CSC 236 Programming Assignment

MAZE

Summary

The MAZE assignment is worth 6% of your grade. For this assignment, you will be submitting a file named **maze.ans**. This file is created by the grading system when you grade your program. The grade is based on correct answers, documentation and the number of executable instructions used and the number of instructions actually executed during testing.

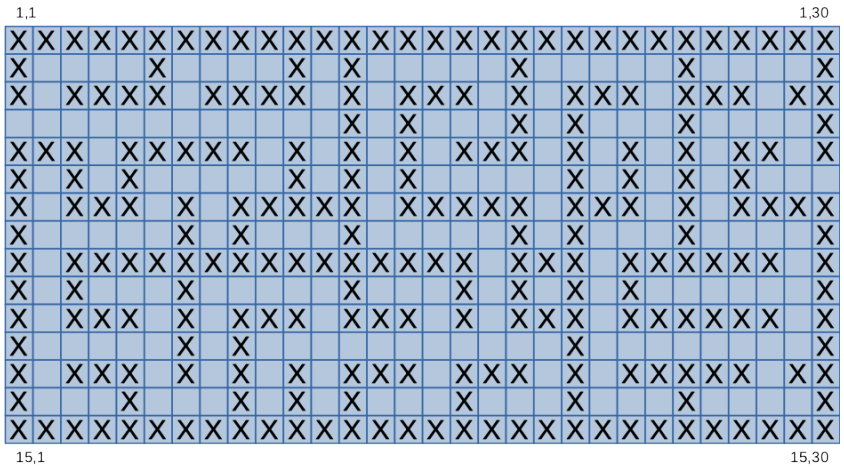
If you have questions about the assignment, feel free to use our Piazza forum, but be sure not to post any code from your solution in a public piazza post (even code that's not working). You can also stop by during office hours to get help from anyone on the teaching staff. Remember, if you need help, you need to make sure your code is documented, including block comments above sections of your code and line comments saying what you are trying to do with each instruction. We'll also expect you to have a design for the program that you're trying to follow.

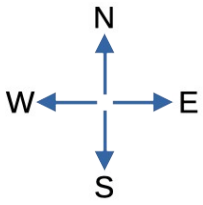
You are going to write a subroutine named **nextval** that will guide a mouse through a maze. The maze is a two-dimensional array of bytes, with 15 rows and 30 columns. The maze that can be traversed using a right- or left-turning mouse. These are navigation strategies are both discussed later in this document. The rules associated with moving the mouse are:

- You may only move one position at a time.
- You may only move into a location in the maze if the value of that location is blank (20h).
- You may not move into a location that is blocked (contains any value other than 20h).
- You may only move left, right, up, down, you may not move diagonally.

A position for the mouse is given as a pair of values, y (row) and x (column). The upper left square of the maze is position 1,1 and the lower right square is 15,30. So, of course, the upper right corner is at 1,30 and the lower left is 15,1.

The top and bottom edges of the maze (y=1, x=1...30 and y=15, x=1...30) will always be blocked (filled with values other than 20h). The left and right edges (x=1 and x=30) will also always be blocked except that there will be one blank square for entry to the maze and one blank square for exit from the maze. Below is a sample maze.





We will use map directions to indicate the direction of travel for the mouse. The mouse will enter the maze on the west side (x=1). In most cases the initial direction will be traveling east. Its goal is to reach the empty location on the east side of the maze (x=30).

Specifications

Your source file will need to be named **nextval.asm**, and the start of your subroutine will need to have the label, **nextval**: Each time your subroutine is called, you will make one move in the maze, east, south, west or north and then return. When you unpack the archive for this program, the starter file for your program will already contain the label you need, but you'll need to copy it from **nextval.m** to **nextval.asm**.

The input parameters passed to nextval are 4 pointers that point to values stored in the data segment. These parameters are passed to the subroutine via four CPU regusters:

- **SI** contains the address of the current value of the x position. X is an unsigned byte in the range 1 to 30.
- **D**I contains the address of the current value of the y position. Y is an unsigned byte in the range 1 to 15.
- **BX** contains the address of the current value of the direction of travel of the mouse. The direction is an unsigned byte with four possible values: E=1 S=2 W=3 N=4
- **BP** contains the address of the maze in memory. Each element is a byte and elements are stored in row-major order. With 15 rows of 30 bytes each, the total size of the maze is 450 bytes. An empty location is represented as a byte with the value 20h, and a blocked location is represented as a byte with some value other than 20h.

The nextval function should change the values in the bytes pointed to by SI, DI and BX to reflect the next move for the mouse.

- Using SI and DI, it should store a new location for the mouse in the X and Y variables stored in the data segment.
- It should set the current direction for the mouse by updating the contents of the data segment byte pointed to by BX.
- The subroutine must not alter the values of registers that might be in use by the caller. For any register it needs to use, it save the contents of that register to the stack and then restore the register's contents before returning.

The nextval subroutine doesn't need to perform error checking. When it is called, the subroutine is guaranteed that the values of the pointer parameters are valid, that the contents of X and Y are an empty space in the maze and that the given mouse direction is north, south, east or west.

- There will always be at least one valid path through the maze.
- The outer perimeter of the maze (the borders) will always be blocked except for the entrance and exit.
- The driver will detect when your mouse has traversed the maze. You do not have to worry about that.
- The X and Y and Direction chosen by the function must represent a legal, resulting state for the mouse. For example, let's assume that the mouse is at location y=6, x=20 and the direction is

east. The nextval function decides to move north to y=5, x=20. I would need to change y from 6 to 5 to indicate a move to the north. I must also change the direction from a 1 to a 4 indicating the mouse changed its direction of movement from east to north.

- The mouse is not allowed to stay in the same place. It must always move to a new location on each call to nextval.
- Once the mouse leaves the starting square, it may not return to that starting square (this wouldn't be a useful move anyway).
- Don't perform any I/O from the nextval subroutine. Don't try to read user input and don't write out anything to the terminal. If you need to debug your subroutine, you can trace through its execution using **cv**.
- The nextval subroutine gets a pointer to the current maze. It is permitted to look at the contents of the maze, but it's not OK to modify the contents of the maze.
- Your mouse must successfully traverse all legal mazes. You can tell if your nextval subroutine work by compiling the nextval.asm source file, linking the resulting object file with the mazedrvr.obj and then running the testmaze.bat file to perform a test. If you receive the "Congratulations! Your mouse has traversed the maze." message, then you've successfully traversed the maze.
- The design information provided with this assignment is intended to illustrate functionality. You may need to worry about making sure you have an efficient implementation.

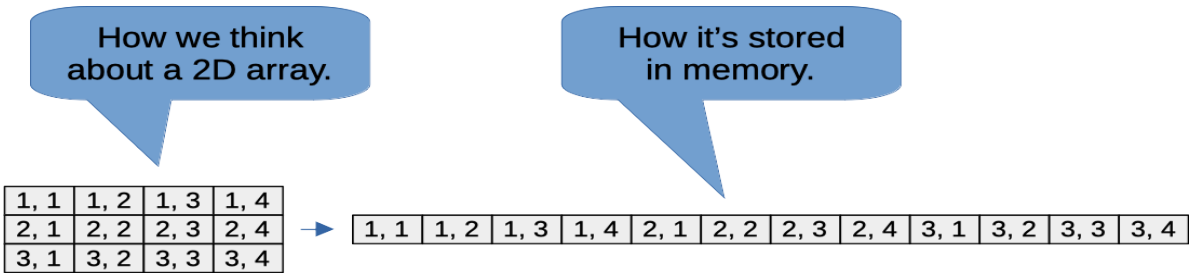
The nextval function should be able to choose its next move based only on the parameters it is given.

- Do not keep any history information between the driver's calls to nextval. Every time nextval is called it must calculate its move based upon the new parameters passed to it.
- Code the nextval behavior based on the specification given in this document. Don't implement any behaviors that are specific to the given driver program or the mazes provided with this assignment.
    - Most of the time when you are first called, the mouse will be on the west side of the maze and the direction will be east. However, that is not part of the specification. Your function should be able to work even if the mouse starts at some other location inside the maze, or if it's not pointing east at the start.
    - Don't reverse engineer the driver code and find the calling program's data areas and remember them or even worse hard code them into your program.

Maze Organization

The calling program defines all the variables. Your nextval subroutine will access all needed data using the four parameters and indirect addressing. The maze is organized as a two-dimensional array stored in row-major order. So when you have an array location Y, X, you need to be able to compute the emory location of the byte storing this cell of the maze. The way this is done is to calculate the offset of the location from the start of the maze.Since the maze is stored in row-major oder, an element at row X and colum Y is stored at an offset of (Y-1) * Width + (X-1). Our maze is always 30 elements wide, so this would be (Y-1) * 30 + (X-1)

The figure below shows how row-major order works for a smaller 2D array with 3 rows and 4 columns. To find a particular cell in memory, you need to calculate the offset of that cell in the row-major representation. For an array with four columns, this would be: offset = (Y-1) * 4 + (X-1)



For example, a cell at Y=2, X=3 would be at offset: (2-1)*4 + (3-1) → 6 in this representation.

Escaping the Maze

Both left and right turning mice are guaranteed to traverse a valid maze and escape. Here's how each strategy works:

A left turning mouse works as follows: • At every location in the maze you try to turn left. • If you cannot turn left then you try to go forward. • If you cannot go forward then you try to turn right. • If you cannot turn right then you go backwards. In other words, the priority of movement is: left, forward, right, backwards.	A right turning mouse works as follows: • At every location in the maze you try to turn right. • If you cannot turn right then you try to go forward. • If you cannot go forward then you try to turn left. • If you cannot turn left then you go backwards. In other words, the priority of movement is: right, forward, left, backwards.

Programming Notes

This assignment focuses in indirect addressing. The inputs to nextval are all pointers, stored in registers SI, DI, BX and BP. These registers don't contain the data you need to work with, instead, they they point to the data.

In some cases, you will need to use data size override and segment override to get the data you need For example:

- To access the current Y value you could copy it to an 8-bit register, like:
  ```
  mov al,[di]                ; load al with the Y value of 1 to 15
  ```
- To set the direction of travel to East you could copy a value to the byte pointed to by BX:
  ```
  mov byte ptr [bx], 1       ; set the direction to east.
  ```
- Remember that the B register in brackets defaults to the stack segment. To access the maze, using the BP register, you will need to use a data segment override

Files You Need

As with the previous project, you will need to download and unpack an archive containing the testing and grading files. All files are packed together in a self-extracting file named **unpack.exe**. You can download this from the class homepage in moodle, or use the following curl command to pull down a copy.

Put a copy of this **unpack.exe** file in the MAZE subdirectory of your shared P23X directory, then execute it from DOSBox. You'll need to:

- Start DOSBox and mount your shared directory as the E drive.
- Switch to the E drive and run **dbset** to set up environment variables.
- Change directory to MAZE and execute **unpack.exe**

When you do this, you'll get a file named NEXTVAL.M. Copy this to a file named NEXTVAL.ASM to get started on your own implementation of the nextval subroutine:

```
copy NEXTVAL.M NEXTVAL.ASM
```

The skeleton NEXTVAL.ASM file will have a place for you to fill in your subroutine. Like you've done on previous assignments, use an editor to add the needed code. Then, you should be able to use the following commands in DOSBox to assemble your NEXTVAL.ASM source file and link it with the mazedrvr.obj program to create an executable.

```
ml   /c  /Zi  /Fl  nextval.asm
link /CO  mazedrvr.obj  nextval.obj
```

Testing

The maze driver program will perform these functions.

- Read an input file that describes the maze, build and display the maze, display the mouse.
- Call your subroutine to request the next position to which the mouse should move.
- Move the mouse based on the values calculated by your subroutine.
- Check for successful completion of traversing the maze.

The starter archive includes test mazes named **maze.01** through **maze.06**. You can run the test program on any of these mazes by entering a command like:

```
testmaze maze.01
```

The starter also lets you test your solution on randomly-created mazes. To run against one of these, enter:

```
testmaze r
```

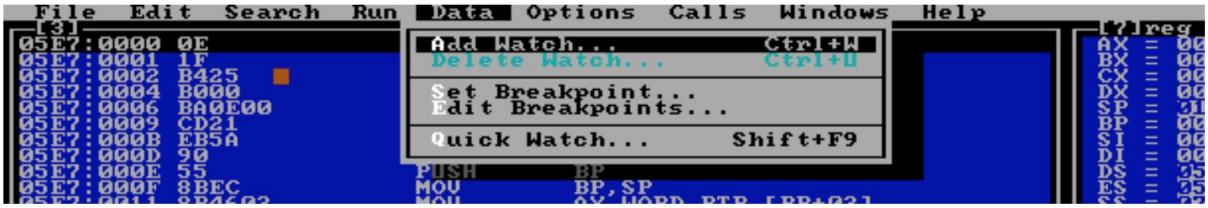
As you develop your solution, you may need to use the debugger to isolate the cause of any defect uncovered by your tests. You can run the After the debugger loads, you can set breakpoints at the entry of your nextval subroutine.

- To run under CodeView under DOSBox, first, enter the following command. This copies over the CodeView configuration file:

  ```
  cvset
  ```

- Then, enter the following command to run the driver (with your function) in the debugger:

  `cv mazedrvr < maze.01`
- After CV starts running, select "Data" from the menu at the top row. Then, select "Set Breakpoint"



- Under "Location:[...]" type nextval so it looks like Location:[nextval]"
- Click ok
- Pres F5 to go. CV will stop at the entry to your nextval routine. This allows you skip over all the driver code and get to the part you implemented.
- In the Memory Window you will see data at xxxx:yyyy Make sure that xxxx matches the value in the DS register. You should be able to just put the cursor on the xxxx field and set it to match the DS register.
- After hitting the breakpoint, use F8 to step through your code.

Grading

Your program must successfully traverse all legal mazes.

After you verify that it successfully traverses the sample mazes, run the grading program by typing:

`gradmaze`

Once nextval is working correctly, you may make additional grading runs to improve efficiency. If you execute the command testmaze mark all subsequent grading runs will be marked as only being done to improve efficiency.

The final grade will be based on:

- 50 points for getting the correct answers to the grading program test cases.
- 15 points for the number of executable instructions written to correctly complete this assignment.
- 15 points for the number of instructions executed in the grading run that had the correct answers.
- 20 points for documentation of a program that functions correctly.

Efficiency and documentation are only a concern after the code works totally correctly, so these parts of your grade will only be calculated and added in after your code passes all the functional tests.

Submit Your Assignment

Electronically submit the **maze.ans** file created by the grading system.

You will want to make sure:

- It contains two concatenated files. First the results file and then your source file **nextval.asm**.

- It contains the line of the following form giving your grade on the assignment:
  `++ Grade ++ xxx = Total grade generated by the Grading System.`

Incorrect electronic submissions will result in your program not being graded and considered late.

This is a version of nextval in pseudocode that you may use to get started. It is functionally correct but not optimized for efficiency.

```
// The notation *direction indicates the value pointed to by the direction pointer

if (*direction == 1) goto testn;    // if direction is east  try to move north first
if (*direction == 2) goto teste;    // if direction is south try to move east  first
if (*direction == 3) goto tests;    // if direction is west  try to move south first
if (*direction == 4) goto testw;    // if direction is north try to move west  first

// Try the desired direction first. If it does not work move to the next direction
// The mouse must be able to move in one direction if the maze is legal

testn:                              // test the north square being empty
offset = (*y-1)*30 + (*x-1);        // calculate offset to current square
if (maze[offset-30] == ' ')         // if north square is empty
   {                                // turn north by taking these actions
   *y = *y-1;                       // - decrement y
   *x = *x;                         // - leave x alone
   *direction = 4;                  // - set direction of travel to north
   return;                          // return
   }

// If we cannot move north, fall through and try east

teste:                              // test the east square being empty
offset = (*y-1)*30 + (*x-1);        // calculate offset to current square
if (maze[offset+1] == ' ')          // if east square is empty
   {                                // turn east by taking these actions
   *y = *y;                         // - leave y alone
   *x = *x+1;                       // - increment x
   *direction = 1;                  // - set direction of travel to east
   return;                          // return
   }

// If we cannot move east, fall through and try south

tests:                              // test the south square being empty
offset = (*y-1)*30 + (*x-1);        // calculate offset to current square
if (maze[offset+30] == ' ')         // if south square is empty
   {                                // turn south by taking these actions
   *y = *y+1;                       // - increment y
   *x = *x;                         // - leave x alone
   *direction = 2;                  // - set direction of travel to south
   return;                          // return
   }

// If we cannot move south, fall through and try west

testw:                              // test the west square being empty
offset = (*y-1)*30 + (*x-1);        // calculate offset to current square
if (maze[offset-1] == ' ')          // if west  square is empty
   {                                // turn west  by taking these actions
   *y = *y;                         // - decrement x
   *x = *x-1;                       // - leave y alone
   *direction = 3;                  // - set direction of travel to west
   return;                          // return
   }

// If we cannot move west, fall through and try north

goto testn;
}
```