



C++ - Module 03

Héritage

Résumé :

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

Version : 7.1

Contenu

je	Introduction	2
II	Règles générales	3
III	Exercice 00 : Et... OUVERT !	6
IV	Exercice 01 : Serena, mon amour !	8
V	Exercice 02 : Travail répétitif	9
VI	Exercice 03 : Maintenant c'est bizarre !	10
VII	Soumission et évaluation par les pairs	12

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 apprentissage du C++. De nombreux langages sont recommandés pour apprendre la POO. Nous avons choisi le C++, car il est dérivé du 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 à bien des égards. Si vous souhaitez devenir un développeur C++ compétent, il vous appartient de poursuivre vos études après le Common Core 42 !

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 de cette façon : `ex00`, `ex01`, ... , `exn`
- Nommez vos fichiers, classes, fonctions, fonctions membres et attributs comme requis dans les lignes directrices.
- Écrivez les noms de classe en majuscules . Les fichiers contenant le code de classe seront `ClassName.hpp/ClassName.h`, `ClassName.cpp` ou `ClassName.tpp`. Ainsi, si vous disposez d'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, chaque message de sortie doit être terminé par une nouvelle ligne caractère et affiché sur la sortie standard.
- Adieu 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 ne comprennent pas est un code qu'ils ne peuvent pas noter. Faites de votre mieux pour écrire un code clair et lisible.

Autorisé/Interdit

Vous ne codez plus en C. Passez au C++ ! Par conséquent :

- Vous pouvez utiliser presque tout ce qui est proposé dans la bibliothèque standard. Ainsi, plutôt que 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 C++11 (et ses dérivés) et les bibliothèques Boost sont interdits. Les fonctions suivantes sont également interdites : `*printf()`, `*alloc()` et `free()`. Si vous les utilisez, votre note sera de 0, point final.

- Notez que, sauf indication contraire explicite, 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.) ni d'algorithmes (tout ce qui nécessite l'inclusion de l'en-tête <algorithm>) d'ici 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 mention 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. Ils doivent donc inclure toutes les dépendances nécessaires. 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 nécessaire (par exemple, pour fractionner votre code). Ces devoirs n'étant pas vérifiés par un programme, n'hésitez pas à le faire, à condition de fournir les fichiers obligatoires.
- Parfois, les lignes directrices d'un exercice semblent courtes, mais les exemples peuvent montrer exigences qui ne sont pas explicitement écrites dans les instructions.
- Lisez chaque module en entier avant de commencer ! Vraiment, foncez.
- Par Odin, par Thor ! Utilisez votre 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 puissiez écrire des scripts 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

manquez beaucoup d'informations utiles ! N'hésitez pas à lire

concepts théoriques.

Chapitre III

Exercice 00 : Et... OUVERT !

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

D'abord, il faut 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 vide (const std::string& target);
- void takeDamage(unsigned int amount);
- 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 coûtent 1 point d'énergie chacun. Bien sûr, ClapTrap ne peut rien faire s'il n'a plus de points de vie ni de points d'énergie. Cependant, ces exercices servant 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 décrivant l'événement. Par exemple, la fonction `attack()` peut afficher un message similaire (sans les chevrons, bien sûr) :


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 garantir 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, cpp}, 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 ClapTrap. Cependant, ses constructeurs, son destructeur et sa fonction attack() afficheront des messages différents. Après tout, les ClapTraps sont conscients de leur individualité.

Notez que le 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 s'effectue 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 guardGate();
```

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.

Implémentez maintenant une classe FragTrap héritant de ClapTrap. Elle est très similaire à ScavTrap. Cependant, ses messages de construction et de destruction doivent être différents. Un enchaînement correct de construction/destruction doit être démontré 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 :

- 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 :


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

Cette fonction membre affiche une demande positive de high fives 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, moitié FragTrap, moitié ScavTrap. Il s'appellera DiamondTrap et héritera à la fois du FragTrap et du ScavTrap. C'est vraiment risqué !

La classe DiamondTrap possède un attribut privé name. Donnez à cet attribut le même nom de variable (sans parler 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 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.



????????????? XXXXXXXXXXXX = \$3\$\$cf36316f07f871b4f14926007c37d388