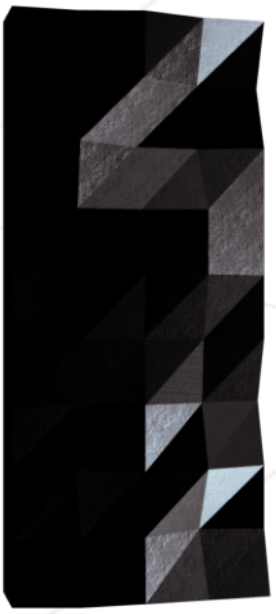

C++ - Module 03

Inheritance

Summary:
This document contains the exercises of Module 03 from C++ modules.

Version: 7.1

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, but 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 that modern C++ is very different in many 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 message must end with a newline character and be 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 code your peer evaluators can't understand is code they can't grade. Do your best to write 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 the C++-ish versions of the C functions you are used to as much as possible.
- 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 only in Modules 08 and 09.** That means: no **Containers** (vector/list/map, and so forth) and no **Algorithms** (anything that requires including 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 explicitly 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!!!

Regarding the Makefile for C++ projects, the same rules as in C apply (see the Norm chapter about the Makefile).

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: Aaaaand... OPEN!

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

First, you have to implement a class! How original!

It will be called **ClapTrap** and will have the following private attributes initialized to the values specified in brackets:

- Name, which is passed as a parameter to the constructor
- Hit points (10), representing the health of the ClapTrap
- Energy points (10)
- Attack damage (0)

Add the following public member functions so that the ClapTrap behaves more realistically:

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

When ClapTrap attacks, it causes its target to lose `<attack damage>` hit points. When ClapTrap repairs itself, it regains `<amount>` hit points. Attacking and repairing each cost 1 energy point. Of course, ClapTrap can't do anything if it has no hit points or energy points left. However, since these exercises serve as an introduction, the ClapTrap instances should not interact directly with one another, and the parameters will not refer to another instance of ClapTrap.

In all of these member functions, you need to print a message to describe what happens. For example, the `attack()` function may display something like (of course, without the angle brackets):

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

The constructors and destructor must also display a message, so your peer-evaluators can easily see that they have been called.

Implement and turn in your own tests to ensure your code works as expected.

Chapter IV

Exercise 01: Serena, my love!

	Exercise : 01
Serena, my love!	
Turn-in directory : *ex*01/	
Files to turn in : `Files from the previous exercise + ScavTrap.{h, hpp}, ScavTrap.cpp`	
Forbidden functions : `None`	

Because you can never have enough ClapTraps, you will now create a derived robot. It will be named **ScavTrap** and will inherit the constructors and destructor from ClapTrap. However, its constructors, destructor, and `attack()` will print different messages. After all, ClapTraps are aware of their individuality.

Note that proper construction/destruction chaining must be shown in your tests. When a ScavTrap is created, the program starts by constructing a ClapTrap. Destruction occurs in reverse order. Why?

ScavTrap will use the attributes of ClapTrap (update ClapTrap accordingly) and must initialize them to:

- Name, which is passed as a parameter to the constructor
- Hit points (100), representing the health of the ClapTrap
- Energy points (50)
- Attack damage (20)

ScavTrap will also have its own special ability:

```
void guardGate();
```

This member function will display a message indicating that ScavTrap is now in Gate keeper mode.

Don't forget to add more tests to your program.

Chapter V

Exercise 02: Repetitive work

	Exercise : 02
Repetitive work	
Turn-in directory : *ex*02/	
Files to turn in : `Files from previous exercises + FragTrap.{h, hpp}, FragTrap.cpp`	
Forbidden functions : `None`	

Making ClapTraps is probably starting to get on your nerves.

Now, implement a **FragTrap** class that inherits from ClapTrap. It is very similar to ScavTrap. However, its construction and destruction messages must be different. Proper construction/destruction chaining must be shown in your tests. When a FragTrap is created, the program starts by constructing a ClapTrap. Destruction occurs in reverse order.

Same goes for the attributes, but with different values this time:

- Name, which is passed as a parameter to the constructor
- Hit points (100), representing the health of the ClapTrap
- Energy points (100)
- Attack damage (30)

FragTrap has a special ability too:

```
void highFivesGuys(void);
```

This member function displays a positive high-fives request on the standard output.

Again, add more tests to your program.

Chapter VI

Exercise 03: Now it's weird!

	Exercise : 03
Now it's weird!	
Turn-in directory : $ex03/$	
Files to turn in : `Files from previous exercises + DiamondTrap.{h, hpp}, DiamondTrap.cpp`	
Forbidden functions : `None`	

In this exercise, you will create a monster: a ClapTrap that's half FragTrap, half ScavTrap. It will be named **DiamondTrap**, and it will inherit from both FragTrap AND ScavTrap. This is so risky!

The DiamondTrap class will have a private attribute named `name`. This attribute must have exactly the same variable name as in the ClapTrap base class (without referring to the robot's name).

To be clearer, here are two examples:
If ClapTrap's variable is `name`, give the DiamondTrap's variable the name `name`.
If ClapTrap's variable is `_name`, give the DiamondTrap's variable the name `_name`.

Its attributes and member functions will be inherited from its parent classes:

- Name, which is passed as a parameter to a constructor
- `ClapTrap::name` (parameter of the constructor + `"_clap_name"` suffix)
- Hit points (FragTrap)
- Energy points (ScavTrap)
- Attack damage (FragTrap)
- `attack()` (ScavTrap)

In addition to the special functions from both parent classes, DiamondTrap will have its own special ability:

```
void whoAmI();
```

This member function will display both its name and its ClapTrap name.

Of course, the ClapTrap instance of DiamondTrap will be created once, and only once. Yes, there's a trick.

Again, add more tests to your program.

Do you know the -Wshadow and -Wno-shadow compiler flags?

You can pass this module without completing exercise 03.

Chapter VII

Submission and Peer-Evaluation

Submit your assignment in your `Git` repository as usual. Only the work within your repository will be evaluated during the defense. Don't hesitate to double-check the names of your folders and files to ensure they are correct.

```
???????????  XXXXXXXXXX = $3$$cf36316f07f871b4f14926007c37d388
```