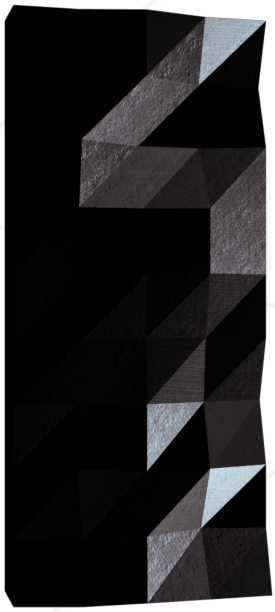

C++ - Module 01

Memory allocation, references, pointers to members, switch

Summary: This document contains the subject for Module 01 of the C++ modules.

Contents

Chapter I

General rules

- 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 basically 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 actually 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. If something seems ambiguous, you don't understand `C++` enough.
- 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 derivates, nor `Boost` or anything your awesomely skilled friend told you `C++` can't exist without.

- 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! Really, do it.
- The compiler to use is `clang++`.
- 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 contains every other includes they are depending on, obviously.
- 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 she or he can't grade.
- Important stuff now : You will NOT be graded by a program, unless explictly 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 absolutely 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: BraiiiiiiinnnzzzZ

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

First, make a `Zombie` class. The zombies have a private name and are able to `announce` themselves like:

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

Yes, `announce( void )` is a member function. Also, add a debugging message in the destructor including the name of the `Zombie`.

After this, write a function that will create a `Zombie`, name it, and return it to be used somewhere else in your code. The prototype of the function is:

```
Zombie* newZombie( std::string name );
```

You will also have to write another function that will create a `Zombie`, and make it `announce` itself. The prototype of the function is:

```
void randomChump( std::string name );
```

Now the actual point of the exercise: your `Zombies` must be destroyed at the appropriate time (when they are not needed anymore). They must be allocated on the stack or the heap depending on its use: sometimes it's appropriate to have them on the stack, at other times the heap may a better choice.

Chapter III

Exercise 01: Moar brainz!

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

Re-using the `Zombie` class, now we are going to create an horde of zombies!

Write a function that takes an integer N. When called, it allocates N `Zombie` objects. It must allocate all the N `Zombie` objects in a single allocation. Then, it should initialize each `Zombie` by giving to each one a name. Last, it should return the pointer to the first `Zombie`. The function is prototyped as follows:

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

Submit a main to test that your function `zombieHorde` works as intended. You may want to do so by calling `announce()` on each one of the `Zombies`. Do not forget to delete ALL the `Zombies` when you don't need them anymore.

Chapter IV

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`	

Make a program in which you will create a string containing `"HI THIS IS BRAIN"`. Create a `stringPTR` which is a pointer to the string; and a `stringREF` which is a reference to the string.

Now, display the address in memory of the string. Next, display the address of the string by using `stringPTR` and `stringREF`.

Last, display the string using the pointer, and finally display it using the reference.

That's all, no tricks. The goal of this exercise is to make you demystify references. It isn't something completely new, it is just another syntax for something that you already know: addresses. Even there are some tiny-little-minuscule details.

Chapter V

Exercise 03: Unnecessary violence

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

Make a `Weapon` class, that has a `type` string, and a `getType` method that returns a const reference to this string. It also has a `setType`, of course!

Now, create two classes, `HumanA` and `HumanB`, that both have a `Weapon`, a name, and an `attack()` function that displays:

```
NAME attacks with his WEAPON_TYPE
```

`HumanA` and `HumanB` are almost-almost the same; there are only two tiny-little-minuscule details:

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

Make it so the following code produces attacks with "crude spiked club" THEN "some other type of club", in 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();
	}
}
```

In which case is it appropriate to store the `Weapon` as a pointer? And as a reference? Why?
These are the questions you should ask yourself before turning in this exercise.

Chapter VI

Exercise 04: Sed is for losers

	Exercise : 04
Sed is for losers	
Turn-in directory : $ex04/$	
Files to turn in : `Makefile, main.cpp, and whatever else you need`	
Forbidden functions : `None`	

Make a program called `replace` that takes a filename and two strings, let's call them s1 and s2, which are NOT empty.

It will open the file, and write its contents to FILENAME.replace, after replacing every occurrence of s1 with s2.

All the member functions of the class std::string are allowed, except replace. Use them wisely!

Of course, you will handle errors as best you can. Do not use the C file manipulation functions, because that would be cheating, and cheating's bad, m'kay?

You will turn in some test files to show your program works.

Chapter VII

Exercise 05: Karen 2.0

	Exercise : 05
Karen 2.0	
Turn-in directory : $ex05/$	
Files to turn in : `Makefile, main.cpp, Karen.hpp, and Karen.cpp`	
Forbidden functions : `None`	

Do you know Karen? We all do, no? In case you don't, here are the kind of comments that Karen does:

- "DEBUG" level: Messages in this level contain extensive contextual information. They are mostly used for problem diagnosis. Example: "I love to get extra bacon for my 7XL-double-cheese-triple-pickle-special-ketchup burger. I just love it!"
- "INFO" level: These messages contain some contextual information to help trace execution in a production environment. Example: "I cannot believe adding extra bacon cost more money. You don't put enough! If you did I would not have to ask for it!"
- "WARNING" level: A warning message indicates a potential problem in the system. The system is able to handle the problem by itself or to proceed with this problem anyway. Example: "I think I deserve to have some extra bacon for free. I've been coming here for years and you just started working here last month."
- "ERROR" level: An error message indicates a serious problem in the system. The problem is usually non-recoverable and requires manual intervention. Example: "This is unacceptable, I want to speak to the manager now."

We are going to automate Karen, she says always the same things. You have to create a class named `Karen` which will contain the following private member functions:

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

Karen also has to have a public function that calls to the private functions depending on the level that is passed as parameter. The prototype of the function is:

```
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/elseif/else, she doesn't hesitate or think twice!

Submit a main to test that Karen complains a lot. It is okay if you want to use the complains we give as example.

Chapter VIII

Exercise 06: Karen-filter

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

We are going to implement a system to filter if what Karen says is important or not, because sometimes we don't want to pay attention to everything Karen says.

You have to write the program `karenFilter` that will receive as a parameter the log level you want to listen to and display all the info that is at this level or above it. For example:

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

There are many ways to filter Karen, but one of the best ones is to SWITCH her off ;)
You must use, and maybe discover, the switch statement in this exercise.