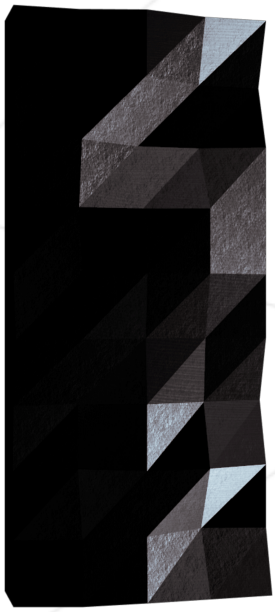

C++ - Module 08

Containers templates, itérateurs, algorithmes

Résumé:
Ce document contient les exercices du Module 08 des C++ modules.

Version: 7

Table des matières

Chapitre I

Introduction

C++ is a general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language, or "C with Classes" (source : Wikipedia).

C++ est un langage de programmation compilé permettant la programmation sous de multiples paradigmes, dont la programmation procédurale, la programmation orientée objet et la programmation générique. Ses bonnes performances, et sa compatibilité avec le C en font un des langages de programmation les plus utilisés dans les applications où la performance est critique (source : Wikipedia).

Ces modules ont pour but de vous introduire à la **Programmation Orientée Objet**. Plusieurs langages sont recommandés pour l'apprentissage de l'OOP. Du fait qu'il soit dérivé de votre bon vieil ami le C, nous avons choisi le langage C++. Toutefois, étant un langage complexe et afin de ne pas vous compliquer la tâche, vous vous conformerez au standard C++98.

Nous avons conscience que le C++ moderne est différent sur bien des aspects. Si vous souhaitez pousser votre maîtrise du C++, c'est à vous de creuser après le tronc commun de 42 !

Chapitre II

Consignes générales

Compilation

- Compilez votre code avec `c++` et les flags `-Wall -Wextra -Werror`
- Votre code doit compiler si vous ajoutez le flag `-std=c++98`

Format et conventions de nommage

- Les dossiers des exercices seront nommés ainsi : `ex00, ex01, ... , exn`
- Nommez vos fichiers, vos classes, vos fonctions, vos fonctions membres et vos attributs comme spécifié dans les consignes.
- Rédigez vos noms de classe au format **UpperCamelCase**. Les fichiers contenant le code d'une classe porteront le nom de cette dernière. Par exemple : `NomDeClasse.hpp`/`NomDeClasse.h`, `NomDeClasse.cpp`, ou `NomDeClasse.tpp`. Ainsi, si un fichier d'en-tête contient la définition d'une classe "BrickWall", son nom sera `BrickWall.hpp`.
- Sauf si spécifié autrement, tous les messages doivent être terminés par un retour à la ligne et être affichés sur la sortie standard.
- *Ciao Norminette !* Aucune norme n'est imposée durant les modules C++. Vous pouvez suivre le style de votre choix. Mais ayez à l'esprit qu'un code que vos pairs ne peuvent comprendre est un code que vos pairs ne peuvent évaluer. Faites donc de votre mieux pour produire un code propre et lisible.

Ce qui est autorisé et ce qui ne l'est pas

Le langage C, c'est fini pour l'instant. Voici l'heure de se mettre au C++ ! Par conséquent :

- Vous pouvez avoir recours à quasi l'ensemble de la bibliothèque standard. Donc plutôt que de rester en terrain connu, essayez d'utiliser le plus possible les versions C++ des fonctions C dont vous avec l'habitude.
- Cependant, vous ne pouvez avoir recours à aucune autre bibliothèque externe. Ce qui signifie que C++11 (et dérivés) et l'ensemble `Boost` sont interdits. Aussi, certaines fonctions demeurent interdites. Utiliser les fonctions suivantes résultera

en la note de 0 : `*printf()`, `*alloc()` et `free()`.

- Sauf si explicitement indiqué autrement, les mots-clés `using namespace <ns_name>` et `friend` sont interdits. Leur usage résultera en la note de -42.
- **Vous n'avez le droit à la STL que dans le Module 08**. D'ici là, l'usage des **Containers** (vector/list/map/etc.) et des **Algorithmes** (tout ce qui requiert d'inclure `<algorithm>`) est interdit. Dans le cas contraire, vous obtiendrez la note de -42.

Quelques obligations côté conception

- Les fuites de mémoires existent aussi en C++. Quand vous allouez de la mémoire (en utilisant le mot-clé `new`), vous ne **devez pas avoir de memory leaks**.
- Du Module 02 au Module 08, vos classes devront se conformer à la forme **canonique, dite de Coplien, sauf si explicitement spécifié autrement**.
- Une fonction implémentée dans un fichier d'en-tête (hormis dans le cas de fonction template) équivaudra à la note de 0.
- Vous devez pouvoir utiliser vos fichiers d'en-tête séparément les uns des autres. C'est pourquoi ils devront inclure toutes les dépendances qui leur seront nécessaires. Cependant, vous devez éviter le problème de la double inclusion en les protégeant avec des **include guards**. Dans le cas contraire, votre note sera de 0.

Read me

- Si vous en avez le besoin, vous pouvez rendre des fichiers supplémentaires (par exemple pour séparer votre code en plus de fichiers). Vu que votre travail ne sera pas évalué par un programme, faites ce qui vous semble le mieux du moment que vous rendez les fichiers obligatoires.
- Les consignes d'un exercice peuvent avoir l'air simple mais les exemples contiennent parfois des indications supplémentaires qui ne sont pas explicitement demandées.
- Lisez entièrement chaque module avant de commencer ! Vraiment.
- Par Odin, par Thor ! Utilisez votre cervelle !!!

Vous aurez à implémenter un bon nombre de classes, ce qui pourrait s'avérer ardu... ou pas ! Il y a peut-être moyen de vous simplifier la vie grâce à votre éditeur de texte préféré.

Vous êtes assez libre quant à la manière de résoudre les exercices. Toutefois, respectez les consignes et ne vous en tenez pas au strict minimum, vous pourriez passer à côté de notions intéressantes. N'hésitez pas à lire un peu de théorie.

Chapitre III

Consignes spécifiques au Module

Comme vous le verrez, il est possible de réussir les exercices de ce module SANS les containers standards et SANS les algorithmes standards.

Cependant, **le but de ce Module est précisément de les utiliser**. Vous avez le droit à la STL. Oui, vous pouvez utiliser les **Containers** (vector/list/map/etc.) et les **Algorithmes** (définis dans le fichier d'en-tête `<algorithm>`). Vous devez vraiment les utilisez le plus possible dans ce Module. Par conséquent, faites de votre mieux pour y avoir recours de manière pertinente.

Dans le cas contraire, vous obtiendrez une très mauvaise note. Ceci, même si votre code fonctionne comme attendu. Ne faites pas preuve de paresse.

Vous pouvez définir vos templates dans les fichiers d'en-tête comme d'habitude. Ou alors, vous pouvez aussi écrire les déclarations de vos templates dans les fichiers d'en-tête et leurs implémentations dans des fichiers .tpp. En tout cas, les fichiers d'en-tête restent obligatoires tandis que les fichiers .tpp sont optionnels.

Chapitre IV

Exercice 00 : Easy find

	Exercice : 00
Easy find	
Dossier de rendu : $ex00/$	
Fichiers à rendre : `Makefile, main.cpp, easyfind.{h, hpp}` `fichier optionnel : easyfind.tpp`	
Fonctions interdites : `Aucune`	

Commençons tranquillement avec un premier exercice facile.

Écrivez une fonction template `easyfind` acceptant un type `T`. Elle prend deux paramètres. Le premier est de type `T` et le second un nombre entier.

Partant du principe que `T` est un container **d'entiers**, cette fonction doit trouver la première occurrence du second paramètre dans le premier paramètre.

Si aucune occurrence n'a été trouvée, vous pouvez soit jeter une exception, soit retourner une valeur d'erreur de votre choix. Si vous avez besoin d'inspiration, jetez un oeil au comportement des containers standards.

Bien entendu, implémentez et rendez vos propres tests pour démontrer que tout marche comme attendu.

`Vous n'avez pas à gérer les containers associatifs.`

Chapitre V

Exercice 01 : Span

	Exercice : 01
Span	
Dossier de rendu : $ex01/$	
Fichiers à rendre : `Makefile, main.cpp, Span.{h, hpp}, Span.cpp`	
Fonctions interdites : `Aucune`	

Créez une classe **Span** pouvant stocker un maxium de `N` entiers. `N` est une variable de type entier non-signé et sera le seul paramètre passé au constructeur.

Cette classe aura une fonction membre appelée `addNumber()` afin d'ajouter un seul nombre à la Span. On l'utilisera pour remplir cette dernière. Toute tentative d'ajouter un nouvel élément s'il y en a déjà `N` autres stockés jettera une exception.

Ensuite, implémentez deux fonctions membres : `shortestSpan()` et `longestSpan()`

Elles devront respectivement trouver la plus petite distance et la plus grande distance entre les nombres stockés, puis la retourner. S'il n'y a aucun nombre stocké, ou juste un, aucune distance ne peut être trouvée. Par conséquent, jetez une exception.

Bien sûr, implémentez vos propres tests qui devront être bien plus complets que celui donné ci-dessous. Testez votre Span avec au moins 10 000 nombres. Vous pouvez tester avec plus de nombres, c'est encore mieux.

Exécuter ce code :

```
int main()
{
  Span sp = Span(5);

  sp.addNumber(6);
  sp.addNumber(3);
  sp.addNumber(17);
  sp.addNumber(9);
  sp.addNumber(11);

  std::cout << sp.shortestSpan() << std::endl;
  std::cout << sp.longestSpan() << std::endl;

  return 0;
}
```

Devrait afficher :

```
$> ./ex01
2
14
$>
```

Le meilleur pour la fin. Une possibilité qu'il serait utile d'avoir, c'est celle de pouvoir remplir votre Span en utilisant une **plage d'itérateurs** (*range of iterators*). Faire des centaines d'appels à `addNumber()`, c'est plutôt répétitif. Implémentez une fonction qui permet d'ajouter plusieurs nombres à votre Span en un appel.

Si vous ne voyez pas quoi faire, étudiez les Containers. Certaines fonctions membres prennent une plage d'itérateurs afin d'ajouter une séquence d'éléments au container.

Chapitre VI

Exercice 02 : Abomination mutante

	Exercice : 02
Abomination mutante	
Dossier de rendu : $ex02/$	
Fichiers à rendre : `Makefile, main.cpp, MutantStack.{h, hpp}` `fichier optionnel : MutantStack.tpp`	
Fonctions interdites : `Aucune`	

Il est temps de passer aux choses sérieuses. Créons une classe mutante.

Le container `std::stack` est top. Malheureusement, c'est l'un des seuls de la STL à ne PAS être itérable. Trop nul.

Mais d'où devrions-nous tolérer cela ? Surtout quand on peut s'octroyer le droit de charcuter la stack originale pour ajouter cette caractéristique manquante.

Afin de réparer cette injustice, vous allez rendre le container `std::stack` itérable.

Implémentez une classe **MutantStack**. Elle sera **implémentée en termes de** la `std::stack`. Elle offrira toutes ses fonctions membres avec en plus des **itérateurs**.

Bien entendu, implémentez et rendez vos propres tests afin de démontrer que tout marche comme attendu.

Vous trouverez un exemple de test ci-dessous.

```
int main()
{
MutantStack<int>    mstack;

mstack.push(5);
mstack.push(17);

std::cout << mstack.top() << std::endl;

mstack.pop();

std::cout << mstack.size() << std::endl;

mstack.push(3);
mstack.push(5);
mstack.push(737);
//[...]
mstack.push(0);

MutantStack<int>::iterator it = mstack.begin();
MutantStack<int>::iterator ite = mstack.end();

++it;
--it;
while (it != ite)
{
    std::cout << *it << std::endl;
    ++it;
}
std::stack<int> s(mstack);
return 0;
}
```

Si vous l'exécutez une première fois avec votre MutantStack, puis une seconde fois en remplaçant la MutantStack, par exemple par `std::list`, les deux résultats devront être identiques. Bien sûr, lorsque vous testez cet exemple avec un autre container, modifiez le code avec les fonctions membres correspondantes (`push()` peut devenir `push_back()`).