



C++ - Module 03

Héritage

Résumé: Ce document contient le sujet du module 03 des modules C++ de 42.

Table des matières

I	Règles Générales	2
II	Exercice 00 : Eeeeeet.... c'est parti !	4
III	Exercice 01 : Serena, mon amour !	5
IV	Exercice 02 : Travail à la chaine	6
V	Exercice 03 : Maintenant, c'est bizarre !	7

Chapitre I


Règles Générales

- Toute fonction déclarée dans une header (sans pour les templates) ou tout header non-protégé, signifie 0 à l'exercice.
- Tout output doit être affiché sur stdout et terminé par une newline, sauf autre chose est précisé.
- Les noms de fichiers imposés doivent être suivis à la lettre, tout comme les noms de classe, les noms de fonction, et les noms de méthodes.
- Rappel : vous codez maintenant en C++, et plus en C. C'est pourquoi :
 - Les fonctions suivantes sont **INTERDITES**, et leur usage se soldera par un 0 : `*alloc`, `*printf` et `free`
 - Vous avez l'autorisation d'utiliser à peu près toute la librairie standard. CÉPENDANT, il serait intelligent d'essayer d'utiliser la version C++ de ce à quoi vous êtes habitués en C, plutôt que de vous reposer sur vos acquis. Et vous n'êtes pas autorisés à utiliser la STL jusqu'au moment où vous commencez à travailler dessus (module 08). Ça signifie pas de Vector/List/Map/etc... ou quoi que ce soit qui requiert une include `<algorithm>` jusque là.
- L'utilisation d'une fonction ou mécanique explicitement interdite sera sanctionnée par un 0
- Notez également que sauf si la consigne l'autorise, les mot-clés `using namespace` et `friend` sont interdits. Leur utilisation sera punie d'un 0.
- Les fichiers associés à une classe seront toujours nommés `ClassName.cpp` et `ClassName.hpp`, sauf si la consigne demande autre chose.
- Vous devez lire les exemples minutieusement. Ils peuvent contenir des prérequis qui ne sont pas précisés dans les consignes.
- Vous n'êtes pas autorisés à utiliser des librairies externes, incluant C++11, Boost, et tous les autres outils que votre ami super fort vous a recommandé
- Vous allez surement devoir rendre beaucoup de fichiers de classe, ce qui peut paraître répétitif jusqu'à ce que vous appreniez à scripter ça dans votre éditeur de code préféré.

- Lisez complètement chaque exercice avant de le commencer.
- Le compilateur est `clang++`
- Votre code sera compilé avec les flags `-Wall -Wextra -Werror`
- Chaque include doit pouvoir être incluse indépendamment des autres includes. Un include doit donc inclure toutes ses dépendances.
- Il n'y a pas de norme à respecter en C++. Vous pouvez utiliser le style que vous préférez. Cependant, un code illisible est un code que l'on ne peut pas noter.
- Important : vous ne serez pas noté par un programme (sauf si précisé dans le sujet). Cela signifie que vous avez un degré de liberté dans votre méthode de résolution des exercices.
- Faites attention aux contraintes, et ne soyez pas fainéant, vous pourriez manquer beaucoup de ce que les exercices ont à offrir
- Ce n'est pas un problème si vous avez des fichiers additionnels. Vous pouvez choisir de séparer votre code dans plus de fichiers que ce qui est demandé, tant qu'il n'y a pas de moulinette.
- Même si un sujet est court, cela vaut la peine de passer un peu de temps dessus afin d'être sûr que vous comprenez bien ce qui est attendu de vous, et que vous l'avez bien fait de la meilleure manière possible.

Chapitre II

Exercice 00 : Eeeeeet.... c'est parti!

	Exercice : 00
Eeeeeet.... c'est parti!	
Dossier de rendu : <i>ex00/</i>	
Fichiers à rendre : Makefile ClapTrap.cpp ClapTrap.hpp main.cpp	
Fonctions interdites : Aucune	

Ici, il faut faire une classe! Quelle originalité!

La classe s'appellera **ClapTrap**, et aura les attributs privés suivants suivants, initialisés aux valeurs spécifiées :

- Name (paramètre à passer au constructeur)
- Hit points (10)
- Energy points (10)
- Attack damage (0)

Vous allez aussi lui donner quelques fonctions publiques pour lui donner un semblant de vie :

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


Dans toutes ces fonctions, vous devez afficher quelque chose pour décrire ce qui se passe. Par exemple, la fonction **attack** peut afficher quelque chose du genre :

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

Vous devez rendre un **main** qui contiendra assez de tests pour prouver que votre code est fonctionnel.

Chapitre III

Exercice 01 : Serena, mon amour !

	Exercice : 01
Serena, mon amour !	
Dossier de rendu : <i>ex01/</i>	
Fichiers à rendre : Pareil que les exercices précédents + <code>ScavTrap.cpp</code> <code>ScavTrap.hpp</code>	
Fonctions interdites : Aucune	

Parce qu'on n'a jamais assez de Claptraps, vous allez en faire un autre.

La classe sera nommée **ScavTrap**, et héritera de **ClapTrap**, le constructeur, le destructeur, et les attaques doivent utiliser des sorties différentes. Après tout, un Claptrap doit avoir une certaine mesure d'individualité.

La classe **ScavTrap** aura ses messages de construction et de destruction. De plus, l'enchaînement correct de construction/destruction doit être présent (Lorsque vous construisez un **ScavTrap**, vous devez commencer par construire un **ClapTrap**.... La destruction se fait dans l'ordre inverse), et les tests doivent le montrer.

ScavTrap utilisera les attributs de Claptrap (Vous devez modifier Claptrap en conséquence)! Et doit les initialiser à :


- Name (Parameter of constructor)
- Hitpoints (100)
- Energy points (50)
- attack damage (20)

ScavTrap disposera également d'une nouvelle fonction spécifique : `void guardGate()`; cette fonction affichera un message sur les sorties standard pour annoncer que ScavTrap est passé en mode gardien de porte.

Étendez votre `main` pour tout tester.

Chapitre IV

Exercice 02 : Travail à la chaine

	Exercice : 02
Travail à la chaine	
Dossier de rendu : <i>ex02/</i>	
Fichiers à rendre : Pareil que les exercices précédents + <code>FragTrap.cpp</code> <code>FragTrap.hpp</code>	
Fonctions interdites : Aucune	

Faire des Claptraps commence probablement à vous taper sur les nerfs.

Vous allez maintenant créer une classe `FragTrap` qui hérite de `ClapTrap`.

- Name (Parameter of constructor)
- Hitpoints (100)
- Energy points (100)
- attack damage (30)


La classe `FragTrap` aura ses messages de construction et de destruction de construction et de destruction. De plus, un enchaînement correct de construction/destruction doit être présent (lorsque vous construisez un `FragTrap`, vous devez commencer par construire un `ClapTrap`... La destruction se fait dans l'ordre inverse), et les tests doivent le montrer.

La fonction spécifique pour `FragTrap` sera `void highFivesGuys(void)` et affichera une demande positive de high fives sur la sortie standard.

Étendez votre `main` pour tout tester.

Chapitre V

Exercice 03 : Maintenant, c'est bizarre !

	Exercice : 03
Maintenant, c'est bizarre !	
Dossier de rendu : <i>ex03/</i>	
Fichiers à rendre : Pareil que les exercices précédents + DiamondTrap.cpp DiamondTrap.hpp	
Fonctions interdites : Aucune	

Maintenant, vous allez créer un monstre en fabriquant un Claptrap qui est moitié Fragtrap, moitié Fragtrap. :

- Name (Parameter of constructor)
- Claptrap : :Name (Parameter of constructor + "_clap_name")
- Hitpoints (Fragtrap)
- Energy points (Scavtrap)
- Attack damage (Fragtrap)
- attack (Scavtrap)

Il aura les fonctions spéciales des deux.

Comme d'habitude, votre `main` sera étendu pour tester la nouvelle classe.

Bien sûr, la partie Claptrap du Diamondtrap sera créée une fois, et seulement une fois... Oui, il y a une astuce.

DiamondTrap disposera d'une nouvelle fonction `void whoAmI();` qui affichera son nom et celui de son clapTrap. Bien sûr, la partie Claptrap du Diamondtrap sera créée une fois, et seulement une fois... Oui, il y a une astuce.



Regardez au niveau des flags de compilation comme `-Wshadow` et `-Wno-shadow`!