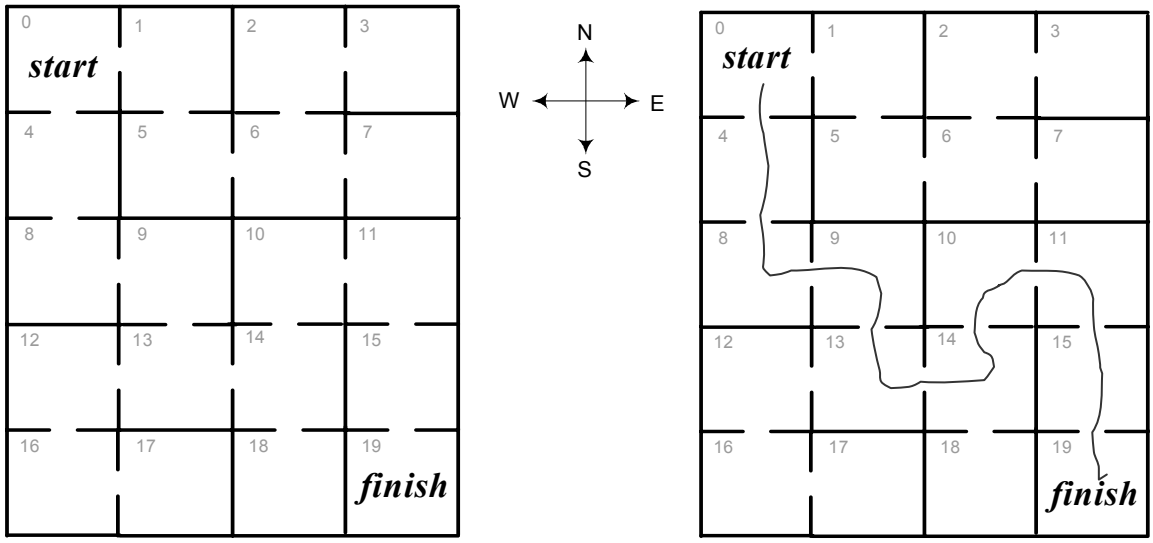
MCS 360 Programming Project #1 — Fall 2007

This project consists of writing a C program to find a path though a maze, using a stack.

Below at left is an example of a maze. It consists of a 5×4 array of "rooms", numbered 0 to 19, as shown. Certain adjacent rooms are connected by doors. For example, rooms 5 and 6 are connected by a door, but rooms 5 and 9 are not. The goal is to find a path from the *start* (room 0, at the upper left) *finish* (room 19, at the lower right), or, alternatively, to determine that no path exists. Below at right is the same maze, with a path from *start* to *finish* shown. The path is **0, 4, 8, 9, 13, 14, 10, 11, 15, 19**. For this maze, the path is unique. In general, it is possible that there is no path from start to finish, or that there is more than one path. For example, if we remove the door connecting rooms 13 and 14, there would be no path. If instead we add a door connecting rooms 17 and 18, there would be a second path **0, 4, 8, 9, 13, 12, 16, 17, 18, 14, 10, 11, 15, 19**. If there is no path, your program should discover and report that fact. If there are multiple paths, your program may find any one of them.



More generally, a maze will consist of an $m \times n$ array of rooms (m rows, n columns), numbered from 0 to $mn-1$. The goal will be to find a path from *start* (room 0) to *finish* (room $mn-1$), or to discover that no such path exists. If we number the rows 0, 1, ..., $m-1$ and the columns 0, 1, ..., $n-1$, the relationship between row, column, and room number is given by:

i) Room number k is in row k/n and column $k\%n$. (Here / denotes truncated integer division.)

ii) Row r, column c contains room number $nr+c$.

We may represent an m by n maze by two integers (m and n), followed by mn sequences, each consisting of one to four letters, each `E`, `S`, `W`, or `N`. For example, the maze above could be represented by:

```
5    4
ES   SW   ES   W
SN   EN   EWN  W
EN   SW   ES   SW
ES   EWN  SWN  SN
EN   W    N    N
```

The 5 and 4 on line 1 indicate that the maze has 5 rows and 4 columns. The `ES` at the start of line 2 indicates that room 0 has doors leading to the east and south. The next entry, `SW`, indicates that room 1 has doors leading to the south and west. Continuing in this manner, the last entry on the last line, `N`, indicates that room 19 has a door leading north.

Your program should first read in the data describing the maze, and then search for a path from *start* to *finish*.

Your compiled program (the executable file) should be named **`maze`** (or **`maze.exe`** on Windows). It can then be run with the command

`maze` ***file_name*** (on Windows)

`./maze` ***file_name*** (on Unix)

where ***file_name*** is the name the file describing the maze, in the format discussed earlier. You program should read the maze from the file named ***file_name*** and perform the appropriate initializations.

Your program should then use a stack `s` of integers, initially empty, to search for a path from *start* to *finish* (or to determine that no path exists). In addition to the stack `s`, you may wish to use a boolean-valued array of size *mn* with all entries false initially, perhaps named `entered`, to keep track of which rooms we have already entered. Once we have entered a room, there is no reason to enter it again in moving forward, although we may reenter it when we back up. And you may wish to use an array of strings, say named `doors`, or size *mn* to keep track of the doors. For example, if `doors[73]` is `"EN"`, then room 73 has doors leading east and north.

The search begins by pushing 0 (start) onto the stack `s` and setting `entered`[0] to true.

As we proceed, the top of the stack `s` will always represent the current room, and the rooms on the stack, from bottom to top, will represent the path we traversed from the starting room to the current room. From our current room i = `StackTop(s)`, we can move forward along our path

east to room $k = i+1$, if `doors`[i] contains `'E'` and `entered`[k] is false, *or*

south to room $k = i+n$, if `doors`[i] contains `'S'` and `entered`[k] is false, *or*

west to room $k = i-1$, if `doors`[i] contains `'W'` and `entered`[k] is false, *or*

north to room $k = i-n$, if `doors`[i] contains `'N'` and `entered`[k] is false.

If more than one forward move is possible, we can choose arbitrarily among them. (Later we will back up to the current room to try the other possible moves, unless we reach *finish* first.) If we move forward to the new room k, then we push k onto stack `s` and set `entered`[k] to true. If no forward move is possible, we are stuck at room i = `StackTop(s)`. When this occurs, we must back up by popping a room from `s`. Then the new top of the stack becomes the current room

We repeat the process of moving forward from the current room if possible, and otherwise backing up from the current room, until either

a) We reach room *mn*–1 (*finish*), in which case we have found our way through the maze (The path consists of the rooms on the stack `s`, in bottom to top order. Your program should print rooms on the path, from start to finish, using only stack operations.), *or*

b) The stack becomes empty, in which case we have searched all possible paths from *start* to *finish*, and determined that no path from *start* to *finish* exists. Your program should print a message that there is no path.

Please observe the following requirements for your program.

1) You may incorporate any (or all) code from the stack implementations given in class, in files **`Stack.h`**, **`ArrayStack.c`**, and **`LinkedStack.c`**. These files are available on the web site. If you choose to implement your own stack using an array, the array must be a resizable dynamic array with an initial size no larger than 16. In any event, there must be no fixed bound on the size of the stack. Likewise, you may incorporate **`mcs360.h`** and **`mcs360.c`** into your program and use the facilities they provide. (Be sure to get the latest version of **`mcs360.h`**, posted late on 09/23/2007.)

2) There can be no fixed bound on the number of rows, number of columns, or number of rooms in the maze. Effectively this will mean that any arrays you use related to the maze will need to be dynamic.

3) You must access the stack only by means of the stack operations as defined in the header file for stacks. You can write any code that would depend on the specific implementation of a stack (except, of course, in writing the code for the stack itself, if you choose to do this.) To print the path from *start* to *finish* , you could create an array of integers whose size equals the size of the stack, invoke `Traverse_Stack()` with an appropriate function `visit()` to copy the stack entries to the array in top-to-bottom order; and then print the array in entries in reverse order.

4) Try to employ good style in your programs. Use meaningful variable names, except possibly for temporary variables used in a small region of the program. Separate major sections of the program by blank lines, and put a meaningful comment before each section. Most importantly, indent code by a fixed number of spaces (about 4) when the level of nesting increases. (For examples, see the code in the textbook for this course, or in the MCS 260 textbook.)

5) The code for your project should be organized in files, as follows:

   `maze.c`: Contains your function main() and all your other code, except as noted below.

   `maze.h`: Header file for **`maze.c`** (recommended, but not necessarily required)

   `Stack.c`: Source file for stacks. You may use either **`ArrayStack.c`** or **`LinkedStack.c`**, presented in class and available on the web site, but rename the one you choose as **`Stack.c`**. Or you may write your own, as long as it conforms to the requirements. Must contain all the code for stacks, and nothing else.

   `Stack.h`: Header file for **`Stack.c`**. Again, you may use the file **`Stack.h`** on the web site.

`mcs360.c`: Optional (required if you use **`Stack.h`** from the website, useful otherwise).

`mcs360.h`: Optional (required if you use **`Stack.h`** from the website, useful otherwise).

6) You may test your program with the maze given earlier in this handout, or with the larger 20 by 10 maze given at the end of this handout. I will prepare some additional mazes that you may use.

This program should be turned in by Tuesday, October 9. I will give you instructions for submitting the program.

Maze #2: A 20 by 10 maze