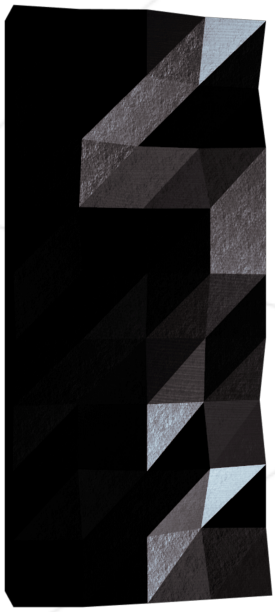

# C++ - Module 04

## Subtype polymorphism, abstract classes, interfaces

*Summary: This document contains the subject for Module 04 of the C++ modules.*

*Version: 10*

# Contents

# Chapter I

# General rules

For the C++ modules you will use and learn C++98 only. The goal is for you to learn the basic of an object oriented programming language. We know modern C++ is way different in a lot of aspects so if you want to become a proficient C++, developer you will need modern standard C++ later on. This will be the starting point of your C++ journey it's up to you to go further after the 42 Common Core!

- Any function implemented in a header (except in the case of templates), and any unprotected header means 0 to the exercise.
- Every output goes to the standard output and will be ended by a newline, unless specified otherwise.
- The imposed filenames must be followed to the letter, as well as class names, function names, and method names.
- Remember: You are coding in `C++` now, not in `C` anymore. Therefore:
    - The following functions are FORBIDDEN, and their use will be punished by a `0`, no questions asked: `*alloc`, `*printf` and `free`.
    - You are allowed to use everything in the standard library. HOWEVER, it would be smart to try and use the C++-ish versions of the functions you are used to in C, instead of just keeping to what you know, this is a new language after all. And NO, you are not allowed to use the STL until you are supposed to (that is, until module 08). That means no vectors/lists/maps/etc... or anything that requires an include <algorithm> until then.
- Actually, the use of any explicitly forbidden function or mechanic will be punished by a `0`, no questions asked.
- Also note that unless otherwise stated, the `C++` keywords `"using namespace"` and `"friend"` are forbidden. Their use will be punished by a `-42`, no questions asked.
- Files associated with a class will always be `ClassName.hpp` and `ClassName.cpp`, unless specified otherwise.
- Turn-in directories are `ex00/`, `ex01/`, ..., `exn/`.

- You must read the examples thoroughly. They can contain requirements that are not obvious in the exercise's description.
- Since you are allowed to use the `C++` tools you learned about since the beginning, you are not allowed to use any external library. And before you ask, that also means no `C++11` and derivatives, nor `Boost` or anything else.
- You may be required to turn in an important number of classes. This can seem tedious unless you're able to script your favorite text editor.
- Read each exercise FULLY before starting it! Do it.
- The compiler to use is `c++`.
- Your code has to be compiled with the following flags : `-Wall -Wextra -Werror`.
- Each of your includes must be able to be included independently from others. Includes must contain every other includes they are depending on.
- In case you're wondering, no coding style is enforced during in `C++`. You can use any style you like, no restrictions. But remember that a code your peer-evaluator can't read is a code they can't grade.
- Important stuff now: You will NOT be graded by a program, unless explicitly stated in the subject. Therefore, you are afforded a certain amount of freedom in how you choose to do the exercises. However, be mindful of the constraints of each exercise, and DO NOT be lazy, you would miss a LOT of what they have to offer
- It's not a problem to have some extraneous files in what you turn in, you may choose to separate your code in more files than what's asked of you. Feel free, as long as the result is not graded by a program.
- Even if the subject of an exercise is short, it's worth spending some time on it to be sure you understand what's expected of you, and that you did it in the best possible way.
- By Odin, by Thor! Use your brain!!!

# Chapter II

# Exercise 00: Polymorphism

|  | Exercise : 00 |
|---|---|
| Polymorphism | |
| Turn-in directory : *ex*00/ | |
| Files to turn in : `Makefile, main.cpp, Every other files you need` | |
| Forbidden functions : `None` | |

For every exercise, your main must test everything.
Constructors and destructors of each class must have specifics output.
Create a simple and complete base class Animal.
The animal class got one protected attribute:

- std::string type;

Create a class Dog that inherits from Animal.
Create a class Cat that inherits from Animal.
(for the animal class the type can be left empty or put at any value).
Every class should put their name in the Type field for example:
The Dog class type must be initialized as "Dog".
Every animal must be able to use the method makeSound().
This method will display an appropriate message on the standard outputs based on the class.

```
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();
```

```
    ...
}
```

This should output the specific makeSound of the Dog and cat class, not the animal one.
To be sure you will create a WrongCat class that inherits a WrongAnimal class that will output the WrongAnimal makeSound() when test under the same conditions.

# Chapter III

# 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 : `Makefile, main.cpp, plus the needed files for your tests` | |
| Forbidden functions : `None` | |

You'll reuse the Ex00 classes.
Create one class called Brain.
Brain will contain an array of 100 std::string called ideas
Now, Dog and cat will have a private Brain* attribute.

`Not every animal got a brain!`

Upon construction Dog and Cat will initialize their Brain* with a new Brain();
Upon destruction Dog and Cat will delete their Brain.

Your main will create and fill an Array of Animal half of it will be Dog and the other half will be Cat.

Before exit, your main will loop over this array and delete every Animal.
You must delete directly Cat and Dog as an Animal.

A copy of a Cat or Dog must be "deep".
Your test should show that copies are deep!

Constructors and destructors of each class must have specifics output.
The appropriate destructors must be called.

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


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

}
```

# Chapter IV

# Exercise 02: abstract class

|  | Exercise : 02 |
|---|---|
| This code is unclean. PURIFY IT! | |
| Turn-in directory : $ex02/$ | |
| Files to turn in : `Makefile, main.cpp, plus any needed files` | |
| Forbidden functions : `None` | |

A general Animal doesn't make sense after all.
For example, it makes no sound!
To avoid any future mistakes, the default animal class should not be instantiable.
Fix the Animal class so nobody instantiates it by mistakes.

The rest should work as before.

# Chapter V

# Exercise 03: Interface & recap

|  | Exercise : 03 |
|---|---|
| Interface & recap | |
| Turn-in directory : *ex*03/ | |
| Files to turn in : `Makefile, main.cpp, plus any needed files` | |
| Forbidden functions : `None` | |

There's no interface in C++98 (not even in C++20) but it's common to call pure abstract class Interface. So for this last exercise let's try interfaces and recap everything!

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);
};
```

Create the concrete Materias `Ice` and `Cure` . Their type will be their name in lowercase ("ice" for Ice, etc...).

Their `clone()` method will, of course, return a new instance of the real Materia's type.

Regarding the `use(ICharacter&)` method, it'll display:

- Ice: "* shoots an ice bolt at NAME *"
- Cure: "* heals NAME's wounds *"

(Of course, replace NAME by the name of the `Character` given as parameter.)

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

Create the `Character` class, 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 Materia at most, empty at the start. He'll equip the Materia in slots 0 to 3, in this order.

In case we try to equip a Materia in a full inventory, or use/unequip a nonexistent Materia, don't do a thing.

The `unequip` method must NOT delete Materia!

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

Of course, you'll have to be able to support ANY AMateria in a Character's inventory.

Your `Character` must have a constructor taking its name as a parameter. Copy or assignation of a `Character` must be deep, of course. The old `Materia` of a `Character` must be deleted. Same upon the destruction of a `Character` .

Create the `MateriaSource` class, which will have to implement the following interface:

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

`learnMateria` must copy the Materia passed as a parameter, and store it in memory to be cloned later. Much in the same way as for `Character`, the Source can know at most 4 Materia, which are not necessarily unique.

`createMateria(std::string const &)` will return a new Materia, which will be a copy of the Materia (previously learned by the Source) which type equals the parameter. Returns 0 if the type is unknown.

In a nutshell, your Source must be able to learn "templates" of Materia and re-create them on demand. You'll then be able to create a Materia without knowing its "real" type, just a string identifying it.

As usual, here's a test main that you'll have to improve on:

```
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;
}
```

Output:

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