

## Assignment 3: cant

---

*Oh, yes you can!*

Ship Date: March 8, 2005, 11:59pm (despite what it says on the syllabus)

This assignment focuses on two major C++ language features – operator overloading and inheritance. In Part 1, you'll add some operators to an existing XML class. In Parts 2 and 3, you'll implement an XML-driven build system for C++ projects.

This project involves more code and more design decisions than previous assignments. So be sure to start early!

Also, an important general note about this assignment: This assignment involves working with XML documents and executing code in response to the data. You may assume that the XML input is valid, both in terms of syntax and semantics.

### Part 1: Complete XMLElement

In Part 1, you'll begin with an existing XML class and add operator overloading. Your new class will support concatenation (through + and +=), comparison (through == and !=), and easy querying (through []).

#### The Modified XMLElement

The textbook (Ch. 24) presents a class called XMLElement, which lets you easily build basic XML structures in code, and later output the structure as a well-formed XML document. In this assignment, we'll give you a modified version of XMLElement that no longer uses pointers. For example, the following code constructs an XML document with a root element, and two subelements:

```
XMLElement myElement;  
myElement.setElementName("root");  
  
XMLElement subElement1;  
subElement1.setElementName("sub");  
subElement1.setAttribute("foo", "bar");
```

```

XMLElement subsub;
subsub.setElementName("subsub");
subElement1.addSubElement(subsub);
myElement.addSubElement(subElement1);

XMLElement subElement2;
subElement2.setElementName("anotherSub");
subElement2.setTextNode("hi there");
myElement.addSubElement(subElement2);

cout << myElement << endl; // output the XML document

```

Since the XMLElement class has overloaded the << operator, the final line will output the XML document, which looks like this:

```

<root>
  <sub foo="bar">
    <subsub>
    </subsub>
  </sub>
  <anotherSub>hi there</anotherSub>
</root>

```

The modified version of XMLElement also contains getters that allow you to query the contents of an XML structure. For example, the following code will output the text node of each sub element of myElement:

```

vector<XMLElement> subElements = myElement.getSubElements();
for (int i = 0; i < subElements.size(); i++) {
    cout << subElements[i].getTextNode() << endl;
}

```

You should grab the new version of XMLElement from `/usr/class/cs193d/assignments/cant`. Take a look at the `.h` file to see the additional features that we're providing.

## XMLParser

We're also giving you a very simple XML parser. We're not kidding when we say *simple*. This parser is bare bones: it only handles basic, well-formed XML. The input can't have a prolog or DOCTYPE (it must start with the root element), it can't have comments, and there's very little error handling. In Part 2, you'll be using XMLParser to create an XML-driven build system for C++. For now, you should grab the source files for XMLParser and give them a quick scan so you can see XMLElement in action and understand the basic usage of XMLParser. We've also included the unit tests that we wrote when developing XMLParser in XMLParserTest.cpp. If you want to run the tests, just write a quick main() that calls XMLParserTest::test().

### Step 1: Implement Implicit Constructors

Remember in lecture when we talked about *implicit constructors*? No? Okay, well an implicit constructor is a constructor that takes a single argument of a particular type. The compiler will automatically construct objects of your class from objects of that type. This will come in handy when you implement operator+.

The XMLElement class could use a constructor that takes a string. Constructing an XML element with a string should use XMLParser to turn the string of XML into a valid XMLElement. Here's the prototype:

```
class XMLElement
{
public:
    XMLElement(); // the existing constructor
    XMLElement(const string& inString); // the implicit string ctor
    [etc]
};
```

Once this constructor is in place, you should be able to construct an XMLElement from a string:

```
XMLElement myElement("<root><sub foo=\"bar\"/></root>");
```

Having this implicit constructor should make it a little easier to write tests. Instead of creating a string stream and an XMLParser, you can just create the element as shown above in a single line of code. You will almost certainly want to use an XMLParser in your implementation of the string constructor. Take a look at XMLParserTest to see how to convert a string into a stream.

You'll also want to add a `const char*` implicit constructor so that your operators will work with C-style strings. It turns out that C++ doesn't make two-degree connections of implicit constructors for you. With only the string constructor, you could do this:

```
result = myXMLElement + myString;
```

But you couldn't do this:

```
result = myXMLElement + "<foo><bar></bar></foo>";
```

To get the latter to work, simply add this constructor:

```
XMLElement(const char* inString);
```

### Step 2: operator+ and operator+=

What does addition mean on an XML element? We'll define addition as the concatenation of two elements into a new, common parent. For example, suppose we start with these two elements:

```
<elementOne>
  <subElementOne/>
</elementOne>
```

```
<elementTwo>
  <subElementTwo/>
</elementTwo>
```

If we add `elementOne + elementTwo`, we'd get this result:

```
<result>
  <elementOne>
    <subElementOne/>
  </elementOne>
  <elementTwo>
    <subElementTwo/>
  </elementTwo>
</result>
```

Note that the + operation creates a new element named "result". The caller will most likely rename this element, but it's important to set it to *something* so that it's a valid element.

The += operator works a little differently. Instead of creating a new element with the two items being added, it appends the added element onto the destination. Given the two elements above, if we say elementOne += elementTwo, the result would be:

```
<elementOne>
  <subElementOne/>
  <elementTwo>
    <subElementTwo/>
  </elementTwo>
</elementOne>
```

Note that this is a little different than the way that += normally works. In most cases, you can think of += as being equivalent to a + operation followed by an assignment. In the case of XMLElement, the two operations have slightly different implementations.

### Step 3: operator== and operator!=

Comparing XML elements would be really useful. The tests in XMLParserTest compare XML elements manually by looking through element names, attribute names, and subelements. Once operator== is implemented, comparing XML elements becomes as easy as comparing ints.

When implementing == and !=, keep in mind the following:

- For subelements, order *does* matter.
- For attributes, order *does not* matter.
- For two elements to be equal, they must have the same name, same text node (if any), same attributes (if any), and same subelements (if any).

### Step 4: operator[] const

XML has a companion technology called *XPath* that lets you describe a location within an XML document. XPath is almost a language unto itself – there is syntax in XPath to select and manipulate data within an XML structure. For our purposes, we'll just look at a very specific part of the XPath syntax. The following XPath selects all of the sub-nodes of "sub", which is itself a sub-node of "root" in the example on page 2:

```
/root/sub/*
```

An XPath is relative to a particular XMLElement. In the above case, the path is relative to the <root> element since it begins with "/root". You can assume that the XPath is always starting with the first element in the path. A slash in an XPath means that you're moving from one node to one of its child nodes. And "\*" means that you're asking for all sub-elements of the current element. For our purposes, you don't need to worry about XPaths that *don't* end in \*. You can assume that the caller is always ultimately looking for the child elements of some element.

operator[] const should allow a user to select the children of an element using a simple XPath, like the one above. If the XPath is invalid (e.g. one of the elements in the path doesn't exist), you can return a vector of 0 elements. If the element at any point in the path has multiple subelements with the given name, you can pick the first one. Here is the prototype for operator[] const:

```
std::vector<XMLElement> operator[](std::string inXPath) const;
```

So imagine we've parsed the XML from page 2 into an XMLElement object named rootElement. If we want to find all of the children of the element "sub," we just build a path to it:

```
std::vector<XMLElement> subChildren = rootElement["/root/sub/*"];
```

Below is a function that may help in your implementation. It takes a string as input and returns the next /-delimited token in the string. It will also modify the input string so that you can call the function repeatedly in a loop to get all the tokens.

```
/**
 * Reads the next token, up to a '/' character. If there is no
 * '/' in the string, it returns the string itself. The input string
 * is modified in place.
 */
string nextToken(string& data)
{
    int nextLoc = data.find('/');
    if (nextLoc == -1) {
        // no '/' found!
```

```
    string retVal = data;
    data = "";
    return retVal;
}
string retVal = data.substr(0, nextLoc);
data = data.substr(nextLoc + 1);
return retVal;
}
```

Feel free to use, modify, or ignore nextToken() as you see fit.

### **Epilogue: Part 1**

As you were writing your unit tests in Part 1, you may have noticed that client code of XMLElement was getting easier to write as you implemented operator overloading. That's one of the nice things about C++. There are many language features that aren't strictly necessary but that can help you write classes that behave as expected and allow simple syntax like + and ==. See also: the string class – the most perfect class ever written in C++.

Some of the operators you implemented in Part 1 may come in handy in Part 2, although you shouldn't feel forced to use them unless it's appropriate. You might even decide that you want to overload additional operators to make your Part 2 solution cleaner! In particular, you might want to extend your XPath support to avoid iterating through XMLElements in Parts 2 and 3.

## Part 2: The Build System Skeleton

Note: You do **not** need to turn in unit tests for Parts 2 and 3, but you may want to write them for your own testing purposes.

Now that we have some flexible XML tools at our disposal, it's time to create our XML-driven build system for C++. In class, we talked about *make*, the traditional build system of choice for C++ projects. *make* has a number of advantages over command-line invocation of *g++*, but it also has a number of shortcomings. Makefiles are notoriously difficult to read, and the *make* system isn't as extensible as one might like.

In the Java world, there is a popular build system called *ant*. *Ant* uses XML files to describe the build system, which are generally more readable than Makefiles. *Ant* also divides build operations into *tasks*. Third party developers can write their own new types of tasks, allowing a high level of extensibility.

*Ant* isn't necessarily limited to compiling Java code. Even though *ant* itself is a Java program, it can be used for C++. However, there is a fair amount of overhead (installing a JVM, setting up environment variables, etc.) involved in doing that. C++ programmers would prefer to have a build system written in C++. So that's what we're going to write.

### Introducing cant

*cant* (C++ version of *ant* – get it?) is similar to Java's *ant*, but has a subset of the functionality and a slightly different syntax. *cant* projects are driven by an XML file, usually called *build.xml*. Here's a sample build file:

```
<project name="textr">
  <property name="basedir" value="/home/klep/cs193d/textr"/>
  <property name="srcdir" value="${basedir}/src"/>
  <property name="bindir" value="${basedir}/bin"/>
  <property name="executable" value="${bindir}/textr.app"/>

  <fileset name="srcset">
    <file path="${srcdir}/*.cpp"/>
    <file path="/usr/class/cs193d/common/testutils.cpp"/>
  </fileset>

  <fileset name="tempfiles">
```

```

        <file path="${executable}"/>
    </fileset>

    <target name="default">
        <compile fileset="srcset" output="${executable}"/>
    </target>

    <target name="update">
        <echo value="Touching all source files and exec..."/>
        <touch fileset="tempfiles"/>
        <touch fileset="tempfiles"/>
    </target>
</project>

```

### build.xml Walkthrough

Let's take a look at the details of the build file shown above.

**<project>** - Every ant build file has a root element named "project". The project element has a mandatory attribute called "name", which is the name of the project we're building.

**<property>** - Properties are simply key / value pairs that are used elsewhere in the build file. By defining properties, you can keep your build file clean and avoid repeating the same file paths and other values over and over. Properties can also be defined in terms of other properties by surrounding the other property name with \${ and }. This is called *variable substitution*. When ant encounters a value that has these symbols, it replaces that part of the value with the value of the corresponding property. Note that properties can be defined in any order and at any time, as long as they are subelements of <project>. So even if we moved the "basedir" property down to the bottom of the file (just above the closing </project> tag), everything would still work. Also note that properties can be used within any attribute value elsewhere in ant. For example, the "output" attribute of the <compile> tag is using the property "executable".

**<fileset>** - Since much of what ant does is simply dealing with files, it has a special notion of a fileset – a group of files. Any operation that deals with one or more files (deleting, compiling, etc.) will be processing a particular fileset. A fileset has a name and one or more <file> elements.

**<file>** - A file element simply has a "path" attribute that gives the location of the file. You can use properties within file paths and you can use wildcards (e.g. \*.cpp) to refer to multiple files within a single <file> tag.

**<target>** - A target is the label for a set of tasks that are performed when cant is invoked. A given project generally has several targets. In the example above, there is a default target, which compiles the project. There is also a "clean" target that deletes the executable program.

**Tasks** – Inside of the <target> tag are a series of actions, called *tasks*. Each task performs a single action, such as compiling a fileset of source files, or moving a directory. Tasks are where the extensibility of XML really shines. When cant is processing a target, it looks at each sub-element in order, instantiates the corresponding task, and tells that task to do its thing. A cant user will be able to easily add their own new types of tasks. Note that tasks can have an arbitrary number of attributes, but they can't contain subelements or text nodes.

### **Implementing the Build System Basics**

A barebones Cant.cpp and Cant.h has been provided in the assignment directory. There's not a whole lot there to begin with. Basically, the Cant class serves as both a build file parser and the build execution engine. The main program will construct a Cant object with a stream (could be a file stream, could be a string stream for testing) and then call build(target) to instruct the Cant object to execute the tasks corresponding to the given target.

The implementation of Cant is up to you, but here are some guidelines and suggestions:

- You'll want to turn the stream into an XMLElement inside the constructor and extract the information you need out of it. You'll may choose to store the XMLElement as a data member of Cant, but you'll probably also want to do some pre-processing. In other words, there are certain data, like properties, that are easier to extract out of the XMLElement one time up front and store in some other format, like a map.
- Evaluating properties is tricky (due to variable substitution) and you have to do it in multiple places. Our suggestion is to look at the find() method of string and to write lots of unit tests for \${x} replacement. Also, think carefully about whether you want to do the replacements all up front, or only when the property is requested. And don't forget that a value could have multiple \${x} references!

- For now, the `build()` method can just return `true`, since we haven't implemented Tasks yet. In Part 3, you'll fill in the `build()` method.
- We highly recommend adding a `testing_getProperty()` method so that you can test parsing of properties and variable substitution using `#{x}`.

### **Evaluating Part 2**

At this point, your Cant program isn't quite functional yet, but there's actually quite a bit in place already. You've built a program that can read and process XML. It also has its own XML-based language that supports variable substitution, and a data structure through filesets. Ultimately, that's precisely what you're doing – writing an interpreter for an XML-based language.

## Part 3: Build a Library of Tasks

Before you jump into writing Tasks, you have to decide how they'll be organized and implemented. Keep in mind that the goal is to use tasks as a point of extensibility. So it needs to be easy for someone to add a new task<sup>1</sup>.

You'll want to start with a base class called Task. Task developers (including yourself) will write subclasses of Task that can be referred to in the build.xml file. The design of this class is totally up to you, but keep the following things in mind:

- The Task class will have to integrate with your Cant class. So it'll have to be pretty easy to instantiate a Task, provide it with its attributes, and tell it to execute at the proper time. Each task should be able to return a success or failure. If a task fails, the build immediately stops and prints a helpful error, indicating which task has failed.
- The Cant object will need to look at a subelement of <target> and somehow determine which Task to instantiate. This is a key design decision – how do you provide a link between an XMLElement called <compile> and a task called CompileTask? And how can you do that in a way that makes it easy for people to add new Tasks with minimal changes to other code?
- Tasks can have state. So each time the Cant object encounters a task, it should create a new instance. It is *not* valid to have a single instance of each type of Task that simply gets reused.
- Tasks have a name, a variable number of arguments (the XML attributes of the corresponding element), and some behavior. That's it! So all the base class needs to do is provide a way for a task to get its arguments and a method that tells it to "go".
- Task arguments can include variable substitutions. For example, the <compile> task up above refers to the "executable" property. You'll want to have the build() method perform any necessary variable substitutions *before* handing the arguments over to the task.

---

<sup>1</sup> In the Java version of ant, you can add tasks dynamically. That is, you can drop a task in a directory, and just start using it right away. Cant tasks won't be quite so simple to add. Assume that the user will recompile cant with their new task.

You'll probably end up developing the Task class and the build() method of Cant in tandem. As you start implementing build(), you'll discover some new features that Task needs to have. Most of the logic will live in Cant::build() at this point – Task will just be a pure virtual base class that has some basic functionality to support its subclasses.

### First Task: <echo>

The first task you'll implement provides a simple "echo" feature. This is useful for debugging and providing output while a target is being executed. All the <echo> task does is output the associated value to the screen, followed by a carriage return. Here's a sample echo.xml build file:

```
<project name="myproject">
  <property name="foo" value="bar"/>
  <property name="biz" value="{foo}"/>

  <target name="default">
    <echo value="Starting to echo!"/>
    <echo value="Biz is: {biz}!"/>
    <echo value="done!"/>
  </target>
</project>
```

If everything is working, the output of running the "default" target should be:

```
Starting to echo!
Biz is bar!
done!
```

Since this first task is so simple, you should use it as an opportunity to iterate on your Task base class design. Remember that once your infrastructure is in place, adding a new Task should be possible with minimal (on the order of 1 line of code) changes to other parts of the code.

Now that you actually have a task in place, you can swap in our CantMain.cpp file to see it in action. Our main will open a file called "build.xml" and execute the "default" target. So for this part, you should also grab "echo.xml" and rename it "build.xml". Make sure it's in the same directory that your application will execute from!

### Next Up: <compile> and <touch>

The <compile> and <touch> tags are the first two useful ones. After implementing these tags, you should be able to process a file like the Textr build file given at the beginning of the assignment.

These tasks will use the `system()` function to issue commands to the UNIX shell. If you're developing on Windows, these commands probably won't work. I recommend that you have the tasks just output the strings that they *would have* executed in UNIX. Then, when you switch to UNIX for testing, turn the output strings into `system()` calls.

The `system()` function simply executes a string argument as a shell command. For example, I could compile the file "test.cpp" in the current directory by doing this:

```
int result = system("g++ test.cpp");
```

The result of `system()` is the return value from the command. So you can check the result for something other than 0, which indicates an error. Note that `system()` takes a C-style string. So if you have a C++ string object, you'll need to call `myString.c_str()` to convert it.



#### **Pensive Scott says:**

In general, you should be very careful with `system()` because it can be a major security hole. We won't worry about it for this assignment, but imagine the damage users could do if they knew that one of their input values was being passed to `system()` – they could read an arbitrary file or delete your whole filesystem!

For the <compile> task, the user should be required to provide a "fileset" attribute with the file(s) to compile and, optionally, an "output" attribute with the destination of the executable. The task should simply call `system()` with "g++ [files] -o [output]". If the <compile> task fails (i.e. g++ returns a non-zero result), the build system should report an error.

The <touch> task simply executes the UNIX command "touch" on the files specified in the "fileset" attribute. All touch does is update the file's modification date, creating the file if it doesn't already exist.

You should write your own build.xml file to test your new tags.

### **Last Task: <debug>**

Last up is a task called <debug>. This task is very similar to <compile> (hint hint) except that it also passes the "-g" flag to g++. Other than that, all of the attributes are the same. The command it issues should be something like "g++ -g [files] -o [output]".

The <debug> task will allow you to make build files that have a separate "debug" target that builds the application with debugging information.

### **Evaluating Part 3**

Congratulations! You've built a functional bare-bones ant-like build system. Sure, it only knows how to compile and touch files, but it also has some pretty advanced features, like file groups and variable substitutions. And since you designed Task to be extensible, you (or another developer) could easily add new functionality. Perhaps a "move" task? A "delete" task? Maybe implement conditionals? There are tons of features like this that have been created for ant, and you've built a solid foundation for extending the feature set of cant.

## **Getting Started, Grading, etc.**

We've provided several source files, including the revised XMLElement, the fabulous XMLParser, the main() for Part 3, an echo.xml file, and some unit tests in /usr/class/cs193d/assignments/cant

As part of this assignment, you'll need to **build your own Makefile!** That's right, you're writing a Makefile that builds a C++ version of ant. You should also create and turn in a build.xml that builds your cant implementation using cant. It may not be as full-featured as your Makefile, but it'll be more readable.

### **Deliverables**

Here is everything you'll need to turn in:

- Your modified XMLElement.h and XMLElement.cpp
- XMLElementTest.cpp where you've added tests for the new features of XMLElement
- Your modified Cant.h and Cant.cpp
- Your Task class and header
- All of your Task subclasses and headers
- Your Makefile
- Your build.xml file for the Cant project
- Your README file

Your README file should contain:

- Your name
- Your email address
- Any comments about the assignment
- A 1-2 paragraph description of how you've implemented Tasks (e.g. How are new Tasks added? How does Cant instantiate Tasks?)

When you're ready to turn in your assignment, type `"/usr/class/cs193d/bin/submit"` on the elaines and follow the directions.

### **Grading**

The following areas will be considered when we grade your assignment:

*Correctness.* We will run a suite of our own tests against your code.

*Testing.* We will examine your unit tests for Part 1 to ensure that they adequately cover the use cases. You don't need to go nuts with testing. Just make sure that you have a test for each feature and you've covered a number of edge cases.

*Scalability.* We will inspect your code for memory leaks and make sure that your classes are good C++ citizens (e.g. copy ctors when needed, good use of operator overloading, etc.)

*Readability.* We will look over your code expecting to see appropriate variable and method names, good functional decomposition, and appropriate use of language features.