CMPT 225 - Lab 1: C++ Class

Recommendation

We suggest you read the entire Lab 1 before jumping into solving the problem outlined in this lab.

Why? Because then you will have discovered the way this lab is structured and that there is a section that explains how to write classes in C++ right after the section in which you are given a problem to solve.

Lab Preparation

Steps:

- Within your **sfuhome/cmpt-225** directory, create a **Lab1** folder.
- **cd** into **sfuhome/cmpt-225/Lab1**.
- Download into this current folder any files you are asked to download as part of this lab. If you are asked to download a **zip** file, make sure you extract its content first. Never edit files that are still part of a **zip** file and have not yet been extracted from it.
- Make sure you save all the files you create as part of Lab 1 in this folder: **sfuhome/cmpt-225/Lab1**.

General Requirements

For all programs you shall write this semester, here is a list of general requirements which must always be satisfied, unless there is a requirement that explicitly states otherwise in the lab.

- If you need to use a literal value (i.e., “hard code” a literal value), you must declare it as a constant.
- You must descriptively name ...
  - Your program (implementation file, i.e., the file with a .cpp extension), executable file and classes
  - Your constants, local variables, parameters and member attributes (x, y, a, b are usually not descriptive variables names)
  - Your functions and member methods of a class
- Your user interface must clearly describe the expected input to the user (which input you are expecting the use to enter, its format and range of value, if possible) and the resulting output.
- You must comment your programs.
- You must indent the statements in your programs.
- You must include a header comment block at the top of each of your source files listing the name of the source file, a description of the program found in the source file, the name of the author of the program (this should be you) and the date of creation of the source file/program.
- When creating classes, you must put the class member attributes (also called "data members") and member method declarations (also called "headers" or "prototypes") in a header file (.h extension) and the implementation of the class member methods (and functions) in an implementation (or source) file (.cpp extension).
- You must give your source files (e.g., class implementation files and test driver files) a .cpp extension and your header files a .h extension (when writing C++ code).
- You must not use **goto** statements.
- You must construct proper conditions for your conditional and iteration statements, unlike: **while (1)**.

Circles - Problem Statement and Requirements

Write a complete C++ ADT class to represent a circle in two-dimensional Cartesian space. You locate this circle in this space by specifying the coordinates (x,y) of its centre.

Your class must consist of a header file (**Circle.h**) and an implementation file (**Circle.cpp**) as described below. The Circle class must have the following (private) member attributes and (public) operations (i.e., member methods):

Attributes

The Circle class must have the following (private) member attributes:

- x coordinate (an int), can be negative
- y coordinate (an int), can be negative
- radius (a double), must be greater than 0.0

Operations

The Circle class must have the following (public) operations (i.e., member methods):

- A default constructor that creates a circle of radius 10 and centred at coordinates (0,0)
- A constructor with parameters for x, y, and radius
- Do you need a destructor?
- int getX() - returns the circle's centre x coordinate
- int getY() - returns the circle's centre y coordinate
- double getRadius() - returns the circle's radius
- void move(int horiz, int vert) - moves the circle to the new given location (horiz,vert) (therefore changing its x and y member attributes to the given horiz and vert parameter values, respectively)
- void setRadius(double r) - changes the circle's radius to r, or to 10.0 if r is invalid
- double computeArea() - computes and returns the circle's area
- void displayCircle() - displays the circle's member attributes like this: **x = 0, y = 11, radius = 0.2**
- bool intersect(Circle c) - returns true if c intersects the calling circle

Writing a C++ ADT Class

.h and .cpp Files

C++ ADT classes are made up of a header file and an implementation file. Both files should have the same name except that the header file has a .h extension while the implementation (code) file has a .cpp extension. The header file contains the class definition, i.e., the class name, the name (and type) of the member attributes and the declaration (header or prototype) for each of the class member methods. The .cpp file consists of the implementation of these class member methods.

Finally, both files must contain the same class documentation: a header comment block at the top of each file as well as description, precondition (if any) and postcondition (if any) for each method.

Public or Private?

Class member attributes and methods should be categorised as being either **private** or **public**. Private member attributes and methods can only be accessed from within the class, whereas public member attributes and methods can be accessed from outside the class, i.e., from client code. There are a couple of good general design principles to follow when deciding whether or not to make a member attribute or member method public:

- Only make something public if it needs to be public, i.e., accessible by client code (code outside this class).
- Designed classes as ADT classes: make all member attributes private (behind the wall) and provide public getter and setter member methods for each member attributes, if appropriate. This way, you can ensure that any class invariants are maintained (e.g. such as ensuring that a circle's radius is a positive number) by adding validation code in the implementation of the setter member methods, as well as in other member methods, if appropriate.

Constructors and Destructors

Every class requires a constructor to create new objects of that class. A class will often have multiple constructors that build new objects in slightly different ways: a default constructor and some parameterised constructors. A constructor is a method that has exactly the same name as the class and has no return type. It is responsible for constructing (instantiating) an object of the class and setting the initial values of the member attributes of this object.

C++ classes require destructors. A destructor is responsible for de-allocating any dynamically allocated memory that an object uses. If you don't write a destructor for a class a default one is created for you. It is OK to rely on the default destructor if your class does not use any dynamically allocated memory. However, if your class does, then you must implement a destructor.

Syntax

It's easier to show the syntax by presenting a simple example. Here is a class that models a rectangle. It has the following member attributes and methods:

- height
- width
- Rectangle - constructor to create a new rectangle with the given height and width
- getHeight - returns the height of the rectangle
- getWidth - returns the width of the rectangle
- setHeight - sets the height of the rectangle
- setWidth - sets the width of the rectangle
- computeArea - computes and returns the area of the rectangle
- displayRectangle - prints the height and width of the rectangle

Here is its header file:

```
/*
* Rectangle.h
*
* Description: This class models a rectangle ...
* Class Invariant: ???
*
* Author:
* Creation date:
*/

class Rectangle {

private:

   // Everything that follows is private and cannot be "seen" and
   // directly accessed from outside the class (from client code).
   // To access these private member attributes, client code needs to use the getters and setters.

   unsigned int width;
   unsigned int height;

   // Some classes have private member methods like our Temperature class. This one doesn't!

public:

   // Everything that follows is public and can be "seen" and directly accessed
   // from outside the class (from client code).

   // Default constructor
   // Note that the default constructor has no parameters.
   Rectangle();

   // Parameterized constructor
   // Description: Create a new rectangle with the given values.
   Rectangle(unsigned int w, unsigned int h);

   // Getters return information about the rectangle.
   // Note the const at the end of each of the methods.
   // This guarantees that a method will not alter the member attributes as it executes.
   unsigned int getWidth() const;
   unsigned int getHeight() const;

   // Setters change the values of the class' member attributes.
   void setWidth(unsigned int w);
   void setHeight(unsigned int h);

   // Description: Compute and return the area of "this" rectangle.
   unsigned int computeArea() const;

   // Description: Prints the rectangle's height and width.
   void displayRectangle() const;

}; // Note the ";" - don't forget it!
// End of Rectangle.h
```

And here is the implementation file:

```
/*
* Rectangle.cpp
*
* Description: This class models a rectangle ...
* Class Invariant: ???
*
* Author:
* Creation date:
*/

#include <iostream>       // As you need to print data
#include "Rectangle.h"    // The header file for the class - you need this!

using std::cout;          // The implementation of this class uses the object cout
using std::endl;          // and the object endl

// Now, read each of the method implementations.
// The Rectangle:: preceding each method indicates that the method belongs to the Rectangle class.
// If it is omitted, the compiler will attempt to create a separate function
// (not belonging to the class). This is not what you want.

// Default constructor
// Note: This part of the constructor's header " : width(1), height(1)"
// is called the "initialization list".
// Check what our textbook and online resources have to say about it!
Rectangle::Rectangle() : width(1), height(1) { }

// Parameterized constructor
// Description: Create a new rectangle with the given values.
Rectangle::Rectangle(unsigned int w, unsigned int h)
{
   if (w > 0)
      width = w; // Don't need {}s if there is only one line in the body.
   else
      width = 1;

   if (h > 0){
      height = h; // But you can use them if you want.
   }else{
      height = 1;
   }
}

// Getters return information about the rectangle.
// Note the const at the end of each of the methods.
// This guarantees that a method will not alter the member attributes as it executes.
unsigned int Rectangle::getWidth() const
{
   return width;
}

int Rectangle::getHeight() const
{
   return height;
}

// Setters change the values of the class' member attributes.
void Rectangle::setWidth(unsigned int w)
{
   if (w > 0)
      width = w;
   else
      width = 1;
   return;
}

void Rectangle::setHeight(unsigned int h)
{
   if (h > 0)
      height = h;
   else
      height = 1;
   return;
}

// Description: Compute and return the area of "this" rectangle.
unsigned int Rectangle::computeArea() const
{
   return width * height;
}

// Description: Prints the rectangle's height and width.
void Rectangle::displayRectangle() const
{
   cout << "width = " << getWidth();
   cout << ", height = " << getHeight() << endl;
   return;
}

// End of the implementation file
```

Note

There is (a lot) more to C++ classes than described above but this is enough to get you started.

Compiling and Running

Go ahead and implement your Circle class.

If you actually want to use and test your Circle class, you will need a main function (contained in its own file), which you will compile with your Circle class files. This program is often known as a **test driver program** or simply **test driver**.

As an example, for testing your Circle class we have provided two test driver programs. Download this zip file and unzip it in your Lab1 directory. This zip file contains **testcirc1.cpp** and **testcirc2.cpp**, the two test driver programs. Each one creates some Circle object and tests their member methods: **testcirc1.cpp** tests the getters, setters and the **computeArea** method while **testcirc2.cpp** tests the **displayCircle** method and the **intersect** method.

Open these two test driver files in the text editor and have a good look at their content. Can you figure out the test cases (test data) these two test drivers are using? Can you figure out the results you are expecting (expected results) when you execute these two test drivers?

Compiling the Test Driver program

Compile these using the following **g++** commands:

```
g++ -o testcirc1 Circle.cpp testcirc1.cpp
g++ -o testcirc2 Circle.cpp testcirc2.cpp
```

There is also a test script (again **test.py**), which you can run as follows:

```
your_username@hostname:~$ ./test.py
```

If you have correctly built the executables **testcirc1** and **testcirc2**, you will see:

```
Running test 1... passed
Running test 2... passed
Passed 2 of 2 tests.
```

on the computer monitor screen.

If you are unable to run **test.py** because of permission problem, enter the following at the command line:

```
$ chmod 755 test.py
```

This command changes the permission of **test.py** such that it should now be **executable**. Try running **test.py** again. If you are curious about this command and how permissions work in Linux, feel free to ask the instructor, the TA or the Internet. :)

If you are not seeing the above results on your screen, you will need to go over your program and figure out the "bug", fix it and recompile your code.

Now, clean up your directory in order to prepare us for the next section of this lab. You achieve this by typing:

```
your_username@hostname:~$ make clean
```

This command will remove any **.o** (object) files as well as the executables **testcirc1** and **testcirc2** you may have already created. We shall come back to the topic of **makefiles** later on in this lab.

Object files

If you wanted to compile your Circle class on its own, you could do so. But wait, there's no main function?! You can compile your Circle class using the **-c** option in **g++**:

```
your_username@hostname:~$ g++ -c Circle.cpp
your_username@hostname:~$ ls C*
Circle.cpp    Circle.h      Circle.o
```

But wait, what does it mean to compile a class with no main function? Why would you want to do such a thing? What is that **Circle.o** file that was just created?

Circle.o is called an **object file**. One way to think about object files is that they contain machine instructions for the member methods in a class. However, no client code is using these member methods yet, since there is no main function.

Why would you want to do this? (1) To make sure some part of your code compiles. (2) To reuse the compilation -- don't need to recompile **Circle.cpp** if you only change **testcirc1.cpp** and/or **testcirc2.cpp**. This may seem trivial in a small example like this one, but makes a big difference when you are dealing with more complex software with many classes.

For example, you can compile **testcirc1.cpp** using the **Circle.o** object file:

```
your_username@hostname:~$ g++ -o test testcirc1.cpp Circle.o
```

You can then execute this new executable as follows:

```
your_username@hostname:~$ ./test
```

Technically, this process involves **linking**, in which the compiled machine instructions in **Circle.o** are linked into the new executable called **test**, rather than being recompiled.

Perhaps you're thinking "Hum... This sounds rather troublesome! First you need to run a **g++** command on **Circle.cpp**, then you need to type in another long **g++** command to compile **testcirc.cpp**."

Would it not be great if there were some way to avoid all this?

Of course, there is.

makefiles

This is where **makefiles** come in. **makefiles** are like recipes that specify how all the components of an application (software) should be compiled and linked. And, by convention, the file is called **makefile** (or **Makefile**).

Here is a short introductory tutorial (video) on **makefile** Check it out!

We provided a **makefile** in **Lab1-Files.zip** which you downloaded. Open this makefile in the text editor and have a good look at its content.

Type the following once again:

```
your_username@hostname:~$ make clean
```

This command will remove the .o file created above.

Now try:

```
your_username@hostname:~$ make testcirc1
```

If you have code that properly compiles, you should see:

```
g++ -Wall -c testcirc1.cpp
g++ -Wall -c Circle.cpp
g++ -Wall testcirc1.o Circle.o -o testcirc1
```

Magic! The recipe in the makefile specifies how to build **testcirc1**. The executable **testcirc1** should now exist.

Now try making a small change to **testcirc1.cpp** (anything at all -- just add an empty line in the file). If you execute **make testcirc1** again, you should see:

```
your_username@hostname:~$ make testcirc1
g++ -Wall -c testcirc1.cpp
g++ -Wall testcirc1.o Circle.o -o testcirc1
```

Note that **Circle.cpp** was **not** recompiled. It did not need to be since no changes were made to it.

makefiles also have a default, called **all**. If you run **make all** or just **make**, it will build everything specified for **all**. In this case, this is **testcirc1** and **testcirc2**. Try the following:

```
your_username@hostname:~$ make clean
rm -f testcirc1 testcirc2 *.o
your_username@hostname:~$ make
g++ -Wall -c testcirc1.cpp
g++ -Wall -c Circle.cpp
g++ -Wall testcirc1.o Circle.o -o testcirc1
g++ -Wall -c testcirc2.cpp
g++ -Wall testcirc2.o Circle.o -o testcirc2
```

You will now have both of your test driver programs, and can run **test.py** on them.

makefiles allow you to easily recompile your code after making changes, while debugging, for example. So, from now on, you don't need to keep typing in **g++** commands, you can simply run **make**.

Of course, there is a lot more to **makefiles** than what has been stated above. Feel free to see what the Internet says about **makefiles**.

In the labs of this course, we will often provide you with a **makefile**, if it is needed.

It is interesting to note that integrated development environments (IDEs) such as MS Visual Studio and Eclipse are essentially executing **makefiles** when you build your projects. As a computing scientist and software developer, it's important for you to become familiar with **makefiles**, i.e., it is important to know what is actually happening "under the hood" of an IDE.

Enjoy!