Concepts of Programming Languages (CS 4308) 12/07/15
Nathaniel Klein

Final Project: Hangman

My approach to this program started simply, but became more complex as my coding progressed. After declaring my variables - one of which was an array of character arrays called 'lines' to store all the words in the file - I opened and stored a text file containing ten words on ten separate lines into a file pointer. I then placed each word from each separate line in the file pointer into 'lines'. Now that the data was in usable form, I welcomed the player to the game and drew its initial scaffolding.

I then created a seed from the clock time and used this to create a random number. I applied a mod-10 (%10) to the number to access only the last digit of the number (0-9). I initialized and created space in memory (using malloc()) for a character array pointer called 'word', and I assigned it a size in memory slightly larger than the number of lines in the text file. I was then able to use the digit derived from the randomized number as an index to get the word in 'lines' that corresponded to the line index, and I stored the word on that line in the aforementioned character array 'word'. It was necessary to use the strlen() function in the <string.h> library to get word length 'wordLength' (an int variable). This variable was absolutely essential to most of the comparisons in the game, as it was used as a control structure for loops throughout.

The last phase of the game's initial setup was to create a new char array called 'compareWord' and, using a loop in drawBlanks(), to place blanks in 'compareWord' whose number matched the number of characters stored in 'word', and then to print out the contents of 'compareWord' to the console. This served two purposes. It gave me a way to print the blanks to the console, but it also provided another char array initialized with the correct length to later be compared to 'word' to verify a win. This completed the initial game setup.

In order to play the game, the player must provide a letter. I called this letter 'playerLetter'. Every loop and while statement needs an ending condition. I chose to put the algorithm for comparing 'playerLetter' to 'word' and everything that goes with this inside a while loop that would end only once the number of incorrect guesses (initialized to zero) totaled at least six (one each for the head, body, left and right arms, and left and right legs). Rather than start right away within this while loop with a for-loop to compare playerLetter with 'word' at index i, I chose to first test if 'playerLetter' was found anywhere in 'word'. I utilized strchr() == NULL for this test. If not, then the guess would be marked down as incorrect (variable incorrect would be incremented), the hangman image associated with this incorrect guess number would be printed to the screen, and the game would ask the player for a new letter. Only if playerLetter was found somewhere in 'word' (the same method was again utilized, this time as strchr() != NULL) did it make sense to compare 'playerLetter' with 'word' one index at a time. If a match at index i occurred, int variable 'matchLttr' would increment, the blank at index i mentioned above in 'compareWord' would be replaced by 'playerLetter', and the contents of 'compareWord' would be printed to the console. When every index in 'word' was exhausted from comparison with 'playerLetter', the game would ask the player to enter another letter, and the process would begin again starting from the top of the while loop.

At a certain point in the development of my code, I realized that a player may accidentally enter the same letter more than once. We already tested that letter for its presence in 'word' once. If it wasn't there the first time, then it won't be there the second time, and it will simply be counted as an incorrect guess. But if it was there the first time, the code as I have explained it thus far would find it again and

try to store it over again. But in the game of Hangman, a player is not allowed to guess the same letter twice; it's considered an incorrect guess. So I had to modify the first “if” statement that tested for the existence of 'playerLetter' in 'word' using strchr() == NULL, and prior to doing this test, I had to test if 'playerLetter' had already been found. If so, then that would also be considered an incorrect guess! The placement of this test was crucial to picking up this nuance and dealign with it effectively.

When playerLetter is not found in 'word' and incorrect equals six, the player loses, his loss total increases, and he is asked in a separate method playAgain() if he wants to play again. If so, then the game starts from the beginning with a call back to main(). If not, the player is shown their win and loss totals during the session, and the game terminates. Similarly, if the number of matched letters equals the length of 'word', then the player has won, his win total increases, and he is again asked in the same separate method playAgain() if he wants to play again. There are two things here of note. First of all, the game either starts from the top or it ends, so the outer-most while loop that tests 'incorrect' so long as it's less than seven is never actually tested, but it does keep the structure of the algorithm well by helping newly acquired letters to loop back to the testing method strchr() as soon as they are entered. Second, the win and loss totals are initialized outside of main, and this is what allows their incremented totals over numerous games to not be reset to zero. Another way of saying this is that their data persists, no matter how many times the game is replayed. This is what allows the game to show the user an accurate win/loss count when the user no longer wants to play anymore.

I'd like to make additional points. The first pertains to the game's ability to parse the character brought into stdin by the Enter key upon gaining input from the user. I had to employ a technique not unique within c, but probably unique vis-a-vis many other programming languages. The getchar() command gets not only the letter entered, but the letter that follows it upon the user pressing the Enter key. This causes all sorts of processing problems, so a second getchar() command had to be implemented to catch the character from the Enter key every time the user entered a new letter. The first getchar() got the actual letter and stored it in 'playerLetter' which the game went on to utilize, while the second getchar() stored the Enter character in a temp variable that the program never accessed, thus rendering the issue moot. The second point pertains to the fact that when drawing ASCII art, one must be extremely careful when using characters in a string that are normally escape characters. Thus, for characters such as '\' that fall into this category (and which were important in the making of this game), one must “escape” the escape character to get it to show in the console. In other words, if I needed a '\', I had to code it like this: '\\'.

There are a number of methods I didn't mention directly in this description, but this is the essence of how the game operates.