Programming Assignment 1 – Part II

Assignment Details

Assigned: February 10th, 2013. Due: February 22nd, 2013, midnight.

Background

Now that we've got the class library set up from Part I, we're going to add XML serialization functionality. This assignment should provide a reasonable introduction to XML as well as strengthen skills with recursion and object oriented design.

So, what is "Serialization"?

Serialization is, broadly, the conversion of objects in memory to files on disk (and vice versa). You'll see the term "XML serialization" a good bit, as it's common for someone designing a large program to want to be able to convert something – whether it's data from a database, configuration information, or as we're going to be doing, world definitions for a game – into a form that can be easily edited on disk, and then easily read back in to the program itself.

And how does this apply to PA1?

You've got the eight classes from PA1 Part I set up so that they properly inherit from each other and hold the data they need to. Now we're going to add the functionality to both save a set of objects of those classes to XML, and then read them back from the XML to create objects in memory.

Getting started

We have a nice class hierarchy, and we're going to add another level to it. You are going to create an `XMLSerializable` class, and the `Entity` class will inherit from it. This means any functionality we implement in `XMLSerializable` will be accessible from all of the classes in our hierarchy (inheritance of behavior!) and we'll also have a number of virtual methods that will need to be overridden (inheritance of interface!) to be called in the appropriate class.

`XMLSerializable` will define all of the methods for XML handling; note that most of these methods will require overrides in most or all derived classes.

Basic methods of `XMLSerializable` (all should be virtual!):

- `void dumpObject()` – dumps the contents of the object to the console. We need some way to verify that the objects we read in from XML are correct after being read in! This method should be overridden in *every* derived class, to make sure that all data is properly written to the console. Will provide the name of the class and call `dumpObjectData`.
- `void dumpObjectData()` – dumps just the member variables to the console. Again, should be overridden in all derived classes – and all of these should call the version in their immediate base class to make sure that all data long the inheritance chain gets output…

- `void writeFragment(ostream & output)` – writes the current object as an XML fragment to the given output stream. Will provide the tags for the class itself (e.g., "`<Item>`" and "`</Item>`") and call `writeDataAsFragment` to output properties.
- `void writeDataAsFragment(ostream & output)` – writes the member variables of the current object as an XML fragment to the specified output. Much like `dumpObjectData`, it will call this in the base class to ensure that all data is output without adding redundant code.
- `void setElementData(string sElementName, string sValue)` – during deserialization (that is, reading the data back from XML stored on disk), the deserialization process will call this in an object when it reads in an element and the contents of that element. Much like `dumpObjectData` and `writeDataAsFragment`, this should be overridden in all classes and call its base class to ensure that all classes get a shot at trying to store the data.

A framework for XML deserialization will be discussed in class on Friday (2/14) and a skeleton of it will be provided. This is the hardest part of the assignment, and the most help will be provided regarding it.

Requirements

The final program will need to, in order:

1. Ask for a filename to read in from the disk
2. Read in the XML and deserialize the data into objects in memory
3. Display the contents of those objects (via `dumpObject`, above)
4. Ask for a filename to write the objects back out to disk in XML format
5. Serialize the objects back to the disk

Additionally, the program must be able to detect bad XML input and output a message when bad XML has been detected. And no, "Segmentation Fault" does *not* qualify as this message.

The program must also utilize function pointers (or a similar mechanism) to dynamically construct classes during deserialization. Note that "a similar mechanism" means something that looks up a name in some structure and then executes what gets looked up; a chain of if statements is *not* valid. We'll discuss this in class at least once more.

The entire program plus a simple `makefile` must be provided – `make prog1` should cause the program to be built, and the executable must be called `prog1`.