Snake Game

Brief Statement

Your assignment is to write a computer game called worm or snake. You can play this game by running the following command on blue.cs.sonoma.edu. Your program should function essentially identically to this program. The four characters “h”, “j”, “k”, and “l” move the worm to left, down, up, and right, respectively. These keys move the cursor in those directions in vi and vim, two of the Unix-based operating systems.

Goals

1. Learn curses, the Linux Screen Manipulation utility.
2. Development of more advanced, but simple data structures.
3. Become more aware of efficiency considerations and demonstrate the role of data structures in efficient program development.
4. Writing Makefiles and using the Linux command, make.

Details

The playing field is a large rectangle of cells surrounded by a wall of asterisks. Each cell corresponds to a specific row and column on the video screen of your terminal. At any one time every cell is either:

1. blank (i.e. there is nothing in it),
2. contains a segment of the worm; either a body segment (the character ‘o’) or the head of the worm (the character ‘@’), or
3. contains a “munchie” (a digit from 1 − 9.)

The goal is to direct the worm to the munchie. The four characters “h”, “j”, “k”, and “l” move the worm to left, down, up, and right, respectively. Every time one of these keys is struck, the worm logically moves in the indicated direction by moving the last segment of the worm into the cell occupied by the next to last segment, the next to last segment of the worm into the cell occupied by the next to the next to the last segment, etc., until the second segment of the worm moves into the cell currently occupied by the head, and finally the head moves to a new cell. If the worm should bump into something, either the wall or itself, it dies and the game is over. If the worm eats the munchie, i.e., lands on the cell containing the munchie, the worm gets to grow and a new munchie is generated in a random empty square.

The worm grows in the following manner: if the worm has just eaten a munchie of value n, then for the next n moves the tail of the worm does not move forward as the worm inches along. That is, for the next n moves the worm grows a new segment at the current location of its tail. The score of the game is the total value of all the munchies eaten.

The number of rows and columns should be command-line inputs to your program. Your program should check that the user provides valid values for these inputs. The number of rows can be between 9 and 25, inclusive, and the number of columns can be between 9 and 80, inclusive. Therefore, you would run your application as follows. numRows and numCols are two integers representing the number of rows and columns of the game, respectively.

./mySoln.x numRows numCols

If we number the cells of the screen, calling the upper left hand corner of the screen position (0, 0), then row 0 is a message row (contains the name of the game in columns 1 − 4 and the score on the right-hand side). The wall occupies the top and bottom rows and the leftmost and rightmost columns, and the playing field is the area inside the wall. Initially the worm is one segments long, consisting of the head of the worm. The munchie is placed randomly in any unoccupied cell. Every time the munchie is eaten, a new munchie of random value appears in a randomly selected unoccupied cell.

There are a number of issues that make writing the program interesting. The immediately obvious problem is how to write to various locations of the screen. We use “curses” for this purpose. However, our application only uses a few of the commands provided by this extensive package. On the projects page for this course, we have posted a working program (sampleScreen.tar) that demonstrate the use of the “curses” library functions that you need for this project.

getChar.cpp, one of the files in screenExample.tar, contains the implementation for get_char(), a function that reads a character from the standard-input; it is the equivalent of C++’s cin.get(c), where c is a variable of type char. There is one difference: if a character has not been typed

by the user in the last 400,000 microseconds or so, get_char() will automatically type one for him/her. In the absence of a character typed in by the user it simply returns the last character typed (it returns blank if the user has not typed a character yet.) The spacing in time between characters is dictated by the symbolic constants SEC_DELAY and USEC_DELAY, defined in getChar.cpp (SEC_DELAY specifies the number of seconds and USEC_DELAY the number of microseconds.) The problem is this: if you move the worm as described above (by moving the last segment on to the next to last segment, etc.) then as the worm gets longer it will take longer and longer to move the

worm and your program won't have time to update the screen before characters are automatically returned by get_char(). This causes the screen to fall further and further behind what is happening inside the program and the user won't know what is going on.

The solution is to do everything in constant time. That is, every move of the worm should take the same amount of time irrespective of the length of the worm. Since every body segment looks like any other, we can simply erase the last segment and move the head in the correct direction

and it will look like the worm inched along.

Keeping track of the location of the head is easy, but how do you keep track of the location of the tail? The first thing that comes to mind is to store the row and column number of the tail segment, and when the worm moves forward find the new location for the tail. But how do you find the new location of the tail segment? You cannot simply search the neighborhood of the current tail look for another body character (why)! This is where data structures come into play. To solve this problem we will use a circular queue of cursors as depicted in the PDF file that you can download from the project web-pages for this course. The size of the circular queue should be equal to the number of squares in the playing area.

Another related problem is how to place a new munchie in constant time, i.e., the same amount of time no matter how much of the playing field is occupied by the worm. You might think that you could randomly pick a location (using the rand() function.) Of course if the random location is not empty (it contains a worm segment) then you would have to pick again. If the worm almost fills the screen then your program might have to pick so many times that the worm would be forced to keep moving (get_char() keeps returning characters) before the user knows where to direct the worm! Again we will solve this problem with a data structure, this time known as an inverted list. This data-structure is also included in the PDF file that is available to you to download.

While debugging your program, if your program terminates abnormally you may find that your terminal will not echo what you type. If this occurs, you might have to close that screen and open a new one.