EECS 281 – Winter 2020
Programming Project 1
Puzzle Solver

Due Monday February 3, 2020, 11:59pm

Project Identifier

EECS 281 Project 1: Puzzle Solver
Version 01-13-20
Credits: Nathan Fenner, David Paoletti

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: A8A3A33EF075ACEF9B08F5B9845569ECCB423725
```

Introduction: The Puzzle

EECS281Games™ is developing a new puzzle game that they want to bring to the market in the next few weeks. They've already built the engine and designed thousands of levels, but they realized that they're not sure how to solve many of them! What's worse, they think some of them are unsolvable!

You are tasked with developing a C++ program that will take as input a sample map and some command-line flags indicating how your program will behave. You will process the map, check that the input is valid, then output information about the puzzle's solvability and a description of the puzzle's solution (if it has any).

The game is played on a grid, filled with open space and walls that block player movement. Here is a small sample input that your program could receive:

```
2 4 7
// A simple example puzzle
// 2 colors (A, B)
// 4x7 grid
@..A..b
.a.#B##
####...
?..B.^^
```

The goal of the game is for the player to get from their starting location (marked with `@`) to the target (marked with `?`), by pressing buttons (`a`, `b`, ..., `z` and `^`) that open and close doors (`A`, `B`, ..., `Z`) that stand in their way.

The player's available moves are to travel one tile north, east, south, or west (up, right, down, or left) as long as there's no wall (`#`) or closed door (`A`, `B`, ..., `Z` when closed) in their way. The player cannot move diagonally, or step off of the grid (you can imagine the specified map is surrounded by impassable walls).

The valid map symbols:

- `@` is the player starting position. A valid input has exactly one @.
- `?` is the target position. A valid input has exactly one ?.
- `.` represents an open space that the player can stand in.
- `#` represents a wall. The player cannot move into a #.
- `A`, `B`, `C`, ..., `Z` are *doors*. The player can only move onto them when they are *open*. They all start *closed*. The letter marking the door is called its "color" (in the game, the doors are displayed with different colors).
- `a`, `b`, `c`, ..., `z` are *buttons*. The player can always move onto them, which causes them to *trigger* when they are *active*. Buttons all start *active*. Each button opens the corresponding type of door (an `a` opens the `A` doors, `b` opens the `B` doors, `c` opens the `C` doors, etc.), and simultaneously closes all other types of doors.
- `^` is a *trap*. It's like a special button that starts *inactive*. Stepping on a trap closes *all* doors.

Doors and Buttons

The maps aren't just simple mazes -- they include special puzzle elements called "buttons" and "doors". When the player moves onto an *active* button (e.g. `b`), it gets triggered:

- `B` doors open; all other doors close
- `b` buttons become inactive; all other buttons and traps become active.
- Informally, buttons open doors of the same color and close everything else.

When the player moves onto an *active* trap (^), it gets triggered:

- All doors close.
- All buttons become active; all traps become inactive.
- Informally, traps close all the doors.

Example Input

```
2 4 7
// A simple example puzzle
// 2 colors (A, B)
// 4x7 grid
@..A..b
.a.#B##
####...
?..B.^^
```

With the game's rules in mind, let's see how the above puzzle can be solved:

1. Walk east and south to the `a`, which opens the `A` doors.

2. Walk north and then 5 east through the `A` door into the upper right room and stand on the `b` button, which closes the `A` doors and opens the `B` doors.
3. Walk 2 west, 3 south (going through the first `B` door), then west (through the second `B` door) until you reach the target (avoiding the traps, which would have closed the `B` doors).

Solving Puzzles

Your program will attempt to find a solution to the puzzle, and present it as output. The exact details of this process will be described in the next section, but **it won't make sense unless you read this section first**. Our goal is to take a map as input and find a sequence of moves (moving north, east, south, and west, or triggering buttons/traps) that bring us to the target, or confirm that this isn't possible. There's no obvious strategy to solving these puzzles- we have to make a lot of decisions and the "best" decision in each case is not obvious or clear.

So instead of trying to cleverly reason about how to best solve these puzzles, *we'll just try to make every possible move and see where that gets us.* This is possible because of a few observations relating to the player's *state*.

The player has a **position** in the grid (`row`, `col`) (the top left corner is position (`0`, `0`) and the bottom right is (`height-1`, `width-1`)). The player's initial position is wherever the `@` symbol is in the input grid. In addition, there is an active button **color**: the color of the last trap/button pressed. The initial color is `^`, since all doors start closed.

The player's complete **state** is the combination of their **color** and **position** (row and column).

In other words, you can think of the player as being inside of a **3D maze, instead of a 2D maze**. The number of "rooms" in this maze is equal to the *number of colors* + 1. Room `0` is the `^` room. Room `1` is where you go after pressing button `a`; room `1` has no `A` walls. Room `2` is where you go after pressing button `b`, etc. Room `0` has no traps; the traps exist in all other rooms. Think about saving memory here; see the "Tips" section later on.

The player's state tells us *everything* about where they are and whether they can win.

First, *how* the player got to their state `S` doesn't affect whether or how they can win; only the state `S` itself (and the map) does. If you know how to get from state `S` to the target, then *any* method to get from the start to state `S` will work.

Secondly, there are at most (number of colors) * width * height possible states.

Together, these two observations lead to the following strategy:

- We know how to get to the initial location (we start there).
- If we know how to get to a location S, and T is just one move away from S (for example, pressing a button or moving north/east/south/west) then we know how to get to T: get to S, then perform that one move. For example, if the current location is (row 0, col 0) and you move east, you arrive at location (row 0, col 1).
- If we repeatedly "learn" how to get to each possible state, then we will either learn a way to the target, or find out that one doesn't exist.

In more detail:

1. The initial state is “reachable”. Recall that a “state” is composed of a color, row, and column.
2. Create a *`reachable_collection`* that will hold the “reachable” states (it starts with just the initial state in it).
3. Take the “next state” from *`reachable_collection`*. Call this *`current_state`*.
4. Find the states *`next_state_1`*, *`next_state_2`*, etc. that you can reach in one step from *`current_state`*. Add them to *`reachable_collection`*, but only if they have never been added before.
5. Repeat from step 3 until you’ve found the target, or the *`reachable_collection`* is empty.

Based on the observations above, once we’ve completed this process, we will have either found the target, or every state that can possibly be reached will have been in the *`reachable_collection`* at some point. Thus, we can use it to decide whether or not the puzzle is solvable.

It’s very important that you understand *why* this works.

Notice that this tells us **which** tiles are reachable, so we know whether the solution exists, but it doesn’t tell us **how** to solve the puzzle. Since we’ll be interested in *how* to solve the puzzle too, we’ll have to modify step 4:

4. Find the states *`next_state_1`*, *`next_state_2`*, ..., that you can reach in one step from *`current_state`*. Add them to *`reachable_collection`*, but only if they have never been added before. When adding each state (*`next_state_1`*, *`next_state_2`*,, etc.) to the *`reachable_collection`*, record that you reached it from *`current_state`*.

Now, if we want to know how to get from the initial state to the target, we just “recall” which state we reached the target from. Then we recall which state we reached that state from. Then we recall which state we reached *that* state from. We repeat until we get all the way back to the initial state. We call this process *backtracking*.

*How **you** accomplish this is an implementation detail that is up to you. Part of this project is figuring out how and where you should store this information in your program.*

The Inputs

Your program will receive two forms of input, the map and command line flags:

The Map

The puzzle map will be received through standard input. **Your program will not be reading any files.** For your convenience, when you’re testing your program on CAEN (or on your computer) you can use a feature of your shell called *input redirection*, which lets you use the contents of a file to fill standard input. By writing

```
./puzzle < some_file.txt
```

the operating system (OS) will provide the contents of `some_file.txt` as the standard input (on `std::cin`) to `puzzle`. Your program just reads from `std::cin`, but it receives the contents of `some_file.txt`. You don’t have to write any code for this, the OS does it for you. Your program will receive a map like the following on standard input:

```
2 4 7
// A simple example puzzle
// 2 colors (A, B)
// 4x7 grid
@..A..b
.a.#B##
####...
?..B.^^
```

A valid map input starts with 3 integers: `num_colors height width`, separated by single spaces. `num_colors` is the number of different colors of doors that exist in the input map. These will be the first `num_colors` letters of the English alphabet. For example, if `num_colors` is 4 then the valid door and button symbols are `A, B, C, D` and `a, b, c, d` respectively. The trap symbol `^` is always valid, even if `num_colors` is `0`. In a valid input file, `num_colors` must be between `0` and `26`, inclusive.

In a valid input, `width` and `height` must both be at least `1`.

Following the first line, there can be 0 or more comments. Comments are lines that start with 2 forward slashes. They should be ignored by your program. Comments will not appear in the middle of the map.

Following the comments are the puzzle itself. The puzzle consists of `height` rows of text, each of `width` characters.

In a valid map input, both `@` and `?` appear exactly once.

Note that a puzzle with no solution can still be a valid input.

Sometimes, your program will receive invalid input, meaning input that doesn't follow some of these rules. Your program is required to detect and reject some types of these invalid inputs. See the "Input Guarantees and Error Checking" section for more on which kinds of invalid inputs you'll have to check for.

The Command Line Flags

In addition to receiving the map through standard input, your program will be given several options as command line flags. The first lab will help with details about handling command line flags.

The flags your program must support:

- `--help` (short: `-h`)
- `--queue` (short: `-q`)
- `--stack` (short: `-s`)
- `--output {TYPE}` (short `-o {TYPE}`) where `{TYPE}` is either the word `map` or `list`.

If your program receives the flag `--help` (long or short) then you should print a help message and return 0, to exit, without expecting any input (the exact thing you print doesn't matter, as we don't check for it).

The type of collection used in the algorithm will be specified by the presence of `--queue` or `--stack`. Exactly one of these should be provided to the program, exactly once. See the next section for details on what these do.

The type of output your program produces will be specified by `--output {TYPE}`. The `--output` flag takes an argument which should be one of `map` or `list`. These describe the kind of output your program should produce. If the `--output` flag isn't provided to your program, then your program should act as though `--output map` was provided. See the section on output for details on what these do.

Examples of valid command line calls:

- `./puzzle --queue -o map`
- `./puzzle -so map`
- `./puzzle --output list --stack`
- `./puzzle --help`
- `./puzzle -qo list`

If your program receives an invalid flag (that is, one that is different from those listed above), it should reject the input.

The Algorithm

Make sure you understand the "Solving Puzzles" section first!
The algorithm will depend on the command line arguments. If `--queue` is provided, then `<reachable_collection>` means "queue", and if `--stack` is provided, then `<reachable_collection>` means "stack". Think of the `<reachable_collection>` as the list of locations that you know you should consider, but haven't finished considering yet.

1. Initially, mark all states as not reachable.
2. Mark the start state as reachable and add it to the `<reachable_collection>`.
3. Loop while the `<reachable_collection>` is NOT empty.
4. Remove the "next" item from the `<reachable_collection>`. Call it `S`. This S consists of a color `c` plus `row` and `col` coordinate `(c, (row, col))`.
5. If `S = (c, (row, col))` is standing on a button `b` or trap of a different color than `c`, and `b` is not yet reachable, add `(b, (row, col))` to the `<reachable_collection>` and mark it as reachable.
6. If `S` is NOT a button, mark the following states as reachable and add them to the `<reachable_collection>` in the following order (provided that they are not off the edge of the map, and have not already been marked as reachable):
    a. north/up `(c, (row - 1, col))`
    b. east/right `(c, (row, col + 1))`
    c. south/down `(c, (row + 1, col))`
    d. west/left `(c, (row, col - 1))`
7. Repeat from step 3.

Remember that you can only add a state to the `<reachable_collection>` if it has not already been reached. When you mark something as reachable, remember to save enough information to know where you came from.

If at step 3 the `<reachable_collection>` is empty, you can stop the loop. Because there is nowhere left to look that you haven't already checked, there is no solution. If you find the target, you've found the solution, and can leave the loop. Trace the path backwards from the target using backtracking.

Even if your program finds some solution, if it's not the same as the one obtained by this algorithm, your program won't pass on the autograder, because your output will be different.

The Output

If there is a solution to the puzzle, the output depends on the output mode, which can be configured by the `--output` command line flag's argument.

- `list`: print the list of states in the solution path, in the form `(c, (row, col))` with `row` being the 0-index row of the state, `Y` being the 0-index column of the state, and `c` being a lowercase letter or the symbol `^` to indicate the active color. The first state printed should be the initial state, whose color is always `^`. The last state printed should be the target state, in whichever color is reached first by the algorithm.
- `map`: print the map, similar to the way it was given in the input (excluding the first line containing its dimensions and any comments). Print each color in a separate map, preceded by a comment indicating the color being output. Also make the following replacements: tiles on the solution path (excluding the initial `@` and final `?`) should be replaced by +, *triggered* buttons or paths should be replaced by `%` (instead of +), and the place that you arrive after triggering a button are replaced by `@` (instead of +). In the "`color ^`" map, traps should be displayed as open floor (`^` becomes `.`), in the "`color a`" map, button `a` and wall `A` should be displayed as floor (`a` and `A` become `.`), "`color b`" should display `b` and `B` as floor, etc.

If there is no solution to the puzzle, instead print the following, **regardless** of the output mode:
No solution.
Reachable:
followed by the map as it was provided as input, *but with all unreachable tiles replaced by #*. A tile is *unreachable* if the player cannot enter any state with that tile as its position. This includes open tiles, doors, buttons, and the target: if the player cannot reach a state where they are standing on the same tile as the door/button/target, then replace it with a #. If there is no solution to the puzzle, the target will therefore always be replaced by a #.

Let's look at some sample input/output. Given the same input from page 1 (continued on next page, so that the input and output are together):

```
2 4 7
// A simple example puzzle
// 2 colors (A, B)
// 4x7 grid
@..A..b
.a.#B##
####...
?..B.^^
```

Here's the map output when run with `--queue`. Look at what's shown in the map: we went from the starting location `@` in the upper left, east to the `+`, then south onto the `a` button, which was replaced by `%`. Where we arrive on color `a` is replaced by `@`, then the trail of `+` shows how we went north, east, and where we reach button `b` is replaced by `%`. Arriving on color b (the `@` sign) we go 2 west, 3 south, and west until reaching the target (`?`).

```
// color ^
@+.A..b
.%.#B##
####...
?..B...
// color a
.+++++%
.@.#B##
####...
?..B.^^
// color b
...A++@
.a.#+##
####+..
?++++^^
```

Here's the map output when run with `--stack`:

```
// color ^
@..A..b
+%.#B##
####...
?..B...
// color a
..++++%
.@+#B##
####...
?..B.^^
// color b
...A++@
.a.#+##
####+..
```

```
?++++^^
```

Notice that stack and queue output are almost the same, only differing in the first two moves. In a larger map, you would see more differences. Here's the list mode output for a queue:

```
(^, (0, 0))
(^, (0, 1))
(^, (1, 1))
(a, (1, 1))
(a, (0, 1))
(a, (0, 2))
(a, (0, 3))
(a, (0, 4))
(a, (0, 5))
(a, (0, 6))
(b, (0, 6))
(b, (0, 5))
(b, (0, 4))
(b, (1, 4))
(b, (2, 4))
(b, (3, 4))
(b, (3, 3))
(b, (3, 2))
(b, (3, 1))
(b, (3, 0))
```

What if there were no solution? Suppose we used the same input file, but replaced the B in the bottom row with either `#` or `A` (solid wall or color A door). Then stack or queue doesn't matter, nor does map or list output, the result would be:

```
No solution.
Reachable:
@..A..b
.a.#B##
####...
####.^^
```

Input Guarantees and Error Checking

A small portion of your grade will be based on error checking. Your program will be provided with deliberately invalid inputs, and if your program does not reject them, you will lose points.

If your program receives any invalid input (either from the map on `std::cin` or as command line flags) it should print a message to `std::cerr` (the error message is highly recommended to help you debug, but not required; the autograder will not "grade" the error message), then immediately `return 1` from `main()`, or call `exit(1)`. If the input is valid (even if there is no solution), your program should `return 0` from `main()`, and should NOT call `exit()` at all.

You cannot make any assumptions about the input, except for the following:

- The first line of the map will be 3 unsigned integers, each in the range 0 to 200,000, separated by spaces.
- The map and flags will contain only non-null ASCII characters.
- `num_colors * width * height < 400,000,000`
- If the `--output` (short: `-o`) flag is provided, then there WILL be some argument after it (but that argument might not be a valid value)

The errors you must check for are:

- `0 <= num_colors <= 26` (0 colors is valid, it just means that there are no doors)
- `1 <= width`
- `1 <= height`
- Exactly one of `--stack` and `--queue` are provided (long or short form)
- The argument to the `--output` flag (if the output flag is provided) is either `list` or `map`
- No invalid command line flags are provided (such as `-x` or `--eecs281`)
- No invalid door or button appears in the map (for example, if `num_colors` is 3, then `'z'` is invalid)
- No invalid characters appear in the map (for example, `'+'` can't appear in the map, but it could appear in a comment)
- `@` appears exactly once in the input map
- `?` appears exactly once in the input map

In all of these cases, print an informative error message to standard error and `exit(1)`.
You do not need to check for any other errors.

Buggy Solutions and Test Files

A portion of your grade will be based on creating test files that can expose bugs. In addition to the correct solution, the autograder has several buggy implementations of the project. Each buggy implementation is based on the instructor solution, but has small bugs (either based on implementation mistakes or misreadings of the specification).

Your goal is to submit test files that cause the buggy implementations to produce different output than the correct implementation. Each time you submit, you can include up to 15 test files to the autograder (though it is possible to get full credit with fewer test files). Each test file must be at most 10x10 (neither `width` nor `height` can exceed 10). The instructor solution will reject test files that are invalid (see the section on error checking). If you submit a test file that causes the instructor solution to exit 1, then that file will not be used to detect bugs.

Part of the reason to submit these test files is to create tests that you would like to know if your program gets the correct output. If you submit a test file that your program produces wrong output, for the first such file that the autograder encounters it will include your output and the correct output in your autograder feedback! So help the autograder help you, and submit test files of your own, early. Don't wait until the last minute to submit these.

Each test file should be a valid input file named *test-n-flags.txt* where 1 <= n <= 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, the autograder will run in `map` output mode. If `m` or `l` is present, it should follow the `s` or `q`. For example, a valid test file might be named `test-1-sl.txt` (stack mode, list output). Given this test file name, the AG would run your program with a command line similar to the following (it might use long or short options, such as `--output` instead of `-o`):

```
./puzzle --stack -o list < test-1-sl.txt > test-1-output.txt
```

When the autograder starts scoring your test files, it will display how many bugs exist, how many are required to start earning points, and how many you need to find to receive full points (the number required for full points will be less than the total number of bugs).

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:

- 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! Use `make debug` and `valgrind` to find it.
- 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 version 6.2.0 of 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 6.2.0 running on Linux. In order to compile with g++ version 6.2.0 on CAEN you must put the following at the top of your Makefile:

```
PATH := /usr/um/gcc-6.2.0/bin:$(PATH)
LD_LIBRARY_PATH := /usr/um/gcc-6.2.0/lib64
LD_RUN_PATH := /usr/um/gcc-6.2.0/lib64
```

Turn in all of the following files:

- All your .h and .cc or .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. Go into this directory and run this command:

```
dos2unix *; tar czf ./submit.tar.gz *.cpp *.h *.cc *.c 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/. **When the autograders are turned on and accepting submissions, there will be an announcement.** 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 (double that in the Spring). For this purpose, days begin and end at midnight (Ann Arbor local time). We will use your best submission for final grading. Part of programming is knowing when you are done (when you have achieved your task and have no bugs); this is reflected in this grading policy. We realize that it is possible for you to score higher with earlier submissions to the autograder; however this will have no bearing on your grade. We strongly recommend that you use some form of revision control (ie: GIT, SVN, etc.) and that you 'commit' your files every time you upload to the autograder so that you can always retrieve an older version of the code as needed. Please refer to your discussion slides and Canvas regarding the use of version control.

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`).

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.

Autograder Testcase Legend

There are a number of testcases visible to you on the autograder. Whenever you submit, your program will be run with each of these as input, and your output compared against the instructor solution's output. If they match, then you'll receive points for that testcase depending on how much time and memory your program

used. Each testcase receives full points if you're under on both time and memory, points are reduced faster the further your program goes over the limit for that testcase.

Most of the names end with `_QL`, `_QM`, `_SL` or `_SM`. These indicate the command-line arguments passed to the program: `Q` means `--queue`, `S` means `--stack`, `M` means `--output map` and `L` means `--output list`. Note that the arguments might actually be in either the long or short form (--queue or -q, etc.).

The first few letters of the test name identifies the map that is provided to the program as input.

- `Spec`: the sample input files found in the spec. A '1' in the name means it comes from page 1, 'A' means it comes from Appendix A. The word 'No' means it is the version that has no solution. So a testcase starting with '`SpecANo`' means the input file from Appendix A that has no solution.
- `Sml`: these are **sm**a**l**l, hand-made test cases that are less than 20x20 in size.
- `Med`: these are "**med**ium-sized" machine-generated testcases, between approximately 100x100 and 400x400 in size. Med1 does not contain any traps. Some of these have no solution, while others have solutions.
- `Big`: these are "**big**" machine-generated testcases, between approximately 500x500 and 2000x2000 in size. Big1 does not contain any traps. Some of these have no solution, while others have solutions.
- `Edge`: these are hand-made "**edge**-case" tests. Make sure your program handles the extremes that the specification allows!
- `INV`: these are "**INV**alid" inputs that your program must *reject* to receive points. See the section on error checking.

Once the project has closed, some additional "hidden" tests will be added. These will be machine-generated tests that were made by the same program that generated the medium and big test cases, and will be of similar sizes (between approximately 200x200 and 2000x2000). The AG is not holding back any invalid, small, hand-made, or tricky maps to trip you up! We just want to be sure that your program works in general.

Tips

Read through this document, but don't feel you have to understand or be ready to code everything before you get started! You can start with what you've just learned (processing the command line) while you do more thinking about the rest of the project. Do some planning before you start coding!

Read through the Optimization Tips (Canvas / Files / Resources) for help on optimizing your program for speed and memory. When storing the map, think about the minimum that you need. You definitely need the map that was read in, and you need to know how to backtrack from any location in the map. Do you really *need* a copy of every "room", or can you figure out that when you're in "room 1" (or color `'a'`), that walls `'A'` can be treated as if they are open floor spaces? How much information do you need for backtracking; for each square to know how you got there? How many different places could you have been before you got "here"? However, don't be paralyzed by trying to optimize everything before your program even works! For 2D and/or 3D data structures, order the subscripts in the same order they are listed in the first line of the file: `num_colors height width`.

Read the "Project 1 the STL and You.pdf" file, it contains information specific to this project to save you time and memory, such as using a deque instead of stack or queue, how to set the size of a multidimensional vector, etc.

Get one output mode working first before working on the other output mode. Figure out which output mode might be easier to code, and do that one first. Also consider when there is no solution: the output in this case might be easier than any other.

When you need to convert between single characters (like `'a'`) and an integer, DO NOT use `atoi()`! The `atoi()` function assumes you have a character pointer to a C-string which is terminated by '`\0'`, and a single character does not meet these constraints. Instead you can use the fact that the ASCII values for letters are consecutive, along with a little math. For example, if `char currentChar = 'd'`, then `currentChar - 'a' + 1 == 4`, meaning if you press button `'d'`, you end up in "room 4". You might have to do some typecasting to get the compiler to accept it, but subtracting the letter `'a'` tells you how far `currentChar` is from `'a'`, etc. You can also convert between button and door characters by subtracting `'a'` and adding `'A'`, or vice-versa. For example, `currentChar - 'a' + 'A' == 'D'` (same `currentChar` as above), meaning that stepping on `'d'` opens the `'D'` doors.

When you do math on characters, DO NOT use magic numbers like 65 for `'A'`, etc. Just use `'A'`.

Appendix A: Two More Samples

Input:

```
3 11 11
// Example levels
// 3 colors (A, B, C)
// 11x11 grid
###########
#...@.#.?.#
#.^...B...#
#.^^^######
#....a....#
#B#.......#
#bC...#B#C#
###A###.#.#
#....B.c#C#
#b...##.C.#
###########
```

How is this puzzle solved?

1. Walk through (or around) the `^` (they start inactive so either is fine) to the `a`, which opens the `A` doors.
2. Walk through the `A` door into the bottom left room and stand on the `b` button, which closes the `A` doors and opens the `B` doors.
3. Walk through the `B` door to the right and stand on the `c` button. **We can't walk over it without triggering it.**
4. Walk through the 3 `C` doors near the bottom right, then through the `C` door near the left-center and trigger the `b` button behind it.

5. Go up through the open `B` door, around the traps (`^`), and through the `B` door into the target room and stand on the target.

Output (when run with `--queue --output map`):

```
// color ^
###########
#...@.#.?.#
#...+.B...#
#...+######
#...+%....#
#B#.......#
#bC...#B#C#
###A###.#.#
#....B.c#C#
#b...##.C.#
###########
// color a
###########
#.....#.?.#
#.^...B...#
#.^^^######
#....@....#
#B#..+....#
#bC+++#B#C#
###+###.#.#
#..+.B.c#C#
#%++.##.C.#
###########
// color b
###########
#++++++#+?.#
#+^..+++..#
#+^^^######
#+...a....#
#+#.......#
#@C...#.#C#
###A###.#.#
#++++++%#C#
#@...##.C.#
###########
// color c
###########
#.....#.?.#
#.^...B...#
#.^^^######
#....a....#
```

```
#B#..+++++#
#%++++#B#+#
###A###.#+#
#....B.@#+#
#b...##+++#
###########
```

Output (when run with `--stack --output map`):

```
// color ^
###########
#+++@.#.?.#
#+....B...#
#+...######
#+++.%....#
#B#+.+....#
#bC+++#B#C#
###A###.#.#
#....B.c#C#
#b...##.C.#
###########
// color a
###########
#.....#.?.#
#.^...B...#
#.^^^######
#..++@....#
#B#+......#
#bC+..#B#C#
###+###.#.#
#+++.B.c#C#
#%...##.C.#
###########
// color b
###########
#+++..#.?.#
#+^++++++.#
#+^^^######
#+...a....#
#+#.......#
#@C...#.#C#
###A###.#.#
#...+++%#C#
#@+++##.C.#
###########
// color c
###########
```

```
#.....#.?.#
#.^...B...#
#.^^^######
#....a....#
#B#++++++++#
#%++..#B#+#
###A###.#+#
#....B.@#+#
#b...##+++#
############
```

Output (when run with `--queue --output list`):

```
(^, (1, 4))
(^, (2, 4))
(^, (3, 4))
(^, (4, 4))
(^, (4, 5))
(a, (4, 5))
(a, (5, 5))
(a, (6, 5))
(a, (6, 4))
(a, (6, 3))
(a, (7, 3))
(a, (8, 3))
(a, (9, 3))
(a, (9, 2))
(a, (9, 1))
(b, (9, 1))
(b, (8, 1))
(b, (8, 2))
(b, (8, 3))
(b, (8, 4))
(b, (8, 5))
(b, (8, 6))
(b, (8, 7))
(c, (8, 7))
(c, (9, 7))
(c, (9, 8))
(c, (9, 9))
(c, (8, 9))
(c, (7, 9))
(c, (6, 9))
(c, (5, 9))
(c, (5, 8))
(c, (5, 7))
(c, (5, 6))
```

```
(c, (5, 5))
(c, (6, 5))
(c, (6, 4))
(c, (6, 3))
(c, (6, 2))
(c, (6, 1))
(b, (6, 1))
(b, (5, 1))
(b, (4, 1))
(b, (3, 1))
(b, (2, 1))
(b, (1, 1))
(b, (1, 2))
(b, (1, 3))
(b, (1, 4))
(b, (1, 5))
(b, (2, 5))
(b, (2, 6))
(b, (2, 7))
(b, (1, 7))
(b, (1, 8))
```

Another Input:

```
3 11 11
// Has no solution
###########
#.....#.?.#
#..@..B...#
#.....#####
#......A.b#
#.a....A..#
#......A.B#
#......CCC#
#......C..#
#......C.b#
###########
```

Output (since there is no solution, the output is the same no matter which flags are specified):

```
No solution.
Reachable:
###########
#.....#####
#..@..#####
#.....#####
#......A.b#
#.a....A..#
#......A.B#
#......####
#......####
#......####
###########
```