Parsing XML using TinyXML

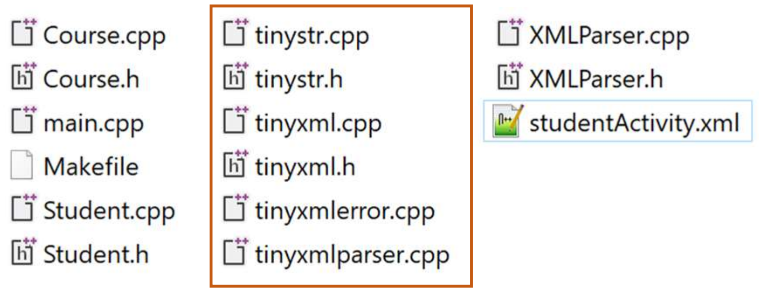
ECE 39595C Object Oriented Programming with C++

September 21st, 2021

Links

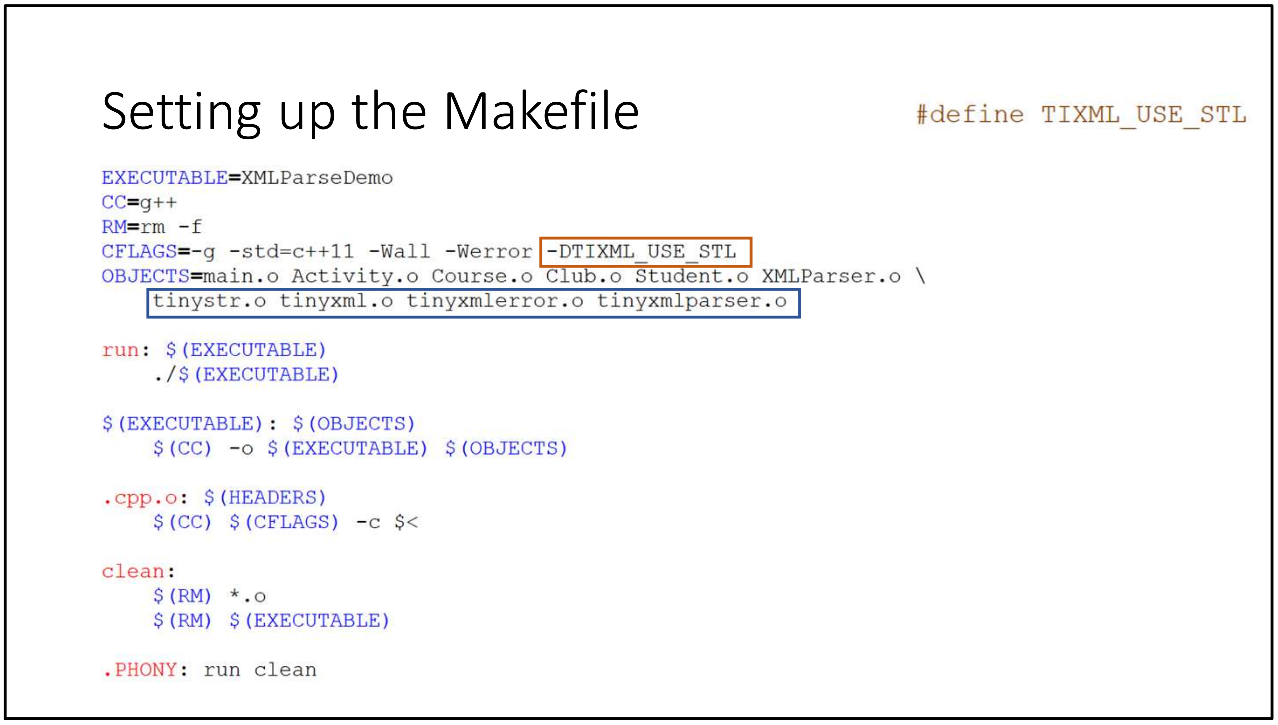
- Website:
  - http://www.grinninglizard.com/tinyxml/
- Download:
  - https://sourceforge.net/projects/tinyxml/files/latest/download
- Documentation:
  - http://www.grinninglizard.com/tinyxmldocs/index.html
- Example XML parsing project on Brightspace

Setting up the project



In addition to your defined classes and source files for your project (e.g. Course, Student, XMLParser, main), TinyXML uses a total of 4 source files and 2 header files to define all its logic. These are the two tinystr files, the two tinyxml files, tinyxmlerror, and tinyxmlparser. You can safely copy these into your project.

One advantage to these being source and header files is we can use them on any OS, rather than having to worry about the OS used to compile the library.



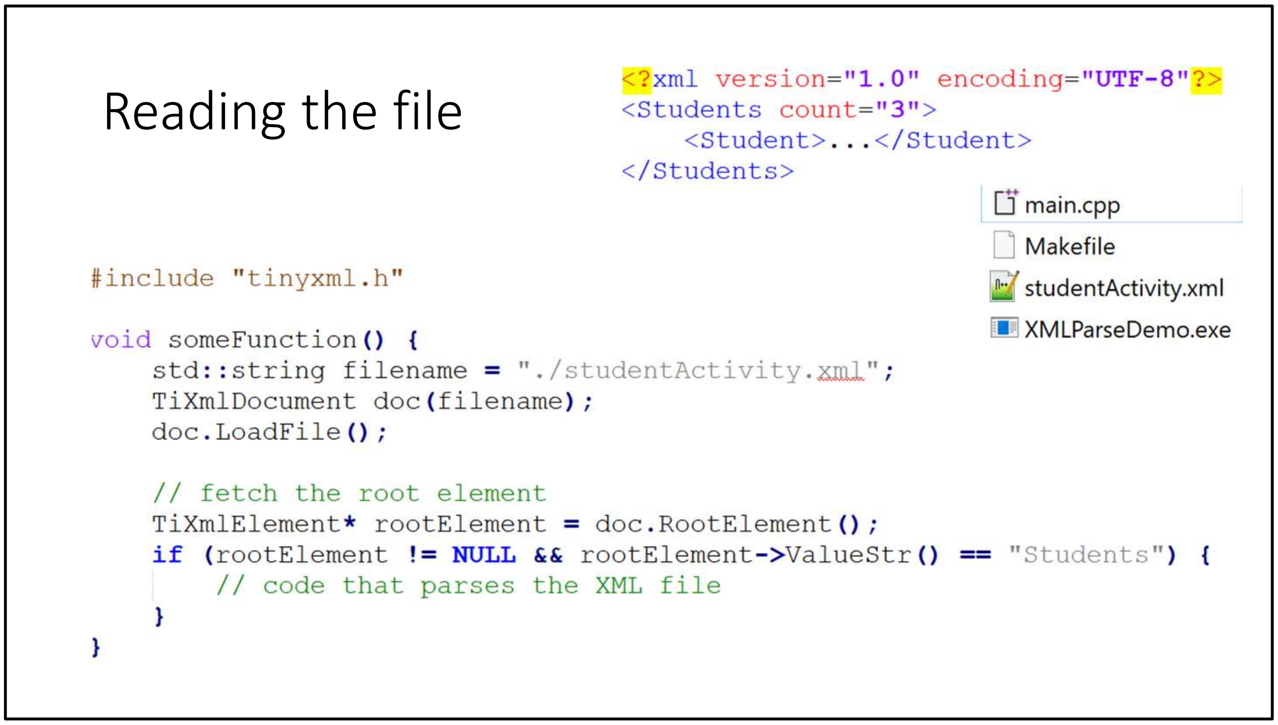
Setting up the Makefile

```
#define TIXML_USE_STL
```

```
EXECUTABLE=XMLParseDemo
CC=g++
RM=rm -f
CFLAGS=-g -std=c++11 -Wall -Werror -DTIXML_USE_STL
OBJECTS=main.o Activity.o Course.o Club.o Student.o XMLParser.o \
    tinystr.o tinyxml.o tinyxmlerror.o tinyxmlparser.o

run: $(EXECUTABLE)
    ./$(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
    $(CC) -o $(EXECUTABLE) $(OBJECTS)

.cpp.o: $(HEADERS)
    $(CC) $(CFLAGS) -c $<

clean:
    $(RM) *.o
    $(RM) $(EXECUTABLE)

.PHONY: run clean
```

To set up a Makefile using TinyXML, we simply need to include all four of the source files as part of the objects we use to compile.

In addition, it makes things a lot easier if we define the compile flag –DTIXML_USE_STL as that enables using the TinyXML functions using std::string. If we did not enable that, we would be required to use const char* for strings. Setting this flag in the compiler is equivalent to adding the top right define statement to all our source and header files that include the tinyxml header.



Reading the file

```
<?xml version="1.0" encoding="UTF-8"?>
<Students count="3">
    <Student>...</Student>
</Students>
```

main.cpp
Makefile
studentActivity.xml
XMLParseDemo.exe

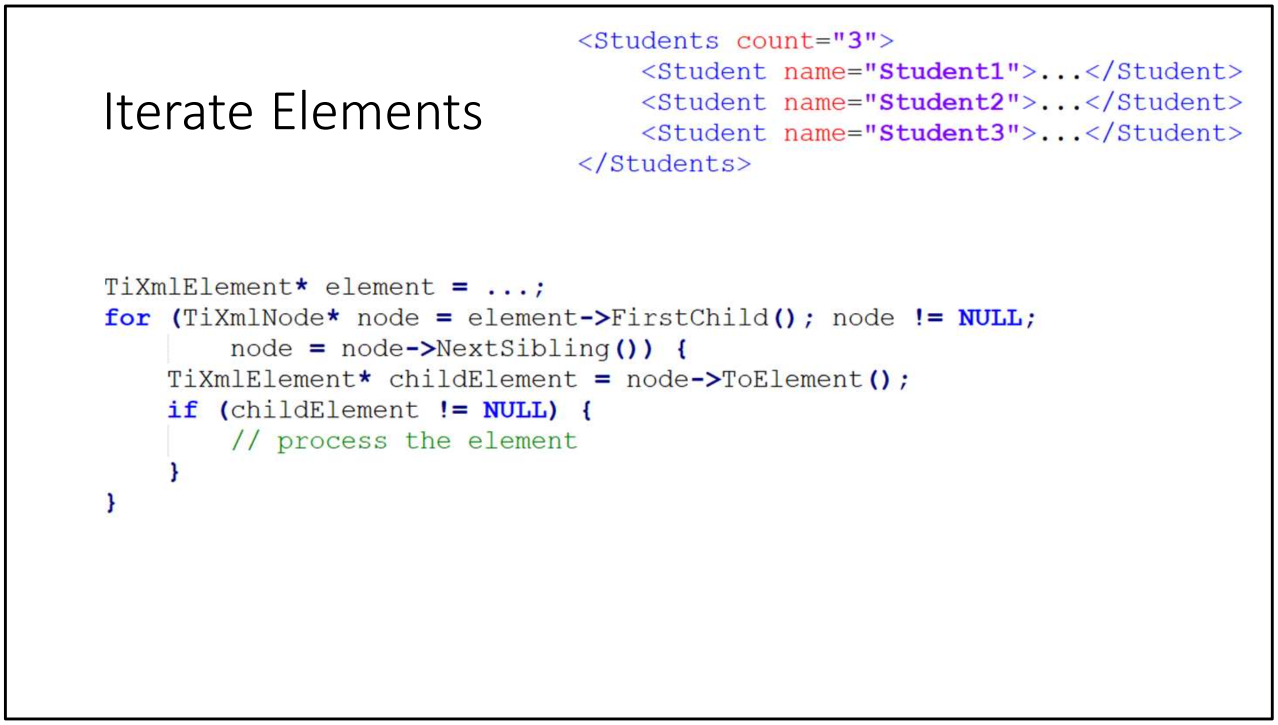
```
#include "tinyxml.h"

void someFunction() {
    std::string filename = "./studentActivity.xml";
    TiXmlDocument doc(filename);
    doc.LoadFile();

    // fetch the root element
    TiXmlElement* rootElement = doc.RootElement();
    if (rootElement != NULL && rootElement->ValueStr() == "Students") {
        // code that parses the XML file
    }
}
```

First, to use TinyXML, we need to include the tinyxml.h header file. This single header includes all the relevant classes for the library, as opposed to using a separate header for each class.

TiXmlDocument doc(filename);
Initializes the XmlDocument object with the given filename. Note the path is relative to the directory where you run your code, in the above example “studentActivity.xml” is in the same directory as the compiled executable.

TiXmlDocument::LoadFile()
Reads the XML data into the XmlDocument object

TiXmlDocument::RootElement()
Returns the first element in the XML. In the example top right, this is the Students element
If the file is missing or invalid, returns NULL.



Iterate Elements

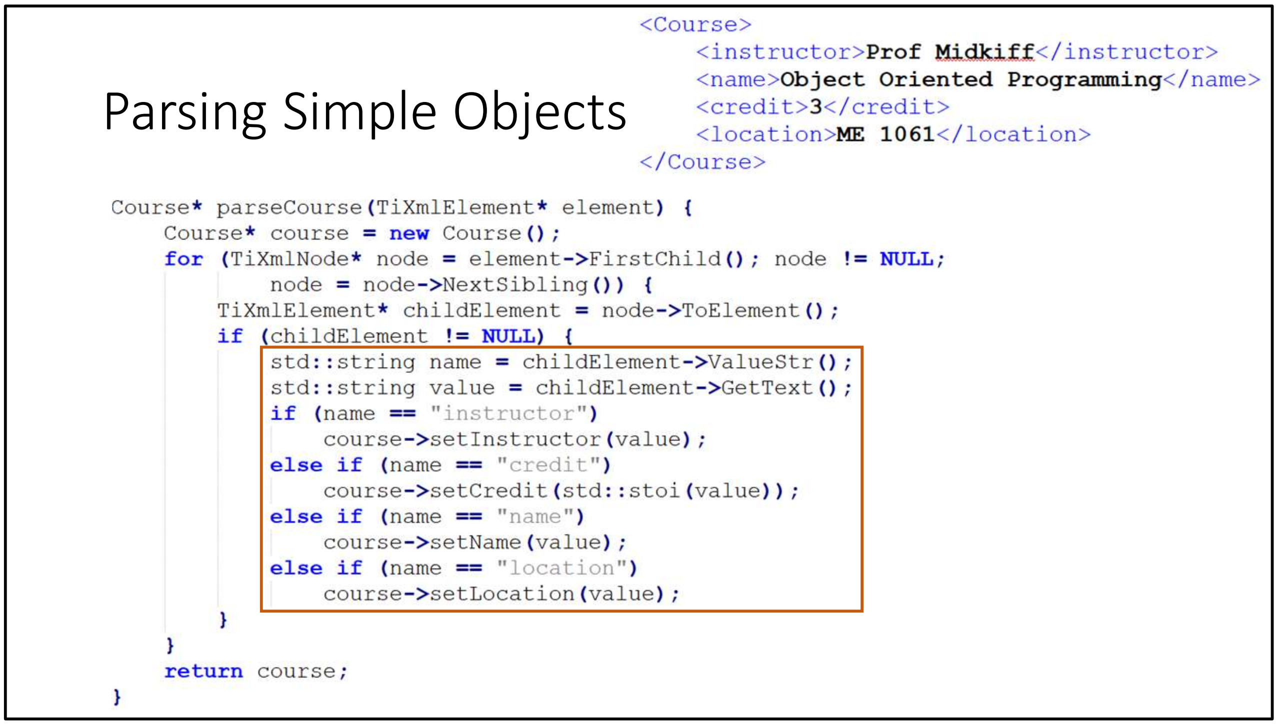
```xml
<Students count="3">
    <Student name="Student1">...</Student>
    <Student name="Student2">...</Student>
    <Student name="Student3">...</Student>
</Students>
```

```cpp
TiXmlElement* element = ...;
for (TiXmlNode* node = element->FirstChild(); node != NULL;
        node = node->NextSibling()) {
    TiXmlElement* childElement = node->ToElement();
    if (childElement != NULL) {
        // process the element
    }
}
```

Internally, elements are stored as a linked list, so the easiest way to iterate though them is like a linked list.

TiXmlNode::FirstChild() returns the element for Student1.
Calling TiXmlNode::NextSibling on Student1 will return Student2
Calling TiXmlNode::NextSibling on Student2 will return Student3
Calling TiXmlNode::NextSibling on Student3 will return NULL, breaking the loop

Within the loop, we call TiXmlNode::ToElement() to convert from a TiXmlNode to a TiXmlElement. This method returns an element pointer if the node is an element, and NULL if it is not an element. Note that there are a few other subclasses of TiXmlNode, but for the sake of the project the main one you need to know about is TiXmlElement.



Parsing Simple Objects

```
<Course>
    <instructor>Prof Midkiff</instructor>
    <name>Object Oriented Programming</name>
    <credit>3</credit>
    <location>ME 1061</location>
</Course>
```

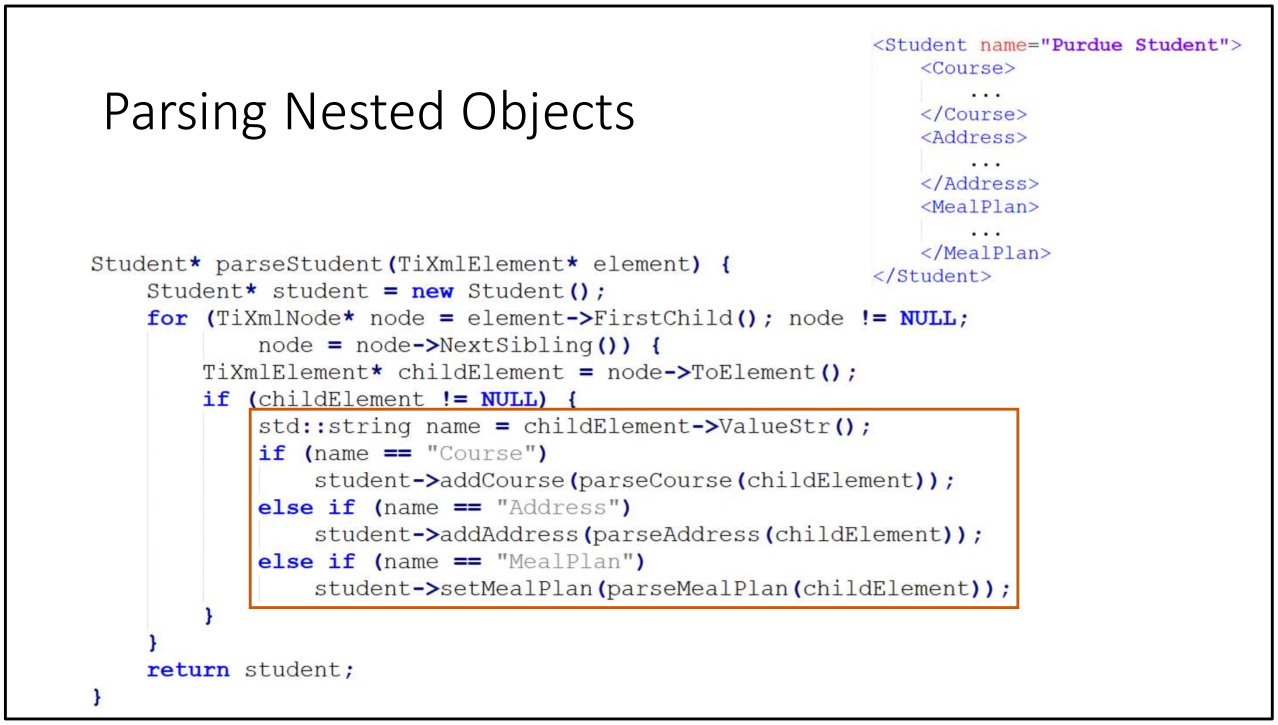
```
Course* parseCourse(TiXmlElement* element) {
    Course* course = new Course();
    for (TiXmlNode* node = element->FirstChild(); node != NULL;
            node = node->NextSibling()) {
        TiXmlElement* childElement = node->ToElement();
        if (childElement != NULL) {
            std::string name = childElement->ValueStr();
            std::string value = childElement->GetText();
            if (name == "instructor")
                course->setInstructor(value);
            else if (name == "credit")
                course->setCredit(std::stoi(value));
            else if (name == "name")
                course->setName(value);
            else if (name == "location")
                course->setLocation(value);
        }
    }
    return course;
}
```

One of the easiest ways to parse the XML is to create a bunch of parseNoun functions, which take a TiXmlElement* as a parameter and return a pointer to the object type.

To parse a simple object like the course above, we will loop through the elements like in the previous example, but make use of two functions:

TiXmlElement::ValueStr(): Returns the name of the XML element being parsed. For example, “instructor” or “credit”
TiXmlElement::GetText(): Returns the text contained inside an XML element. For example, “Prof Midkiff” or “ME 1061”. If the element contains no text, this returns an empty string (for example, calling GetText() on the Course element would return an empty string)

Logic is pretty simple, for each child element of course, if the name matches one of our expected types, we set the proper value in the course object.



Parsing Nested Objects

```
<Student name="Purdue Student">
    <Course>
        ...
    </Course>
    <Address>
        ...
    </Address>
    <MealPlan>
        ...
    </MealPlan>
</Student>
```

```
Student* parseStudent(TiXmlElement* element) {
    Student* student = new Student();
    for (TiXmlNode* node = element->FirstChild(); node != NULL;
            node = node->NextSibling()) {
        TiXmlElement* childElement = node->ToElement();
        if (childElement != NULL) {
            std::string name = childElement->ValueStr();
            if (name == "Course")
                student->addCourse(parseCourse(childElement));
            else if (name == "Address")
                student->addAddress(parseAddress(childElement));
            else if (name == "MealPlan")
                student->setMealPlan(parseMealPlan(childElement));
        }
    }
    return student;
}
```

Often in XML, we encounter objects that contain other objects which we wish to parse. We can handle that by simply calling one of our parseNoun functions when we encounter a given tag.

Like the Course object on the last slide, we make use of the ValueStr() function to determine which element we are parsing. However, instead of calling GetText(), we will pass the element directly to the relevant parsing method.

For example, if we encounter a course object, we can call our parseCourse() method from the previous slide, which returns a Course*. That can then be passed into a function on Student to store the object.

Questions?