Brandon Swanson

August 817h 2014

CS 165

Week 8 Assignment Report

UNDERSTANDING

This week the lectures and readings were focused on file I/O. These techniques were something I explored early on in the class because of the exciting enhancements it can add to a program. Like importing a large set of data, or saving states between program executions.

From the reading and lectures though I learned new ways of reading the files until the end. I had previously used the .eof() method but have found times where this was unreliable and have since been learning to use what the book calls the macho way of whlie(ifstream >> value). Upon seeing this method of reading in file streams I was very curios about the mechanisms behind it. I know from operator overloading that the >> extraction operator will return a reference to the stream, and this is how the >> and << are able to "chain". The question was how was it being evaluated as a boolean. After discussing with others in class it is the case that upon fail the pointer to the stream is set to null (which as a bool is cast to false as it is value 0).

I also learned different ways of formating the output to the screen using flags like ios::width and ios::right. I have up and till now using strings and methods like .append() to achieve these effects. Additionally I have not often overridden the << operator and instead have built a toString() method for objects and then called

```
cout << myObject.toString();
```

but This week I did make an object for the logger exercise with an overridden << operator. The effects

are obviously indistinguishable to the user but I think that operator overload can provide a more consistent fell to the code.

I also got a chance to write a function that took advantage of the inheritance relationship of streams. It is in the WalkerMaker class. It uses combinations of uppercase and lowercase letters to generate a starting set of living cells for game of life. By having an istream as the parameter I was able to pass in either cin, an istringstream (constructed from a string), or an ifstream and use the >> operator them to get the characters and construct the world. It allowed for testing via files, storing configurations in strings, or getting input from keyboard.

EXERCISE COMPONENTS

For the logger exercise I did not track more than just the occupants names, but I did take the time to create an occupant class, so that if more info was added to occupants was added latter the program would continue to function and could be more easily changed, as occupants are passed as parameters and then their .name accessed instead of passing just strings for their names.

For the quartile exercise we were explicitly instructed to re-open the file in order to find the appropriate numbers after discovering how many there were. Unless there were more items than available memory it seems a better way would be to read the values into memory. My implementation opens the file one time more than necessary (or two if you grab the third quartile after the median) but I wanted to be able to test the behavior so I made a function of grabbing an item or getting the average of 2 items. From here the only work was to check if the set or subset had even or odd elements and correctly calculating the position of the middle

the GetInt exercise seems a little late in the class, as some on piazza have noted this is an operation we have been required to preform many many times this term. In week 2 I created a function

for validating input and returning an int. It used getline and checked for stream errors, alpha characters and used strtol (string to long) over other available methods for the benefit of having overflow checking. Over time there have come to be obvious limitations to this function, I would like to adapt it to receive input and output stream references so that it can be used more portably (files, ncurses) but has worked well throughout the term. For this exercise I used a string stream for the extraction of the int value after checking it contained a valid combination of characters (0-9 and leading only '-').

DESIGN

My final implementation of Game of Life stayed very close to the original design. The classes God and Angel were able to remain almost unchanged, and the planned concepts behind the World interface held up. Many of the changes were not a redesign of concept but rather a struggle to find the best way to wrangle c++ classes into behaving like abstract classes and interfaces in the ways I was familiar with from Java.

One of the first changes was the addition of the separate interfaces (class with exclusively pure virtual functions) for display and for calculating the game rules. As per my original design I was seeking to keep the data structure of the world entirety encapsulated. The GOL namespace defined two types, coordinates and cells, and the world needed to be able to deliver these items one at a time upon request, but not surrender its actual data structure members or iterators to them. But the original design had the world class implementation holding on to an iterator to its data sets and handing out elements and iterating, but the weakness here was that it could only be accessed by one agent, if there were two displays or two classes that executed the rules this would not work. So I wanted to be able to create many instances of an object that could access the cells of the world in the abstracted and encapsulated way. Therefore the reaping and display interface classes were created in the WorldTool.hpp

This file defines the 3 classes required by any implementation of WORLD. (these may have been better as WORLD nested classes) The third after the interfaces was a WorldBuilder object, whose only responsibility is to hold the necessary values for calling the constructor of an implementation (subclass) of WORLD and upon request delivering a pointer to a WORLD object that is pointing to a constructed and concrete class. The purpose of this, is to ensure that only one class, the god controller class, owns the instance of the World. In this way the god class can be used with any kind of world that implements the World interface (aka subclasses)

In other programing language it would be possible to achieve this same behavior without needed to write implementations for 4 separate classes. The subclass called MapSetWorld could declare that it implements a certain interface, and then the owner of that object could pass it to other objects casted as an interface.

REFLECTION

The code in the GOL/ folder that defines the interfaces and types for the game of life rules is very short. Surprisingly short compared to the time spent on it. It is an obvious truism that less code is harder than more. I believe if I had not done this in an object oriented way but simply wrote the code for manipulating the map and sets that I used as a data structure it would have taken less than one day of work to have the functionality up and running, but it would not be nearly as robust or modifiable.

As mentioned in my report last week I always try to design in an un-scoped way, attempting to build encapsulated objects with well defined interfaces. As there was only one week to finish this implementation of game of life (a week that included a midterm); many features I wanted to include had to be left on the drawing board. The creation of the interface classes was a complex endeavor and

the accurate execution of the simulation took careful and extensive testing. Once the engine of the game was successfully up and running and outputting the world to the screen was working, I had less than desirable time remaining to make a user interface that could take advantage of many of the parameterized functionality. Additionally the true benefits of the polymorphism achieved in the class design, the ability to achieve runtime behavior changes through differing subclasses, is not currently demonstrated.

But I hope to later continue to improve this game, and the time spent on the core classes, is not time wasted. Good class design can go a long way toward future portability and flexibility.

The method by which I checked for still life/oscillating/ or repetition of x period was easy to implement, functioning, but left me unsatisfied. It is a functionality I would really prefer to be encapsulated within the WORLD or GOD class, so that It would always be available to whatever subclass or display method is implemented. I sketched out a couple of ways to perform such a task, the primary decision was whether the class WORLD should implement methods to calculate the repeat period defining a type to store the generations and requesting them from the world through a pure virtual function, and possibly defining a version of Generation() to be called by god that would then call a separate virtual method, allowing these calculations to be executed before and after the subclasses generation switching. Or, Alternatively, virtual functions of calculating and accessing repeat period could be added to the display interface and WORLD interface, leaving the responsibility of storing and equating successive generations to implemented subclasses.

Sadly on my production timeline (Sunday 12am deadline) changing the class interfaces was not only too time-intensive but also rather risky to the establish and tested stability of the program.

I am increasingly frustrated with the limited ability of the Menu class I created earlier in the

course of this term. As it used void() function pointers the functions executed have no parameters or returns and then depend on global values for operation. There is usually no need for global objects or values and I prefer (as do most) to keep these as local as possible. But to achieve the world creation method selection in my program using the menu class it necessitated a global world builder. In the logger exercise I played with a more local version of the menu idea, I would really like to re-visit this class and come up with a more flexible way to control program flow or choose an object type from a list.

TESTING

To keep final code clean I compiled separate testing and experimenting files while developing. The file GOLexperimentENV.cpp contains my work in figuring out the mechanics of the map and set data types, building a < for sorting coordinates, and checking small pointer pertains. Once the interfaces were established TESTreap.cpp was used to test out the instantiation and operation of various classes and to get a verbose output of the workings of the MapSetWorld class

The testing process for game of life was more defined than any other project I have made up to this point. After getting the game running and building a way to represent the living cells I used the display interface and reaping interface to check the calculations.

One error was found via an assert statement that was designed to catch if a cell was reported have more than 8 neighbors. This error wasn't found until near completion because all of my test cases were being done in a square world, and upon switching to 80X22 there was a problem, I knew that somewhere along the way these dimensions were mixed up in the parameter passing. The assert statement helped me catch this error that would not be easily detectable by just watching the worlds

display.

Another error was more difficult to find, I was comparing my glider case output to the expected output. It worked for many generations but on the 5th generation lost a cell. I found it surprisingly while attempting to trace the problem using GDB. The surprising part was that it was found only because gdb was outputting my code to me one line at a time and I caught the fact that a bug I had fixed in two other places that was causing the last element in the set of cells to be missed was not fixed in the Angel ReapAndSow method. This was a good reminder to read my code line by line when the behavior is suspicious.

The code for generating test output can be seen in TESTreap.cpp, and generated test cases in the folder TEST. Please see samples below, in the included TEST folder there is also output of small test case worlds as defined in my previous weeks submission as well as tests of the random generation. These test outputs were generated using Linux stream re-direction like so

```
./TESTreap>test.txt
```

TEST CASES

testing neighbor calculation in a 10by10 zero counting (0-9) world
Y axis increases downward for ease of print outcordinates form: x/y

neighbors of 3/4
2/3 3/3 4/3
2/4 4/4
2/5 3/5 4/5

neighbors of 2/2
1/1 2/1 3/1
1/2 3/2
1/3 2/3 3/3

neighbors of 9/9
8/8 9/8 0/8
8/9 0/9
8/0 9/0 0/0

neighbors of 4/0
3/9 4/9 5/9
3/0 5/0
3/1 4/1 5/1

neighbors of 0/0
9/9 0/9 1/9
9/0 1/0
9/1 0/1 1/1

testing neighbor calculation in a 10by10 zero counting (0-9) world
Y axis increases downward for ease of print outcordinates form: x/y

neighbors of 3/4

###
# #
###

---

neighbors of 2/2

###
# #
###

---

neighbors of 9/9

##

##
# #
---

neighbors of 4/0

# #
###

###
---

neighbors of 0/0

# #
#

#
---

testing neighbor calculation in a
10by10 zero counting (0-9) world
Y axis increases downward for

test case a

```
----------
    #



     #





----------
```

neighbor number

```
----------
 1      1
11      1

   111
   1 1
   111



11      1
----------
```

test case b

```
----------





     # #
      #



----------
```

neighbor number

```
----------




    11211
    11311
    12221
     111


----------
```

test case c

```
----------
    #
     #


    # #
     # #
      #



----------
```

neighbor number

```
----------
 1111   1
111 1   1
   111
  11211
1 212211
11222311
   12221
    111

 11     1
----------
```