Brandon Swanson
Week 3 Assignment Report
CS 165 Summer 2014

UNDERSTANDING

This week we learned about recursion and also more about arguments/parameters. I was very surprised that we were not required to solve the tower of hanoi, as I have done this so many times I thought it was a mandatory teaching tool for the subject of recursion. Simply put recursion involves solving a problem using a function that calls itself (or two functions that call each other), but more generally it is a way to approach a problem where the algorithm used can be broken up into smaller instances of the same algorithm. For example Binary Search seeks to find a value in a sorted array, it can look at the middle value and decide if the value is less than or greater than that value and then restrict its search to half of the array, the algorithm can continue this process until it is has divided the chunk it is search down to only the single element it was after.

Default arguments was a new topic for me, before this lecture I was writing Java style cascading function calls, where foo() would return foo(a,b). instead of defining foo(val1=a,val2=b). This is more evident what the desired default behavior is.

DESIGN

see attached image for my design sketch for Word Guessing game Part 2. My original game from week 2 had almost all of these features, so I wanted to take this opportunity to try to find something about the game to improve. I was looking at classmates code and saw one person had used defined string of all the letters in the alphabet as their primary database and output, when the player entered a character the program searched this string for that character, it it wasn't in there than it either wasn't a letter or had already been guessed. If it was in there it replaced it with a space, and outputted this modified string as the indicator of remaining letters available.

I compared this to my functioning but rather oblique data structure. my week 2 game had 3 arrays with 3 ints tracking the number of elements in them. And many of my functions called a self written Contains(char) method that had to iterate of the array. One of the elements missing from my game was an output of the remaining letters of the alphabet available to guess. I reflected on the necessary operations to print this out in my current schema, and it would have been 26 * (Num Guesses Made) memory accesses. I wanted to come up with something better.

I opted to switch to a boolean array to hold a true/false value for each letter of the alphabet and I wrote a simple inline function to map the 26 letters onto the 0-25 addresses of the array. I generally don’t like this kind of approach, tracking for all possible elements if that element exists in our set, its not very scalable, but in this case I knew we were restricted to only tracking the letters of the alphabet. This ended up reducing my game tracking variables to only 2 boolean arrays a counter for number of guesses (from 3 char arrays, and 3 ints tracking elements in the arrays)

I also changed the design of my input cycle for character guessing, many of my functions for input up to this point were given full access to the output stream, they gave the user a prompt, asked for input, then gave the user error messages, and then finally returned a proper input. I was trying to design a more unified feeling for this game and realized that this would be a problem, so I wrote out a new cycle of function calls / messages and decided to return a struct that had a flag for successful input and a corresponding message, or an input value. (more on this in reflection)

TESTING

I wrote the following test cases before implementing my design

SCENARIO	**INPUT CASE**	**DESIRED OUTPUT**
Get Secret Word	hellos89wjhw	did you mean hello(y/n):
	. hello again	did you mean hello(y/n):
	BrandonSwanson	That is too long
Make a Guess	gh	one letter at a time please
	7	only letters please
Word:“banana”	b	good job, in the word
	n	good job, in the word 2 times
	j	not in the word
	b	you already guessed that

REFLECTION

A lot of the problems I solved with recursion this week fall under the category of tail recursion. I don’t particularly like using recursion for these purposes because it seems that an iterative approach would be easier to write and to understand. I like recursive functions that have some computation after the recursive function call, I suppose only because they are interesting, they lead to a sort of “ah-ha” moment after drawing out a trace to understand what they are doing.

In this weeks program I changed two things, the data structure (the easy part) and the way the information was outputted (the hard part). I realized that I needed to begin decoupling my user input methods from the output stream. Similar to a development model like MVC (model view controller). I have written a lot of functions for getting a number or a word from the user and they all make use of cout << in order to prompt or tell the user that their input was invalid. For this game I used a method I had written that indiscriminately returns a string from getline() and then used a struct to hold any error messages about the user inputed more than one character, numbers etc, or a correctly imputed single character. I could then loop over this function until it told me the input was correct, but also decide how to display the error message.

I want to work on designing input functions that are independent from any output stream so that i can reuse them in more contexts (like if I tried to make an ncurses program). My first thought is to either pass in a library defined stream object, or create my own output object that I can pass to these functions.

ACCOMPLISHMENTS

- A dictionary object for retrieving the player one secret word entry, At first I thought this would not be feasible, but I was working on next weeks phrase guess game and was curious about using source files to generate lists of words. I wrote a little word parsing program and threw a whole book at it. I was amazed at how quickly it was able to read the file. From there I got a dictionary text file that was formatted for easy reading, and then learned about map objects for quick searching. I added a command line argument to turn this feature off, and also left in the old method as a failsafe in case of any file read error.
- Menu and Menu-item classes created for the interface of the exercise components. A menu item is instantiated with a pointer to a void() function and a few strings for output. This defines a selection, introduction, and repeating behaviour for all the exercise component functions.
- Use of command line parameters
    - menu can receive command line parameters to select an item (i used this during debugging)
    - Secret Word game has an alternate clear screen method invoked with the command line argument ‘-s’, in case the system() call method causes any bizarre outcomes.
- an ASCII art style user interface for the secret word game, Guesses are represented graphically, and due to the screen refreshing the display appears constant while the guesses, available letters, and secret word appear to change
- Template functions for Contains(), ie is element in this set, that works for types: long, int, float, double, char, string (at least so far that ive tested), and an insert element function that works on chars, ints, floats, longs, doubles (not strings). After the refactor of the hangman game these are not as needed but i’m sure I will have a use for them in the future,