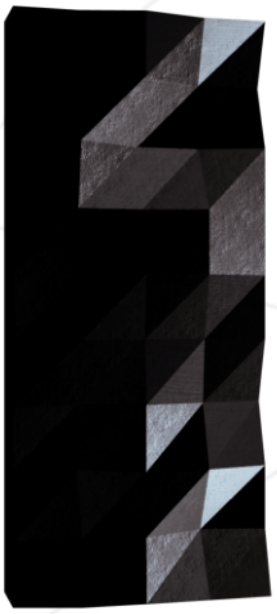

C++ - Module 01

Allocation mémoire, pointeurs sur membres, références, switch instruction

Résumé:
Ce document contient les exercices du Module 01 des C++ modules.

Version: 9.2

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

Exercice 00 : BraiiiiiiinnnzzzZ

	Exercice : 00
BraiiiiiiinnnzzzZ	
Dossier de rendu : $ex00/$	
Fichiers à rendre : `Makefile, main.cpp, Zombie.{h, hpp}, Zombie.cpp, newZombie.cpp, randomChump.cpp`	
Fonctions interdites : `Aucune`	

Pour commencer, implémentez une classe **Zombie**. Elle possède un attribut privé `name` (*nom*) de type string.

Ajoutez-lui une fonction membre `void announce( void );`. Les zombies se présentent (s'annoncent) ainsi :

```
<nom>: BraiiiiiiinnnzzzZ...
```

N'affichez pas les chevrons (< et >). Pour un zombie nommé Foo, le message serait :

```
Foo: BraiiiiiiinnnzzzZ...
```

Ensuite, implémentez les fonctions suivantes :

- `Zombie* newZombie( std::string name );`
  Crée un zombie, lui donne un nom et le retourne afin qu'il puisse être utilisé en dehors de la portée de la fonction.
- `void randomChump( std::string name );`
  Crée un zombie, lui donne un nom et le fait se présenter.

Quel est donc le but de l'exercice ? Vous devez déterminez dans quel cas il est plus judicieux d'allouer les zombies sur le tas, et dans quel autre sur la pile.

Les zombies doivent être détruits lorsque vous n'en avez plus besoin. Le destructeur doit afficher un message de débug comportant le nom du zombie.

Chapitre IV

Exercice 01 : Encore plus de cerveaux !

	Exercice : 01
Encore plus de cerveaux !	
Dossier de rendu : $ex01/$	
Fichiers à rendre : `Makefile, main.cpp, Zombie.{h, hpp}, Zombie.cpp, zombieHorde.cpp`	
Fonctions interdites : `Aucune`	

Il est maintenant temps de créer une **horde de Zombies** !

Implémentez la fonction suivante dans le fichier correspondant :

```
Zombie*     zombieHorde( int N, std::string name );
```

Cette fonction doit allouer `N` objets Zombie en une seule allocation. Ensuite, elle doit initialiser les zombies en donnant à chacun le nom passé en paramètre. Elle retourne un pointeur sur le premier zombie.

Écrivez vos propres tests afin de vous assurer que votre fonction `zombieHorde()` se comporte comme demandé. Essayez d'appeler `announce()` pour chacun des zombies.

N'oubliez pas de tous les `delete` et de vérifiez que vous n'avez pas de **fuites de mémoire**.

Chapitre V

Exercice 02 : BONJOUR ICI LE CERVEAU

	Exercice : 02
BONJOUR ICI LE CERVEAU	
Dossier de rendu : $ex02/$	
Fichiers à rendre : `Makefile, main.cpp`	
Fonctions interdites : `Aucune`	

Créez un programme comportant :

- Une variable de type string initialisée à "HI THIS IS BRAIN".
- `stringPTR` : Un pointeur sur la string.
- `stringREF` : Une référence sur la string.

Votre programme doit afficher :

- L'adresse de la string en mémoire.
- L'adresse stockée dans `stringPTR`.
- L'adresse stockée dans `stringREF`.

Puis :

- La valeur de la string.
- La valeur pointée par `stringPTR`.
- La valeur pointée par `stringREF`.

C'est tout. Il n'y a pas de pièges. Le but de cet exercice est de vous faire démystifier les références qui ne sont pas un concept totalement nouveau. Bien qu'il y ait de légères différences, il s'agit seulement d'une autre syntaxe pour représenter quelque chose que vous faites déjà : manipuler des adresses.

Chapitre VI

Exercice 03 : Violence inutile

	Exercice : 03
Violence inutile	
Dossier de rendu : $ex03/$	
Fichiers à rendre : `Makefile, main.cpp, Weapon.{h, hpp}, Weapon.cpp, HumanA.{h, hpp}, HumanA.cpp, HumanB.{h, hpp}, HumanB.cpp`	
Fonctions interdites : `Aucune`	

Implémentez une classe **Weapon** (*arme*) qui possède :

- Un attribut privé `type` de type string.
- Une fonction membre `getType()` retournant une référence constante sur `type`.
- Une fonction membre `setType()` qui attribue à `type` la nouvelle valeur passée en paramètre.

Maintenant, créez deux classes **HumanA** et **HumanB**. Toutes deux possèdent une `Weapon` et un `name`, ainsi qu'une fonction membre `attack()` affichant (sans les chevrons bien sûr) :

```
<name> attacks with their <weapon type>
```

HumanA et HumanB sont presque identiques, à l'exception de deux petits détails :

- Alors que le constructeur de HumanA prend une Weapon comme paramètre, ce n'est pas le cas de celui de HumanB.
- HumanB n'aura **pas toujours** une Weapon, tandis que HumanA en aura **forcément** une.

Si votre implémentation est correcte, l'exécution du code suivant affichera une attaque avec "crude spiked club", puis une attaque avec "some other type of club", pour les deux tests :

```
int main()
{
  {
    Weapon  club = Weapon("crude spiked club");

    HumanA bob("Bob", club);
    bob.attack();
    club.setType("some other type of club");
    bob.attack();
  }
  {
    Weapon  club = Weapon("crude spiked club");

    HumanB jim("Jim");
    jim.setWeapon(club);
    jim.attack();
    club.setType("some other type of club");
    jim.attack();
  }
  return 0;
}
```

N'oubliez pas de vérifier que vous n'avez pas de **fuites de mémoire**.

Dans quel cas pensez-vous plus judicieux d'utiliser un pointeur sur Weapon ? Et une référence sur Weapon ? Pourquoi ? Prenez le temps d'y réfléchir avant de commencer.

Chapitre VII

Exercice 04 : Sed, c'est pour les perdant(e)s

	Exercice : 04
Sed, c'est pour les perdant(e)s	
Dossier de rendu : $ex04/$	
Fichiers à rendre : `Makefile, main.cpp, *.cpp, *.{h, hpp}`	
Fonctions interdites : `std::string::replace`	

Concevez un programme prenant trois paramètres dans l'ordre suivant : un nom de fichier et deux strings, `s1` et `s2`.

Le fichier `<filename>` sera ouvert et son contenu copié dans un nouveau fichier `<filename>.replace`, où chaque occurrence de `s1` sera remplacée par `s2`.

Avoir recours aux fonctions C de manipulation de fichiers est interdit et sera considéré comme de la triche. Toutes les fonctions membres de la classe `std::string` sont autorisées sauf `replace`. Utilisez-les intelligemment !

Bien entendu, vous devez gérer les entrées inattendues et les erreurs possibles. Créez et rendez vos propres tests afin de prouver que votre programme fonctionne.

Chapitre VIII

Exercice 05 : Harl 2.0

	Exercice : 05
Harl 2.0	
Dossier de rendu : *ex*05/	
Fichiers à rendre : `Makefile, main.cpp, Harl.{h, hpp}, Harl.cpp`	
Fonctions interdites : `Aucune`	

Connaissez-vous Harl ? Tout le monde connaît Harl, voyons. Au cas où vous n'en auriez jamais entendu parler, vous trouverez ci-dessous le genre de commentaires que Harl est capable de faire. Ils sont classés par niveau de gravité :

- Niveau "**DEBUG**" : Les messages de débug contiennent des informations contextuelles. On les utilise principalement pour établir des diagnostics.
  Exemple : *"I love having extra bacon for my 7XL-double-cheese-triple-pickle-special-ketchup burger. I really do !"*

- Niveau "**INFO**" : Ces messages contiennent des informations détaillées. On les utilise pour tracer l'exécution d'un programme en production.
  Exemple : *"I cannot believe adding extra bacon costs more money. You didn't put enough bacon in my burger ! If you did, I wouldn't be asking for more !"*

- Niveau "**WARNING**" : Les messages d'avertissement indiquent un problème potentiel dans le système. Toutefois, on est en mesure de le gérer ou de l'ignorer.
  Exemple : *"I think I deserve to have some extra bacon for free. I've been coming for years whereas you started working here since last month."*

- Niveau "**ERROR**" : Ces messages indiquent qu'une erreur irrécupérable s'est produite. Il s'agit généralement d'un problème critique qui nécessite une intervention manuelle.
  Exemple : *"This is unacceptable ! I want to speak to the manager now."*

Vous allez automatiser Harl. Ce ne sera pas difficile étant donné qu'il se répète beaucoup. Vous devez créer une classe **Harl** avec les fonctions membres privées suivantes :

- `void debug( void );`
- `void info( void );`
- `void warning( void );`
- `void error( void );`

Harl a également une fonction membre publique qui, selon le niveau passé en paramètre, fait appel à l'une des quatre fonctions membres ci-dessus :

```
void    complain( std::string level );
```

Le but de l'exercice est de vous faire utiliser des **pointeurs sur fonctions membres**. Ce n'est pas qu'une suggestion. Harl doit pouvoir se plaindre sans avoir recours à une forêt de if/else if/else. Il n'y réfléchit pas à deux fois !

Créer et rendre des tests pour montrer que Harl se plaint beaucoup. Vous pouvez utiliser les exemples de commentaires indiqué au dessus dans le sujet ou choisir d'utiliser des commentaires de votre choix.

Chapitre IX

Exercice 06 : Harl filter

	Exercice : 06
Harl filter	
Dossier de rendu : $ex06/$	
Fichiers à rendre : `Makefile, main.cpp, Harl.{h, hpp}, Harl.cpp`	
Fonctions interdites : `Aucune`	

Il arrive que vous n'ayiez pas envie de prêter attention à chacun des commentaires de Harl. Implémentez un système permettant de filtrer ce qu'il dit selon les niveaux que vous acceptez d'entendre.

Concevez un programme prenant en paramètre un des quatre niveaux de gravité. Il affichera le message de ce niveau et tous ceux des niveaux supérieurs.

```
$> ./HarlFilter "WARNING"
[ WARNING ]
I think I deserve to have some extra bacon for free.
I've been coming for years whereas you started working here since last month.

[ ERROR ]
This is unacceptable, I want to speak to the manager now.

$> ./HarlFilter "I am not sure how tired I am today..."
[ Probably complaining about insignificant problems ]
```

Bien qu'il y ait plusieurs moyens de gérer Harl, l'un des plus efficaces est de le SWITCH off.

Appelez votre exécutable `HarlFilter`.
Vous devez utiliser, et peut-être découvrir, l'instruction switch dans cet exercice.

`Vous pouvez valider ce module sans l'exercice 06.`

Chapitre X

Soumission et évaluation par les pairs

Remettez votre travail dans votre dépôt `Git` comme d'habitude. Seul le travail dans votre dépôt sera évalué lors de la soutenance. N'hésitez pas à n'hésitez pas à vérifier les noms de vos dossiers et fichiers pour vous assurer qu'ils sont pour vous assurer qu'ils sont corrects.