CENG 201 PROGRAMMING PROJECT[1]

For this project you will write a C++ program which routes a Robot through a maze to find the Goal. You will develop a number of algorithms for solving the maze and will be graded on correctness, programming style, and speed of execution.

THE MAZE

The Maze consists of open areas (indicated by blank spaces), walls (indicated by `X'), where Robot starts (indicated by `R') and where the Goal is (indicated by `G'). The Robot can only travel north, south, east or west. It may not travel along the diagonals of the maze. Your task will be to indicate the route from the Robot to the Gal by drawing the route using the `*' symbol. So a sample maze might look like this:

```
XXXXXXX
XXG   X
X   XXX
X     X
XXXXX X
XR    X
XXXX XX
```

While the output might look like:

```
XXXXXXX
XXG   X
X * XXX
X ****X
XXXXX*X
XR****X
XXXX XX
```

The maze will be a square of fixed size (N by N) and the Robot is not allowed to leave the bounds of the maze.

ROUTING SCHEMES

You are to develop three routing schemes for the Robot.

- A queue-based routing scheme
- A stack-based routing scheme
- An “optimal” routing scheme

[1] This project is adapted from EECS380 Winter 2001 Project 1
http://www.eecs.umich.edu/courses/eecs380/PROJ/proj1.html

In the queue-based routing scheme you are to have a queue of maze locations. Place the location that the Robot starts at in the queue. And then do the following:

1. Dequeue the next location.
2. Enqueue all locations next to the location you just dequeued which are empty spaces (blanks) that have never before been enqueued. Remember that the Robot can only travel north, south, east or west, so diagonal spaces are not considered next to each other. As you enqueue these spaces, check to see if any of them hold the cheese. If none of them do, go back to step 1.
3. Clearly you have found a route from the Robot to the Goal, but exactly what that route may not be clear. So you need to figure out what route the Robot should take. Your route should not backtrack onto itself. That is, it should not require the Robot visiting the same square twice.

For the stack-based routing scheme, do the same thing, but push and pop the locations rather than enqueueing and dequeueing them. Clearly you will need to expand on the above algorithms, but you must follow the basic algorithm outlined above for the queue and the stack-based routing algorithms.

For the optimal routing scheme, you are free to use any routing algorithm you wish to run. But you must get the Robot to the Goal in as few steps as possible AND you will be graded on how long it takes you to find Robot's route. The first goal is getting an optimal route. Finding a non-optimal or illegal route (going through walls, outside of the Maze, etc.) quickly is worth nothing, even if it is fast...

INPUT AND OUTPUT FORMATS

Your program should be able to deal with two different input and output formats. The first is a coordinate scheme where the Robot, the Goal, and each wall are listed by their XY-coordinates. Any unspecified location is assumed to be empty. The second is a map where the XY coordinates are implicit by their ordering. In both cases the size of the maze is specified on the first line as a single integer "N'" representing an N by N maze. For example the following is a legal map-formatted Maze:

```
4
XXXX
R XG
X
  X
```

While that same Maze could be specified in the coordinate scheme as:

```
4
X 0 0
X 0 1
X 0 2
X 0 3
X 3 3
R 1 0
X 2 0
X 1 2
```

```
G 1 3
```

The output file formats are very similar. For the map format simply add the ``*'' as needed to specify the path in addition to reprinting the rest of the map. For the coordinate solution specify only where to place ``*''s. So for the above Maze one might get:

```
4
XXXX
R*XC
X***
  X
```

as the map format and that same solution could be represented using the coordinate output as:

```
* 1 1
* 2 1
* 2 2
* 2 3
```

In the event of the coordinate solution the path should specify each step taken starting next to the Robot and ending next to the Goal.

In either case, if no route exists your program should leave the output file blank.

After each case print the following message on the screen

- if Goal found, then print
  "Goal found! *EE* positions explored, unexplored list contains *CC* positions."
- else
  "Goal not found! *EE* positions explored."

where *EE* denotes the number of steps in the route and *CC* denotes the number of positions in the stack or queue depending on the case.

TIMING

When you are asked to time your program, you should measure the time it takes for your search algorithm to run. Do not include the time it takes to read the input or write the output. HOWEVER you must include the time for the entire search algorithm including figuring the exact path taken. Don't try to bend the rules here (by starting the search while reading in the data), doing so will result in 0 points for the timing part of the grade (see below).

Timing information in milliseconds should be printed to the standard output in the following format

```
@runtime = xxx.xxxx sec
```

STACKS, QUEUES AND THE STL

You are to write the program using your own stack and queue classes. Those classes should have the same interface as the stack and queue classes in the STL. Further, you will also be expected to use the

STL rather than your own stack and queue implementation. You only need to write the member functions for queue and stack that you actually use!

To be able to experiment with different implementations of the same interface, (stack and queue) you need to do the following:

1. Have the alternative container classes available (each class must have a different name).
   Example:
     Proj1MyContainer, Proj1STLContainer
   Note, you need not use the same names as above (e.g. Proj1STLContainer).
2. Implement your code that uses containers in terms of an alias Example: Container (in other words, Container will be the class name assumed by your code, even though no such class was defined yet).
3. Use the typedef command to make the alias point to one of the actual classes
   Example:
     typedef Proj1MyContainer Container;
   This command should go after your #include the headers of your container implementations.
4. In order to change how the code is compiled, depending on the container implementation you need, use

```
#ifdef  STLon
  typedef Proj1STLContainer Container;
#else
  typedef Proj1MyContainer  Container;
#endif
```

(The # should be in the first column of your code and not indented.)

5. STLon is a "macro" or a "symbol" that, when defined, will make your code use an STL-based container. Otherwise, your own container will be used (but the code implemented in terms of "Container" will be unchanged!)
6. To define STLon, modify Makefile and add -DSTLon after g++. This has the same effect as adding #define STLon 1 to your code (before #ifdef), except that your code is not modified.

When you submit your project, you need to have -DSTLon set in the Makefile. The code should work identically (constant-factor differences in speed are allowed). None of your code should ever perform a #define STLon. If you want to better understand what all of this is doing, read the man pages at http://msdn.microsoft.com/en-us/library/hhzbb5c8(v=vs.110).aspx

COMMAND LINE ARGUMENTS

Your program should take the following command line arguments: (when we say a switch is "set" that means it appears on the command line.)

- -Stack (boolean) If this switch is set you should use the stack-based routing scheme.
- -Opt (boolean) If this switch is set you should use your optimal routing scheme.
- -Queue (boolean) If this switch is set you should use the queue-based routing scheme.
- -Infile (string) Use the string as the input filename. If the -Infile flag does not appear then you should use the file ``maze.txt'' as the input filename.
- -Outfile (string) Use the string as the output file name. If the -Outfile flag does not appear then you should use the file ``mazeout.txt'' as the output filename.
- -Time (boolean) If this switch is set, print the runtime of your program as described above.
- -Incoordinate (boolean) If this switch is set, the input file is in the coordinate format. If the switch is not set, the input file is in the map format.
- -Outcoordinate (boolean) If this switch is set, the output file is to be in the coordinate format. If the switch is not set, the output should be in the map format.
- -Help (boolean) If this switch is set your program should print a brief help message which describes what the program does and what each of the flags are. The program should then exit(0).

Legal command line arguments must include exactly one of -Stack -Opt or -Queue. If none are specified or more than one is specified the program should print an informative message to standard error and exit(1).

Errors to check for

- In the map input format you should check for: illegal characters, maps which are too short, (either not enough characters per line or not enough lines), and those don't start with a number by itself on the first line.
- For the coordinate input format you should check for: illegal characters in the first column, coordinates which would not fit in a maze of the stated size, and those that don't start with a number by itself on the first line.
- Not being able to open the output file for writing.
- Not having exactly one of -Stack -Opt or -Queue on the command line.
- In all these cases, print an informative error message to standard error and exit(1).

What you don't need to check for

- For the map format ignore any extra characters on a given line, or on lines below the last line needed.

- For the coordinate format, assume that the data is really in the correct format. (ie. character integer integer) You don't need to error check that. (Although you do need t check to see if the character and the integers are valid as mentioned above.)

Coding style

Use a reasonable organization for your overall program.

> Find a fairly reasonable class structure. Don't stick everything into one class. Make reasonable use of constructors. Where needed be sure to supply a custom destructor. If the way you design your code feels sloppy to you, it probably is. Utilize multiple files in a way that is consistant with the general use in C++. Our text does a good job describing this.

Use reasonable comments.

> Explain what each class does and what each data member is used for. A one or two line description of most member functions is also desirable. Where you use non-standard coding techniques, document them. List your name and the date last modified for each file.
>
> Remember that a useless comment is worse than no comment at all.
> int temp; // declare temp. variable
> would be an example of a useless comment which just makes code harder to read!

Use reasonable formatting.

> It should be obvious where a given code block ends due to indentation. Use reasonable and informative variable names. On systems where the program "cb" (C beautify) is available, consider using an indentation style similar to what it generates. Avoid lines that wrap in an 80 column display wherever possible.

Variable names

> Using things like "i" and "j" as variables can be reasonable, but you should not use such variables to store meaningful long-term data. Other than LCV (loop control variables) you should use descriptive names for your variables, functions, classes, structures, etc.

Code resuse

> If you find yourself rewriting basically the same algorithm many times, stop and try to see if you can somehow reuse the code.

PART 1: (assigned 13.12.2014; due 27.12.2014)

In this part, you will implement the following sections.

- **INPUT AND OUTPUT FORMATS**
- **STACKS, QUEUES AND THE STL**

PART 2: (assigned 13.12.2014; due 10.1.2015)

In the last part, you will use the files programmed in Part 1 to develop the following sections.

- **ROUTING SCHEMES**
- **TIMING**

GRADING

Working stack and queue algorithms using your own classes -- 50 points.

Working stack and queue algorithms using the STL -- 25 points.

A working optimal algorithm. -- 15 points

The speed of your optimal algorithm -- 10 points

Further, poor coding style will result in potentially significant deductions. Follow the above rules as closely as possible.