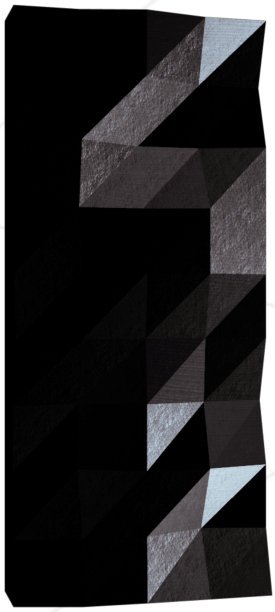

C++ - Module 03

Héritage

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

Table des matières

Chapitre I

Règles Générales

- Toute fonction déclarée dans une header (sauf 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. CEPENDANT, 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). Ca 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 paraitre répétitif jusqu'à ce que vous appreniez a scripter ca dans votre éditeur de code préferé.

- Lisez complètement chaque exercice avant de le commencer.
- Le compilateur est `clang++`
- Votre code sera compilé avec les flags `-Wall -Wextra -Werror -std=c++98`
- 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 préferez. 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 : $ex00/$	
Fichiers à rendre : `FragTrap.cpp FragTrap.hpp main.cpp`	
Fonctions interdites : `Aucune`	

Votre premier exercice sera de faire une classe robot boîtes à chaussures : FR4G-TP. La classe sera nommée FragTrap, et aura les attributs suivants, initialisés à la valeur demandée entre parenthèse :

- Hit points (100)
- Max hit points (100)
- Energy points (100)
- Max energy points (100)
- Level (1)
- Name (paramètre à passer au constructeur)
- Melee attack damage (30)
- Ranged attack damage (20)
- Armor damage reduction (5)

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

- rangedAttack(std : :string const & target)
- meleeAttack(std : :string const & target)
- takeDamage(unsigned int amount)
- beRepaired(unsigned int amount)

Dans toutes ces fonctions, vous devez afficher quelque chose qui décrit ce qu'il se passe.

Par exemple, `rangedAttack` peut afficher quelque chose du type :

```
FR4G-TP <name> attaque <target> à distance, causant <damage> points de dégâts !
```

Le constructeur et le destructeur doivent également afficher quelque chose, afin que les gens voient que celui-ci a été appelé proprement. Point bonus si les messages sont originaux ou des réferences stylées (Si vous ne savez pas ce qu'est un FR4G-TP, allez sur internet pour choisir quelques citations bien senties).

Quelques contraintes :

- Le nombre de points de vie ne peut jamais dépasser les points de vie maximum. Pareil pour les points d'energie. Si, par exemple, vous réparez trop d'HP, vous ne devez pas dépasser la valeur maximale. De la même manière, vous ne pouvez pas descendre en dessous de 0.

- Quand vous prenez des dégâts, vous devez prendre en compte la réduction de dégâts liée à l'armure.

Complétez votre classe en rajoutant une fonction `vaulthunter_dot_exe(std::string const & target)` qui lancera une attaque semi-random sur la cible. Faites en sorte qu'à chaque fois que la fonction est appellée, une attaque amusante (au possible) soit choisie parmis 5 attaques possibles.
Quelque soit la manière, tout est bon, mais rappelez-vous que plus votre méthode est élégante, mieux c'est. Cette fonction coute 25 d'énergie à lancer. Si vous n'avez pas assez d'énergie, vous devez afficher quelque chose disant que votre robot n'a plus d'énergie.

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 : $ex01/$	
Fichiers à rendre : `Pareil que les exercices précedents + ScavTrap.cpp ScavTrap.hpp`	
Fonctions interdites : `Aucune`	

Parce que nous n'avons jamais assez de Claptraps, vous allez en créer un autre qui aura un but différent : tenir la porte d'entrée de votre repère de méchant, et lancer un challenge à tous les gens qui osent s'approcher.

La classe sera nommée `ScavTrap`, et aura ces attributs :

- Hit points (100)
- Max hit points (100)
- Energy points (50)
- Max energy points (50)
- Level (1)
- Name (paramètre à passer au constructeur)
- Melee attack damage (20)
- Ranged attack damage (15)
- Armor damage reduction (3)

Ajoutez les mêmes fonctions que dans `FragTrap`, mais dans le constructeur, le destructeur, et les attaques doivent avoir d'autres outputs.
Après tout, vos Claptraps devraient avoir un peu d'individualité.

L'exception sera que le ScavTrap n'a pas de `vaulthunter_dot_exe`. À la place, elle

aura la fonction `challengeNewcomer`, qui laissera le choix au ScavTrap d'un challenge parmis un set de challenge disponibles (nous espérons qu'ils seront amusants), et devra l'imprimer sur stdout.

Étendez votre main pour tester les deux classes.

Chapitre IV

Exercice 02 : Travail à la chaine

	Exercice : 02
Travail à la chaine	
Dossier de rendu : $ex02/$	
Fichiers à rendre : `Pareil que les exercices précedents + ClapTrap.cpp ClapTrap.hpp`	
Fonctions interdites : `Aucune`	

Faire des Claptraps commence sûrement à vous taper sur les nerfs. Non ?
Afin de travailler moins, vous aller devoir d'abord travailler plus.

Créez une classe `ClapTrap`, de laquelle héritent `FragTrap` et `ScavTrap`.

Les fonctions ou variables communes de `ScavTrap` et de `FragTrap` doivent être placées dans la nouvelle classe `ClapTrap`. Les autres fonctions et variables doivent rester dans leur classe respective.

La classe `ClapTrap` aura son propre constructeur et destructeur. Des messages de construction/destruction sont attendus aussi (par exemple, quand vous construisez un `FragTrap`, vous devez d'abord créer un `ClapTrap`). La destruction doit être proprement effectuée (sens inverse).
Incluez des tests qui montreront que votre rendu marche.

Chapitre V

Exercice 03 : Maintenant c'est facile !

	Exercice : 03
Maintenant c'est facile !	
Dossier de rendu : $ex03/$	
Fichiers à rendre : `Pareil que les exercices précedents + NinjaTrap.cpp NinjaTrap.hpp`	
Fonctions interdites : `Aucune`	

En utilisant ce que vous avez fait plus tôt, faites une classe `NinjaTrap`, avec les attributs suivants :

- Hit points (60)
- Max hit points (60)
- Energy points (120)
- Max energy points (120)
- Level (1)
- Name (paramètre à passer au constructeur)
- Melee attack damage (60)
- Ranged attack damage (5)
- Armor damage reduction (0)

Son attaque spéciale sera la fonction `ninjaShoebox`. Il y aura plusieurs fonctions avec la même signature, chacune prenant la reference d'une instance de ClapTrap différente, avec une action différente (`NinjaTrap` inclus).
Quel dommage que vous ne puissiez pas faire en sorte que la fonction prenne n'importe quel ClapTrap et réagisse différement selon la classe. On verra ca demain j'imagine !
Rendez vos actions amusantes et un main pour vérifier que votre classe fonctionne correctement.

Chapitre VI

Exercice 04 : Boîtes à chaussures d'assaut ultime

	Exercice : 04
Boîtes à chaussures d'assaut ultime	
Dossier de rendu : $ex04/$	
Fichiers à rendre : `Pareil que les exercices précedents + SuperTrap.cpp SuperTrap.hpp`	
Fonctions interdites : `Aucune`	

Maintenant, créez une classe qui combinera le meilleur des deux mondes : moitié FragTrap, moitié NinjaTrap.
Elle sera nommée `SuperTrap`, et doit hériter à la fois de `FragTrap` et `NinjaTrap`.
Ses attributs et fonctions seront choisis dans ses deux classes parentes :

- Hit points (Fragtrap)
- Max hit points (Fragtrap)
- Energy points (Ninjatrap)
- Max energy points (Ninjatrap)
- Level (1)
- Name (paramètre à passer au constructeur)
- Melee attack damage (Ninjatrap)
- Ranged attack damage (Fragtrap)
- Armor damage reduction (Fragtrap)
- rangedAttack (Fragtrap)
- meleeAttack (Ninjatrap)

Elle aura les attaques spéciales des deux classes.

Étendez votre main pour tester cette nouvelle classe.
Bien entendu, la partie ClapTrap du SuperTrap ne sera créée qu'une seule fois. Il y a une astuce, cherchez un peu.