EECS 281 – Fall 2010

Project 1 – Michael's Escape

Assigned: Monday, September 13
Due: Monday, September 27 at 11:55:00 p.m.

Honor Code

As with any project/homework/exam in this class, you must abide by The Engineering Honor Code. Remember, all work submitted should only be work done by you and only you. If you are unsure about the level of allowed collaboration, please consult Professor Jagadish, the GSI, or the IAs.

Overview

Michael is trapped in the prison and is trying to find a way out. Tonight is a good opportunity to escape because Michael is doing a night shift in the storage rooms. His friends outside have dug a tunnel that connected to the last storage room. They are going to destroy the power supply tonight to create a short time period for Michael to escape. Michael has to find the tunnel before the power is back.

Evaluation

In this project you will develop a number of algorithms to help Michael find the tunnel. You will implement the algorithms in C++, and will be graded on your program's correctness, performance and programming style.

The Blueprint

You will be given a map of the storage rooms of the prison. The map will be formatted as a text file with the following characters.

- '`.`' represents open floor, where Michael can walk safely.
- '`@`' represents obstacles, which Michael should avoid.
- '`I`' is the entrance door, where Michael enters each room.
- '`O`' is the exit door, where Michael leaves each room.
- '`T`' is the tunnel, which Michael can use to escape the prison for good.

All of the rooms except for the last room have exactly two doors, one entrance door, and one exit. The last room has one entrance door and one tunnel.

The rooms are not necessarily the same size. If Michael walks into the obstacles in any of the rooms, he will alert the guards and get caught.
A simple room might look like:

```
@@@@@@@
@@....@
I...@@@
@.....@
@@@@@.@
@T....@
@@@@.@@
```

Unfortunately, Michael cannot see anything in the dark, so he can only move in the North, South, East, or West directions. That is, *he may not travel along the diagonals of the room.* Your job is to find a route for him to the tunnel and output this route with a sequence of directions in "`N`", "`S`", "`E`", "`W`". Michael starts his journey at the entering door in the first room. Every time he leaves the room through an exit door, he will begin at the entering door in the next room.

Specifications

Michael knows that there are `K` rooms. He will search for the exit door in the first `K-1` rooms, and search for the tunnel in the last room.

Once Michael finds the exit to a room, he must take one additional step to exit the room, depending on which edge of the room he is at. If there are multiple directions in which he could exit, then choose in order of "`NSEW`".

If Michael cannot find an exit in the first `K-1` rooms, or the tunnel in the last room, then the escape simply fails.

Routing Approach

You will develop two routing approaches for Michael:

1. Find the first valid path using a *queue*.
2. Find the first valid path using a *stack*.

Queue-Based Approach

First, *enqueue* Michael's start position and then do the following:

1. *Dequeue* the next location.
2. *Enqueue* all of the walkable tiles ('.') immediately North, South, East, and West (in that order) of the location you just dequeued. Do not enqueue any tiles that have already been enqueued.

3. Check to see if any of these spaces is connected to the tunnel or is an exit door. If none of them do, go back to Step 1. Remember: Michael cannot move diagonally, so spaces to the Northwest, Northeast, Southwest, and southeast are not considered next to each other and should not be enqueued in this step.

4. Once you have found the tunnel, you still need to guide Michael to the tunnel. Remember, Michael needs to finish before the power is back and therefore *does not want to visit the same square twice*. That is, the path you give to Michael should not contain any loops or backtrack onto itself.

Stack-Based Approach

This will be similar to the queue-based approach except that you will *push* and *pop* the locations instead of *enqueue* and *dequeue* them.

Note that the approaches given here are high-level. You will need to expand upon them to finish this project.

Input and Output Formats

The input file is a text-based map describing the layout of each room. The first line contains an integer indicating the number of rooms (suppose the number is `K`). The rest of the file is divided into `K` sections describing each room. In the first line of each section, the size of the room (`M` rows by `N` columns) is specified as a pair of integers, separated by a single space. The following `M` lines will contain exactly `N` characters for each. For example, the following is a legal input file (though it may be physically unrealistic):

```
2
3 3
@.@
..O
I.@
3 3
..I
.T@
...
```

The output will consist of `K` lines, where `K` is the number of rooms. Each line is a sequence of moves in directions of "`N`", "`S`", "`E`", "`W`", and must end with a "`\n`" character. For instance, the queue-based output for the above map might look like:

```
NEEE
WS
```

If there is no possible way for Michael to escape without knocking over obstacles, print "`Escape failed.\n`" to standard output.

Stacks, Queues, and STL

While you are free to implement your own stack and queue classes, you are encouraged to use the STL's implementations, which will save you substantial programming time and most likely decrease the overall runtime of your program (STL containers have been highly optimized over many years). Helpful links on the STL can be found at SGI STL Reference (`http://www.sgi.com/tech/stl/`) and cppreference (`http://www.cppreference.com/`).

Command Line Arguments

We will run your program with one of the commands:

```
./escape --Stack < [testfile]
./escape --Queue < [testfile]
```

Legal command line inputs include *exactly one* of `--Stack` and `--Queue`. If `--Stack` is present, then use the stack-based approach, and use the queue-based approach if `--Queue` is present. If both are present, neither is present, or if any other command line arguments are included, then print an informative error message to `stderr` and terminate the program by calling `exit(1)`.

(See `http://en.wikipedia.org/wiki/Main_function_%28programming%29` for more information on using command line arguments.)

The notation `< [testfile]` is file input redirection. This means that your program will treat the contents of `[testfile]` as if it were coming in through `cin`. For this reason, you should use `cin` to read input, and `cout` to display output. Do *not* use `ifstream` or `ofstream`.

(See `http://en.wikipedia.org/wiki/Redirection_%28computing%29` for more information on input/output redirection.)

Errors To Check For

If you encounter any of the following errors, print an informative message to `stderr` and then exit using `exit(1)`. In this case, do not print anything to `stdout`.

- In the input file format, you should check for:
  - Illegal characters ('`T`' is illegal in the first `K-1` rooms and '`O`' is illegal in the last room)

  - Maps that are incomplete (not enough characters/line or not enough lines)
  - Files that don't start with a positive number in the first line.
  - Room-sections that don't start with a pair of positive numbers.
  - Rooms other than the last that have extra lines at the end.
  - Rooms that do not have exactly one entering door.
  - Any of the first K-1 rooms that does not have exactly one exit door.
  - Last room that does not have exactly one tunnel.
- In the command line, you should check for:
  - Not having exactly one of `--Stack`, `--Queue`.
  - Having any extra arguments (anything other than `./escape`, `--Stack`, or `--Queue`).

Errors Not To Check For

- Ignore all extra characters on a line, or lines after the very last one needed.
- You may assume that all numbers will be less than the maximum value that an integer can hold (no need to worry about integer overflow).
- Positive numbers will *not* start with a '`+`' sign.
- We will always include '`< [testfile]`' in the command line.
- If a door exists, it will always be on an edge.

Testing

Testing your code to see if it produces the correct solution is essential. We highly recommend that you test your code thoroughly with test cases beyond what we have given. For instance, your code may have a bug that the given examples do not uncover or your code works with a 4×5 grid but not a 10×12. Try to have a variety of maps to make sure that your code is robust and does not have hidden bugs. For example, have small-, medium-, and large-sized maze grids, mazes that are long or wide, and maze that are sparsely-populated (very few obstacles) and densely-populated (many obstacles).

Public Test Cases

The first two test cases on the autograder are published here and won't be graded. It is strongly recommended that you test your program against these cases yourself before submitting to the autograder. If your code fails these test cases, you will not do well.

Test #1

Input: `test0.txt`

```
2
3 3
I..
...
..O
2 2
I.
.T
```

Command Line: `./escape --Stack < test0.txt`

Output:

```
EESSS
ES
```

Test #2

Input: `test1.txt`

```
2
3 3
@.@
..O
I.@
3 3
..I
.T@
...
```

Command Line: `./escape --Queue < test1.txt`

Output:

```
NEEE
WS
```

Note: You are encouraged to use informative print statements for debugging purposes. Statements such as:

```
Approach: Queue-based
Number of Maps: 2
```

could be very helpful in determining exactly what your code is doing.

However, make sure to remove them before submitting, or your program will be marked as incorrect.

File Separation and Makefile

Division of code into separate files that perform different tasks is an essential part of good coding style. For example, using one file for input/output, and another for your algorithm could make bugs much easier to find and could make it easier to write your code in the first place.

For this project, it is required that you use at least one header file and two `.cpp` files. Because each of you will separate your code differently and use different file names, we also require that you include a makefile, called '`Makefile`' to make the compilation process efficient.

Your makefile must have the following properties:

- It must create a working executable called '`escape`'.
- It must create the correct object (`.o`) files.
- When '`make clean`' is run, all object files and the '`escape`' executable are removed.

SVN Repository

You need to set up an SVN repository for your project. You are required to do this in the beginning and use it to keep all of your work during the entire project development. A full subversion log file named "`svn.txt`" is required in the project submission. Make sure that you have at least five commits in the log file in order to gain full points on this part. An easy way to create this file is to run

```
svn log -v > svn.txt
```

SVN will be covered in Discussion 2, the week of September 13–17.

How To Submit Your Project

You can start making submissions to the autograder once you think your program works. The autograder will test your program on its own set of test cases and then report the results back to you. The content of the test cases will not be available for you to see. We highly recommend you also write your own test cases and test your program. Keep in mind that the autograder is on a Red Hat Linux machine (same as the CAEN machines). Although all the code should compile and work on any system, sometimes weird things happen. Thus, we recommend that you test and compile (and develop) your code in Linux on the

CAEN machines and not in other platforms. Do all of your project work with all needed files in some directory other than your home directory. This will be your "submit directory". When you are ready to submit your final version, make sure:

- You have deleted all .o files and all executable(s) and that typing "`make clean`" accomplishes this.
- Your makefile is called `Makefile`.
- Typing the command "`make`" will result in an executable called `escape`.
- You have no other files besides what you need.
- Your code compiles and runs correction using version 4.1.2 of the g++ compiler. This is the standard version on the CAEN Linux systems (such as login.engin.umich.edu). Even if everything seems to work on another operating system or with different versions of GCC, the course staff will not support anything other than GCC 4.1.2 running on Linux.

Turn in the following files:

- all your .h and .cpp files
- Makefile
- svn.txt

You must prepare a compressed tar archive (a `.tar.gz` file) of all of your files to submit to the autograder. One way to do this is to have all of your files for submission (and nothing else) in one directory. Go into this directory and run this command:

```
tar czf ../submission.tar.gz *
```

This will prepare a suitable file in the parent of your working directory. Submit your tar ball directly to the autograder at: `https://grader8.eecs.umich.edu`.

You may submit up to **3** times a calendar day to autograder for feedback. For this purpose, a calendar day begins at midnight (12:00:00 am) and ends at 11:59:59 pm Ann Arbor local time. However, there might be delay over the internet between the time you submit and the time the autograder receive it. We recommend that you upload your last submission of the day before 11:55:00 pm. It's unlikely to have a delay larger than five minutes. If you submit more than 3 times, the autograder will acknowledge that you have submitted but will not provide you with feedback. If you do not receive feedback within 30 minutes, do not panic. This delay depends on the load on the autograder. It is likely to be greatest in the hours immediately before the deadline. If you have any problems with the autograder, email Scott Reed at reedscot@umich.edu. Thus, if possible, submit early!

Coding Style

Your coding style will be graded on what warnings it gives you (and how many) and a hand-grading of the code. Some suggestions to improve your score:

- Use a reasonable structure for your program:
  - Find a fairly reasonable class structure.
  - Don't lump everything into one class or one file. If ideas are clearly separate, then put them in separate files.
  - Make use of constructors and destructors (like when deallocating memory).
  - Use multiple .cpp and .h files as appropriate. For example, class declarations should go into the header files (.h) and their subroutines should go into standard code files (.cpp).
  - In general, if your code seems sloppy to you, then it probably is. Try and write code that you and others can understand.
- Use informative and concise comments:
  - Explain what each class does and what each of its member functions do.
  - A 1-2 line explanation is useful for non-standard coding syntax.
  - A useless comment is worse than no comment at all. For example,

    ```
    int temp; // creates temp variable
    ```

    is not helpful.
- Formatting your code:
  - Use indentations when you have if/for/while/general code blocks.
  - Avoid going past 80 characters per line.
- Variable names:
  - Use reasonable and informative names.
  - Using variables "`i`" and "`j`" are acceptable for LCVs (loop control variables), but consider using other names for long-term data.
- Code Reuse: No large segment of code should appear in more than one place. For example, the only difference between the stack-based and the queue-based algorithms is the use of a stack versus a queue. See if you can find a way to implement the structures such that a stack or a queue can be "plugged in" according to the command line switches.

Runtime Evaluation

This project will be graded somewhat on speed. We will post benchmark times for selected test cases and will deduct points if you do not meet these times. Specifically, you will lose 10% of the grade for that test case if your program takes between 1 and 3 times as long as the benchmark. You will lose 20% if it is between 3 and 10 times worse, 33% if between 10 and 100 times worse, and all credit if greater than 100 times worse. These points are deducted per test case but are not calculated until after the project is due.

You may **NOT** use compiler optimization (the O3 flag, for example) in this project. Use of compiler optimization will result in an automatic 33% penalty (as if your code ran $>$ 10 times slower) on all timed test cases.

Grading

The grading for Project 1 is out of 100 points:

- 80 points - A working (passing all test cases) stack and queue based implementation of the program. (partial credits for passing each test case)
- 10 points - Coding style.
- 5 points - `Makefile`.
- 5 points - `svn.txt`.

Submission Time and Late Policy

The project is officially due at 11:55:00 on Monday, Sep 27, 2010. You have two free late days in total for this semester. An additional late day will be charged at the cost of 1% of your total course grade. (Each project is 8% of your total courses grade). You can use at most 2 late days on each project regardless of whether they are free late days or charged late days.

Hints and Advice

- **Start early!** The earlier you start, the more time you have to write your code, ask questions, and debug.
- Print debugging statements as you develop. Just make sure to remove them before submitting.
- Good Luck!