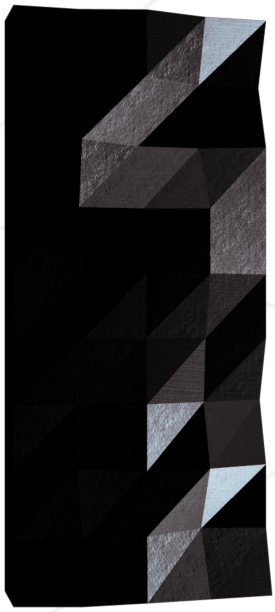

C++ - Module 01

Memory allocation, pointers to members, references, switch statement

Summary:
This document contains the exercises of Module 01 from C++ modules.

Version: 8

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 Module 08 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 08, 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: BraiiiiiiinnnzzzZ

	Exercise : 00
BraiiiiiiinnnzzzZ	
Turn-in directory : $ex00/$	
Files to turn in : `Makefile, main.cpp, Zombie.cpp, Zombie.{h, hpp}, newZombie.cpp, randomChump.cpp`	
Forbidden functions : `None`	

First, implement a **Zombie** class. It has a string private attribute `name`.

Add a member function `void announce( void );` to the Zombie class. Zombies announce themselves as follows:

```
<name>: BraiiiiiiinnnzzzZ...
```

Don't print the angle brackets (< and >). For a zombie named Foo, the message would be:

```
Foo: BraiiiiiiinnnzzzZ...
```

Then, implement the two following functions:

- `Zombie* newZombie( std::string name );`
  It creates a zombie, name it, and return it so you can use it outside of the function scope.
- `void randomChump( std::string name );`
  It creates a zombie, name it, and the zombie announces itself.

Now, what is the actual point of the exercise? You have to determine in what case it's better to allocate the zombies on the stack or heap.

Zombies must be destroyed when you don't need them anymore. The destructor must print a message with the name of the zombie for debugging purposes.

Chapter IV

Exercise 01: Moar brainz!

	Exercise : 01
Moar brainz!	
Turn-in directory : $ex01/$	
Files to turn in : `Makefile, main.cpp, Zombie.cpp, Zombie.{h, hpp}, ZombieHorde.cpp`	
Forbidden functions : `None`	

Time to create a **horde of Zombies**!

Implement the following function in the appropriate file:

```
Zombie*    zombieHorde( int N, std::string name );
```

It must allocate `N` Zombie objects in a single allocation. Then, it has to initialize the zombies, giving each one of them the name passed as parameter. The function returns a pointer to the first zombie.

Implement your own tests to ensure your `zombieHorde()` function works as expected. Try to call `announce()` for each one of the zombies.

Don't forget to `delete` all the zombies and check for **memory leaks**.

Chapter V

Exercise 02: HI THIS IS BRAIN

	Exercise : 02
HI THIS IS BRAIN	
Turn-in directory : $ex02/$	
Files to turn in : `Makefile, main.cpp`	
Forbidden functions : `None`	

Write a program that contains:

- A string variable initialized to "HI THIS IS BRAIN".
- `stringPTR`: A pointer to the string.
- `stringREF`: A reference to the string.

Your program has to print:

- The memory address of the string variable.
- The memory address held by `stringPTR`.
- The memory address held by `stringREF`.

And then:

- The value of the string variable.
- The value pointed to by `stringPTR`.
- The value pointed to by `stringREF`.

That's all, no tricks. The goal of this exercise is to demystify references which can seem completely new. Although there are some little differences, this is another syntax for something you already do: address manipulation.

Chapter VI

Exercise 03: Unnecessary violence

	Exercise : 03
Unnecessary violence	
Turn-in directory : $ex03/$	
Files to turn in : `Makefile, main.cpp, Weapon.cpp, Weapon.{h, hpp}, HumanA.cpp, HumanA.{h, hpp}, HumanB.cpp, HumanB.{h, hpp}`	
Forbidden functions : `None`	

Implement a Weapon class that has:

- A private attribute type, which is a string
- A `getType()` member function that returns a const reference to type.
- A `setType()` member function that set type using the new one passed as parameter.

Now, create two classes: **HumanA** and **HumanB**. They both have a `Weapon` and a `name`. They also have a member function `attack()` that displays (of course, without the angle brackets):

```
<name> attacks with their <weapon type>
```

HumanA and HumanB are almost the same except for these two tiny details:

- While HumanA takes the Weapon in its constructor, HumanB doesn't.
- HumanB may **not always** have a Weapon, whereas HumanA will **always** be armed.

If your implementation is correct, executing the following code will print an attack with "crude spiked club" then a second attack with "some other type of club" for both test cases:

```
int main()
{
  {
    Weapon  club = Weapon("crude spiked club");

    HumanA bob("Bob", club);
    bob.attack();
    club.setType("some other type of club");
    bob.attack();
  }
  {
    Weapon  club = Weapon("crude spiked club");

    HumanB jim("Jim");
    jim.setWeapon(club);
    jim.attack();
    club.setType("some other type of club");
    jim.attack();
  }

  return 0;
}
```

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

In which case do you think it would be best to use a pointer to Weapon? And a reference to Weapon? Why? Think about it before starting this exercise.

Chapter VII

Exercise 04: Sed is for losers

	Exercise : 04
Sed is for losers	
Turn-in directory : $ex04/$	
Files to turn in : `Makefile, main.cpp, *.cpp, *.{h, hpp}`	
Forbidden functions : `std::string::replace`	

Create a program that takes three parameters in the following order: a filename and two strings, `s1` and `s2`.

It will open the file `<filename>` and copies its content into a new file `<filename>.replace`, replacing every occurrence of `s1` with `s2`.

Using C file manipulation functions is forbidden and will be considered cheating. All the member functions of the class `std::string` are allowed, except `replace`. Use them wisely!

Of course, handle unexpected inputs and errors. You have to create and turn in your own tests to ensure your program works as expected.

Chapter VIII

Exercise 05: Karen 2.0

	Exercise : 05
Karen 2.0	
Turn-in directory : *ex*05/	
Files to turn in : `Makefile, main.cpp, Karen.{h, hpp}, Karen.cpp`	
Forbidden functions : `None`	

Do you know Karen? We all do, do we? In case you don't, find below the kind of comments Karen makes. They are classified by levels:

- "**DEBUG**" level: Debug messages contain contextual information. They are mostly used for problem diagnosis.
  Example: *"I love having extra bacon for my 7XL-double-cheese-triple-pickle-special-ketchup burger. I really do!"*

- "**INFO**" level: These messages contain extensive information. They are helpful for tracing program execution in a production environment.
  Example: "I cannot believe adding extra bacon costs more money. You didn't put enough bacon in my burger! If you did, I wouldn't be asking for more!"

- "**WARNING**" level: Warning messages indicate a potential issue in the system. However, it can be handled or ignored.
  Example: "I think I deserve to have some extra bacon for free. I've been coming for years whereas you started working here since last month."

- "**ERROR**" level: These messages indicate an unrecoverable error has occurred. This is usually a critical issue that requires manual intervention.
  Example: "This is unacceptable! I want to speak to the manager now."

You are going to automate Karen. It won't be difficult since she always says the same things. You have to create a **Karen** class with the following private member functions:

- `void debug( void );`
- `void info( void );`
- `void warning( void );`
- `void error( void );`

Karen also has a public member function that calls the four member functions above depending on the level passed as parameter:

```
void    complain( std::string level );
```

The goal of this exercise is to use **pointers to member functions**. This is not a suggestion. Karen has to complain without using a forest of if/else if/else. She doesn't think twice!

Create and turn in tests to show that Karen complains a lot. You can use the example comments.

Chapter IX

Exercise 06: Karen-filter

	Exercise : 06
Karen-filter	
Turn-in directory : $ex06/$	
Files to turn in : `Makefile, main.cpp, Karen.{h, hpp}, Karen.cpp`	
Forbidden functions : `None`	

Sometimes you don't want to pay attention to everything Karen says. Implement a system to filter what Karen says depending on the log levels you want to listen to.

Create a program that takes as parameter one of the four levels. It will display all messages from this level and above. For example:

```
$> ./karenFilter "WARNING"
[ WARNING ]
I think I deserve to have some extra bacon for free.
I've been coming for years whereas you started working here since last month.

[ ERROR ]
This is unacceptable, I want to speak to the manager now.

$> ./karenFilter "I am not sure how tired I am today..."
[ Probably complaining about insignificant problems ]
```

Although there are several ways to deal with Karen, one of the most effective is to SWITCH her off.

Give the name `karenFilter` to your executable.