Brandon Swanson

Week 3 Assignment Report
CS 165 Summer 2014

RECURSION EXTRA CREDIT

I chose to pursue the extra-credit recursive algorithm approach for the phrase game. I have written up an assignment report specifically for this RecursiveGuessChecker.hpp file and attached it separately <RecusionReport.pdf>

Understanding

pointers, arrays (somewhat related), command line arguments and c-strings (also somewhat related)

Pointers

I can't imagine these four topics not being thought in the same week, they are all almost inseparable. To start we have the pointer, something I learned many years ago but when working in more 'high level' languages (or specifically Java, which runs on a virtual machine) I had happily left pointers behind. I was delighted to find some interesting uses for pointers though. Both my switchable clear screen method and my Menu object make use of function pointers. This can create an interesting case of mutable behavior without the need for extraneous sub classing. My first design for the menu had menu item as a virtual class (abstract/interface) and each item would need to be a subclass. But without the convenient lazy instantiation this would have been a cumbersome object to implement. As is now menu items are constructed by passing in a pointer to the void() function that will be run when the menu will be selected see SwansonObjects\ArgReader.hpp ln#30 for another use of pointers. This object stores a list of structs containing a pointer to the boolean value held by the class that instantiated the arg-reader, and then a string. If the string is in the command line arguments, then boolean that is being pointed to (again, this boolean is outside the scope of the arg-reader) is set to true.

Arrays

I didn't realize until this weeks lecture how related pointers and arrays are. When the lecture described the [] operand as being a “pointer offset” it all suddenly snapped into place, why they start at 0 (pointer address plus 0), why there is no array out of bounds exception, why arrays are never pass by value and can be defined as a parameter simply with a pointer to a type. I had been using arrays in my assignments since the very first week but the readings/lectures about pointers made these features of arrays suddenly make sense.

One of the more interesting challenges was the dynamic 2 dimensional array. A 2d array is sort of a pointer to a pointer, or a pointer that points to the beginning of a set of memory locations. And then advancing the pointer with [] would refer to another collection of memory locations and finally [][] would be a double de-refrencing. So in order to make a dynamic int **gameBoard, I needed to first declare an array of pointers gameBoard = new *int[boardsize] and then iterate over its pointers and point them to a dynamically declared int array: gameBoard[i] = new int[boardsize]

see dynttt.cpp ln#57

C-strings and Command Args

c-strings are a fun little way to store characters with a termination (sentinel) character '\0'. because arrays in c++ are size agnostic all functions that manipulate them must also carry an (*int arraysize)* parameter, to know how many values can be accessed. But with this '\0' iterative functions can advance through the string until finding such a character (usually, save for badly made c-string).

The command line args make use of this iterative advantage when passing in a 2d char array *argv[] sometimes written as **argv. It is an array of c-string but by definition this is also a 2d array of characters plus '\0' values.

On Understanding and Arrays

As it happens my phrase.cpp contains almost no arrays, and in fact the backing class for my phrase game all sets, set<string> specifically. Primarily because their one instance of each value, and quick searching made them great for answering, Is this <> on of theses <>, probably the source of the name set, as in is ItemX an element of SetY. This question is the core of the guessing game.

Anyhow, if you need proof of my learning objectives for arrays, look no further than week 2 and 3 both of those were backed by arrays, but in different ways, one was arrays of chars and a counting variable for the number of elements added, and week 3 I moved to a boolean array size 26 with a mapping equation for all letters inSet=lettersIn[char-'a].

As it turned out the above described boolean array is really a poor expression of a set<> type, set only tracks the existence of values in its set, and can do so for types where the universe is not defined (the letter universe is |26|)

Design and Testing

Design documents will be attached in file or in this document. Of which there is a great amount.

The recursive algorithm and my new MVC approach took careful planning, and my design sketches proved to be invaluable.

Approaches used for testing and designing this week

- Stubbing – Function calls were designing with parameters and return types (including structs/objects), I.e. a Messaging Diagram. Then functions with dummy returns were implemented so that each one implemented could be tested against its interactions with other components
- Design/Test in Isolation
    - Objects were developed separately early in the week with clearly defined behaviors to test against. Once functioning they could be quickly dropped in to the larger Phrase.cpp with no (or very little) modification needed
    - game input/output/data functions were made into separate objects/import files from the game controller to make them easy to control and check.
    - The behavior of these objects could be tested without the other components, isolating

possible sources of erroneous behavior, and allowing for automated input

- MVC – Model View Controller design pattern
  - After reflection from previous weeks I decided to de-couple my model from the output, and attempt to run my game through a controller class (the global, int main class). Delegating all cout << statements to only one object (or trying to)
  - A Phrase game object was made that held the data of a single game round and had very limited access to the output stream (it requests the guess from user).
  - One benefit of the PhraseGame object is that the rules of the game are seperate from the operations of the game (PhraseGame could change without needing to adapt the controller) Additionally it manages its own data, so repeating the game only requires instantiating a new PhraseGame object.
- Black Box Testing
  - I implemented a tester just for the dictionaries that would throw a rubric set of words at it to determine if its word lookup was functioning properly, this allowed me to quickly check the validity of updates and fixes to this object. It relied entirely on public method calls of the Dictionary object, meaning that the validity of the Dictionary object was defined only by public appearance and behavior (IE Black Boxing)
  - I left these test code in my final submission, either commented out or in a separate file, as opposed to previous weeks where these were removed, they are there for your benefit.
  - The file SwansonObjects/BlackBoxTesters.hpp is not used in any of the compiles this week but it is in there as evidence of my Testing.

Reflection

This week I learned a great deal about objects in c++ and some special data types like sets (sorted/unsorted) and maps. I briefly considered using a map for the phrase game with the pairing <string,boolean> for the key, value, where in the phrase would be 'letter',true. Thankfully I realized that as each key in this map had the same value (true because it was in the map) the map was useless over a set<>.

I also made my first subclass and dealt with a lot of headaches. Here are the takeaways.

- Initialization lists, a good way to initialize constants for a class
- constructors cannot delegate
- sub classes are constructed through the parent constructor (duh) therefore calls to virtual methods within the parent class constructor are not sent to an override method of the subclass (this one was hard to discover)

Hits and Misses

HITS

- The ThemeDictionary object is not only functioning but quite function
- The ThemeDictionary is also my first successful subclass (after a lot of painful effort), and expresses polymorphism. The controller object retains a Dictionary object, that sometimes is

the base class, and other times is the Derived ThemeDictionary.

- An effective use of the Menu object (even combined in sub-menus) for phrase game, This object was actually designed to hosts the exercise components (before the makefile decision), but because of its robust design it was able to be ported to this purpose (OOP for the win)
- A step in the right direction towards an Input/Output/Data/Controlling Decoupling model, PhraseGame object works nicely at passing back messages to be displayed in the specialized display of the controller.
- Pointers are weird and difficult but are useful and also sort of fun, the important this to remember is that the de-refrencing * has different effects on pointers and variables.

MISSES

- I really would have liked Phrase game to use the dictionary object but I didn't have time to add necessary methods to the object (primarily to add the alphabet to the set) so I made a function to retrieve the set from the dictionary, and then built the phrase game methods to work with sets.
- I would like the Phrase Game to be more agnostic about the display, because it retained the guess checker object, it built the reveal phrase, this is no good for decoupling. Similarily the controller relied on this phrase being equal to the secret to determine if the game had been won. This should have been accomplished in a more encapsulated way.
- I should have not let Feature-Creep (aka scope-creep) get so out of hand, the game I have created is great, but it was not easily finished, next week I hope to start with only the specs, and add spice as time allows.