Brandon Swanson

Assignment Report Week 5

7/25/14

Understanding

This week we learned about Classes, aka Objects. There was also structs (little baby classes with only public members and no methods), more about pointers, and reading on exception handling. Finally we are into the fun world of designing with objects. I spent the bulk of week 2 and 3 doing self studying on the syntax of objects in c++. I could not restrain myself from learning how to use objects in this new language because once you have learned how to use OOP concepts in your design it is very difficult to think without them. In the weeks that introduced functions many of the topics were like an introduction to the ways of thinking that are essential. There was de-composition or breaking the problem into smaller parts, and scope concerns, passing in necessary parameters and isolating mutations of variables. Now with these tools of objects we have the potential to develop code that can be portable between programs/assignments.

After reading the section on exception handling I was curious if I should go back through my library functions for user input and convert some of the “that input was not valid” if statements into the Try/Throw/Catch pattern. The interesting thing is that in this case this would not change the behaviour and therefore would be a syntax change only, but it would make the intention of this code more clear to other readers. In my experience many exception catches result in halting the program. Maybe this is a habit I have erroneously picked up but I have a “program must go on” design pattern. Always trying to keep things running.   My phrase game from last week had a backup dictionary created from a hard-coded array of strings to use in the case that there was a file-read error.

Pointers are become indispensable, especially when working with collections of objects or with subclassed objects. I will touch more on the collection of pointers in Design section, but It was very interesting to learn how dependant on pointers Polymorphism was. My original menu class had a list of MenuItems, but then when passing in MenuItem subclasses, the base class behavior was being executed, this was rectified by switching to holding a list of pointers to MenuItems.   This was doubly important later when MenuItem became an interface class (holding pure virtual functions) as it was then impossible to actually hold an instance of a MenuItem, or pass a MenuItem as a parameter. It seems that other than in specific circumstances when I want to reference my objects it will be through pointers.

Design

Default constructors, I don't like them. Many objects have a default constructor that doesn't initialize its member variables and then a Initialize() method. I dont like this pattern because it either leads to accidental referencing of NULL values, or forces the object to check its initialization before attempting to return a value, but for the class that has an instance of this class to recognize this error there must be an interface for error messages established. I find it better to ensure that the object is fully initialized upon construction. This prevents the ability to create an array of these objects easily but you can instead have a collection of pointers to objects that are yet to be constructed. Ultimately I want to avoid the scenario where I import an object I created months later and forget that it needed special treatment after construction before its members can be accessed or its methods executed.

I ran into one problem with this initialization during construction model and it had to do with subclassing. It turned out that when I wanted a specialized behavior during initialization by my subclass, but the method being overridden was called by the parent constructor, then the parent class method was called not the child class (forgive my use of parent and child over base and derived). There was an easy compromise to be had, which would be to have the constructor return an uninitialized object upon which could be called Initialize(). But I have already elucidated why this method does not appeal to me. Instead I found a solution I called two-stage construction that allowed the subclass to allow the initialization to be avoided and then called from within the subclass.

see here
https://github.com/swanyriver/ASSIGNMENT-4/blob/master/SwansonObjects/Dictionary.hpp#L117

For the design of the Menu class for this week, and the one described in week 6 assignment I am trying to decipher the appropriate scope. My menu acts as a sort of intersection point, executing small pieces of code to make changes to my objects, or running exercise components, but each of these is not aware of its state of being run from a menu, and the menu has no affiliation with the functions it runs.

The design doc mentions validating the input to its options, and in week 6 is described more a “gaming center”. These seem to reach into a broader category than a menu. More of a global controller for these sub programs, managing input and output. This controller model is something I was pursuing in my phrase game submission. I am trying to envision a class that could act as this gaming host. Recieving input from user, sending it to the “model” then displaying the response(sending to view). This is certainly a possibility, each game would need to be made to fit into a common language of operation and attributes, having rounds and guesses. I think the most challenging part would be unifying the display of each of these different games. But specifically on validating the input to each option, these rules of valid input would have to be determined by each game, but the cycle of get input, send input to model, model returns valid/invalid with message, controller displays message. this could be made into an abstract interface that each game implements.

Testing

I tested to assure that when numbers were requested numbers were received and they were in range. For the input of peoples names I decided not to screen this at all. The user could enter numbers, or a as many words as they like, including symbols. For the purposes of this program no input in this field would cause an error as the only operation being performed on it is displaying it through cout <<. And if no input could cause an error in the operation of my program than by what criteria would I decide it is an invalid name. Maybe aliens or robots have used my program. Contrarily though a negative age does seem to be invalid so that input is restricted.

One scenario that took some extra tracing to debug was the adding pointers to People class to the map in the object.cpp. I was ending up with a map containing all duplicate entries (well duplicate values at least). After a lot of testing it would seem like I was about to add a new entry but yet they all had the same value at the end. It turned out that because I was re-initializing the newPerson object variable everytime and then passing a pointer to it, all the pointers were pointing to the same instance, I needed to use the new directive to pass a pointer to a new instance of the object.

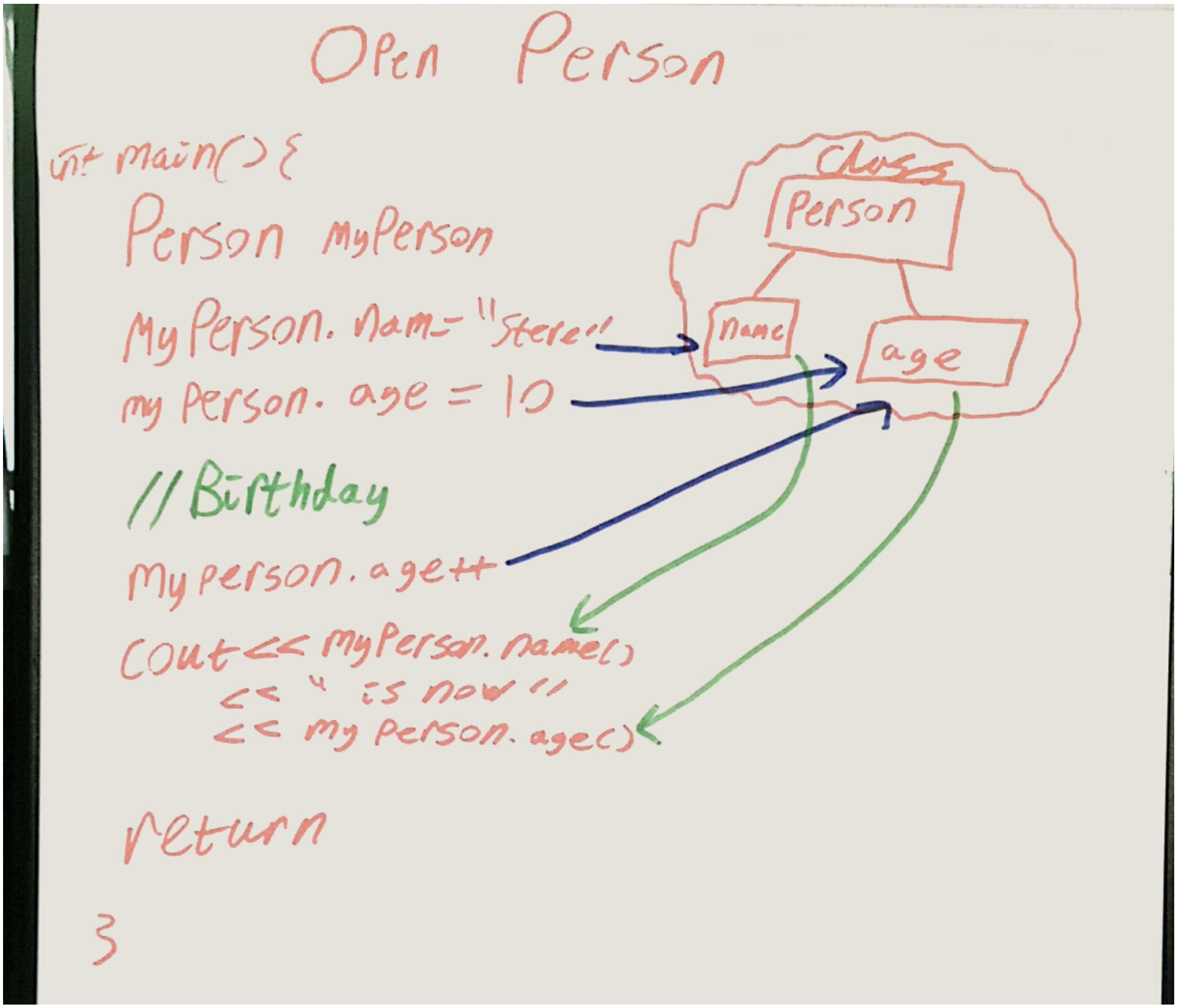
To make testing easier I made a simple program using the compile menu subclass. It compiles all the exercise components via a call to system() and then allows for the running of one are all of them in order. I was able to use this to quickly make changes to one component and the Re-Build them and check their new behavior. If you want to try it yourself compile the exerciseComponents.cpp.

Reflection

The exercise components this week have certainly shed light on the relationship between arrays of objects and the default constructor. As well as the importance of using object pointers. Last week I naively assumed that if I made it through the week about pointers I could move on to holding onto concrete variables and forget about pointers. But it seems that if I want to leave the option for subclassing and polymorphism or hold a reference to an object that will be constructed later, I will need to become comfortable with the * and -> operands.

Some on Piazza have questioned the value of using a struct when class already exists. I find them very helpful as return types from methods of a class. It can be used to define what kinds of values will be produced after an operation, and then internally as the class uses various private methods to produce one instance of this struct (eg GameGuess) then that struct being produced can be passed by reference until it is fully assigned.

SEE BELOW SKETCHES FOR EXERCISE 7



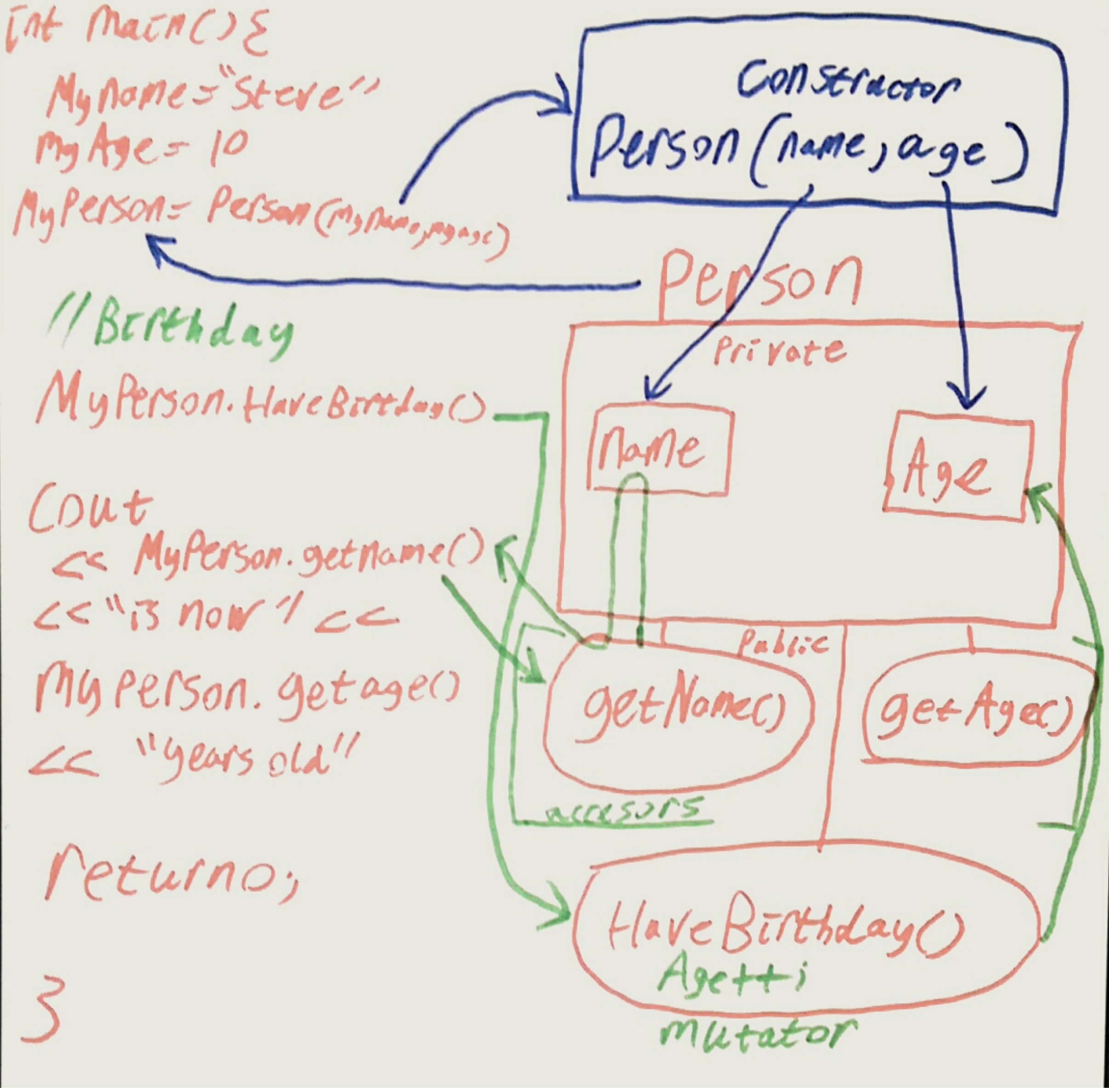
Open Person

```
int main(){
  Person myPerson
  MyPerson.nam="Steve"
  my Person. age = 10
  //Birthday
  Myperson.age++
  cout<< myPerson.name()
      << " is now "
      << my person.age()

  return

}
```

Private Person



```
int main(){
  MyName="Steve"
  myAge=10
MyPerson= Person(myName,myage)

 //Birthday
MyPerson.HaveBirthday()

Cout
 << MyPerson.getName()
 << "is now" <<
 myperson.getage()
 << "years old"

 return0;

}
```