Foreword

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

In this exercise, you will cover the following topics:

- Using encapsulation;
- Overloading functions;
- Writing a C++ class that inherits from another one;
- Using polymorphism;
- Creating abstract classes and/or interfaces ***[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
Modifying the Tree class

1 Encapsulation

At the moment the two attributes **height** and **evergreen** are declared public which allow any function or program to modify these attributes[1].

Instruction

Change the visibility of the two attributes **height** and **evergreen** to private and write the accessors and mutators :

```
//Accessors and mutators
void setHeight(double h);
double getHeight();
void setEvergreen(bool p);
bool isEvergreen();
```

In your main program, call the relevant methods to get or change the attributes of Tree objects.

Question 1

- What happens if you try to change a private attribute inside the main program without using the proper methods?
- What keyword do you need to add to the function declaration to let the compiler check that the Tree attributes are not modified by the **getHeight()** and **isEvergreen()** methods?
- Check if the compiler does its job?

2 Overloading the constructor

The default Tree constructor is always creating trees of size 10.0 and that are not evergreen. The accessors and mutators allow you to modify these attributes later but not at the creation. However, it would be interesting to be able to create trees of all sizes without having to modify their height later.

C++ allows you to overload methods in an object (even operators as we will see later) and that is also true for constructors.

Instruction

Declare a new constructor in the Tree class that has a double parameter h and write the code that allow the creation of a Tree of size h (but still not evergreen).

In the main program create two trees one by default and one of size 5.5 and print out their respective sizes.

[1] You can try to change the **height** of any Tree in the previous program

3 Default constructor

In the previous section we have seen that one can write a default constructor and as many overloaded ones as necessary. However, if you have not declared any constructor in your class, the compiler provides you with a default one that is basically in charge of potentially initializing the allocated memory to a default value (but it may not do it).

Instruction

For this section we will first start by commenting every constructors that have been previously declared (both the default and the one that sets the height).

Try to run a simple program that declares a `Tree` and call `getHeight()` and `isEvergreen()`.

Question 2

- Does the program compile and run ?
- What are the values of the two attributes `height` and `evergreen` ?
- What is the danger of using the default constructor provided by the compiler ?

Instruction

Now remove the comments for the specific height constructor `Tree(double h)` but not for the default one.

Run your simple program that declares a `Tree` and call `getHeight()` and `isEvergreen()` again.

Question 3

- Does the program compile and run ?

Warning

Remove all the comments for the next parts.

Part II

Inherit from the Tree class

4 Writing the class

Instruction

Create a `pine.h` file that contains the definition of a class `Pine` that inherits from the previously defined class `Tree`.

Question 4

- What do you need to include in the `pine.h` file in order to inherit from the `Tree` Class?

Instruction

Create a `pine.cpp` file that contains the source code for the `Pine` class. Here we want to write a specific constuctor and destructor and specific `draw()` and `info()` methods for the `Pine` class.

Modify the `main.cpp` in order to declare a `Pine` and call the `draw()` and `info()` methods. Update the `makefile` in order to compile and link the `Pine` class and run your program. Parts of the expected output are shown below.

```
$ ./main

...

Planting a Pine                               <------ call to the constructor
Drawing a Pine                                <------ call to the draw() method
The Pine is planted at address 0x7fffeeff7430  <------ call to the info() method
Cutting down a Pine                           <------ call to the destructor

...
```

Question 5

In the full output :

- Why is there a call to the `Tree` constructor ?
- Why is there a call to the `Tree` destructor ?
- What happens if you modify both the `Tree` and `Pine` constructors to display the objects addresses ?

Instruction

Create a `Pine` on the stack and on the heap.

Question 6

- Are constructor and destructor called ?
- If not, have you called `delete` for the `Pine` created on the heap ?

5 Accessing the parent class attributes

Instruction

Try to access or modify the attributes of the underlying class `Tree` from a `Pine`, for example try adding the following line to the `draw()` method in the `Pine` class.

```cpp
void Pine::draw(){
  std::cout << "Drawing a Pine of height" << height << std::endl;
}
```

6 Creating a default constructor

One problem with our current implementation is that by default the newly created pines are not evergreen as they depend on the creation of a default tree. We have seen above that you can modify the attribute later. However, just as in the previous part it would be interesting to create pines that are evergreen.

One solution can be to call the `setEvergreen()` method inside the `Pine` constructor. However, it is not really a good practice as this attribute is supposed to be private in the `Tree` class.

Instruction

The correct way to proceed would be to write a specific `Tree` constructor that takes a boolean parameter and sets `evergreen` to that value and call it in the default constructor of the `Pine` class with `true`.

Part III

Polymorphism

Imagine that you want to draw a complete forest but you don't know the specific type of each tree in it. In C++, you can do that by using pointers or references to a `Tree` through polymorphism.

Instruction

In your `main.cpp` file write down the following code :

```cpp
#include "tree.h"
#include "pine.h"

int main(int argc, char* argv[]) {
  //create a Tree on the stack
  Tree t;
  Pine p;

  Tree *tp = &p; //declare a pointer on the pine

  tp->draw();
  tp->info();

  return 0;
}
```

Question 7

- Why is the `Tree draw()` called instead of the `Pine`'s one ?
- How can it be solved ?

Instruction

Declare both the `draw()` and `info()` as `virtual` in the `Tree` and `Pine` class and in your `main.cpp` file write down the following code :

```cpp
#include "tree.h"
#include "pine.h"

int main(int argc, char* argv[]) {
  //create a Tree on the heap
  Tree *tp = new Pine; //allocate a Pine

  tp->draw();
  tp->info();

  delete tp;

  return 0;
}
```

Question 8

- What happens at the destruction of the objects[a]?
- How can it be solved ?

[a] The compiler should warn you if the `-Wall` argument is set

Part IV

BONUS - Create a Tree Interface

7 Make the `Tree` class abstract

In the previous code for the `Tree` class, the three methods : `draw()`,`info()` and ~`Tree()` are declared virtual which means they can be overridden in the derived classes. If we follow the idea of drawing a complete forest, our program will always draw specific trees and not generic one. To do that it can be interesting to make the `Tree` class abstract.

Instruction

Modify the `tree.h` so that both `draw()` and `info()` are declared pure virtual.

Try to create a simple `Tree` in your main program.

Question 9

- Can you compile your code ?
- Remove the simple `Tree` and use a pointer on a `Tree` to create a `Pine`[a]. Does it compile/work?

[a] See instruction for question 6

8 Create a `Tree` interface

In the previous section the `Tree` class has been made abstract but we can take thing a little further by making it into an interface. That is, a contract that derived classes must follow.

Question 10

- In an interface can you have both virtual and non virtual methods ?
- If we give a default implementation (that does nothing) for the virtual destructor in the `tree.h`, do you need to compile `tree.cpp` ?

Instruction

Create the following interface and use it in your main program.

```cpp
#ifndef _TREE
#define _TREE

class Tree{
  virtual ~Tree(){};
  //Public methods
  virtual void draw() =0;
  virtual void info() =0;
};

#endif
```

Modify the `makefile` accordingly.

Warning

In the next labs we will use this simple interface as we add other types of trees and make our program more and more complex.

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
```