Foreword

In order to do this lab you must have access to the school's virtual machine (see wiki on Moodle).

In this exercise, you will cover the following topics:

- Writing a first C++ Class ;
- Separate compilation with g++ ;
- Create a more complex makefile;
- Visualizing memory addresses ***[BONUS]***.

Advice

- Use `man` an especially `man 3` for the development pages ;
- The website `https://en.cppreference.com` is your most valuable friend for the Teaching Unit;

Part I
Writing a new C++ Class

1 Writing the header file

A header file is a file with extension .h which contains a C++ function and/or class declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that comes with your compiler.

You request to use a header file in your program by including it with the C++ preprocessing directive `#include`, like you have seen inclusion of `math.h` header file, which comes along with the math library.

Instruction

Create a `tree.h` file that contains the definition of a simple class "Tree" with no attribute and only two public methods: `void draw()` and `void info()`.

Question 1

- What do you need to include in the `main.cpp` file in order to use the `Tree` Class.
- What can happen if you include the same file multiple times (for example in multiple `.cpp` files)?

To prevent the last phenomenon to occur, one can enclose the entire real contents of the file in a conditional, like this :

```
#ifndef HEADER_FILE
#define HEADER_FILE

the entire header file

#endif
```

This construct is commonly known as a wrapper `#ifndef`. When the header is included again, the conditional will be false, because `HEADER_FILE` is defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

Warning

Using `#define` defines a macro that the preprocessor will replace in all the content of the file. That means that if you use a name that is used in your code for example `#define Tree` instead of `#define TREE` the preprocessor will replace your class definition and the compilation will fail. This is why it is recommended to use upper case names sometimes preceded by an underscore[a].

[a] You can also find `#pragma once` in some files but it is not always supported by the compiler

2 Writing the source code & compiling the class

Now that the header file is complete, we need to write the actual code for the previously defined tree methods (`draw()` and `info()`).

Instruction

Create the following `tree.cpp` file to provide the source code for the `draw()` method.

```
#include "tree.h"

#include <iostream>

void Tree::draw(){
  std::cout << "Drawing tree" << std::endl;
}
```

Compile the class using `g++ -c tree.cpp -Wall`.

Question 2

- Why is there a `Tree::` prefix for the method ?
- Is is possible to directly build an executable program from the class ?

3 Using the Tree class

To be able to use our `Tree` class in an executable program we need to create an additional file containing the `main` function of our program.

Instruction

Write the following program :

```
#include "tree.h"

int main(int argc, char* argv[]) {
  Tree t;
  t.draw();
  return 0;
}
```

Compile the class using `g++ -c main.cpp -Wall`.

Question 3

- Explain why there is no `#include <iostream>` instruction at the start of the file?
- How can we link the `main.o` and `tree.o` so that the executable code for the `draw` method is known ?

We can see that compiling and linking our program requires several calls to the g++ program. In particular, each time the `tree.h` or `tree.cpp` changes, we need to compile the Tree class and go through the linking process again. That's where the `make` utility from the first lab comes in handy.

Instruction

In fact, we can automate the whole process by writing the corresponding `makefile`.

```
main: main.o tree.o
  g++ -o main main.o tree.o -Wall
main.o: main.cpp
  g++ -c main.cpp -Wall
tree.o: tree.cpp tree.h
  g++ -c tree.cpp -Wall
clean:
  rm -rf *.o main
```

Try to change something in either the `tree.h` or `tree.cpp` and see what it does for the compilation.

4 Writing constructors and destructors

At the moment we rely on the compiler to provide both a constructor and a destructor for our `Tree` class. However, we might want to add attributes next and they would need to be initialized. Depending on the compiler we can never know to which value these attribute will be initialized which can lead to unforeseen consequences.

Instruction

Modify both the `tree.h` and `tree.cpp` to write a default constructor & destructor for the `Tree` class that prints that the object has been created and destroyed.

Question 4

- When are these constructors and destructors called ?
- If you try to create two trees, in what order are their constructor & destructor called ?

Part II
BONUS - Exploring the memory

5 Create object on the stack or the heap

In the previous code for the `main` program, the Tree object "`t`" is declared on the Stack. It means that it is automatically destroyed when exiting the main function [1]. However, C++ allows you to create objects in another part of the computer memory called the Heap. This can be useful when you need to dynamically create objects, for example when you don't know the size of an array in advance. To do so, you must use the C++ keyword `new`.

Instruction

Modify the `main` function so that the "`t`" object is created on the Heap.

Question 5

- When allocating memory on the heap, you have to handle the life-cycle of the object yourself, what is the proper way to manually destroy such an object ?
- What are the other changes you have to make to the code in order to call the `draw()` method on object created on the heap ?

6 Creating the `info()` method

While the `draw()` method previously defined is quite straightforward (it draws a tree) the `info()` one will be used to display information about the underlying object.

Instruction

We want to print the following message on the console when calling the `info()` method.

```
$ ./main
The tree is planted at address 0x7fffc02f50ef
```

- Add the source code for the `info()` method in `tree.cpp` that prints the actual memory address of the object [a],
- Create multiple objects on the stack and the heap and call the `info()` on them[b].

[a] Use the pointer `this`
[b] Don't forget to delete the objects allocated with new

[1] aka when exiting the program

Question 6

- How do you explain the differences between the addresses of objects on the Stack vs on the Heap?

7 Adding attributes

Instruction

Add two public attributes : a double height attribute and a boolean evergreen attribute to the `Tree` Class and change the default constructor so that any new `Tree` has `height` set to 10.0 and `everGreen` set to false. Rebuild your previous program where multiple objects were created.

Question 6

- What is the effect on the actual addresses of the objects?
- How can it be explained [a]?

[a] Check the sizeof() function

Appendices : Useful commands

```
$man COMMAND              #display the manual page for the given COMMAND
$man 3 FUNCTION           #display the developer manual for the given FUNCTION
$g++                      #GNU project C and C++ compiler
$make                     #GNU make utility to maintain groups of programs
$ldd                      #print shared object dependencies
$strace                   #trace system calls and signals
$strings                  #print the sequences of printable characters in files
$gdb                      #The GNU Debugger
```