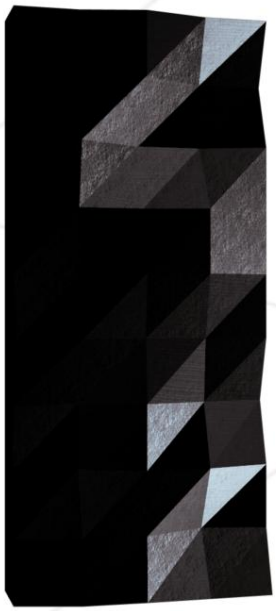

C++ - Module 03

Héritage

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

Version: 7.1

Contenu

Chapitre I

Introduction

C++ est un langage de programmation à usage général créé par Bjarne Stroustrup comme une extension du langage de programmation C, ou « C avec classes » (source : Wikipédia).

L'objectif de ces modules est de vous initier à la programmation orientée objet.
Ce sera le point de départ de votre parcours C++. De nombreux langages sont recommandés pour apprendre la programmation orientée objet. Nous avons décidé de choisir C++ car il est dérivé de votre vieil ami C.
Parce qu'il s'agit d'un langage complexe, et afin de garder les choses simples, votre code sera conforme à la norme C++98.

Nous sommes conscients que le C++ moderne est très différent sur de nombreux aspects. Donc si vous voulez devenir un développeur C++ compétent, c'est à vous d'aller plus loin après le 42 Common Core !

Chapitre II

Règles générales

Compilation

- Compilez votre code avec c++ et les indicateurs -Wall -Wextra -Werror
- Votre code devrait toujours être compilé si vous ajoutez l'indicateur -std=c++98

Conventions de formatage et de dénomination

- Les répertoires d'exercices seront nommés ainsi : ex00, ex01, ... , exn
- Nommez vos fichiers, classes, fonctions, fonctions membres et attributs comme requis dans les lignes directrices.
- Écrivez les noms de classe au format UpperCamelCase . Les fichiers contenant le code de classe doit toujours être nommé en fonction du nom de la classe. Par exemple : ClassName.hpp/ClassName.h, ClassName.cpp ou ClassName.tpp. Ainsi, si vous avez un fichier d'en-tête contenant la définition d'une classe « BrickWall » représentant un mur de briques, son nom sera BrickWall.hpp.
- Sauf indication contraire, tous les messages de sortie doivent être terminés par une nouvelle ligne caractère et affiché sur la sortie standard.
- Au revoir Norminette ! Aucun style de codage n'est imposé dans les modules C++. Vous pouvez suivre votre style préféré. Mais gardez à l'esprit qu'un code que vos pairs évaluateurs ne peuvent pas comprendre est un code qu'ils ne peuvent pas noter. Faites de votre mieux pour écrire un code propre et lisible.

Autorisé/Interdit

Vous ne codez plus en C. Il est temps de passer au C++ ! Par conséquent :

- Vous êtes autorisé à utiliser presque tout ce qui se trouve dans la bibliothèque standard. Ainsi, au lieu de vous en tenir à ce que vous connaissez déjà, il serait judicieux d'utiliser autant que possible les versions C++ des fonctions C auxquelles vous êtes habitué.
- Cependant, vous ne pouvez utiliser aucune autre bibliothèque externe. Cela signifie que les bibliothèques C++11 (et les formes dérivées) et Boost sont interdites. Les fonctions suivantes sont également interdites : *printf(), *alloc() et free(). Si vous les utilisez, votre note sera de 0 et c'est tout.

- Notez que sauf indication explicite contraire, l'espace de noms d'utilisation <ns_name> et Les mots-clés amis sont interdits. Sinon, votre note sera de -42.
- Vous êtes autorisé à utiliser le STL uniquement dans les modules 08 et 09. Cela signifie : pas de conteneurs (vecteur/liste/carte/etc.) et pas d'algorithmes (tout ce qui nécessite d'inclure l'en-tête <algorithm>) jusqu'à ce moment-là. Sinon, votre note sera de -42.

Quelques exigences de conception

- Les fuites de mémoire se produisent également en C++. Lorsque vous allouez de la mémoire (en utilisant le nouveau (mot-clé), vous devez éviter les fuites de mémoire.
- Du Module 02 au Module 09, vos cours doivent être conçus dans le style orthodoxe Forme canonique, sauf indication explicite contraire.
- Toute implémentation de fonction placée dans un fichier d'en-tête (à l'exception des modèles de fonction) signifie 0 pour l'exercice.
- Vous devez pouvoir utiliser chacun de vos en-têtes indépendamment des autres. Ainsi, ils doivent inclure toutes les dépendances dont ils ont besoin. Cependant, vous devez éviter le problème de double inclusion en ajoutant des gardes d'inclusion. Sinon, votre note sera de 0.

Lis-moi

- Vous pouvez ajouter des fichiers supplémentaires si vous en avez besoin (par exemple pour diviser votre code). Comme ces tâches ne sont pas vérifiées par un programme, n'hésitez pas à le faire à condition de rendre les fichiers obligatoires.
- Parfois, les directives d'un exercice semblent courtes, mais les exemples peuvent le montrer. exigences qui ne sont pas explicitement écrites dans les instructions.
- Lisez chaque module dans son intégralité avant de commencer ! Vraiment, faites-le.
- Par Odin, par Thor ! Utilise ton cerveau !!!

Concernant le Makefile pour les projets C++, les mêmes règles qu'en C s'appliquent (voir le chapitre Norme sur le Makefile).

Vous devrez implémenter de nombreuses classes. Cela peut paraître fastidieux, à moins que vous ne soyez capable de créer un script dans votre éditeur de texte préféré.

Vous disposez d'une certaine liberté pour réaliser les exercices.
Cependant, suivez les règles obligatoires et ne soyez pas paresseux. Vous
manque beaucoup d'informations utiles ! N'hésitez pas à lire à propos
concepts théoriques.

Chapitre III

Exercice 00 : Et... OUVERT !

	Exercice : 00
Et... OUVERT !	
Répertoire de remise : ex00/	
Fichiers à rendre : Makefile, main.cpp, ClapTrap.{h, hpp}, ClapTrap.cpp Fonctions interdites : Aucune	

Il faut d'abord implémenter une classe ! Quelle originalité !

Il s'appellera ClapTrap et aura les attributs privés suivants initialisés aux valeurs spécifiées entre parenthèses :

- Nom, qui est passé en paramètre à un constructeur
- Les points de vie (10) représentent la santé du ClapTrap
- Points d'énergie (10)
- Dégâts d'attaque (0)

Ajoutez les fonctions membres publiques suivantes pour que le ClapTrap soit plus réaliste :

- attaque void(const std::string& target);
- void takeDamage(unsigned int montant);
- void beRepaired(unsigned int montant);

Lorsque ClapTrap attaque, il fait perdre <dégâts d'attaque> points de vie à sa cible. Lorsque ClapTrap se répare, il récupère <quantité> points de vie. Attaquer et réparer lui coûtent 1 point d'énergie chacun. Bien entendu, ClapTrap ne peut rien faire s'il n'a plus de points de vie ou d'énergie. Cependant, comme ces exercices servent d'introduction, les instances de ClapTrap ne doivent pas interagir directement entre elles, et les paramètres ne doivent pas faire référence à une autre instance de ClapTrap.

Dans toutes ces fonctions membres, vous devez afficher un message pour décrire ce qui se passe. Par exemple, la fonction attack() peut afficher quelque chose comme (bien sûr, sans les crochets angulaires) :

ClapTrap <nom> attaque <cible>, causant <dégâts> points de dégâts !

Les constructeurs et le destructeur doivent également afficher un message, afin que vos pairs évaluateurs on peut facilement voir qu'ils ont été appelés.

Implémentez et remettez vos propres tests pour vous assurer que votre code fonctionne comme prévu.

Chapitre IV

Exercice 01 : Serena, mon amour !

	Exercice : 01
Serena, mon amour !	
Répertoire de remise : ex01/	
Fichiers à rendre : Fichiers de l'exercice précédent + ScavTrap.{h, hpp}, ScavTrap.cpp Fonctions interdites : Aucune	

Parce qu'on n'a jamais assez de ClapTraps, vous allez maintenant créer un robot dérivé.
Il s'appellera ScavTrap et héritera des constructeurs et du destructeur de Clap-Trap. Cependant, ses constructeurs, son destructeur et attack() afficheront des messages différents.
Après tout, les ClapTraps sont conscients de leur individualité.

Notez qu'un chaînage de construction/destruction approprié doit être montré dans vos tests.
Lorsqu'un ScavTrap est créé, le programme commence par construire un ClapTrap. La destruction se fait dans l'ordre inverse. Pourquoi ?

ScavTrap utilisera les attributs de ClapTrap (mettre à jour ClapTrap en conséquence) et doit les initialiser à :

- Nom, qui est passé en paramètre à un constructeur
- Les points de vie (100) représentent la santé du ClapTrap
- Points d'énergie (50)
- Dégâts d'attaque (20)

ScavTrap aura également sa propre capacité spéciale :

```
void gardeGate();
```

Cette fonction membre affichera un message informant que ScavTrap est désormais en mode gardien de porte.

N'oubliez pas d'ajouter plus de tests à votre programme.

Chapitre V

Exercice 02 : Travail répétitif

	Exercice : 02
Travail répétitif	
Répertoire de remise : ex02/	
Fichiers à rendre : Fichiers des exercices précédents + FragTrap.{h, hpp}, FragTrap.cpp Fonctions interdites : Aucune	

Faire des ClapTraps commence probablement à vous énerver.

Maintenant, implémentez une classe FragTrap qui hérite de ClapTrap. Elle est très similaire à ScavTrap. Cependant, ses messages de construction et de destruction doivent être différents. Un enchaînement de construction/destruction correct doit être montré dans vos tests. Lorsqu'un FragTrap est créé, le programme commence par construire un ClapTrap. La destruction s'effectue dans l'ordre inverse.

Même chose pour les attributs, mais avec des valeurs différentes cette fois-ci :

- Nom, qui est passé en paramètre à un constructeur
- Les points de vie (100) représentent la santé du ClapTrap
- Points d'énergie (100)
- Dégâts d'attaque (30)

FragTrap a également une capacité spéciale :

vide highFivesGuys(void);

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

Encore une fois, ajoutez plus de tests à votre programme.

Chapitre VI

Exercice 03 : Maintenant c'est bizarre !

	Exercice : 03
Maintenant c'est bizarre !	
Répertoire de remise : ex03/	
Fichiers à rendre : Fichiers des exercices précédents + DiamondTrap.{h, hpp}, DiamondTrap.cpp Fonctions interdites : Aucune	

Dans cet exercice, vous allez créer un monstre : un ClapTrap qui est à moitié FragTrap, à moitié ScavTrap. Il s'appellera DiamondTrap et héritera à la fois du FragTrap ET du ScavTrap. C'est très risqué !

La classe DiamondTrap aura un attribut privé name. Donnez à cet attribut exactement le même nom de variable (je ne parle pas ici du nom du robot) que celui de la classe de base ClapTrap.

Pour être plus clair, voici deux exemples.
Si la variable de ClapTrap est name, donnez le nom name à celle du DiamondTrap.
Si la variable de ClapTrap est _name, donnez le nom _name à celle du DiamondTrap.

Ses attributs et fonctions membres seront choisis parmi l'une de ses classes parentes :

- Nom, qui est passé en paramètre à un constructeur
- ClapTrap::name (paramètre du constructeur + suffixe "_clap_name")
- Points de vie (FragTrap)
- Points d'énergie (ScavTrap)
- Dégâts d'attaque (FragTrap)
- attaque() (Scavtrap)

En plus des fonctions spéciales de ses deux classes parentes, DiamondTrap aura sa propre capacité spéciale :

void qui suis-je();

Cette fonction membre affichera à la fois son nom et son nom ClapTrap.

Bien sûr, le sujet ClapTrap du DiamondTrap sera créé une fois, et une seule fois. Oui, il y a une astuce.

Encore une fois, ajoutez plus de tests à votre programme.

Connaissez-vous les indicateurs de compilateur -Wshadow et -Wno-shadow ?

Vous pouvez réussir ce module sans faire l'exercice 03.

Chapitre VII

Soumission et évaluation par les pairs

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

?????????? XXXXXXXXXX = $3$$cf36316f07f871b4f14926007c37d388