ECE 2036 Lab 3: Robot Swarm

Due: March 1, 2020 at 11:59 PM

In this lab we will be creating a C++ class, and an object of that class. Classes can be used to represent anything: cars, gradebooks, students, professors, or in this case, robots. For this lab, you will create the class **`Robot`**. However, one robot by itself is not very interesting so you will also be expected to create a class called **`RobotSwarm`** that will manage a group of **`Robot`** objects, letting them move around within a grid without colliding with one another. As a final piece, you will also maintain a record of each **`Robot`** object's "history" – the locations it touched on the grid and the number of times it touched the points on the grid. You will write this history out to a file for each Robot.

The learning objectives of this lab give you practice:

1. Using basic C++ classes;
2. Creating basic vectors of user-defined objects;
3. Creating constructors and overloading constructors;
4. Using and creating set and get member functions in C++ classes;
5. Using C++ string objects;
6. Using basic text file I/O objects.

You will create the class **`Robot`**. Your class will have two private data members: **`xPosition`** and **`yPosition`**. Your class will also have an additional data member, a two-dimensional array (your choice of C-style array or a C++ vector of vector of integers) for tracking each time the Robot touches a particular position. Your class will also have several public member functions for manipulating objects of the class:

- **`setXPosition(int)`**
- **`setYPosition(int)`**
- **`getXPosition()`**
- **`getYPosition()`**
- **`moveForward()`**
- **`moveLeft()`**
- **`moveRight()`**
- **`moveBackward()`**
- **`inputMove()`**
- **`displayPosition()`**
- **`printHistory()`**

In addition, your class will have a constructor which initializes **`xPosition`** and **`yPosition`** to zero.

One robot working by itself is not very interesting though so you will be expected to also design a class called **`RobotSwarm`**. The constructor of the **`RobotSwarm`** will accept a single integer argument that represents the number of **`Robot`** objects that need to be created. The constructor

will create that many **Robot** objects as a vector of **Robot** objects. When each **Robot** object is created, you must decide where to place each **Robot** object in a 32 by 32 grid, i.e. **xPosition** must be bound between 0 and 31, inclusive, and **yPosition** must be bound between 0 and 31, inclusive. Provide these limits using either **#define** or global constant variables. (This will largely depend on your two-dimensional array implementation.) You will also want to make sure that each **Robot** object resides on its own position, i.e. no position is occupied by more than one **Robot** object at a time.

The **RobotSwarm** class will have a public function **moveSwarm** that will have each **Robot** object randomly select to move one space in one of the four directions (**moveForward()**, **moveLeft()**, **moveRight()**, **moveBackward()**. Before permitting a **Robot** object to move in the selected direction, first make sure that no other **Robot** in the swarm occupies the intended space. If another **Robot** already occupies that space, have the current **Robot** object select another direction randomly. If the **Robot** object cannot select a position to move after 5 random selections, do not move that **Robot** object, and move on to the next one. After all **Robot** objects in the swarm have at least attempted to move, let the **moveSwarm** function return.

The **RobotSwarm** class will have another public function **printHistory** that will call the **printHistory** function of each of its Robot objects.

In the **main()** function of the code, you should create one instantiation (object) of **RobotSwarm**, with object name **mySwarm**. The code should prompt the user asking for the number of times the swarm needs to move. After the swarm has moved the specified number of times, the **main()** function should call **printHistory** for **mySwarm** and then exit.

DETAILED LAB PROGRAMMING REQUIREMENTS

Your code must include the following:

1. A constructor for class **Robot** with no arguments. The constructor must initialize the values of **xPosition** and **yPosition** to zero. It must also initialize the two-dimensional integer array for tracking its positions.
2. **setXPosition(int), setYPosition(int), getXPosition(), getYPosition()**: Member functions of class **Robot** to set and get the values of **xPosition** and **yPosition**.
3. **moveForward(), moveBack(), moveLeft(), moveRight()**: Member functions of class **Robot** to move the robot forward, backward, left, or right by one integer unit, by calling **setXPosition** and **setYPosition** to update the robot's position. It should also increment the position in the **Robot** object's history array. NOTE: You do not need to worry about the orientation of the **Robot** in space, i.e. assume the **Robot** object only faces one direction the entire time.
4. **inputMove()**: A member function of class **Robot** to prompt the user for the next move of the robot, and call the move functions accordingly.

5. **`displayPosition()`**: A member function of class Robot to call **`getXPosition()`** and **`getYPosition()`** and display the x and y position of the robot on the screen. This function will be useful for debugging your code.
6. **`printHistory()`**: Prints the history of the positions for the **`Robot`** in the 32x32 grid of integers to a file. For simplicity, this can be a single file that each **`Robot`** object simply appends to.
7. A constructor for class **`RobotSwarm`**. The constructor will take a single integer argument denoting the number of **`Robot`** objects that need to be created. It will also set the x and y positions of each **`Robot`** object randomly, making sure that no two **`Robot`** objects occupy the same spot.
8. **`moveSwarm`**: Moves each **`Robot`** object in a randomly selected direction according to the above instructions. For testing purposes, you can allow the random selections to use the default seed, but for the final submission, your code should seed the random number generator using the current time, i.e. **`srand(time(NULL));`**
9. **`printHistory`**: Calls the **`printHistory`** function of each **`Robot`** object.
10. In the **`main()`** function of the code, you should create one instantiation (object) of **`RobotSwarm`**, with object name **`mySwarm`**. The code should prompt the user asking for the number of **`Robot`** objects to create and the number of times the swarm needs to move. After the swarm has moved the specified number of times, the **`main()`** function should call **`printHistory`** for **`mySwarm`** and then exit.

Your code should be in at least 5 separate files. Robot.h should contain only the class definition for your Robot class. Robot.cc should contain the implementations for all of the methods defined in Robot.h. RobotSwarm.h should contain only the class definition for your RobotSwarm class. RobotSwarm.cc should contain the implementations for all of the methods defined in RobotSwarm.cc. main.cc should contain your main method. You may want an additional file for any constant or global information that Robot, RobotSwarm, and main may need to be aware of.

To compile your files, you will need to use the following command (if you are compiling outside of an IDE):

```
g++ Robot.cc RobotSwarm.cc main.cc -o swarmIt
```

(If you needed any other .cc files as helpers, make sure to include them in your build command as well.)

This will allow you to run your code as follows: **`./swarmIt`**

To submit, compress the files described above into yourusernameLab3.tar.gz and submit via Canvas. In order to create this tarfile (a.k.a. tarball), you will need to run the following command:

```
tar –cvzf yourusernameLab3.tar.gz ./Lab3
```

(This command assumes that your source code is contained in a folder named Lab3.)

Appendix A: Sample File Output

(Example only shows a 16x16 grid for one Robot. Actual output will be the 32x32 grid for each Robot object in the RobotSwarm.)

Use tabs between each position on a line. For readability, it is up to you if you would want to print a blank new line between each row. It is not required though. Between each Robot printout, print “Robot” and the index of that Robot in the swarm.

Output file should simply be a text file named “robotOutput.txt”

```
Robot 0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	0
0	0	0	0	1	0	1	1	1	0	0	0	0	0	0	0
0	0	0	1	1	0	0	1	1	0	0	0	0	0	0	0
0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0
1	1	2	1	1	0	1	0	1	1	0	0	0	0	0	0
1	1	1	1	2	2	2	1	2	1	0	0	0	0	0	0
0	1	1	1	1	2	1	0	1	0	0	0	0	0	0	0
1	2	2	2	0	1	1	1	2	1	0	0	0	0	0	0
1	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0
2	2	1	2	0	1	0	0	0	0	0	0	0	0	0	0
1	1	1	2	1	1	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
```

APPENDIX B: ECE 2036 Lab Grading Rubric

If a student's program runs correctly and produces the desired output, the student has the potential to get a 100 on his or her lab; however, TA's will **randomly** look through this set of "perfect-output" programs to look for other elements of meeting the lab requirements. The table below shows typical deductions that could occur.

In addition, if a student's code does not compile on **`pace-ice.pace.gatech.edu`**, then he or she will have an automatic 30% deduction on the lab. Code that compiles but does not match the sample output can incur a deduction from 10% to 30% depending on how poorly the output matches the output specified by the lab. This is in addition to the other deductions listed below or due to the student not attempting the entire assignment.

AUTOMATIC GRADING POINT DEDUCTIONS

Element	Percentage Deduction	Details
Does Not Compile on PACE-ICE System	30%	Program does not compile on pace ice cluster!
Does Not Match Output	10%-30%	The program compiles but doesn't match output. For this lab, output will be each 128x128 grid for each Robot's history.

ADDITIONAL GRADING POINT DEDUCTIONS FOR RANDOMLY SELECTED PROGRAMS

Element	Percentage Deduction	Details
Correct file structure	10%	Does not use both .cc and .h files, implementing class prototype correctly
Does not implement classes	10%	
Encapsulation	10%	Does not use correct encapsulation in object-oriented objects
Setters/Getters	10%	Does not use setters and getters for each data member in Task Class.
Constructors	10%	Does not implement constructors with the correct functionality.
Clear Self-Documenting Coding Styles	5%-15%	This can include incorrect indentation, using unclear variable names, unclear comments, or compiling with warnings. (See Appendix B)

LATE POLICY

Element	Percentage Deduction	Details
Late Deduction Function	score - (20/24)*H	H = number of hours (ceiling function) passed deadline note : Sat/Sun count has one day; therefore H = 0.5*Hweekend

Appendix C: Coding Standards

Indentation:

When using *if/for/while* statements, make sure you indent 4 spaces for the content inside those. Also make sure that you use spaces to make the code more readable.
For example:

```
for (int i; i < 10; i++)
{
    j = j + i;
}
```

If you have nested statements, you should use multiple indentions. Each { should be on its own line (like the *for* loop) If you have *else* or *else if* statements after your *if* statement, they should be on their own line.

```
for (int i; i < 10; i++)
{
   if (i < 5)
   {
       counter++;
       k -= i;
   }
   else
   {
       k +=1;
   }
   j += i;
}
```

Camel Case:

This naming convention has the first letter of the variable be lower case, and the first letter in each new word be capitalized (e.g. firstSecondThird). This applies for functions and member functions as well! The main exception to this is class names, where the first letter should also be capitalized.

Variable and Function Names:

Your variable and function names should be clear about what that variable or function is. Do not use one letter variables, but use abbreviations when it is appropriate (for example: “imag" instead of
“imaginary”). The more descriptive your variable and function names are, the more readable your code will be. This is the idea behind self-documenting code.

File Headers:

Every file should have the following header at the top

```
/*
Author: <your name>
Date Last Modified: <date last modified>
Organization: ECE2036 Class

Description:

Describe what is done in this file.

*/
```

Code Comments:

1. Every function must have a comment section describing the purpose of the function, the input and output parameters, the return value (if any).
2. Every class must have a comment section to describe the purpose of the class.
3. Comments need to be placed inside of functions/loops to assist in the understanding of the flow of the code.