Practicum 5 – Maps and Closures

Assignment Details

Assigned: February 18th 2014. Due: February 20th, 2014 at midnight.

Background

One of the requirements of PA1 Part 2 – using a data structure to hold function pointers and call them to construct objects – stands out, to many, as the scariest. This practicum covers how to handle it.

Before we start

If you do not already have a working `XMLSerializable` base class, check the course web page – working .cpp and .h files have been posted. If you use these files (and you are welcome to!), all you should need to do is copy them to your pa1 directory, and then make sure that `Entity` inherits from `XMLSerializable`.

We'll be working in the `program` directory for this assignment; all of the changes are in addition to PA1 Part 2, so both should be able to exist side-by-side.

Getting started

Navigate to your `program` directory; we'll be working there. Copy over the provided files if you need to.

Some Changes to the makefile

For starters, open up your makefile, and make the following changes:

- Remove `main.o` from the `OBJECTS` line (if you had it there, at least; depending on how you have implemented previous practicum assignments, it may not be there…)
- If you have a main.cpp already, add `main.o` to the production that builds `prog1`, so it reads:

```
prog1: $(OBJECTS) main.o
        g++ $^ -o prog1
```

- Add a new production to the makefile as follows:

```
la5: $(OBJECTS) la5.o
      g++ $^ -o la5
```

- And while we're at it, edit the line that does the compilation of the individual C++ files to make sure we're using C++11 features:

```
%.o: %.cpp $(HEADERS)
        g++ --std=c++0x -c $< -o $@
```

So what changes here? This lets us have two targets for the makefile. From the command line, we can use "`make prog1`" to work as it always has, or "`make la5`" to make an executable for this lab.

Remember that to build this lab, we need to use "`make la5`" each time!

Note that if you try to make the new executable right now, it will fail, complaining about not being able to find `la5.o`; so let's go ahead and fix that.

la5.cpp

To make `la5.o`, we need a `la5.cpp`.

Open it up in your favorite editor, and set it up. Go ahead and `#include <iostream>`, `<map>`, `<string>` and `<functional>`. Add `using namespace std;`, too, if you prefer (if not, you'll need a lot of `std::` in the program; it's your preference, but you probably want the `using` statement in there…).

Then, inside the main function, we're going to set up a `map`. A `map` is a C++ STL class which, er, maps something to something else – in this case, we're going to map strings (which will be class names) to function pointers (to construct an object of that class, as to meet the requirement for PA1 Part 2).

`map` is a parameterized class, like `vector`, except it takes two arguments – the type mapped from (the "key") and the type mapped to (the "value"). In our case, we want to map from a `string` to a function pointer that returns an `XMLSerializable` pointer – which is, if we use the C++11 syntax, `function<XMLSerializable*()>`. As such, to declare a map of this type, add to main:

```
map<string,function<XMLSerializable*()>> mapConstructor;
```

Now, before we do anything with the map, we need to fill it. And to fill it, we'll need to know what classes we can use; there are really only four classes we'll see in the XML for this assignment, so let's go ahead and use those -- `#include` your header files for your Armor, Weapon, Item, and Creature classes.

Now we're going to populate the `map`. Let's start with Item; outside of your main function, define a function that constructs an Item object, and returns it as an `XMLSerializable` pointer:

```
XMLSerializable * constructItem()
{
	return new Item;
}
```

(Note that depending on how you declared your constructors, you may need to modify this slightly!)

Make sure you understand what this function does – we construct an Item object via the `new` operator (which gives us an `Item*`), and then return it as an `XMLSerializable*`. Remember that you can always cast a pointer to a derived class to a pointer to a base class.

Then, inside your main function, after you've declared your map, associate that function with "Item":

```
mapConstructor["Item"] = constructItem;
```

Just like we've talked about before – `constructItem` here is an expression, the expression is of function pointer type, and evaluates to the location in memory where `constructItem` resides, so we can then store that variable in the map.

Now we can certainly do it exactly that way for the remaining classes, but let's do at least one of them in a slightly easier way:

```
mapConstructor["Enemy"] = []() { return new Enemy; };
```

Ok, now, this is a bit new. The C++11 spec allows you to declare anonymous functions in the middle of code, called closures. The `[]` is what tells C++ that we're trying to declare one; the code has (basically) the same result as doing it the other way – a function gets declared, stored in memory, and its memory location then stored in the map. And, again – we have logic stored as data, which we can look up later. This is pretty cool.

And speaking of the looking up…

To finish up the practicum, we're going to ask the user for a class name[1], then look it up in our `map`, and then try to construct an object based off of it.

And how do we do this? Just about the same way we set the values:

```
function<XMLSerializable*()> pFunc = mapConstructor[sLookup];
```

And this gives us a function pointer from the mapping[2]. But, since we're just asking the user to come up with text from the keyboard, we might not have found anything. How do we tell? By seeing if the value we get back is `NULL`:

```
if( pFunc == NULL )
{
	// do stuff if we didn't find anything…
}
else
{
	// do stuff if we did find something in the map
}
```

And now to finish off the assignment:

If there's nothing there, tell the user.

If there is, call the function pointer[3], and tell the user if a non-`NULL` value came back from the call.

Aaaand we're done.

---

[1] `string sLookup; cin >> sLookup;` is a pretty reasonable way of reading it in. Make sure to prompt the user with something useful…

[2] There is an important caveat here; when you use the `[]` operator on a `map`, it *always* constructs a new object at that location if there is not one. In our example, it works well because the default value of a `function` object is equivalent to `NULL`, but is a common gotcha in C++ programming.

[3] As covered previously, if `pFunc` is our variable holding a function pointer, `pFunc()` will call that function. Note that calling a `NULL` function pointer does not have pleasant consequences.

Requirements

- Submitted in your `program` directory
- You should handle `Item`, `Enemy`, `Weapon`, and `Armor` – the four classes that exist in the sample XML file on the class web site.
- The `la5` program, when run, should ask the user for a class name, and construct an object of that class, and report success when done, or failure if not. Nothing more, nothing less.
- The construction of the class should be done via looking up a function pointer in a `map`.
- We'll be talking about this in lecture on Friday, as well as how to combine everything together…