CS2201
Fall 2017
Assignment No. 7

Purpose: This project is meant to give you experience in creating and using a generic template class. In particular, you will implement a template stack ADT and a template queue ADT that will be capable of holding any type of data. This project will also introduce you to inheritance and polymorphism in C++. Then you will combine all the above to create a program that can explore a maze to find the exit/goal.

The Assignment: You will complete this assignment in several steps. To be successful, you will need to carefully complete each step in order.

Step I:
Download the file "`project7.zip`" from the project page in Brightspace and unzip the files into a project directory where you do your course work. The files you unzip are a CLion project, so you do not need to create a project on your own. You can open the project by staring CLion, select open project, and navigate to the folder you just unzipped. As you follow the subsequent steps, add your code to this project so that it will be compiled and linked appropriately. Contact me immediately if you encounter any problems with the supplied code. If you are not using CLion, extract the necessary files and build a project with whatever IDE you are using.

Step II:
Implement a template stack ADT that will hold data of any type. Call the class `Stack`.

Template **Stack** class functional specification:
This class will provide support for a template stack abstraction that holds data of any specified type. Define two files `Stack_t.h` and `Stack_t.cpp` that implement the class. It must support the following public operations (T is used in place of ItemType in the description just to save space; you will likely want to continue using ItemType):

Stack( )	The default constructor.
Stack(const Stack<T>& rhs)	The copy constructor.
~Stack( )	The destructor.
operator= (const Stack<T>& rhs)	The assignment operator.
bool isEmpty( ) const	Tests whether the stack is empty.
void push(const T& item)	Push an item on top of the stack.
void pop( )	Pop the top item off the stack. If a pop( ) is attempted on an empty stack then throw a std::underflow_error exception.
T top( ) const	Return the top item off the stack without popping it. If a top( ) is attempted on an empty stack then throw a std::underflow_error exception.
size_t size( ) const	Return the number of items on the stack.

Note: Since the source code of a template class must be available to the compiler whenever an instance of the template class is created, I recommend the following strategy:

1. Declare the template class in Stack_t.h as described in lecture. Start with your DblStack class definition by making a copy of your DblStack.h file. The number of changes necessary to make the DblStack class declaration a template is actually quite minimal. Important note: It is easier to templatize the class if your node struct is defined within the scope of the class (usually within the private section) as opposed to being define before the class [I recommend that you make this change first and re-test your concrete DblStack class before proceeding.]
2. Next, copy your DblStack.cpp file into Stack_t.cpp, and start making the changes necessary to the class methods. The changes here are a bit more complex, but not too bad if you are careful and keep track of what you are doing. Remember that ***all*** class method must become template functions. You must templatize the class name whenever it is use as a type declaration. Be sure to carefully review the class specification above.
3. Next, add **`#include "Stack_t.cpp"`** as the next to the last line in your Stack_t.h file (place it just above the **`#endif`** line). Thus to use your template class, one only needs to do a **`#include "Stack_t.h"`** at the top of their source file. Note: since the .cpp file is now included by anyone who wants to use a stack, DO NOT include that file as a part of the CLion project – otherwise the compiler will attempt to compile that file separately which will lead to a lot of error messages. To reiterate the last point: do not include the template source file as a part of the CLion project; the file needs to exist and be in

the same directory/folder as the other files of the project, it is just not declared as part of the project, which means it is not listed in the SOURCE_FILES list of CMakeLists.txt. Note: the Stack_t.cpp file should no longer do a **`#include of "Stack_t.h"`**, since the .h file now includes the .cpp file – so be sure to remove that line from the .cpp file [note: this last step is actually not necessary ***if*** your .h file contains the correct header guards so that it is not compiled a second time during a compilation].

4. Your stack implementation must use a linked-list structure of your own creation. Do not attempt to use the stack that is available in the Standard Template Library. Again: defining your node structure inside the class is preferred to defining it outside the class since it makes templatizing the class easier.
5. All functions should be properly documented with pre- and post-conditions. These comments should appear both with the function declaration (.h file) and the function definition (.cpp file).
6. Finally, fully test your templated stack class before proceeding. One way to do this would be to make a copy of your Project #5, replacing your DblStack code with the new Stack_t code. You would then go through your program that you used to test the DblStack and make the following changes: whenever you created and used a DblStack object, now create a Stack<double> object. You should not have to change how that object is used. Recall from the discussion in lecture that template functions get instantiated & compiled **only if** you actually use them – thus your testing should include at least one call to all methods of your class, otherwise the methods do not even get compiled.
7. Final reminder: Be sure that you name your files, classes, and methods <u>exactly</u> as specified. In particular, the file names use **Stack_t** while the class name is **Stack**.

Step III:
Implement a template queue ADT that will hold data of any type. Call the class `Queue`.

<u>Template **Queue** class functional specification</u>:
This class will provide support for a template queue abstraction that holds data of any specified type. Define two files `Queue_t.h` and `Queue_t.cpp` that implement the class. It must support the following public operations (T is used in place of ItemType in the description):

Queue( )	The default constructor.
Queue(const Queue<T>& rhs)	The copy constructor.
~Queue( )	The destructor.
operator= (const Queue<T>& rhs)	The assignment operator.
bool isEmpty( ) const	Tests whether the queue is empty.
void enqueue(const T& item)	Add an item to the end of the queue.
void dequeue( )	Remove the item off the front of the queue. If a dequeue( ) is attempted on an empty queue then throw a std::underflow_error exception.
T front( ) const	Return the item from the front of the queue without removing it. If a front( ) is attempted on an empty queue then throw an std::underflow_error exception.
size_t size( ) const	Return the number of items in the queue.

<u>**Note:**</u> Since the source code of a template class must be available to the compiler whenever an instance of the template class is created, I recommend the following strategy (identical to what you did for the stack class):

1. Define the template class in Queue_t.h as described in lecture. Start with your DblQueue class declaration by making a copy of your DblQueue.h file. The number of changes necessary to make the DblQueue class definition a template is actually quite minimal. Important note: It is easier to templatize the class if your node struct is defined within the scope of the class (usually within the private section) as opposed to being define before the class [I recommend that you make this change first and re-test your class before proceeding.]
2. Next, copy your DblQueue.cpp file into Queue_t.cpp, and convert your Queue_t.cpp file to define a templated queue class. This will be similar to the effort necessary to define the templated stack class. Be sure to carefully review the class specification above.
3. Next, add **`#include "Queue_t.cpp"`** as the next to the last line in your Queue_t.h file (place it just above the **`#endif`** line). Thus to use your template class, one only needs to do a **`#include "Queue_t.h"`** at the top of their source file. Note: since the .cpp file is now included by anyone who wants to use a queue, DO NOT include that file as a part of the CLion project – otherwise the compiler will attempt to compile that file separately which will lead to a lot of error messages. Note: the Queue_t.cpp file should no longer do a **`#include "Queue_t.h"`**, since the .h file now includes the .cpp file – be sure to remove that line from your .cpp file [note: this last step is actually not necessary ***if*** your .h file contains the correct header guards so that it is not compiled a second time during a compilation].

4. Your queue implementation must use a linked-list structure of your own creation. Do not attempt to use the queue that is available in the Standard Template Library.
5. All functions should be properly documented with pre- and post-conditions. These comments should appear both with the function prototype (.h file) and the function definition (.cpp file).
6. Finally, fully test your templated queue class before proceeding. Use the test program you used in step 1 above.
7. Final reminder: Be sure that you name your files, classes, and methods exactly as specified.

Step IV:
Now that we have some of the scaffolding work out of the way, we are almost ready to explore a maze. Before we can proceed, we need to learn about the other code that has been provided. Besides reading the functional specifications below, be sure to review the provided code in the .h and .cpp files. **Important note:** **DO NOT CHANGE ANY OF THE CODE IN THE FOLLOWING THREE CLASSES.**

Point class functional specification:
This class provides support for a point in the x-y coordinate system. Each Point object simply has an x coordinate and a y coordinate. We will use the Point class to keep track of positions within the maze. The Point class provides the following public functionality:

Point( )	The default constructor. Initializes the Point to the origin.
Point(int x, int y)	The alternative constructor.
operator== (const Point& rhs) const operator!= (const Point& rhs) const	The equality operators.
operator<< (ostream & os, const Point& p)	The insertion operator for output. Thus a Point object knows how to print itself in a nicely formatted manner.
int x int y	The x & y coordinates of a Point are publicly accessible.

The Point class does not perform any dynamic memory allocation, and thus the compiler-provided Big 3 will suffice; so remember that the compiler provides you with a working assignment operator for the Point class.

Maze class functional specification:
This class provides support for representing and investigating a maze. A maze is represented with a 2 dimensional array of char, where different characters represent walls (‘#’), open spaces (‘ ’), the starting point (‘o’), and the ending/goal point (‘*’). The Maze class provides the following public functionality:

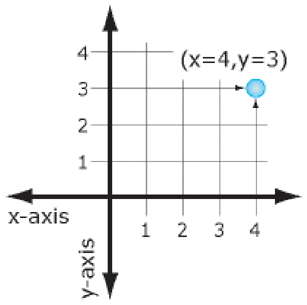
Maze(string filename)	The constructor which initializes the Maze with data from the named file. The file format is described below.
void printMaze( ) const	Display the maze to the console.
void printVisitedMaze( ) const	Display the maze to the console, also indicating which locations have been marked as visited.
int getNumRows( ) const int getNumCols( ) const	Get the row/column dimensions.
char get(int x, int y) const char get(Point location) const	Get the item from the maze for the specified position. Throws an exception if the xy/location is invalid.
void markVisited(int x, int y) void markVisited(Point location)	Mark a location as visited. Throws an exception if the xy/location is invalid.
bool hasBeenVisited(int x, int y) const bool hasBeenVisited(Point location) const	Determine if a location has been visited. Throws an exception if the xy/location is invalid.
bool isOpen(int x, int y) const bool isOpen(Point location) const	Determine if a location is open. Throws an exception if the xy/location is invalid.
bool isWall(int x, int y) const bool isWall(Point location) const	Determine if a location is a wall. Throws an exception if the xy/location is invalid.
void resetVisitedFlags( )	Turn all the visited flags off.
Point getStartLocation( ) const	Get the starting point for the maze.
Point getEndLocation( ) const	Get the ending point for the maze.

Maze file format:
The file format for a maze will be that the first line contains the dimensions of the maze (height and then width), and subsequent lines each contain one row of the maze, with each character representing one square of the maze. Walls are represented by the hash ‘#’

character, open locations by the space ‘ ’ character, the start location is represented by the lowercase ‘o’ character, and the ending location by the asterisk ‘*’.

Here is a sample maze input file:
```
10 10
##########
# #      #
# # #### #
# # #* # #
# # ## # #
# #    # #
# # #### #
# ## o## #
#        #
##########
```

The maze uses zero-based indexing, with the bottom-left corner being the origin: x=0 and y=0. The x-coordinate increases across the screen left-to-right, while the y-coordinate increases up the screen bottom-to-top. In the above example, the ending location is at coordinate (5, 6), while the start location is at coordinate (5, 2). When using a Point object to access a location in the maze, the Point’s x/y fields map naturally to the maze’s coordinate system.



To make things easier, static matrices are used to represent the maze. Thus the largest maze that can be represented is 50 by 50. This also means that we do not need the Big 3 for the Maze class.

You are encouraged to write your own test mazes. If you are creative and want to share your mazes with the rest of the class, simply post them to Piazza.

Note that the maze files are simple ASCII text files. Different operating systems represent the end of a line in different ways. The files I will distribute with the project were made on Windows. You may need to convert these files if you are running on a different OS[1]. Do a web search on “convert dos text file to macos”. Wikipedia has a nice article.

PointAgenda class functional specification:
This class provides support for an agenda of points that you want to visit. This class is what is known as a “*pure virtual class*” or an “*abstract base class*”. That means that this class declares methods, but does not define all of them. What good is a class that does not define its methods? Well, it is used as a base class in an inheritance hierarchy, where all the derived classes override and thus actually define the pure virtual methods. This ensures that all derived classes provide support for the same methods (Java uses the Interface class for this purpose). For more information on abstract classes see pages 44-46 in the class text. The PointAgenda class provides the following public functionality:

~PointAgenda( )	The destructor.
bool isEmpty( ) const	Determine if the agenda has any Points in it.
void add(const Point & item)	Add a Point to the agenda.
void remove( )	Remove the next Point from the agenda. If a remove( ) is attempted on an empty PointAgenda then a std::underflow_error exception is thrown.
Point peek( ) const	Get the next Point from the agenda without removing it. If a peek( ) is attempted on an empty PointAgenda then a std::underflow_error exception is thrown.
size_t size( ) const	Get the count of Points in the agenda.

Main.cpp file:
In addition to the above three class, a Main.cpp file is provided that controls the program. It creates a Maze object, a PointAgenda object (actually one of the derived class objects) and a MazeSolver object. It then calls the solve() method on the MazeSolver object. This Main.cpp program has been provided to you and you should not make any changes to it.

Again: DO NOT CHANGE ANY OF THE CODE IN THE ABOVE THREE CLASSES.

[1] To convert a Windows text file to a Mac text file do this: (A) Open up TextEdit and create a new file. (B) Click the "Format" menu and choose "Make Plain Text". (C) Open up the maze and copy it into the new file. (D) At the beginning of every line, hit backspace (to get rid of the old newline character) then hit enter to put in the Mac newline character. (E) Save the new file.

Step V:
Now that we have defined our abstract base class, we have to create the concrete derived classes that implement our desired interface (before proceeding, you should review pages 40-46 of the class text). In C++ we do this by using public inheritance. This mechanism ensures that all of the public (or protected) members of the base class are visible to the derived class. Because we intend for our derived classes to be usable, we must make them *concrete*. This means that we are not allowed to have any pure virtual functions, whether they are inherited from our base class or explicitly defined. Follow this strategy when creating a concrete derived class:

1. Start out with an empty class declaration and modify the declaration in order to use public inheritance. This should result in code similar to: `class DerivedClass : public BaseClass {...};.` The **public** keyword is crucial here. C++ has three different inheritance mechanisms and if you forget to explicitly use the visibility modifier, it will default to **private**. This is a common cause of a lot of inheritance-related errors.
    a. Make sure that your header file does a **#include** of the BaseClass's header file.
2. At this point, your class is successfully deriving (also called subclassing or extending) the `BaseClass`. However, it is not a concrete class since it contains pure virtual functions (the ones it inherited from our base class). The easiest way to get rid of the pure virtual functions is to copy their signatures into our class declaration and get rid of the **= 0**. This will re-declare those functions as being concrete. After doing this for all of the pure virtual functions we are left with a concrete derived class declaration.
    a. Note that when re-declaring the pure virtual functions as concrete the **virtual** keyword is optional. Once a function is declared as virtual in the base class, it is always virtual in all derived classes. It is however customary to leave the **virtual** keyword in the signature of the derived classes to serve as a reminder to the programmer that will come after you.
3. Although we have successfully declared a concrete derived class, we cannot use it yet since our functions aren't defined. To amend this, we can proceed as usual by defining our functions in the CPP file or directly coding them into the header file. A common mistake with defining virtual functions in the CPP file is to repeat the **virtual** keyword. This will result in compilation errors. **virtual** should only appear in the header files.
4. You may notice that a lot of `BaseClass`es define a virtual destructor with an empty body. Before turning in this assignment you should be able to convince yourself why this is necessary despite the fact that the destructor has no work to do (so it would appear that the default destructor would suffice).

FifoPointAgenda class functional specification:
This class should derive from the PointAgenda class via public inheritance. Since the methods provided are the same as the PointAgenda class, they are not listed again here. Follow the above instructions while creating this concrete derived class. Note that in addition to the methods from PointAgenda, this class should also have one private data member for keeping track of our agenda in a first-in-first-out manner (HINT: do you know of a FIFO container?). Since all functions definitions will be short (no more than 1 line), it is advised that you inline them in the header file [to inline the code in the header file you simply replace the semicolon after each method header with a pair of curly braces containing the desired code]. This class should not perform any dynamic memory allocations so the compiler generated Big-3 will suffice. Note: this is not a template class; this class only holds Point objects.

LifoPointAgenda class functional specification:
This class should derive from the PointAgenda class via public inheritance. Since the methods provided are the same as the PointAgenda class, they are not listed again here. Follow the above instructions while creating this concrete derived class. Note that in addition to the methods from PointAgenda, this class should also have one private data member for keeping track of our agenda in a last-in-first-out manner (HINT: do you know of a LIFO container?). Since all functions definitions will be short (no more than 1 line), it is advised that you inline them in the header file [to inline the code in the header file you simply replace the semicolon after each method header with a pair of curly braces containing the desired code]. This class should not perform any dynamic memory allocations so the compiler generated Big-3 will suffice. Note: this is not a template class; this class only holds Point objects.

Polymorphism at work:
Since both the FifoPointAgenda and LifoPointAgenda classes are derived from the PointAgenda class, any FifoPointAgenda or LifoPointAgenda object can be treated as a PointAgenda object. We take advantage of that fact in the following manner: The supplied main program asks the user what kind of agenda they want to use and then it dynamically creates an instance of the appropriate class (the dynamic creation is done via a pointer). The pointer is then treated as a PointAgenda pointer, which allows us to use the methods of the PointAgenda class, regardless of which type of object the pointer actually refers to.

Step VI:
Now that we have learned about the provided classes and created our two point agenda classes, we are ready to go and explore a maze. You are to write a class called `MazeSolver`.

MazeSolver class functional specification:
This class will provide the necessary support to explore a maze from a given start position until the ending position is found. Define two files `MazeSolver.h` and `MazeSolver.cpp` that implement the class. It must support the following public operations:

MazeSolver(Maze& newMaze, PointAgenda& newAgenda)	The only class constructor. It accepts a Maze and a PointAgenda, both as reference parameters. It should simply store the parameters into ***private reference variables*** of the corresponding type. Note the private variables should be reference variables (declared with an ‘&’) to avoid making copies of the parameters, and since the PointAgenda class is a pure virtual class (and thus it cannot be instantiated).
bool solve(bool trace)	This method will explore the maze that was provided when the MazeSolver object was created. It will use the provided PointAgenda to store points that still need to be investigated. The method should list all Points as they are visited and print a total count of visited Points when done. It should also print an appropriate message indicating if the ending point is reachable or unreachable (and returning true or false respectively). More details are provided below. The trace parameter can be used to visually show the progress of the maze exploration. This is optional – you can ignore this parameter if you want (but you must still declare it).

Besides the above public methods, your MazeSolver class should define appropriate private helper methods as needed. Remember that good programming styles says that we should not create large monolithic methods, but rather create many small helper methods.

The meat of the class will be the writing of the solve method (and associated helper methods), which together will determine whether a maze has a solution---that is, whether you can get from the start to the finish (without jumping over any walls). The algorithm one usually follows goes something like this: start at the start location, and trace along all possible paths to (eventually) all reachable open squares. If at some point you run across the finish, then the maze is solvable. If not, it wasn't.

A rough outline is as follows: Begin by adding the start location to the agenda (the agenda keeps track of all the places we still want to visit). You then repeatedly grab & remove a location from the agenda and see if you have visited it before. If so, there is no need to explore it again, otherwise you explore it. Exploring it means you first print its location. Then check if it is the finish square, in which case you can terminate the search. Otherwise you compute all the adjacent locations (north, east, west, & south – no diagonal moves) that are inside the maze and aren't walls, and add them to the agenda for later exploration. Finally record the fact that you've visited this location so you won't ever have to explore it again. If the agenda ever becomes empty before you find the finish, then the finish is unreachable and you can terminate the search. When you are done, report whether the maze was solvable or not and how many locations were visited. Return true if the maze was solvable, otherwise return false.

Note that this strategy is entirely agnostic as to what kind of agenda you use. The main program allows the user to select an agenda type, but your solver method does not know what type of agenda it has received. You simply write your code more abstractly in terms of the Agenda operations. Additional note: we are not using recursion to explore the maze, rather we are maintaining our own agenda of points that we still need to visit.

The solve method should print out all Points as they are visited, and print out a total count of visited Points when it completes. See the sample execution for an example.

Notes:

1. You may assume that any well-formed maze will have exactly one start and exactly one finish. You may not assume that all valid mazes will be entirely enclosed with walls.

Going above & beyond (optional):
Sometimes it is a little easier to visualize the execution of your maze solver if you can see which locations are visited as they are being visited. You can do this by calling the `printVisitedMaze()` method during each step of your solver. To make this useful, you want to print the maze in the same location on the screen (you can do this by clearing the screen before printing). And to prevent it from moving along too fast, you can do a system call that causes your program to pause temporarily (you can adjust the length of the pause if desired). Here is a snippet of code that works with CLion (in debug mode), Visual Studio and Xcode (requires `#include <thread>`):

```
#if defined _WIN32
      system("cls");
#elif defined __APPLE__
      system("clear");
#endif
      myMaze.printVisitedMaze();
      std::this_thread::sleep_for(std::chrono::milliseconds(100));
```

If you add this code to your solver, be sure to only execute it if the trace parameter to the solve() method is true. If the trace parameter is false, your solver should not produce the visualization, but only produce the list of locations visited. One side effect of this visualization is that it interferes with our maze tracing (the list of locations visited). If visualization is turned on, only one trace point will be displayed per screen image. That is the expected behavior when visualization is turned on. You do not need to keep track of all visited points and display them all at once when the program is done. Rather than seeing a complete list of points visited, we will instead see which points have been visited by viewing the displayed maze.

Logistics:

- **Electronic Turn-in:** You should turn in the following files, named as follows:
  - `MazeSolver.h & MazeSolver.cpp`
  - `Stack_t.h & Stack_t.cpp`
  - `Queue_t.h & Queue_t.cpp`
  - `FifoPointAgenda.h & LifoPointAgenda.h` (and corresponding .cpp files if defined)
  - a **README** document (a .txt text file or a .doc Word document), containing the answers to each write-up question below.

- Each file you turn in should have the standard block comments at the top of the file, including an honor statement**.** Be sure to use good programming style.
- To use the templated classes, a user should only have to a #include of the .h files. The .h files should do a #include of the corresponding .cpp files as discussed in class and described in the instructions above. If you fail to configure your templated classes in this manner, our grading script will not work with your code, and your grade will be deducted accordingly.
- Do not submit any of the supplied files: Main.cpp, Point.cpp|.h, Maze.cpp|.h, or PointAgenda.h.

Electronic turn-in should be done via the assignment page in Brightspace. Be sure to attach all the required files prior to hitting the "Submit" button. Please **do not** zip up your entire project directory for submission, as the file would be extremely large – just submit the required files listed above.

Write-up Questions:

Answer the following questions in the README file (either txt or doc) you are to submit for grading. Your answers to these questions will constitute **10% of your grade** for this project. Since this is worth 10% of your grade, I expect well written responses, using full sentences and proper English. This is a form of project report, similar to a report written for a physics lab or biochemistry lab – please treat it seriously. Your writing skills in this lab write-up are being evaluated as a part of our department's ABET accreditation efforts, and your document will be graded accordingly.

1. State your name and email address.
2. After reviewing this spec and the provided files, please estimate/report how long you think it will take you to complete it.
3. How many hours did you actually spend total on this assignment?
4. Who/What did you find helpful for this project? If you received assistance from a person, who were they and what assistance did they provide?
5. Did you access any other reference material other than this project description and the class text? Please list them.
6. How did you test that the final program is correct? How did you test that your stack and queue classes are correct?
7. Which agenda method, FIFO or LIFO, seems to be faster at finding a solution to a given maze? Justify your answer.
8. If you failed to mark maze locations as visited, how would it have affected your ability to determine if a solvable maze was indeed solvable? Would a LIFO solver still be able to correctly determine if a maze was solvable for all solvable mazes? Would a FIFO solver still be able to correctly determine if a maze was solvable for all solvable mazes?
9. What did you enjoy about this assignment? What did you hate? Did we provide too much code or not enough code to make it interesting?
10. Was this project description lacking in any manner? Please be specific.
11. Do you have any suggestions for improving this assignment?
12. Any other information you would like to include.

Final notes:

1. Each file you turn in should have the standard block comments at the top of the file**.** All text files should have your name on the first line; your name should appear in the comment at the beginning of each source code file you turn in.
2. All methods should be properly commented with pre- & post-conditions. These comments should appear in both the .h file and the .cpp file.
3. Be sure to use good programming style. Style will be a larger portion of your grade on this assignment.

4. Since `Stack_t.cpp` and `Queue_t.cpp` are not specified as a member of the CLion project (because they are not meant to be compiled separately), the IDE does not pay attention when those files are changed. Thus when you edit one of the files and then “Build” your project the compiler will not automatically recompile files that depend upon those template .cpp files. If you change either of those two source files, you should perform a complete rebuild of your entire project so that all files are recompiled. You can specify a rebuild Run→Clean, followed by Run→Build.

Text files on Mac vs. PC:

Please note that the text maze files distributed with the project are DOS/Windows text files. These files can cause problems for Mac/Linux users since the two systems represent text files differently. In particular, they use different characters to represent the end of each line. If you are a Mac user and encounter difficulties with the provided DOS text files, follow these steps to convert them to Mac text files:

1. Open up TextEdit and create a new file
2. Click the "Format" menu and choose "Make Plain Text"
3. Open up the maze and copy it into the new file
4. At the beginning of every line, hit backspace (to get rid of the old newline character) then hit enter to put in the Mac newline character
5. Save the new file

Advanced Mac users can use utilities such as `sed` to do the conversion, but it requires installing new utilities onto your computer and thus is not described here.

Sample execution:

Note that your execution may not exactly mimic what is displayed below, since the order in which you add adjacent locations to an agenda affects the final execution.

```
MAZE SOLVER!!

Please enter name of file containing the maze: test1.txt
Here is the maze to be solved:
##########
# #      #
# # #### #
# # #* # #
# # ## # #
# #    # #
# # #### #
# ## o## #
#        #
##########

Please select the type of agenda you want to use by entering its number:
1: Stack-based agenda
2: Queue-based agenda
Enter choice: 1

Solving the maze with a stack-based agenda:

Do you want to trace the execution of the solver? (Y|N)
n

(5,2)->(4,2)->(4,1)->(5,1)->(6,1)->(7,1)->(8,1)->(8,2)->(8,3)->(8,4)->(8,5)->(8,
6)->(8,7)->(8,8)->(7,8)->(6,8)->(5,8)->(4,8)->(3,8)->(3,7)->(3,6)->(3,5)->(3,4)-
>(4,4)->(5,4)->(6,4)->(6,5)->(6,6)->(5,6)->Solution found!
Number of nodes visited: 29
```

```
Do you want to solve another maze? (Y|N)
y
Please enter name of file containing the maze: test1.txt
Here is the maze to be solved:
##########
# #      #
# # #### #
# # #* # #
# # ## # #
# #    # #
# # #### #
# ## o## #
#        #
##########

Please select the type of agenda you want to use by entering its number:
1: Stack-based agenda
2: Queue-based agenda
Enter choice: 2

Solving the maze with a queue-based agenda:

Do you want to trace the execution of the solver? (Y|N)
n

(5,2)->(5,1)->(4,2)->(4,1)->(6,1)->(3,1)->(7,1)->(2,1)->(8,1)->(1,1)->(8,2)->(1,
2)->(8,3)->(1,3)->(8,4)->(1,4)->(8,5)->(1,5)->(8,6)->(1,6)->(8,7)->(1,7)->(8,8)-
>(1,8)->(7,8)->(6,8)->(5,8)->(4,8)->(3,8)->(3,7)->(3,6)->(3,5)->(3,4)->(3,3)->(4
,4)->(5,4)->(6,4)->(6,5)->(6,6)->(5,6)->Solution found!
Number of nodes visited: 40

Do you want to solve another maze? (Y|N)
n
```

Acknowledgments:

This assignment is based on an assignment from the Nifty Assignments archive that was originally created by Don Blaheta of Knox College.