Recitation 10 - Classes and Command Line
CSCI-1300
Instructor: Dr. David Knox
Fall 2017

Review on Classes: Header and Source files

We learned in Recitation 9 how useful classes can be. By creating this new "type object" we can extend the semantics of our program to more complex structures. However, introducing complex structures can make things more confusing and disorganized. This is why, as we progress with the complexity of a program, we create multiple files: *header and source files*.

Header file

Header files will have *".h"* in their filename extensions. In a header file, we **define** one or more of the complex structures (*class*) we want to develop. In a *class*, we define member functions and member attributes. These functions and attributes are the building blocks of the *class*.

.h file format

```
#include
….
class <ClassName> {

      public:
            <constructor> // can be overloaded
            <deconstructor>
            <member function>
            <member attribute>
            ...
      private:
            <member attribute>
            <member function>
            ….
};
```

Source file

Source files are recognizable by the *".cpp"* extension. In a *source file* we usually **implement** the complex structures (*class*) defined in the header file. Since we are splitting the development of actual code for the class into a **definition** (header file) and an **implementation** (source file), we need to link the two somehow.

In the source file we will include the header file that defines the class so that the source file is "aware" of where we can retrieve the definition of the class. We must define the class definition in every source that wants to use our user defined data type (our class). When implementing each member function, our source files must tell the compiler that these functions are actually the methods defined in our class definition. You must specify the class name before each of the method names using the following syntax:

```
<data type> <class name> :: <method name> ( <param> )
{
      ...
}
```

In general, a source (.cpp) file will look like this:

```
#include "ClassName.h"

<ClassName> :: <ClassName> ( <params> ) {
      // constructor
      . . .
}
<returnType> <ClassName> :: <member func 1> ( <params> ) {
      . . .
}
<returnType> <ClassName> :: <member func 2> ( <params> ) {
      . . .
}
```

We see that any member function that belongs to a specific class is prepended by:

- **returnType**, which tells us the return type of the member function
- **Classname**, which tells us where we find the member function in the .h file *(Whom does this function belong to?)*
- **::** which tells us to look inside the class name specified.

Overall, implementing a member function of a class is similar to how we implement a normal function. We need the return type, the function name and its parameters. The only difference is to add the class name of this function followed by "**::**".

To review classes in more details please refer to the **Recitation 9 writeup**.

Function Overloading

C++ allows more than one definition for a function name. Function overloading means that a function can be declared with the same name as another function in the same scope, *as long as it has different arguments and implementation.*

Command Line: basic commands, linking files

So far we have been using IDEs like codeBlocks, which makes it easier to compile, debug, and analyze our code because IDEs group compilers, debuggers, and editors. Sometimes these tools require time to learn and use because they don't always behave like we expected. A command line approach is often a faster, more versatile, and works across multiple operating systems and environments.

Command Line Interface/Terminal

It is a text interface through which a user can interact with the computer by issuing specific commands. Examples of these commands in Linux Environment are:

- **ls**: list the files in the current directory
- **pwd**: display the complete path of current directory you are at
- **cd <target_directory>**: change directory to the target directory
- **rm <file_name>**: removes the file indicated
- **rm -r <directory_name>**: removes a whole directory
- ….

Tip: Use Tab to autocomplete filenames or directory names.
For a more comprehensive list please refer to: https://help.ubuntu.com/community/UsingTheTerminal

To compile a project, we will need to tell the compiler which file has the *main* function and what are the other source files the project needs.
For example, we know that in our SolarSystem_main.cpp we have instances of the class *Planet* and instances of the class *SolarSystem*. The way we would compile our set of files to compile *createSystem* program is the following:

```
$ g++ -std=c++11 SolarSystem_main.cpp SolarSystem.cpp Planet.cpp -o createSystem
```

By typing ***g++ -std=c++11*** we ask the computer to use a compiler specific to the language c++11. The compiler then expects one or more arguments which will be the source files we want to compile, in our case ***SolarSystem_main.cpp SolarSystem.cpp Planet.cpp***. By passing in all three of them we are telling the compiler to link them together and treat them as if they were a single long source file. The

name after the "**-o**" is the name of the executable after the project is compiled. You can name it anything you want, but when "-o" is not specified, the default executable name is "**a.out**".
After executing the above command, to run the project, type the following into terminal:

```
$ ./createSystem
```

Recitation Activity

***Preamble**: we will now go ahead and implement a program that will generate a Solar System composed by various planets and calculate the difference of the planets' radius. Every solar system is an object of class **solarSystem** and every planet is an object of class **planet**. Let our Solar System have 5 orbiting planets and every planet have a name and a radius.*

Download the Recitation_10_files folder. The folder has ***solarSystem.h planet.cpp*** and ***SolarSystem_main.cpp***. Explore the files and understand them. You will be implementing the source code for the class SolarSystem and the header file for the class Planet accordingly to what you find in the respective header and source files. *Once you are done with creating the classes, start your testing by modifying **solarSystem_main.cpp** as described by the comments in the source code.*

Part 1: Planet Class

Parts 1 and 2 of this recitation activity have a shared main file SolarSystem_main.cpp, provided to you on Moodle.

The class you will implement is called ***Planet***. This time we have given you the class file ***planet.cpp***. It is your job to create the header file for the planet class. Create the file ***planet.h*** in CodeBlocks in the same directory as the main file. The planet class has two constructors, a default constructor (which takes no arguments) and a constructor that takes two arguments. One argument should be the name of the planet and the radius of the planet (in km). It should also have getters and setters for the class attributes.

Part 2: Planet Class

The next class you will implement is called ***solarSystem***. The ***solarSystem*.h** file is provided on moodle, and you will create your class file ***solarSystem.cpp***.

There are four private attributes in this class: ***maxNumPlanets*** (the max number of planets in this solar system)***, systemName*** (the name of the solar system)***, numPlanets*** (how many planets are in this solar system) and ***systemPlanets*** (an array of planets in this solar system)***.***

There should also be one constructor, a destructor, and seven methods to create for the ***solarSystem*** class. Four of the seven methods are getters for the class attributes ***systemName*** and ***numPlanets***. ***addPlanet(string, float)*** should add a new planet into the solar system. ***getPlanet(int)*** should return the the planet object of the index indicated by the argument. Finally, you will have to add a member function,

radiusDifference(Planet, Planet), that calculates the difference in radius between two planets (read the comments in ***solarSystem_main.cpp*** for more information).

Once you have implemented the header file for the planet class, you will need to modify your main to test your classes. In **solarSystem_main.cpp**, you will have to ask the user for names and the radius of five new planets you want to add to the solar system, and print out the difference in radius between every two planets in the system. Find a smart logic to compute all the differences as there are $C(5,2) = \frac{5!}{2!(5-2)!} = 10$ possible combinations and you do not want to hard code all of them. Additional information is found in the source files.

When you are done, zip *all* of your files into one file called **Recitation10.zip** and **submit to COG** (https://web-cog-csci1300.cs.colorado.edu/) you should also **submit to Moodle** by Sunday at 5pm.