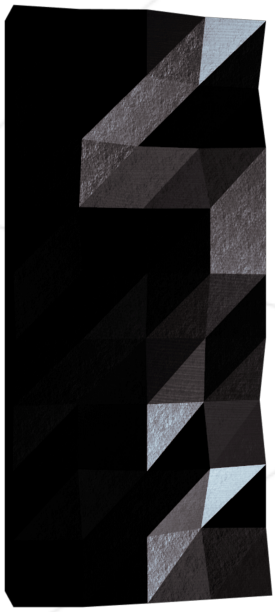

C++ - Module 03

Héritage

Résumé:
Ce document contient les exercices du Module 03 des C++ modules.

Version: 6

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 les Modules 08 et 09**. 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 09, 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 : Eeeeet... ACTION !

	Exercice : 00
Eeeeet... ACTION !	
Dossier de rendu : $ex00/$	
Fichiers à rendre : `Makefile, main.cpp, ClapTrap.{h, hpp}, ClapTrap.cpp`	
Fonctions interdites : `Aucune`	

Pour changer, vous allez implémenter une classe !

Elle s'appellera **ClapTrap** et possèdera les attributs privés suivants initialisés aux valeurs entre parenthèses :

- Name, son nom, qui sera passé en paramètre d'un constructeur
- Hit points (10), ou points de vie
- Energy points (10), ou points d'énergie
- Attack damage (0), ou dommages infligés en attaquant

Ajoutez au ClapTrap ces fonctions membres publiques afin de lui donner vie :

- `void attack(const std::string& target);`
- `void takeDamage(unsigned int amount);`
- `void beRepaired(unsigned int amount);`

Quand ClapTrap attaque, sa cible perd `<attack damage>` hit points. Quand ClapTrap se répare, il regagne `<amount>` hit points. Les actions attaquer et réparer coûtent chacune 1 point d'énergie. Bien entendu, ClapTrap ne peut exécuter aucune action s'il n'a plus de vie ou d'énergie.

Pour toutes ces fonctions membres, vous devez afficher un message descriptif. Par exemple, la fonction `attack()` affichera quelque chose dans le genre (bien sûr, sans les chevrons) :

```
ClapTrap <name> attacks <target>, causing <damage> points of damage!
```

Les constructeurs et le destructeur afficheront également un message. Ceci afin que vos pairs puissent vérifier facilement qu'ils ont effectivement été appelés.

Écrivez et rendez vos propres tests afin de démontrer que votre tout fonctionne comme demandé.

Chapitre IV

Exercice 01 : Serena, mon amour !

	Exercice : 01
Serena, mon amour !	
Dossier de rendu : *ex*01/	
Fichiers à rendre : `Fichiers de l'exercice précédent + ScavTrap.{h, hpp}, ScavTrap.cpp`	
Fonctions interdites : `Aucune`	

Plus il y a de ClapTraps, mieux c'est ! C'est pourquoi vous allez créer un robot dérivé du ClapTrap. Il s'appellera **ScavTrap** et héritera des constructeurs et du destructeur de ClapTrap. Toutefois, ses constructeurs, son destructeur et son `attack()` afficheront des messages différents. Après tout, les ClapTraps sont conscients de leur individualité.

Notez bien que vos tests devront montrer que l'enchaînement des constructeurs/destructeurs s'effectue bien dans le bon ordre. Quand on crée un ScavTrap, le programme commence par créer un ClapTrap. La destruction s'effectue dans l'ordre inverse. Pourquoi ?

ScavTrap utilisera les attributs du ClapTrap (modifiez donc ClapTrap en conséquence) et les initialisera à :

- Name, son nom, qui sera passé en paramètre d'un constructeur
- Hit points (100)
- Energy points (50)
- Attack damage (20)

ScavTrap aura également une capacité spéciale et unique :

```
void guardGate();
```

Cette fonction membre affichera un message informant que ScavTrap est entré en mode Gate keeper.
N'oubliez pas d'enrichir vos tests.

Chapitre V

Exercice 02 : Travail à la chaîne

	Exercice : 02
Travail à la chaîne	
Dossier de rendu : $ex02/$	
Fichiers à rendre : `Fichiers des exercices précedents + FragTrap.{h, hpp}, FragTrap.cpp`	
Fonctions interdites : `Aucune`	

Faire des ClapTraps doit sans doute commencer à vous taper sur les nerfs.

Maintenant, implémentez une classe **FragTrap** héritant de ClapTrap. Elle est très similaire à ScavTrap. Ses messages de construction et de destruction doivent cependant être différents. Vos tests devront montrer que l'enchaînement des constructeurs/destructeurs s'effectue bien dans le bon ordre. Quand on crée un FragTrap, le programme commence par créer un ClapTrap. La destruction s'effectue dans l'ordre inverse.

Même chose pour les attributs, mais avec des valeurs différentes :

- Name, son nom, qui sera passé en paramètre d'un constructeur
- Hit points (100)
- Energy points (100)
- Attack damage (30)

FragTrap possède aussi une capacité spéciale :

```
void highFivesGuys(void);
```

Cette fonction membre affiche une demande de high fives sur la sortie standard.

À nouveau, enrichissez vos tests.

Chapitre VI

Exercice 03 : Ok, ça devient bizarre

	Exercice : 03
Ok, ça devient bizarre	
Dossier de rendu : $ex03/$	
Fichiers à rendre : `Fichiers des exercices précedents + DiamondTrap.{h, hpp}, DiamondTrap.cpp`	
Fonctions interdites : `Aucune`	

Dans cet exercice, vous allez créer un monstre : un Claptrap moitié FragTrap moitié ScavTrap. On l'appellera le **DiamondTrap** et il héritera À LA FOIS du FragTrap et du ScavTrap. Manoeuvre risquée, s'il en est !

La classe DiamondTrap aura un attribut privé `name`. Donnez à cette variable exactement le même nom (le nom de la variable, pas le nom du robot) que celle de la classe de base ClapTrap.

Voici deux exemples pour clarifier ceci :
Si la variable du ClapTrap est `name`, appelez celle du DiamondTrap `name`.
Si la variable du ClapTrap est `_name`, appelez celle du DiamondTrap `_name`.

Ses attributs et ses fonctions membres seront pris chez l'un de ses deux parents :

- Name, son nom, qui sera passé en paramètre d'un constructeur
- `Claptrap::name` (paramètre du constructeur suivi du suffixe "`_clap_name`")
- Hit points (FragTrap)
- Energy points (ScavTrap)
- Attack damage (FragTrap)
- `attack()` (Scavtrap)

En plus des fonctions spéciales de ses deux parents, le DiamonTrap possèdera la sienne propre :

```
void whoAmI();
```

Cette fonction membre affichera à la fois son nom et le nom de son sous-objet ClapTrap.

Bien sûr, le sous-objet Claptrap du DiamondTrap ne sera créé qu'une seule et unique fois. Oui, il existe un moyen de faire cela.

Encore une fois, enrichissez vos tests.

Connaissez-vous les options de compilation -Wshadow et -Wno-shadow ?

Vous pouvez valider ce module sans l'exercice 03.