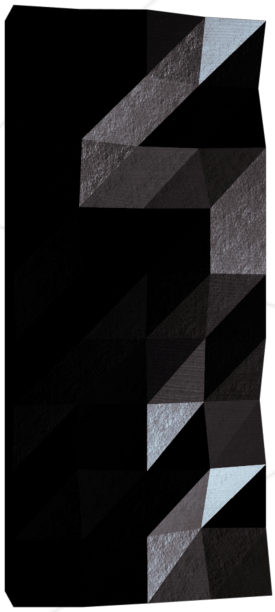

C++ - Module 04

Polymorphisme par sous-typage, classes abstraites, interfaces

Résumé:
Ce document contient les exercices du Module 04 des C++ modules.

Version: 10

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 le Module 08**. 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 08, 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 : Polymorphisme

	Exercice : 00
Polymorphisme	
Dossier de rendu : *ex*00/	
Fichiers à rendre : `Makefile, main.cpp, *.cpp, *.{h, hpp}`	
Fonctions interdites : `Aucune`	

Pour chaque exercice, veuillez fournir les tests **les plus complets** possible.
Les constructeurs et les destructeurs de chaque classe doivent afficher des messages qui leur sont propres. N'utilisez pas le même message pour toutes les classes.

Commencez par implémenter une classe simple de base **Animal**. Elle possède un attribut protégé :

- `std::string type;`

Implémentez une classe **Dog** (*chien*) qui hérite de Animal.
Implémentez une classe **Cat** (*chat*) qui hérite de Animal.

Ces deux classes dérivées doivent initialiser leur `type` en fonction de leur nom. Ainsi, le type de Dog sera "Dog", et celui de Cat sera "Cat". Le type de la classe Animal peut être laissé vide ou initialisé avec la valeur de votre choix.

Chaque animal doit être capable d'utiliser la fonction membre :
`makeSound()`
Elle affichera un son cohérent (les chats n'aboient pas).

Exécuter ce code devrait afficher les sons propres aux classes Dog et Cat, pas celui de la classe Animal.

```
int main()
{
    const Animal* meta = new Animal();
    const Animal* j = new Dog();
    const Animal* i = new Cat();

    std::cout << j->getType() << " " << std::endl;
    std::cout << i->getType() << " " << std::endl;
    i->makeSound(); //will output the cat sound!
    j->makeSound();
    meta->makeSound();
    ...

    return 0;
}
```

Afin de vous assurer d'avoir compris, implémentez une classe **WrongCat** héritant d'une classe **WrongAnimal**. Dans le code ci-dessus, si vous remplacez l'Animal et le Cat par le WrongAnimal et le WrongCat, le WrongCat devrait afficher le son du WrongAnimal.

Écrivez et rendez plus de tests que ceux donnés ci-dessus.

Chapitre IV

Exercice 01 : Je ne veux pas brûler le monde

	Exercice : 01
Je ne veux pas brûler le monde	
Dossier de rendu : *ex*01/	
Fichiers à rendre : `Fichiers de l'exercice précédent + *.cpp, *.{h, hpp}`	
Fonctions interdites : `Aucune`	

Les constructeurs et les destructeurs de chaque classe doivent afficher des messages qui leur sont propres.

Implémentez une classe **Brain** (*cerveau*) contenant un tableau de 100 `std::string` appelé `ideas` (*idées*).
Ainsi, les classes Dog et Cat auront un attribut privé `Brain*`.
À la construction, les classes Dog et Cat créeront leur Brain avec `new Brain();`
À la destruction, les classes Dog et Cat devront `delete` leur Brain.

Dans votre fonction main, créez et remplissez un tableau d'objets **Animal** dont la moitié est composée d'objets **Dog** et l'autre moitié d'objets **Cat**. À la fin de l'exécution du programme, parcourez ce tableau afin de `delete` chaque Animal. Vous devez `delete` directement les chiens et les chats en tant qu'Animal. Les destructeurs correspondants doivent être appelés dans le bon ordre.

N'oubliez pas de vérifier que vous n'avez pas de **fuites de mémoire**.

La copie d'un objet Dog ou d'un objet Cat ne doit pas être superficielle. Par conséquent, vous devez vous assurer que vos copies sont bien des copies profondes.

```cpp
int main()
{
    const Animal* j = new Dog();
    const Animal* i = new Cat();


    delete j;//should not create a leak
    delete i;
    ...

    return 0;
}
```

Écrivez et rendez plus de tests que ceux donnés ci-dessus.

Chapitre V

Exercice 02 : Classe abstraite

	Exercice : 02
Classe abstraite	
Dossier de rendu : $ex02/$	
Fichiers à rendre : `Fichiers de l'exercice précédent + *.cpp, *.{h, hpp}`	
Fonctions interdites : `Aucune`	

Créer des objets Animal ne sert pas à grand-chose au final. Ils ne font aucun bruit !

Afin d'éviter les erreurs potentielles, la classe Animal de base ne doit pas être instanciable. Modifiez-la afin que personne ne puisse l'instancier. Votre code doit fonctionner comme avant.

Si vous le souhaitez, vous pouvez ajouter la lettre A pour préfixer le nom Animal.

Chapitre VI

Exercice 03 : Interface & recap

	Exercice : 03
Interface & recap	
Dossier de rendu : *ex*03/	
Fichiers à rendre : `Makefile, main.cpp, *.cpp, *.{h, hpp}`	
Fonctions interdites : `Aucune`	

Il n'y a pas d'interfaces en C++98 (ni en C++20). Toutefois, les classes purement abstraites sont communément appelées des interfaces. Donc dans ce dernier exercice, afin de s'assurer que ce module est maîtrisé, vous implémenterez des interfaces.

Complétez la définition de la classe **AMateria** suivante et implémentez les fonctions membres nécessaires.

```
class AMateria
{
    protected:
            [...]

    public:
            AMateria(std::string const & type);
            [...]

            std::string const & getType() const; //Returns the materia type

            virtual AMateria* clone() const = 0;
            virtual void use(ICharacter& target);
};
```

Implémentez les Materias **Ice** (*glace*) et **Cure** (*soin*) sous forme de classes concrètes. Utilisez leur noms en minuscules ("ice" pour Ice, "cure" pour Cure) comme types. Bien sûr, leur fonction membre `clone()` retournera une nouvelle instance de même type (en clonant une Materia Ice, on obtient une autre Materia Ice).

Pour ce qui est de la fonction membre `use(ICharacter&)`, elle affichera :

- Ice : `"* shoots an ice bolt at <name> *"`
- Cure : `"* heals <name>'s wounds *"`

`<name>` est le nom du **Character** (*personnage*) passé en paramètre. N'affichez pas les chevrons (< et >).

Quand on assigne une Materia à une autre, copier son type n'a pas grand intérêt.

Créez la classe concrète **Character** qui implémentera l'interface suivante :

```
class ICharacter
{
    public:
            virtual ~ICharacter() {}
            virtual std::string const & getName() const = 0;
            virtual void equip(AMateria* m) = 0;
            virtual void unequip(int idx) = 0;
            virtual void use(int idx, ICharacter& target) = 0;
};
```

Le **Character** a un inventaire de 4 items, soit 4 Materias maximum. À la construction, l'inventaire est vide. Les Materias sont équipées au premier emplacement vide trouvé, soit dans l'ordre suivant : de l'emplacement 0 au 3. Dans le cas où on essaie d'ajouter une Materia à un inventaire plein, ou d'utiliser/retirer une Materia qui n'existe pas, ne faites rien (cela n'autorise pas les bugs pour autant). La fonction membre `unequip()` ne doit PAS `delete` la Materia !

Occupez-vous des Materias laissées au sol par votre personnage comme vous le sentez. Vous pouvez enregistrer l'adresse avant d'appeler unequip(), ou autre, du moment que vous n'avez pas de fuites de mémoire.

La fonction membre `use(int, ICharacter&)` utilisera la Materia de l'emplacement[`idx`], et passera la cible en paramètre à la fonction `AMateria::use`.

L'inventaire de votre personnage devra pouvoir contenir n'importe quel type d'objet AMateria.

Votre **Character** doit comporter un constructeur prenant son nom en paramètre. Toute copie (avec le constructeur par recopie ou l'opérateur d'affectation) d'un Character doit être **profonde**. Ainsi, lors d'une copie, les Materias du Character doivent être `delete` avant que les nouvelles ne les remplacent dans l'inventaire. Bien évidemment, les Materias doivent aussi être supprimées à la destruction d'un Character.

Créez la classe concrète **MateriaSource** qui implémentera l'interface suivante :

```
class IMateriaSource
{
    public:
            virtual ~IMateriaSource() {}
            virtual void learnMateria(AMateria*) = 0;
            virtual AMateria* createMateria(std::string const & type) = 0;
};
```

- `learnMateria(AMateria*)`
  Copie la Materia passée en paramètre et la stocke en mémoire afin de la cloner plus tard. Tout comme le Character, la **MateriaSource** peut contenir 4 Materias maximum. Ces dernières ne sont pas forcément uniques.
- `createMateria(std::string const &)`
  Retourne une nouvelle Materia. Celle-ci est une copie de celle apprise précédemment par la **MateriaSource** et dont le type est le même que celui passé en paramètre. Retourne 0 si le type est inconnu.

En bref, votre **MateriaSource** doit pouvoir apprendre des "modèles" de Materias afin de les recréer à volonté. Ainsi, vous serez capable de générer une nouvelle Materia à partir de son type sous forme de chaîne de caractères.

Exécuter ce code :

```
int main()
{
    IMateriaSource* src = new MateriaSource();
    src->learnMateria(new Ice());
    src->learnMateria(new Cure());

    ICharacter* me = new Character("me");

    AMateria* tmp;
    tmp = src->createMateria("ice");
    me->equip(tmp);
    tmp = src->createMateria("cure");
    me->equip(tmp);

    ICharacter* bob = new Character("bob");

    me->use(0, *bob);
    me->use(1, *bob);

    delete bob;
    delete me;
    delete src;

    return 0;
}
```

Devrait afficher :

```
$> clang++ -W -Wall -Werror *.cpp
$> ./a.out | cat -e
* shoots an ice bolt at bob *$
* heals bob's wounds *$
```

Comme d'habitude, écrivez et rendez plus de tests que ceux donnés ci-dessus.

Vous pouvez valider ce module sans l'exercice 03.