vi = [1]
vi.push_back(-1);    //vi = [1, -1]
vi.pop_back();       //vi = [1]
```

La taille du vecteur s'obtient à l'aide de la méthode `st::vector::size()` ;
L'accès à l'élément en position `i` d'un `std::vector v` se fait via l'expression `v[i]`.

1. Dans la fonction `main`, créer quelques vecteur contenant différents type, insérez des éléments, affichez le contenu du vecteur et retirez-lui des éléments pour vous familiariser avec son utilisation.
2. Utilisez un vecteur pour réaliser le bonus de l'exercice 1.

3.2 Itérateurs

Un concept fondamental de la STL est celui d'itérateur :

— Un itérateur est un objet permettant d'accéder aux éléments d'un conteneur de manière indépendante du conteneur en question.

— La valeur du conteneur pointée par un itérateur s'obtient avec l'opérateur `*` comme s'il s'agissait d'un pointeur : `*it`.

— Nous allons ici utiliser le type d'itérateur le plus courant : le *forward iterator*. C'est un type d'itérateur qui permet de parcourir un conteneur du premier au dernier élément.

— Un itérateur sur le premier élément d'un conteneur tel qu'un `std::vector v` s'aobtient par l'appel de la méthode `begin()`.

— L'itérateur pointant sur l'élément *imaginaire* positionné après le dernier élément du conteneur [3] s'obtient en appelant la méthode `end()`.

— Un *forward iterator* de `std::vector<int>` est de type `std::vector<int>::iterator`. C'est long et fastidieux à tapper. Il est cependant possible de laisser le compilateur déduire automatiquement le type de l'itérateur lors de sa déclaration via le mot clef `auto` qui remplace le type de la variable itérateur : `auto it = v.begin();`. `auto` peut déduire n'importe quel type lors de la déclaration de n'importe quel variable.

— Le passage à l'élément suivant se fait en appliquant l'opérateur `++` à un itérateur.

Voici un exemple complet faisant usage des itérateurs.

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

int main(){
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    printf("[");
    for(auto it = v.begin(0 ; v != v.end() ; ++v){ //or v++
        printf("%d, ", *it);
    }
    printf("]\n");
}
```

On comprend ici la raison pour laquelle l'itérateur obtenu par l'appel de `end()` pointe après le dernier élément. S'il pointait sur le dernier élément, la condition de sortie de la boucle de l'exemple ci-dessus aurait été vrai avant que le dernier élément n'ait été affiché. Cet itérateur un peu particulier n'est donc là que pour tester la fin du parcours du conteneur et non pas pour accéder à un élément.

5. Utiliser des itérateurs chaque fois que cela est possible dans la classe `stack`. En particulier, faites en sorte de ne plus utiliser la notation `v[i]`. Référez-vous à la documentation de `std::vector` sur l'un des deux sites de référence pour trouver des alternatives : `cppreference.com` ou `cplusplus.com`.
6. Remplacez dans la classe `Stack` le `std::vector<int>` par un `std::list<int>`. Tout devrait continuer de fonctionner sans modification supplémentaire.

3. https ://en.cppreference.com/w/cpp/container/vector/end