EECS 281 – Fall 2021
Programming Assignment 1
Back to the Ship![1]

Due Thursday September 23, 11:59pm

Project Identifier

You <u>MUST</u> include the project identifier at the top of every file you submit to the autograder as a comment. This includes all source files, header files, and your Makefile (in the first TODO block):

```
// Project Identifier: 950181F63D0A883F183EC0A5CC67B19928FE896A
```

Overview

You have just broken out of the detention level of a large and strangely moon-like space station. You need to find your way back to your old spacecraft of questionable space-worthiness to escape. Your adorable robot friend has hacked into the elevator system of the space station to assist you in your escape.

Space Station Layout

The space station is composed of up to 10 square levels with elevators to travel between them. A description of the space station will be provided by an input file using a 3-D coordinate system and special characters. The special characters and what they represent:

- `'.'` floor space
- `'#'` walls (the only character that cannot be walked on/through)
- `'S'` your starting location at the detention level
- `'H'` the hangar of the spacecraft location
- `'E'` corresponds to elevators that transport you from that location to the same row and column of the levels that have an elevator on the same location

You can travel north, east, south or west from any 'floor space' tile as well as your starting location. You may not travel diagonally. Elevators allow you to travel to an elevator located in the same row and column on any other level. You may **only** travel between levels via elevators.

Some levels may not be enclosed by walls; you are to treat level edges and walls as impassable terrain.

Ultimately, your task will be to indicate your route to the hangar tile from your starting location using directional indicators ('n' for north, 'e' for east, 's' for south, 'w' for west, and 0-9 to indicate where you got off the elevator). More details on this are provided below.

Input file format

The program gets its description of the space station from a file that will be read from standard input (cin). This input file is a simple text file specifying the layout of the space station. We require that you make your program

[1] Credits: David Paoletti

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 graphs with relatively few lines of a file, map input mode exists to express dense graphs (ones for which most of the tiles are not just floor space) in the smallest number of lines. You should use `sync_with_stdio(false)` - this makes a very significant runtime difference.

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 on the command-line (see below), this is part of the file.**
The second line will be a single integer indicating the number of levels in the space station, whose value will be between 1 and 10.
The third line will be a single integer `N`, indicating the size of each (and every) level of the space station (each level is `NxN` in size and all levels are the same size). The "Assumptions you may make" tells you how large this number might be; also see the Optimization Tips file on Canvas.

Comments may also be included in any input file. 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 space station in the test file. Any levels with noteworthy characteristics for testing purposes should also be commented. By convention, a comment line identifying the level number is placed before the map of that level. 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 (but do not have to check for this).

Map input mode (M):

For this input mode, the file should contain a map of each level, in order. The levels are specified beginning with the lowest level and working up. **The lowest level is level 0**; that is to say, the levels are 0-indexed.

A valid `M` mode input example file for a space station that has two 4x4 levels:

```
M
2
4
//sample M mode input file with two 4x4 levels
//level 0
.H..
....
E..S
#..#
//level 1
....
#...
E#..
#...
```

Copy/pasting text from a PDF file may yield unexpected results! PDF files contain elisions (invisible characters) that may cause your code (or the autograder) to behave in unexpected ways. You should either retype the test file manually on your machine or use an editor such as vim or emacs to see (and remove) invisible characters.

Coordinate list input mode (L):

The file should (after the first three lines) contain a list of coordinates for *at least* all non-floor space coordinates in the space station. 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: `(level,row,column,character)`. The `row` and `column` positions range from `0` to `N-1`, where `N` is the value specified on line 3 of the file. By default, all unspecified coordinates within the space station are of type `'.'` (floor space); however, it is not invalid to redundantly specify them to be so.

Valid coordinates (for a space station with three 4x4 levels):

`(0,0,1,#)`	
`(2,2,2,H)`	
`(1,1,3,.)`	-- While it is valid to specify floor space, it is redundant

Invalid coordinates (for a space station with three 4x4 levels):

`(9,1,2,#)`	-- level 9 does not exist
`(3,4,2,.)`	-- row 4 does not exist
`(2,1,5,#)`	-- column 5 does not exist
`(0,0,1,F)`	-- 'F' is an invalid map character

Here is a valid `L` mode input file that describes levels that are identical to those that the sample `M` input file did:

```
L
2
4
//sample L mode input file, two 4x4 levels
(1,1,0,#)
(1,2,1,#)
(1,3,0,#)
(0,0,1,H)
(0,2,0,E)
(1,2,0,E)
(0,2,3,S)
(0,3,0,#)
(0,3,3,#)
```

Routing schemes

You are to develop two routing schemes to get from the starting location to the hangar location tile:

- A routing scheme that uses a queue as the search container
- A routing scheme that uses a stack as the search container

Both of these routing schemes will always lead you to the hangar (if a path exists). Whether you're supposed to be using stack or queue based, the algorithm is basically the same. Read the "Project 1 the STL and You" file, it will tell you (among many other useful things) how to use a single data structure, a `deque`, and allow you to write one version of the code for both routing schemes.

We're going to use a few terms here, and try to be consistent about them in the project specification, office hours, Piazza, etc.

- *Search container* (the deque) is where you add things to help you find the hangar. Think of this as the places that might help you get to the hangar, but haven't had a chance to *investigate* yet.
- *Discovered* means that this square has been added to the search container. You can never *discover* the same location twice (this prevents infinite loops and other problems). You must keep track of what has been discovered and what has not.
- *Investigate* is when a location comes out of the deque, and you start to determine what other locations it helps you to *discover*. There is no need to track what has or has not been investigated.

Create a search container. Before you start looping, add the starting location to the search container, and mark it as discovered. As long as the search container is not empty, do the following:

1. Take the "next location" from the search container; we will refer to this as the current location. Remove it from the search container.
2. Investigate all of the locations that you can reach from the current location. For each one, if it is off the edge, a wall, or already discovered before, ignore it. Otherwise, add it to the search container and mark it as discovered. You should check in this order: N, E, S, W. If the current location is an elevator, you should **also** investigate every other level, from the lowest level (0) to the highest level, in the same row and column as the current location: if any other levels have an 'E' in the same row/column, and they are not discovered, each should be added to the search container and marked as discovered.
3. As soon as you find the hangar tile `H`, you should stop searching. If you don't stop, loop back to step 1.

The only difference between an elevator and a non-elevator tile is what positions you are allowed to discover. You still have to check that you haven't discovered a location before discovering it.

One important point to remember: when we talk about the routing scheme, adding things to the search container, etc.: you (the person) are not moving yet! Until your trusty robot companion uses this program to find the route that will get you to the hangar, you stay put. If you start asking questions like "how did I teleport from level 0, row 1, column 2 to level 3, row 5, column 4?", remember that YOU didn't move yet. The program investigated a different location that just happened to be far away from the last location it investigated. Once your robot is done running the program, it will produce output (a map or list of directions) for you to follow, and that set of directions will make sense for you: you'll go north, go east, take an elevator to level 2, go south, etc.

The program must run to completion within 35 seconds of total CPU time (user time). In most cases 35 seconds is more time than you should need. See the **`time`** manpage for more information (this can be done by entering “man time” to the command line). We may test your program on very large space stations (up to several million locations). Be sure you are able to navigate to the hangar tile in large space stations within 35 seconds. Smaller space stations should run MUCH faster.

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 the section on command line arguments below for more details.

For both output formats, you will show the path you took from start to finish.

Map output mode (M):

For this output mode, you should print the map of the levels, replacing existing characters as needed to show the path you took. Beginning at the starting location, overwrite the characters in your path with `'n'`, `'e'`, `'s'`, `'w'`, or `'0-9'` (where you got off the elevator) to indicate which tile you moved to next. Elevator tiles that you use in your final path should be overwritten with the number of the level where you exited the elevator. Do not overwrite the `'H'` 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 space station space. *You should discard all existing comments from the input file for the output file*. However, do create comments to indicate the level numbers and format them as shown below. Before any other output, state the starting location, so that the map is easier to follow.

Thus, for the sample input file specified earlier, using the **queue**-based routing scheme and map (`M`) style output, you should produce the following output:

```
Start in level 0, row 2, column 3
//level 0
.Hww
...n
E..n
#..#
//level 1
....
#...
E#..
#...
```

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

```
Start in level 0, row 2, column 3
//level 0
eH..
n...
nwww
#..#
//level 1
....
#...
E#..
#...
```

We have highlighted the modifications to the output in red to call attention to them; do not attempt to color your output (this isn't possible, as your output must be a plain text file).

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 'H' (but you should not print the coordinate for 'H'). The coordinates should be printed in the same format as they are specified in coordinate list input mode (the `(level,row,column,character)` format). The character of the coordinate should be `'n'`, `'e'`, `'s'`, `'w'`, or `'0-9'` (where you got off the elevator) to indicate spatially which tile is moved to next. You should discard all comments from the input file, but you should add one comment on line 3, just before you list the coordinates of the path that says `"//path taken"`. There's no need to state the starting location in this mode, since it would be the first line after the `"//path taken"`.

The following are examples of correct output format in (L) coordinate list mode that reflect the same solution as the Map output format above:

For the queue solution:

```
//path taken
(0,2,3,n)
(0,1,3,n)
(0,0,3,w)
(0,0,2,w)
```

For the stack solution:

```
//path taken
(0,2,3,w)
(0,2,2,w)
(0,2,1,w)
(0,2,0,n)
(0,1,0,n)
(0,0,0,e)
```

There is only one acceptable solution per routing scheme for each map of the space station. If no valid route exists, the program should simply reprint the space station with no route shown for Map output mode, and should have no coordinates listed after the “`//path taken`” comment in List output mode.

The mode of input and output can differ. That is, the input mode may be List mode, but the output may be requested in Map mode and vise-versa. They may also be the same (but are not guaranteed to be).

Command line arguments

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 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 return 0 from main. We don't grade this command line option, but it's a standard practice for command-line programs.
- 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_long()`** for how to do this.

Legal command line arguments must include exactly one of --stack or --queue (or their respective shortforms -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 exit(1).

Examples of legal command lines:
- `./ship --stack < infile > outfile`
  - This will run the program using a stack and map output mode.
- `./ship --queue --output M < infile > outfile`
  - This will run the program using a queue and map output mode.
- `./ship --stack --output L < infile > outfile`
  - This will run the program using a stack and coordinate list output mode.

We are using input and output redirection here. While we are reading our input from a file and sending out output to another file in this case, we are NOT using file streams! The operating system “connects” `cin` to read the input file, and when you send output to `cout`, the operating system redirects it to the output file. Come to office hours if this is confusing!

Examples of illegal command lines:
- `./ship --queue -s < infile > outfile`
  - Contradictory choice of routing
- `./ship < infile > outfile`
  - You must specify either stack or queue

Test files

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 files. If you have a single silly bug that causes most of the test cases on the autograder 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 suite 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 space station in either map (`M`) or coordinate list (`L`) format. Each test file file should be named *test-n-flags.txt* where 0 < n <= 15 for each test file. 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 (we might use long or short options, such as --output instead of -o):

```
./ship -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 tests the autograder runs on your solution are **NOT** limited to 10x8x8; your solution should not impose any size limits smaller than 65,535 (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:

- Input errors: illegal map characters, such as `F`. Check this for both map/list input modes.
- For coordinate list input mode, you must check that the row, column, and level numbers of each coordinate are all valid.
- 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`' following `--output` or `-o`).

In all of these cases, print an informative error message to standard error and exit(1). Read the file `Error_messages.txt` for standard error messages and why to use them. **You do not need to check for any other errors.**

Assumptions you may make

- You may assume we will not put extra characters after the end of a line of the map or after a coordinate.
- You may assume that coordinates in coordinate list input mode will be in the format `(level,row,col,character).`
- You may assume that there will be exactly one start location 'S' and exactly one spacecraft hangar tile 'H' in the space station.
- You may assume that we will not give you the same coordinate twice for the coordinate list input mode.
- You may assume the input mode line and the integer dimensions of the space station on lines two and three at the beginning of the input file will be by themselves, without interspersed comments, and that they will be correct.
- If you see a single '/' character at the beginning of a line, you can assume it is a comment. This makes reading the list mode input MUCH easier.
- You can assume that we will never give you negative numbers for the number of levels, the size of the map, or any portion of a coordinate in list input mode. You can assume that all of these numbers will fit within an `unsigned int` or `uint32_t` variable.

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: 950181F63D0A883F183EC0A5CC67B19928FE896A`
- The Makefile must also have this identifier (in the first TODO block).
- DO NOT copy the above identifier from the PDF! It might contain hidden characters. Copy it from the `Identifier.txt` file from Canvas instead.
- 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 `puzzle`. 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:

```
tar czvf ./submit.tar.gz *.cpp *.h *.hpp Makefile test*.txt
```

This will prepare a suitable file in your working directory.

Note that if you’re using the `Makefile` that we provided as part of Project 0 and Lab 1, most of these things will be done for you! When you want to prepare a submission, you can use either `'make fullsubmit'` or `'make partialsubmit'.` The difference is that a “full submit” includes test files, a “partial submit” does not. Use the command `'make help'` to get our `Makefile` to tell you everything it can do!

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/. 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 use your best submission for final grading**. 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).

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 regular expressions library (it is not fully implemented on gcc) or the thread/atomics libraries (it spoils runtime measurements). Do **not** use the STL's unique or shared pointers.

Grading

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. A leader board will be posted on the course website, and will be updated frequently during the project. You may track your progress relative to other students and instructors there.
10 points -- Test file coverage (effectiveness at exposing buggy solutions).
10 points -- Your program does not lose any memory as reported by valgrind. Memory that is “still reachable” is fine, but any of the three types of “lost” will lose points.

Grading will be done by the autograder.

Coding style

Your grade will not be directly based upon your coding style. However, a better coding style will make your code easier to write, debug, and get help on. 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 global variables, 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. 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

Go watch the Project 1 Ship Tutorial video on the EECS 281 YouTube Channel.

- 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.
  - We haven't gotten to Lecture 7 yet, but when we do you'll see that linked lists are almost never the answer in EECS 281.
  - Don't do linear searching if you don't have to.
- Only print the specified output to standard output.
- You may print whatever diagnostic information you wish to standard error. 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 return 0. If the input is valid but no route exists, also have main return 0.
- Do not `exit(0)` for a valid input file! The `exit()` function bypasses normal program shutdown, meaning that object destructors are not called. Thus `valgrind` will report memory as lost if those objects (who never got destructed) had used new to allocate dynamic memory.
- The only times you should use `exit()` are for the "Errors you must check for" (and possibly `-h` or `--help`). Having no solution is not an "error", it's just something that sometimes happens.
- *This is not an easy project.* ***Start it immediately!*** Remember what your fellow students said in the Computing CARES video: you don't have to start coding immediately, but start understanding the project, planning your work, etc.

Appendix A - Another Sample

An additional simple test file containing an elevator that must be used. The input is on one page, followed by queue output, and stack output, each on a separate page.

Map Input:

```
M
2
4
//sample using elevator
//level 0
...E
....
....
H...
//level 1
...E
....
....
S...
```

List Input:

```
L
2
4
//sample using elevator
(1,0,3,E)
(1,3,0,S)
(0,0,3,E)
(0,3,0,H)
```

Map Output (Queue):

```
Start in level 1, row 3, column 0
//level 0
...s
...s
...s
Hwww
//level 1
eee0
n...
n...
n...
```

List Output (Queue):

```
//path taken
(1,3,0,n)
(1,2,0,n)
(1,1,0,n)
(1,0,0,e)
(1,0,1,e)
(1,0,2,e)
(1,0,3,0)
(0,0,3,s)
(0,1,3,s)
(0,2,3,s)
(0,3,3,w)
(0,3,2,w)
(0,3,1,w)
```

Map Output (Stack):

```
Start in level 1, row 3, column 0
//level 0
swww
s...
s...
H...
//level 1
...0
...n
...n
eeen
```

List Output (Stack):

```
//path taken
(1,3,0,e)
(1,3,1,e)
(1,3,2,e)
(1,3,3,n)
(1,2,3,n)
(1,1,3,n)
(1,0,3,0)
(0,0,3,w)
(0,0,2,w)
(0,0,1,w)
(0,0,0,s)
(0,1,0,s)
(0,2,0,s)
```

Appendix B - Autograder Test Cases

Each test case on the autograder has a unique name which gives some information about how the test was run. The number within the test name is meaningless, but the prefix and suffix provide information that you will find useful when debugging your program:

Spec*: One of the tests from this project specification. An ‘E’ indicates the file from Appendix A (elevator needed), otherwise it is the input file from page 2. An ‘L’ or ‘M’ indicates the input file format.
INV*: An invalid test case (see the “Errors you must check for” section).
S*: A “small” input file, usually hand-written, possibly tricky.
M*: A “medium” input file, generated by a program.
B*: A “big” input file, also generated by a program.
*s: Run with the stack flag, map output.
*S: Run with the stack flag, list output.
*q: Run with the queue flag, map output.
*Q: Run with the queue flag, list output.

When you start submitting test files to the autograder, it will tell you (in the section called "Scoring student test files") how many bugs exist, the number needed to start earning points, and the number needed for full points. It will also tell you how many are needed to start earning an extra submit/day!