p1-rescue-the-countess

EECS 281: Project 1 - Rescue the Countess

Due Tuesday, May 14, 2024 at 11:59 PM

Overview

Marco of the Fungus Province has just received news that Louser has taken Countess Cherry captive in his dark, maze-like castle. With his twin brother, Luigino, Marco has traveled many miles to reach the castle, dodging flying shells and jumping on angry mushrooms. Now that he has entered the castle, he needs your help to rescue the Countess. Using C++, you must implement some basic path finding algorithms to guide Marco to the Countess and foil Louser's evil plans!

Starter Files

The tarball, p1-rescue-the-countess-files.tgz, contains a Makefile, a project identifier, a listing of valid error messages, various text files (.txt) containing sample inputs and outputs, and a README about the sample files.

Castle Layout

You must help Marco navigate through Louser's castle. The castle contains up to 10 rooms, with warp pipes to travel between them. The rooms are all squares of the same size.

Your program will be given a map or description of the castle that uses the following symbols:

- `.` walkable room space
- `#` walls (which cannot be walked on or through)
- `!` one of Louser's minions (which cannot be walked on or through)
- A digit `0-9` which indicates a warp pipe traveling to the same spot in the room with the corresponding number
- `S` Marco's starting location
- `C` location of Countess Cherry

You can assume, for the files that we give you, that there is exactly one location for Countess Cherry and exactly one starting point for Marco. You still might want to verify this, just in case an input file that you create breaks this rule.

Because you have a copy of the castle map, your task is to plan a path for Marco from the starting position (S) to the Countess (C) that does not pass through any walls (#) or minions (!). Marco can walk next to the minions, but cannot walk on top of them. Marco doesn't have time to get in a fight today!

Rooms are to be treated as if they are surrounded by walls; you are to treat both walls and room edges as impassable terrain.

Marco can discover new locations north, east, south or west from any room space (.) as well as the starting location; **no discoveries may be made diagonally**. Your path planning program must check that it does not move Marco off of the map, onto walls (#), or onto minions (!).

In addition, when Marco steps onto a warp pipe (digits 0-9), he will only be able to discover the location in the same row and column of the room corresponding to the warp pipe's number. If Marco steps on a warp pipe with digit, d, at row and column (r, c), in any room, then he'll be able to discover (r, c) in the room d. It is possible for this to continue from warp pipe to warp pipe, if there are multiple rooms with warp pipes in the same location.

If room d doesn't exist, no new location can be discovered. You'll need to check for this. For example, a castle with 2 rooms might have a pipe labeled 5, but room 5 does not exist, so Marco won't use this pipe.

Input File Format

The program gets a castle map from a file that will be read from standard input (cin). This input file is a simple text file specifying the layout of the castle. We require that you make your program compatible with two input modes: map (M) and coordinate list (L).

The reason for having two input modes is that a large percentage of the runtime of your program will be spent on reading input or writing output. Coordinate list mode exists so that we can express very large maps with a minimal amount of input data; map input mode exists to express maps whose input files are also easily human readable.

For Both Input Modes (M and L):

- The first line of every input file will be a single character specifying the input mode, M or L . Unlike the output mode, which is given with the Command Line Options, input mode is found only in the input file.

- The second line will be a single integer $1 \le R \le 10$ indicating the total number of rooms in the castle.

- The third line will be a single integer $1 \le N$, indicating the length of the sides of every room in the castle (all rooms are $N \times N$ squares).

You can assume that N (and thus a row or a column index) will fit inside of an unsigned integer variable (unsigned int or uint32_t).

Comment lines begin with // and they are allowed anywhere in the file after the first three lines. When developing your test files, it is good practice to place a comment on line 4 describing the nature of the castle in the test file. Any castles with noteworthy characteristics for testing purposes should be commented. The

map file is also easier to read if there is a comment line identifying the room number before the description of that room, but this is not required. Comments are allowed in either input mode.

Additionally, there may be extra blank/empty lines at the end of any input file, your program should ignore them. If you see a blank line in the file, you may assume that you have hit the end.

Map Input Mode (M):

For this input mode, the file should contain a map of each room, in order from 0 (bottom) to $R - 1$ (top).

An example map mode input file for a castle with two 4x4 rooms:

```
Example map mode input file for a castle with two 4x4 rooms
 1   M
 2   2
 3   4
 4   //sample M mode input file with two 4x4 rooms
 5   //castle room 0
 6   .C..
 7   ....
 8   ...S
 9   #.1#
10   //castle room 1
11   ....
12   #...
13   .!..
14   #...
```

Coordinate List Input Mode (L):

A coordinate list input file will contain a list of coordinates for at least all walls, minions, and warp pipes in the rooms. Only one coordinate should appear on a given line. We do not require that the coordinates appear in any particular order. A coordinate is specified in **precisely** the following form: `(room,row,col,character)`. The row and col positions range from 0 to $N - 1$, where N is the value specified at the top of the file. By default, all unspecified coordinates within the rooms are of type `.` (room space); however, it is still valid to redundantly specify them.

Here is a valid `L` mode input file that describes a castle identical to the one previously specified in `M` mode:

```
Example coordinate list mode input file for a castle with two 4x4 rooms
 1   L
 2   2
 3   4
 4   //sample L mode input file, two 4x4 rooms
 5   //castle room 0
 6   (0,0,1,C)
```

```
 7   (0,1,0,.)
 8   (0,2,3,S)
 9   (0,3,0,#)
10   (0,3,2,1)
11   (0,3,3,#)
12   //castle room 1
13   (1,1,0,#)
14   (1,2,1,!)
15   (1,3,0,#)
16   (1,3,3,.)
```

Invalid coordinates (for a castle with two 4x4 rooms):

```
1   (2,1,2,#)      -- Room 2 does not exist!
2   (1,4,2,.)      -- Row 4 does not exist!
3   (0,1,5,!)      -- Col 5 does not exist!
4   (0,0,1,F)      -- F is an invalid map character!
```

Routing Schemes

You are to develop two routing schemes to help Marco get from the starting location to the location of Countess Cherry:

- A queue-based routing scheme
- A stack-based routing scheme

Be sure to read the document titled "Project 1 the STL and You"; it has some excellent tips, plus a way to implement the two different routing schemes with a single search container (the deque<>).

In the routing scheme use a search container of locations within the castle. First, initialize the algorithm by adding the start position S to the search container. Then, loop through the following steps:

1. Remove the next position from the search container.
2. If that position has a warp pipe, add the corresponding position that the pipe leads to, i.e., the same row/col position but with the specified room number. As noted below, only add this position if it has not been added to the search container before.
3. If the position is not a warp pipe, then add all locations adjacent to the location you just removed that are available to move into (walkable room space, warp pipes, or the Countess's location). **Add any locations that you are allowed to move to from your present location in the following order: north, east, south, and west.**
4. As you add any of these spaces (step 2 or 3) to the search container, check to see if any of them is the location of the Countess C ; if so, stop searching for a path.

If the search container becomes empty before you reach the Countess C , the search has failed and there is no path to rescue the Countess.

Whenever you add a position to the search container, you have "discovered" it. ***Do not discover any position more than once***. Remember that from a walkable room space tile `.` or your starting location `S` you can only discover locations north, east, south, or west. Warp pipes are the only way of moving between rooms.

When visiting a warp pipe, you still have to check that you haven't added its target location to the data structure when you see it. This will help you prevent Marco from following warp pipe loops. Also, make sure that Marco doesn't warp onto an enemy or into a solid wall! This is considered unhealthy.

The program must run to completion within 35 seconds of total CPU time (user + system) - programs running for longer than this period will receive 0 points on that test case. In most cases 35 seconds is much more time than you should need; even if you finish before 35 seconds, you may still receive no points because your solution is too slow. See the [time manpage](#) for more information (this can be done by entering `man time` at the command line). We may test your program on very large maps (up to several million locations). Be sure you are able to navigate to the Countess in large castle maps within 35 seconds. Smaller castle maps should run MUCH faster.

Libraries and Restrictions

Unless otherwise stated, you are allowed and **encouraged** to use all parts of the C++ STL and the other standard header files for this project. You are not allowed to use other libraries (eg: boost, pthread, etc). You are not allowed to use the `shared_pointer` or `unique_pointer` constructs from the memory include file (we want you to learn proper use of pointers yourself). You are not allowed to use the C++11 regular expressions library (it is not fully implemented in gcc 6.x) or the thread/atomics libraries (it spoils runtime measurements).

Output File Format

The program will write its output to standard output (`cout`). Similar to input, we require that you implement two possible output formats. Unlike input, however, the output format will be specified through a command line option `--output`, or `-o`, which will be followed by an argument of M or L (M for map output and L for coordinate list output). See [Command Line Options](#) for more details.

There is only one acceptable solution per routing scheme for each castle. For both output formats, you will show the path you took from start to finish. If no valid route exists, then regardless of output mode, the program should print only

```
    No solution, N tiles discovered.
```

(followed by a newline), where N is the number of tiles discovered while searching, regardless of the output mode. If this occurs, do not output the starting location or `Path taken:` line.

Map Output Mode (M):

For this output mode, you should first print the starting location (see the example below). Then print the map of the castle rooms, replacing existing characters as needed to show the path you chose. Beginning at the starting location, overwrite the characters in the path with `n`, `e`, `s`, `w`, or `p` to indicate which tile Marco

moved to next. All pipes Marco used in the final path should be overwritten by `p`, regardless of what room they lead to or from. Do not overwrite the location of the Countess `C` at the end of the path, but do overwrite the `S` at the beginning. For all spaces that are not a part of the path, simply reprint the original room map space. You should only create comments to indicate the room numbers and format them as shown below, all other comments from the input files should be omitted.

Thus, for the sample input file specified earlier, using the queue-based routing scheme and map output mode, you should produce the following output:

```
 1   Start in room 0, row 2, column 3
 2   //castle room 0
 3   .Cww
 4   ...n
 5   ...n
 6   #.1#
 7   //castle room 1
 8   ....
 9   #...
10   .!..
11   #...
```

Using the same input file but with the stack-based routing scheme, you should produce the following output:

```
 1   Start in room 0, row 2, column 3
 2   //castle room 0
 3   eC..
 4   n...
 5   nwww
 6   #.1#
 7   //castle room 1
 8   ....
 9   #...
10   .!..
11   #...
```

Coordinate List Output Mode (L):

For this output mode, you should print only the coordinates that make up the path you traveled. You should print them in order, from (and including) the starting position to the `C` (but you should not print the coordinate for `C`). The coordinates should be printed in the same format as they are specified in coordinate list input mode (room,row,col,character). The character of the coordinate should be `n`, `e`, `s`, `w` or a `p` to indicate spatially which tile is moved to next (as with map output, the `p` indicates that a warp pipe was taken). You should discard all comments from the input file, but you should add one line, just before you list the coordinates of the path that says "Path taken:".

The following are examples of correct output in coordinate list mode that reflect the same solution as the map output mode above.

For the queue solution:

```
1    Path taken:
2    (0,2,3,n)
3    (0,1,3,n)
4    (0,0,3,w)
5    (0,0,2,w)
```

For the stack solution:

```
1    Path taken:
2    (0,2,3,w)
3    (0,2,2,w)
4    (0,2,1,w)
5    (0,2,0,n)
6    (0,1,0,n)
7    (0,0,0,e)
```

Be aware that input and output modes are independent of each other. Each test can use either input mode and either output mode. They may or may not be the same.

Command Line Options

Your program should take the following case-sensitive command line options (when we say a switch is "set", it means that it appears on the command line when you call the program):

- `--stack` , `-s` : If this switch is set, use the stack-based routing scheme
- `--queue` , `-q` : If this switch is set, use the queue-based routing scheme
- `--output (M|L)` , `-o (M|L)` : Indicates the output file format by following the flag with an M (map format) or L (coordinate list format). If the `--output` option is not specified, default to map output format (M), if `--output` is specified on the command line, the argument (either M or L) to it is required. See the examples below regarding use.
- `--help` , `-h` : If this switch is set, the 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)` or `return 0;` from `main()` .

When we say `--stack` , or `-s` , we mean that calling the program with `--stack` does the same thing as calling the program with `-s` . See getopt for how to do this.

Legal command line arguments must include exactly one of `--stack` or `--queue` (or their respective short forms `-s` or `-q`). If none are specified or more than one is specified, the program should print an informative message to standard error (`cerr`) and call `exit(1)` .

Legal Command Line Examples

```
$ ./superMarco --stack < infile > outfile
```
Run the program using the stack algorithm and map output mode

```
$ ./superMarco --queue --output M < infile > outfile
```
Run the program using the queue algorithm and map output mode
```
$ ./superMarco --stack --output L < infile > outfile
```
Run the program using the stack algorithm and coordinate list output mode

Notice that we are using input and output redirection here. While we are reading our input from a file and sending our output to another file in this case, we are NOT using file streams! The < command redirects the file specified by the next command line argument to be the standard input (`cin`) for the program. The > command redirects the output of the program (things sent to `cout`) to be printed to the file specified by the next command line argument. The operating system makes calls to `cin` to read the input file and it makes calls to `cout` to write to the output file. Come to office hours or post on Piazza, if this is confusing!

Illegal Command Line Examples

```
$ ./superMarco --queue -s < infile > outfile
```
Contradictory choice of routing (stack and queue)
```
$ ./superMarco < infile > outfile
```
No routing mode (stack or queue) specified

Testing Your Solution

It is extremely frustrating to turn in code that you are "certain" is functional and then receive half credit. We will be grading for correctness primarily by running your program on a number of test cases. If you have a single silly bug that causes most of the test cases to fail, you will get a very low score on that part of the project *even though you completed 95% of the work*. Most of your grade will come from correctness testing. Therefore, it is imperative that you test your code thoroughly. To help you do this we will require that you write and submit a suite of test files that thoroughly test your project.

Your test files will be used to test a set of buggy solutions to the project. Part of your grade will be based on how many of the bugs are exposed by your test files. (We say a bug is exposed by a test file if the test file causes the buggy solution to produce different output from a correct solution.)

Each test file should be an input file that describes a castle in either map or coordinate list format. Each test file should be named test-n-flags.txt, where $1 \le n \le 15$. The "flags" portion should include a combination of letters of flags to enable. Valid letters in the flags portion of the filename are:

- s: Run stack mode
- q: Run queue mode
- m: Produce map mode output
- l: Produce list mode output

The flags that you specify as part of your test filename should allow us to produce a valid command line. For instance, don't include both s and q, but include one of them; include m or l, but if you leave it off, we'll run in map output mode. For example, a valid test file might be named test-1-sl.txt stack mode, list output). Given

this test file name, we would run your program with a command line similar to the following (the autograder will use a random combination of long and short options, such as `--output` instead of `-o`):

```
$ ./superMarco -s --output L < test-1-sl.txt > test-1-output.txt
```

Test files may have no more than 10 levels, and the size of a level may not exceed 8x8. You may submit up to 15 test files (though it is possible to get full credit with fewer test files). The test cases the autograder runs on your solution are NOT limited to 10x8x8; your solution should not impose any size limits (as long as sufficient system memory is available).

Errors you must check for

A small portion of your grade will be based on error checking. You must check for the following errors:

- More or less than one `--stack` or `--queue` or `-s` or `-q` on the command line. You may assume the command line will otherwise be correct (this also means that we will not give you characters other than `M` or `L` to `--output`).
- Input errors: illegal map characters, such as a `Z` in the map (you should check in both input modes).
- For L input mode, you must check that the room, row, and column numbers of each coordinate are all valid positions.

In all of these cases, print an informative error message to standard error (`cerr`) and call `exit(1)`. In the starter files, look at the file named Error_messages.txt. If you output any one of those standard error messages when the input is actually valid, the autograder will repeat the error message back to you. This can be helpful when your program thinks that a valid input is invalid, because you can more easily figure out what went wrong.

You do not need to check for any other errors.

Safe Assumptions

- We will not put extra characters after the end of a line of the map or after a coordinate.
- All coordinates in coordinate list input mode will be in the format (room,row,col,character).
- There will be exactly one start location `S` and exactly one location of the Countess `C` in the map.
- We will not give you the same coordinate twice for coordinate list input mode.
- The input mode and the integer dimensions of the castle on lines two and three at the beginning of the input file will be by themselves, without interspersed comments, and they will be correct.

Submission to the Autograder

Do all of your work (with all needed files, as well as test files) in some directory other than your home directory. This will be your "submit directory". Before you turn in your code, be sure that:

- Every source code and header file contains the following project identifier in a comment at the top of the file: `// Project Identifier: B99292359FFD910ED13A7E6C7F9705B8742F0D79`

- The Makefile must also have this identifier (in the first TODO block).

- You have deleted all .o files and your executable(s). Typing `make clean` shall accomplish this.

- Your makefile is called Makefile. Typing `make -R -r` builds your code without errors and generates an executable file called `superMarco`. The command line options `-R` and `-r` disable automatic build rules, which will not work on the autograder.

- Your Makefile specifies that you are compiling with the gcc optimization option `-O3`. This is extremely important for getting all of the performance points, as `-O3` can often speed up code by an order of magnitude. You should also ensure that you are not submitting a Makefile to the autograder that compiles with the debug flag, `-g`, as this will slow your code down considerably. If your code "works" when you don't compile with `-O3` and breaks when you do, it means you have a bug in your code!

- Your test files are named `test-n-flags.txt` and no other project file names begin with test. Up to 15 test files may be submitted.

- The total size of your program and test files does not exceed 2MB.

- You don't have any unnecessary files or other junk in your submit directory and your submit directory has no subdirectories.

- Your code compiles and runs correctly using the g++ compiler. This is available on the CAEN Linux systems (that you can access via 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 running on CAEN Linux. At the moment, the default version installed on CAEN is 4.8.5, however we want you to use version 6.2.0 (available on CAEN with a command and/or Makefile); this version is also installed on the autograder machines.

Turn in all of the following files:

- All your .h, .hpp, and .cpp files for the project
- Your Makefile
- Your test files

You must prepare a compressed tar archive (.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. Our Makefile provides the command `make fullsubmit`. Alternately you can go into this directory and run this command:

```
$ COPYFILE_DISABLE=true tar -cvzf ./submit.tar.gz *.cpp *.h *.hpp Makefile test*.txt
```

This will prepare a suitable file in your working directory.

Submit your project files directly to either of the two autograders at: https://g281-1.eecs.umich.edu or https://g281-2.eecs.umich.edu. **When the autograders are turned on and accepting submissions, there will be an announcement on Piazza.** The autograders are identical and your daily submission limit will be shared (and kept track of) between them. You may submit up to three times per calendar day (more during the Spring). For this purpose, days begin and end at midnight (Ann Arbor local time). **We will count only your best submission for your grade.** If you would instead like us to use your LAST submission, see the

autograder FAQ page, or use this form. **If you use an online revision control system, make sure that your projects and files are PRIVATE; many sites make them public by default! If someone searches and finds your code and uses it, this could trigger Honor Code proceedings for you.**

Please make sure that you read all messages shown at the top section of your autograder results! These messages will help explain some of the issues you are having (such as losing points for having a bad Makefile or using disallowed libraries).

Grading

All grading will be done by the autograder, with points distributed as follows.

80 points – Your grade will be primarily based on the correctness of your algorithms. Your program must have correct and working stack and queue algorithms and support both types of input and output modes. **Additionally:** Part of your grade will be derived from the runtime performance of your algorithms. Fast running algorithms will receive all possible performance points. Slower running algorithms may receive only a portion of the performance points.

10 points – No memory leaks. Solutions that are proven to run (passing certain autograder test cases) will also be run through valgrind, to check for memory leaks at exit. This is something you should do yourself, to check for memory leaks as well as a number of other issues that valgrind can enumerate. Valgrind is available at the CAEN command prompt, and can also be downloaded to run on Linux-based personal environments.

10 points – Testing and Bug Finding (effectiveness at exposing buggy solutions with user-created test files).

Empirical Efficiency

We will check for empirical efficiency both by measuring the memory usage and running time of your code and by reading the code. We will focus on things such as whether you use unnecessary temporary variables, whether you copy data when a simple reference to it will do, and whether you use an $O(n)$ algorithm or an $O(n^2)$ algorithm.

Coding Style

Although your project will not be graded on style, style is a very important part of programming and software development. Among other things, good coding style consists of the following:

- Clean organization and consistency throughout your overall program
- Proper partitioning of code into header and cpp files
- Descriptive variable names and proper use of C++ idioms
- Effective use of library (STL) code
- Omitting globals, unnecessary literals, or unused libraries
- Effective use of comments
- Reasonable formatting - e.g. an 80 column display

- Code reuse/no excessive copy-pasted code blocks
- Effective use of comments includes stating preconditions, invariants, and postconditions, explaining non-obvious code, and stating big-Oh complexity where appropriate

It is **extremely helpful** to compile your code with the gcc options: `-Wall -Wextra -pedantic -Wconversion` . This will help you catch bugs in your code early by having the compiler point out when you write code that is either of poor style or might result in behavior that you did not intend.

Hints and Advice

- Design your data structures and work through algorithms on paper first. Draw pictures. Consider different possibilities before you start coding. If you're having problems at the design stage, come to office hours. After you have done some design and have a general understanding of the assignment, re-read this document. Consult it often during your assignment's development to ensure that all of your code is in compliance with the specification.
- Always think through your data structures and algorithms before you code them. It is important that you use efficient algorithms in this project and in this course, and coding before thinking often results in inefficient algorithms.
    - If you are considering linked lists, be sure to review the lecture slides or measure their performance against vectors first (theoretical complexities and actual runtime can tell different stories).
- Only print the specified output to standard output.
- You may print whatever diagnostic information you wish to standard error (`cerr`). However, make sure it does not scale with the size of input, or your program may not complete within the time limit for large test cases.
- If the program does find a route, be sure to have `main()` do a `return 0` .
- If the input is valid but no route exists, also have `main()` do a `return 0` .
- This is not an easy project. **Start it immediately!**

Have fun coding!