CSCI 1300 - Spring 2018
Instructor: Fleming/Gupta
Recitation 11
Due Saturday, April 7th, by 6:00pm

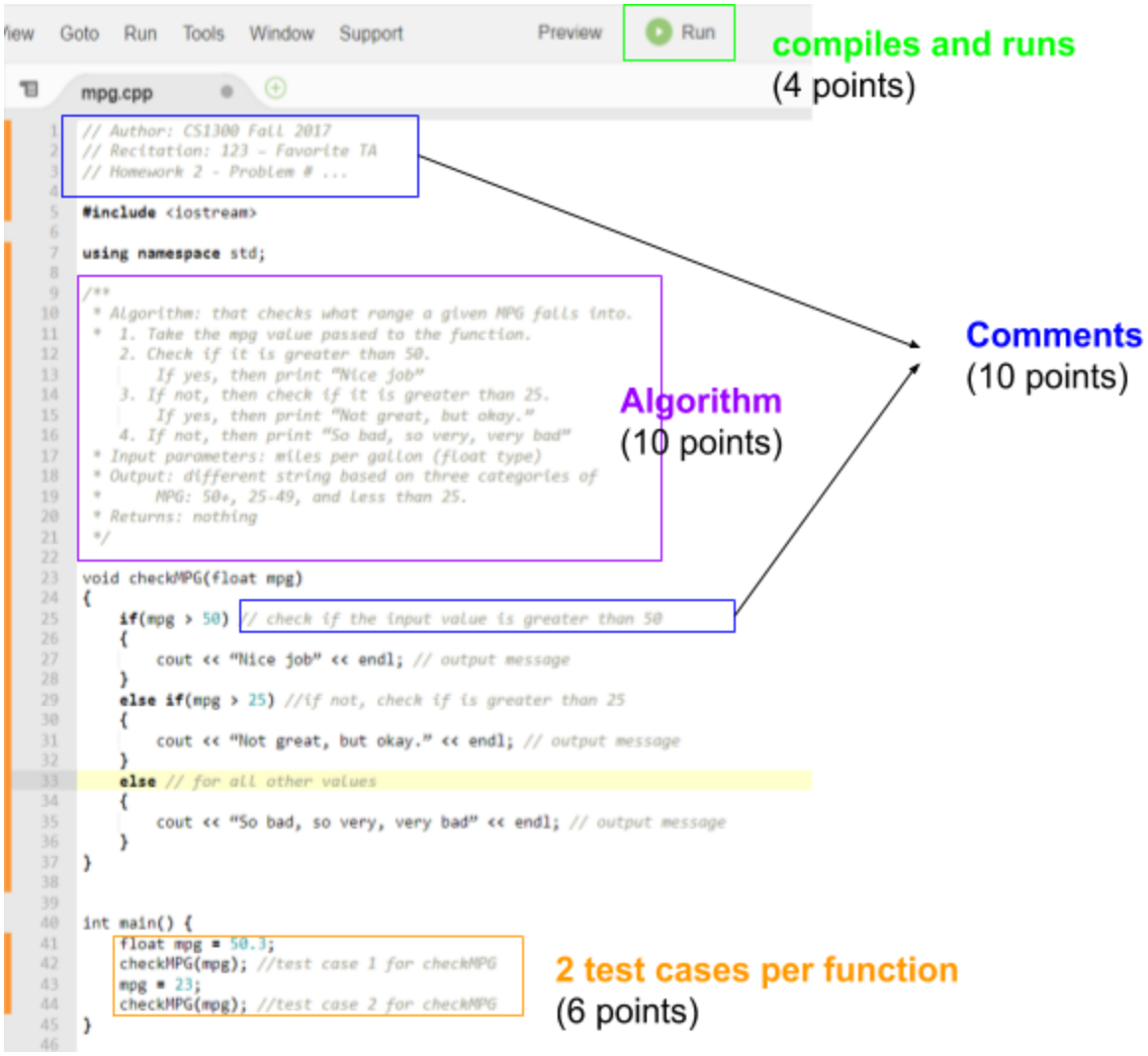
- All components (Cloud9 workspace and zip file) must be completed and submitted by **Saturday, April 7th 6:00 pm** for your solution to receive points.
- Recitation attendance is required to receive credit.
- **Develop in Cloud9:** For this recitation assignment, write and test your solution using Cloud9.
- **Submission:** Both steps must be fully completed by the submission deadline for your homework to receive points. Partial submissions will not be graded.
    1. Make sure your Cloud9 workspace is shared with your TA: Your recitation TA will review your code by going to your Cloud9 workspace. TAs will check the last version that was saved before the submission deadline.
        - Create a directory called *Rec11* and place all your file(s) for this assignment in this directory.
    2. **Submit a zip file to Moodle:** After you have completed all the questions and checked them on Moodle, you must submit a zip file with all the .cpp file(s) you wrote in Cloud9. Submit this file going to *Rec 11 (File Submission)* on moodle.

There is no autograder(Moodle quiz) for this recitation.

- 4 points for code compiling and running
- 6 points for sample runs
    - a sample run with input and output for createSystem (questions 1-3)
    - sample runs with input and output for question 4
- 10 points for an algorithm description for each function
- 10 points for comments
    - required comments at top of file

```
// CS1300 Spring 2018
// Author:
// Recitation: 123 – Favorite TA
// Cloud9 Workspace Editor Link: https://ide.c9.io/…
// Recitation 11 - Problem # ...
```

    - comments for functions / test cases




```cpp
// Author: CS1300 Fall 2017
// Recitation: 123 - Favorite TA
// Homework 2 - Problem # ...

#include <iostream>

using namespace std;

/**
 * Algorithm: that checks what range a given MPG falls into.
 *  1. Take the mpg value passed to the function.
    2. Check if it is greater than 50.
        If yes, then print "Nice job"
    3. If not, then check if it is greater than 25.
        If yes, then print "Not great, but okay."
    4. If not, then print "So bad, so very, very bad"
 * Input parameters: miles per gallon (float type)
 * Output: different string based on three categories of
 *      MPG: 50+, 25-49, and less than 25.
 * Returns: nothing
 */

void checkMPG(float mpg)
{
    if(mpg > 50) // check if the input value is greater than 50
    {
        cout << "Nice job" << endl; // output message
    }
    else if(mpg > 25) //if not, check if is greater than 25
    {
        cout << "Not great, but okay." << endl; // output message
    }
    else // for all other values
    {
        cout << "So bad, so very, very bad" << endl; // output message
    }
}


int main() {
    float mpg = 50.3;
    checkMPG(mpg); //test case 1 for checkMPG
    mpg = 23;
    checkMPG(mpg); //test case 2 for checkMPG
}
```

Review on Classes: Header and Source files

We learned in Recitation 10 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 10 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 Cloud9, 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:

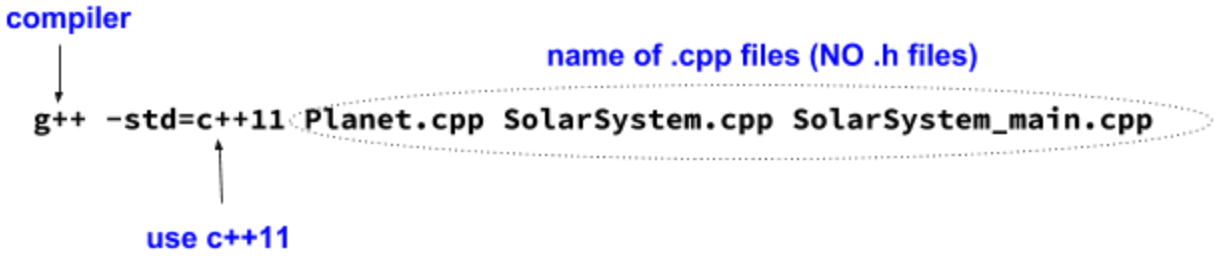
- `ls: list the files in the current directory`
- `pwd: display the complete path of current directory you are at`
- `cd <target_directory>: change directory to the target directory`
- `rm <file_name>: removes the file indicated`
- `rm -r <directory_name>: removes a whole directory`

Tip: Use *Tab* to autocomplete filenames or directory names.

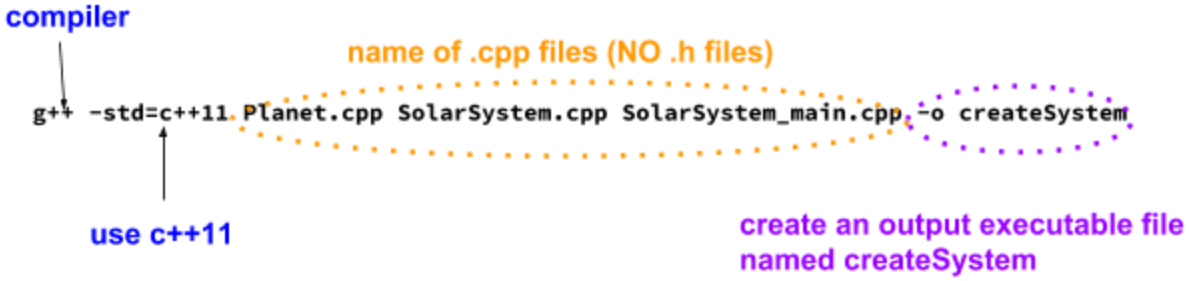
For a more comprehensive list please refer to:
https://help.ubuntu.com/community/UsingTheTerminal

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, assume we have a driver file named **SolarSystem_main.cpp** which uses instances of the class **Planet** and instances of the class **SolarSystem**. We also want our executable to be named *createSystem*. The way we would compile our set of files to produce the *createSystem* program is the following:



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
```

Questions 1 - 3

There are no autograder questions for these questions. The files you submit should run and produce the expected output. Otherwise, points will be deducted.

You will also need to submit (as comments or a separate text file) at least one additional sample run you used to test your program. The sample run should have the input you provided as well as the output your program generates.

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_11_files`** folder. The folder has:

- **`solarSystem.h`**
- **`planet.cpp`**
- **`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.

Questions 1, 2, and 3 of this recitation activity have a shared main file **`SolarSystem_main.cpp`**, provided to you on Moodle.

1. Planet Class (Create a Header File)

The class you will implement is called *planet*. We have given you the class file **`planet.cpp`**. Create the header file for the planet class.

Create the file planet.h in Cloud9 in the same directory as the main file. The planet class has:

- two constructors
    - **`planet()`** a default constructor (which takes no arguments)
    - **`planet(string, float)`** a constructor that takes two arguments, the name of the planet and the radius of the planet (in km)
- getters and setters for the class attributes.

2. Solar System Class (Create an Implementation File)

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:

1. **`maxNumPlanets`** (the max number of planets in this solar system. This should be set to 10)
2. **`systemName`** (the name of the solar system)
3. **`numPlanets`** (how many planets are in this solar system currently)
4. **`systemPlanets`** (an array of planets in this solar system).

The class should have the following methods (functions):

- one constructor
- a destructor (no code needed in it, just a comment is enough)
- 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
- **`radiusDifference(planet, planet)`**, that calculates the difference in radius between two planets (read the comments in **`solarSystem_main.cpp`** for more information).

3. Driver File

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. There will always be 5 planets for this object.

Find a smart logic to compute the difference between the radii of all the planets in this solar system. There will be **C(5,2) =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.

Here is a sample solar system:

planet	**radius**
Earth	3959.0
Mars	2106.0
Jupiter	43441.0
Mercury	1516.0
Neptune	15299.0

Radius difference	Earth	Mars	Jupiter	Mercury	Neptune
Earth	-	1853	39482	2443	11340
Mars	1853	-	41335	590	13193
Jupiter	39482	41335	-	41925	28142
Mercury	2443	590	41925	-	13783
Neptune	11340	13193	28142	13783	-

This is the expected output of this function for the above solar system.

```
Radius difference between planet Earth and planet Mars is 1853
Radius difference between planet Earth and planet Jupiter is 39482
Radius difference between planet Earth and planet Mercury is 2443
Radius difference between planet Earth and planet Neptune is 11340
Radius difference between planet Mars and planet Jupiter is 41335
Radius difference between planet Mars and planet Mercury is 590
Radius difference between planet Mars and planet Neptune is 13193
Radius difference between planet Jupiter and planet Mercury is 41925
Radius difference between planet Jupiter and planet Neptune is 28142
Radius difference between planet Mercury and planet Neptune is 13783
```

You will also need to submit (as comments or a separate text file) at least one additional sample run you used to test your program. The sample run should have the input you provided as well as the output your program generates.

Introduction to Vectors

Let's start with something we already know about- Arrays.

To recap, an array is a contiguous series that holds a *fixed number of values* of the same datatype.

A vector is a template class that uses all of the syntax that we used with vanilla arrays, but adds in functionality that relieves us of the burden of keeping track of memory allocation for the arrays. It also introduces a bunch of other features that makes handling arrays much simpler.

How Vectors Work- Syntax and Usage

First things first. We need to include the appropriate header files to use the vector class.

```
#include <vector>
```

We can now move on to declaring a vector. This is general format of any vector declaration:

```
vector <datatype_here> name(size);
```

The size field is optional. Vectors are *dynamically-sized*, so the size that you give them during initialization isn't permanent- they can be resized as necessary.

Let's look at a few examples

```
//Initializes a vector to store int values of size 10.
vector <int> vec1(10);

//Initializes an empty vector that can store strings.
vector <string> vec2;
```

You can access elements of a vector in the same way you would access elements in an array, for example **`array[4]`**. Remember, indices begin from 0.

You can find a quick reference to the functions available in C++ vector class in a nice PDF form here, but following are the ones you will need in this recitation:

All these functions belong to the vector *class*, so you need to call them on an instance of a vector. (vec1.size(), for example).

- **`size()`** returns the size of a vector.
- **`at()`** takes an integer parameter for index and returns the value at that position

Adding elements to the vector is done primarily using two functions

- **`push_back()`** takes in one parameter (the element to be added) and appends it to the end of the vector.

```
vector <int> vector1; // initializes an empty vector
vector1.push_back(1); //Adds 1 to the end of the vector.
vector1.push_back(3); //Adds 3 to the end of the vector.
vector1.push_back(4); //Adds 4 to the end of the vector.
cout<< vector1.size(); //This will print the size of the
vector - in this case, 3.
//vector1 looks like this: [1, 3, 4]
```

- **`insert()`** can add an element at some position in the middle of the vector.

```
//vectorName.insert(vectorName.begin() + position, element)
vector1.insert(vector1.begin() + 1, 2);
cout << vector1.at(1) << endl; // 2 is at index=1
//vector1 looks like this: [1, 2, 3, 4]
```

  Here, the **begin** function returns an iterator to the first element of the vector. You can think of it as an arrow pointer that points to the memory location just before the first element. The **`begin()`** would thus refer to the first element and **`begin()+k`** would refer to the kth element in the vector, k starting at 0.

Elements can also be removed.

- **`pop_back()`** deletes the last element in the vector
- **`erase()`** can delete a single element at some position

```
//vector_name.erase(vector_name.begin() + position)
vector1.erase(vector1.begin() + 0);
cout << vector1.at(0) << endl; //2 is at index=0
//vector1 looks like this: [2, 3, 4]
```

The vector class has many more functions so look at: http://www.broculos.net/2007/11/c-stl-vector-class-cheatsheet.html if you are interested in learning more. (You may want to use vectors instead of arrays for project 3).

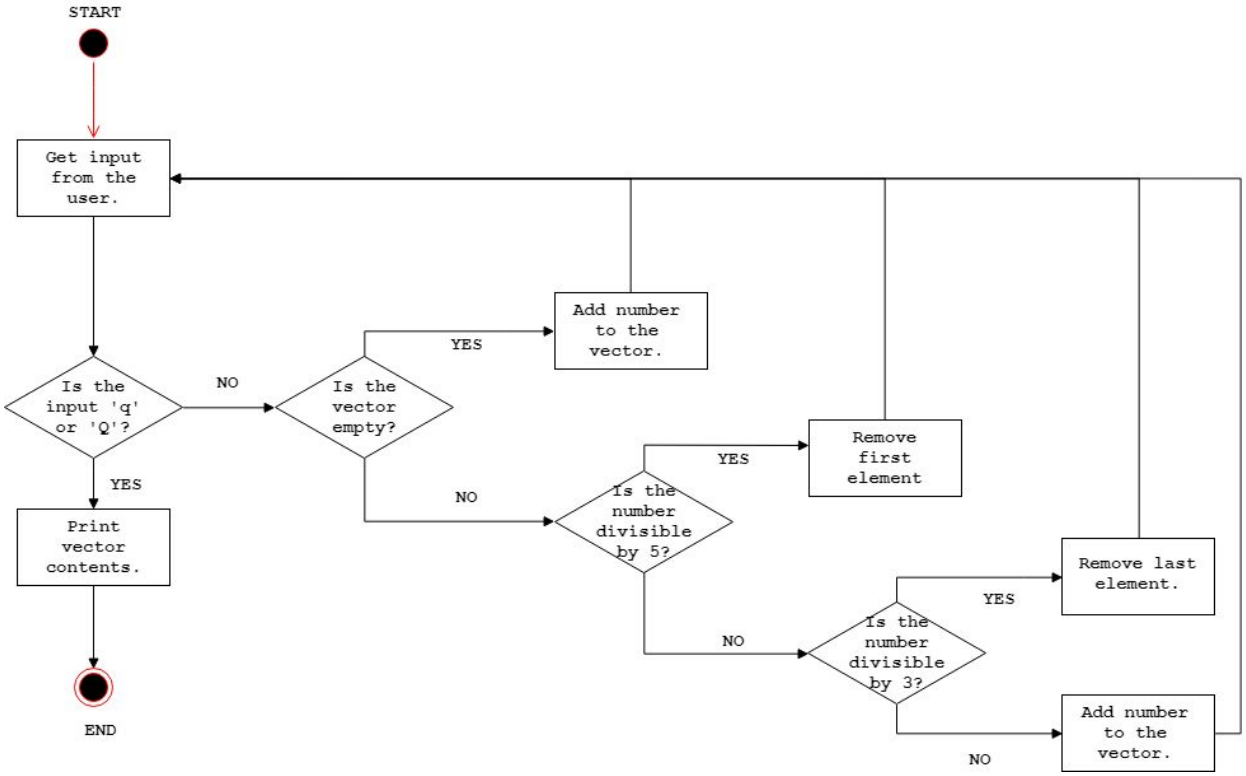
4. Vectors

There is no autograder for this question.

You will need to submit (as comments or a separate text file) at least one additional sample run you used to test your program. The sample run should have the input you provided as well as the output your program generates.

Write a program that asks the user to repeatedly enter **positive integers** and stops when the user enters 'q' or 'Q'. Your program should create a vector to handle the user input. At the beginning of the program, the vector should be empty. Then, as the user enters values, the vector will be modified as follows:

- If the vector is empty, insert the user input value into the vector.
- If the vector is not empty and the user input value is divisible by 5, then remove an element from the front(beginning) of the vector.
- If the vector is not empty and the user input value is divisible by 3, then remove an element from the end of the vector.
- Otherwise, insert the user input value at the end of the vector.



After the user is done entering values, your program should display all the elements in the vector, in order, separated by spaces, followed by (on the next line) the minimum and the maximum value.

Note: your program should make sure to only handle correct input (only positive integers are allowed).

Here's a possible sample run:

```
Please enter an integer: 3

Please enter an integer: 4

Please enter an integer: 7
```

```
Please enter an integer: 5

Please enter an integer: 8

Please enter an integer: 10

Please enter an integer: q

The elements in the vector are: 7 8

Min = 7

Max = 8
```

You will need to submit (as comments or a separate text file) at least one additional sample run you used to test your program. The sample run should have the input you provided as well as the output your program generates.