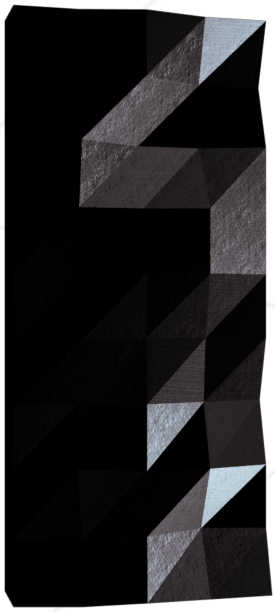

C++ - Module 04

Subtype polymorphism, abstract classes, interfaces

Summary:
This document contains the exercises of Module 04 from C++ modules.

Version: 10

Contents

Chapter 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).

The goal of these modules is to introduce you to **Object-Oriented Programming**. This will be the starting point of your C++ journey. Many languages are recommended to learn OOP. We decided to choose C++ since it's derived from your old friend C. Because this is a complex language, and in order to keep things simple, your code will comply with the C++98 standard.

We are aware modern C++ is way different in a lot of aspects. So if you want to become a proficient C++ developer, it's up to you to go further after the 42 Common Core!

Chapter II

General rules

Compiling

- Compile your code with `c++` and the flags `-Wall -Wextra -Werror`
- Your code should still compile if you add the flag `-std=c++98`

Formatting and naming conventions

- The exercise directories will be named this way: `ex00, ex01, ... , exn`
- Name your files, classes, functions, member functions and attributes as required in the guidelines.
- Write class names in **UpperCamelCase** format. Files containing class code will always be named according to the class name. For instance: `ClassName.hpp`/`ClassName.h`, `ClassName.cpp`, or `ClassName.tpp`. Then, if you have a header file containing the definition of a class "BrickWall" standing for a brick wall, its name will be `BrickWall.hpp`.
- Unless specified otherwise, every output messages must be ended by a new-line character and displayed to the standard output.
- *Goodbye Norminette!* No coding style is enforced in the C++ modules. You can follow your favorite one. But keep in mind that a code your peer-evaluators can't understand is a code they can't grade. Do your best to write a clean and readable code.

Allowed/Forbidden

You are not coding in C anymore. Time to C++! Therefore:

- You are allowed to use almost everything from the standard library. Thus, instead of sticking to what you already know, it would be smart to use as much as possible the C++-ish versions of the C functions you are used to.
- However, you can't use any other external library. It means C++11 (and derived forms) and `Boost` libraries are forbidden. The following functions are forbidden too: `*printf()`, `*alloc()` and `free()`. If you use them, your grade will be 0 and that's it.

- Note that unless explicitly stated otherwise, the `using namespace <ns_name>` and `friend` keywords are forbidden. Otherwise, your grade will be -42.
- **You are allowed to use the STL in the Module 08 and 09 only.** That means: no **Containers** (vector/list/map/and so forth) and no **Algorithms** (anything that requires to include the `<algorithm>` header) until then. Otherwise, your grade will be -42.

A few design requirements

- Memory leakage occurs in C++ too. When you allocate memory (by using the `new` keyword), you must avoid **memory leaks**.
- From Module 02 to Module 09, your classes must be designed in the **Orthodox Canonical Form, except when explicitely stated otherwise**.
- Any function implementation put in a header file (except for function templates) means 0 to the exercise.
- You should be able to use each of your headers independently from others. Thus, they must include all the dependencies they need. However, you must avoid the problem of double inclusion by adding **include guards**. Otherwise, your grade will be 0.

Read me

- You can add some additional files if you need to (i.e., to split your code). As these assignments are not verified by a program, feel free to do so as long as you turn in the mandatory files.
- Sometimes, the guidelines of an exercise look short but the examples can show requirements that are not explicitly written in the instructions.
- Read each module completely before starting! Really, do it.
- By Odin, by Thor! Use your brain!!!

You will have to implement a lot of classes. This can seem tedious, unless you're able to script your favorite text editor.

You are given a certain amount of freedom to complete the exercises. However, follow the mandatory rules and don't be lazy. You would miss a lot of useful information! Do not hesitate to read about theoretical concepts.

Chapter III

Exercise 00: Polymorphism

	Exercise : 00
Polymorphism	
Turn-in directory : *ex*00/	
Files to turn in : `Makefile, main.cpp, *.cpp, *.{h, hpp}`	
Forbidden functions : `None`	

For every exercise, you have to provide the **most complete tests** you can.
Constructors and destructors of each class must display specific messages. Don't use the same message for all classes.

Start by implementing a simple base class called **Animal**. It has one protected attribute:

- `std::string type;`

Implement a **Dog** class that inherits from Animal.
Implement a **Cat** class that inherits from Animal.

These two derived classes must set their `type` field depending on their name. Then, the Dog's type will be initialized to "Dog", and the Cat's type will be initialized to "Cat". The type of the Animal class can be left empty or set to the value of your choice.

Every animal must be able to use the member function:
`makeSound()`
It will print an appropriate sound (cats don't bark).

Running this code should print the specific sounds of the Dog and Cat classes, not the Animal's.

```
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;
}
```

To ensure you understood how it works, implement a **WrongCat** class that inherits from a **WrongAnimal** class. If you replace the Animal and the Cat by the wrong ones in the code above, the WrongCat should output the WrongAnimal sound.

Implement and turn in more tests than the ones given above.

Chapter IV

Exercise 01: I don't want to set the world on fire

	Exercise : 01
I don't want to set the world on fire	
Turn-in directory : $ex01/$	
Files to turn in : `Files from previous exercise + *.cpp, *.{h, hpp}`	
Forbidden functions : `None`	

Constructors and destructors of each class must display specific messages.

Implement a **Brain** class. It contains an array of 100 `std::string` called `ideas`.
This way, Dog and Cat will have a private `Brain*` attribute.
Upon construction, Dog and Cat will create their Brain using `new Brain();`
Upon destruction, Dog and Cat will `delete` their Brain.

In your main function, create and fill an array of **Animal** objects. Half of it will be **Dog** objects and the other half will be **Cat** objects. At the end of your program execution, loop over this array and delete every Animal. You must delete directly dogs and cats as Animals. The appropriate destructors must be called in the expected order.

Don't forget to check for **memory leaks**.

A copy of a Dog or a Cat mustn't be shallow. Thus, you have to test that your copies are deep copies!

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


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

    return 0;
}
```

Implement and turn in more tests than the ones given above.

Chapter V

Exercise 02: Abstract class

	Exercise : 02
Abstract class	
Turn-in directory : $ex02/$	
Files to turn in : `Files from previous exercise + *.cpp, *.{h, hpp}`	
Forbidden functions : `None`	

Creating Animal objects doesn't make sense after all. It's true, they make no sound!

To avoid any possible mistakes, the default Animal class should not be instantiable. Fix the Animal class so nobody can instantiate it. Everything should work as before.

If you want to, you can update the class name by adding a A prefix to Animal.

Chapter VI

Exercise 03: Interface & recap

	Exercise : 03
Interface & recap	
Turn-in directory : *ex*03/	
Files to turn in : `Makefile, main.cpp, *.cpp, *.{h, hpp}`	
Forbidden functions : `None`	

Interfaces don't exist in C++98 (not even in C++20). However, pure abstract classes are commonly called interfaces. Thus, in this last exercise, let's try to implement interfaces in order to make sure you got this module.

Complete the definition of the following **AMateria** class and implement the necessary member functions.

```
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);
};
```

Implement the Materias concrete classes **Ice** and **Cure**. Use their name in lowercase ("ice" for Ice, "cure" for Cure) to set their types. Of course, their member function `clone()` will return a new instance of the same type (i.e., if you clone an Ice Materia, you will get a new Ice Materia).

The `use(ICharacter&)` member function will display:

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

`<name>` is the name of the Character passed as parameter. Don't print the angle brackets (< and >).

While assigning a Materia to another, copying the type doesn't make sense.

Write the concrete class **Character** which will implement the following interface:

```
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;
};
```

The **Character** possesses an inventory of 4 slots, which means 4 Materias at most. The inventory is empty at construction. They equip the Materias in the first empty slot they find. This means, in this order: from slot 0 to slot 3. In case they try to add a Materia to a full inventory, or use/unequip an unexisting Materia, don't do anything (but still, bugs are forbidden). The `unequip()` member function must NOT delete the Materia!

Handle the Materias your character left on the floor as you like. Save the addresses before calling unequip(), or anything else, but don't forget that you have to avoid memory leaks.

The `use(int, ICharacter&)` member function will have to use the Materia at the slot[`idx`], and pass the target parameter to the `AMateria::use` function.

Your character's inventory will be able to support any type of AMateria.

Your **Character** must have a constructor taking its name as a parameter. Any copy (using copy constructor or copy assignment operator) of a Character must be **deep**. During copy, the Materias of a Character must be deleted before the new ones are added to their inventory. Of course, the Materias must be deleted when a Character is destroyed.

Write the concrete class **MateriaSource** which will implement the following interface:

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

- `learnMateria(AMateria*)`
  Copies the Materia passed as a parameter and store it in memory so it can be cloned later. Like the Character, the **MateriaSource** can know at most 4 Materias. They are not necessarily unique.
- `createMateria(std::string const &)`
  Returns a new Materia. The latter is a copy of the Materia previously learned by the **MateriaSource** whose type equals the one passed as parameter. Returns 0 if the type is unknown.

In a nutshell, your **MateriaSource** must be able to learn "templates" of Materias to create them when needed. Then, you will be able to generate a new Materia using just a string that identifies its type.

Running this 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;
}
```

Should output:

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

As usual, implement and turn in more tests than the ones given above.

You can pass this module without doing exercise 03.