Incorporate XML-Based Services

What you need

Visual C++ 6.0

IBM alphaWorks XML for C++ Parser 3.0.1

by Brian K. Holman

XML is here to stay, so take advantage of it. Use the DOM API to create and change XML data.

Extensible Markup Language (XML) is revolutionizing the way applications exchange data—particularly on the Internet, where no one vendor can force a proprietary format. XML, a World Wide Web Consortium (W3C) standard, has received tremendous adoption in the two short years since it was created. One reason for XML's widespread use is its simplicity: It's a nonbinary format that conforms to a normalized grammar. Many vendors, including Microsoft, IBM, and Novell, are embracing XML.

In this article, you'll gain a basic understanding of the XML format in the context of an XML-parsing application. You'll learn how to put constraints on the contents of an XML document and how to traverse and manipulate an XML document structure. You'll also learn about another W3C standard, the Document Object Model (DOM). DOM defines a language-independent object-oriented API for manipulating XML. Here, you'll work with IBM's C++ implementation of DOM. With the basic knowledge and techniques you'll learn in this article, you can begin taking advantage of XML in your C++ applications.

In HTML—XML's precursor—you see a defined set of tags with a fixed meaning. For example, a <TITLE> tag has a fixed meaning as the title of an HTML document, and it must appear within the <HEAD> section of the document. A browser also has a defined place where it displays this information. If there is a need for a new tag called <SUBTITLE>, the W3C must extend the standard, and at some point new browsers would incorporate support for this tag. When the new browsers achieve reasonable adoption, we can take advantage of the <SUBTITLE> tag.

Unlike HTML, XML has no predefined set of tags. In the sample XML document for sharing information about movies, I make use of a <SUBTITLE> tag within the <NAME> section of a <MOVIE> (see Listing 1). The application reading the document determines the <SUBTITLE> tag's meaning. Whether the <SUBTITLE> tag is valid in this context depends on the Document Type Declaration (DTD). A DTD defines what tags are valid for a particular document, how often they occur, and where they can occur (see Listing 2).

You should also become familiar with companion standards to the base XML standard, including Stylesheets, Schemas, and Namespaces. Although an in-depth discussion of these standards is outside the scope of this article, you can refer to Links for additional information.

XML | X(ML)-Rated Movies

```
<?xml version="1.0"?>
<MOVIE>

<NAME>
<TITLE>2010</TITLE>
<SUBTITLE>The Year We Make Contact</SUBTITLE>
</NAME>

<RATING>PG</RATING>
<STAR>Roy Scheider</STAR>
<STAR>John Lithgow</STAR>
<DIRECTOR>Peter Hyams</DIRECTOR>

<RELEASE>
<STUDIO>MGM</STUDIO>
<DATE>1984</DATE>
<LENGTH>116 minutes</LENGTH>
</RELEASE>

</MOVIE>
```

Listing 1 | In this sample XML document for sharing information about movies, note the <SUBTITLE> tag included within the <NAME> section of a <MOVIE>. The application reading the document determines the <SUBTITLE> tag's meaning.

A New Application Type

When a Web-based application is written, it usually queries a database based on some parameters submitted by the end user, wraps the resulting data in HTML, and ships it off to the browser. The end user then views the results. Most Web-based applications today are written to be directly consumable by the end user.

Although developers will continue to create these traditional Web-based applications, you'll start to see a new type of application in which the generated content will be consumed by another application and not by the end user directly. These new applications will take advantage of

specialized middle-tier services available on the network. This capability will eliminate the need to develop monolithic Web-based applications that include all the required functionality internally. XML is the ideal open format for these new applications to share structured data and communicate with one another over the network.

Any application that has structured data should be able to represent that data in XML. Microsoft Office 2000 uses embedded XML in an HTML document to preserve the nonviewable data. This enables "round-trip" editing between the Office native formats and HTML.

Let's look at a practical example. If you're building a shopping site, you might choose to build and develop the core shopping cart system based on product data stored in a local database. However, you might have an order fulfillment house that actually keeps the inventory, a clearinghouse that verifies purchase orders and credit cards, and a shipping company that tracks and delivers the order. Even with all these parties involved, you want the order creation, updating, and status process to be seamless to the customer.

Resources

- Objects & Architecture, "Transport Data With XML," by Jim Beveridge [*VCDJ* March 2000]
- Objects & Architecture, "Load XML Documents in C++," by Jim Beveridge [*VCDJ* April 2000]

XML | DTD Player

```
<!DOCTYPE MOVIE [
<!ELEMENT MOVIE (NAME, RATING, STAR+, DIRECTOR, RELEASE)>
<!ELEMENT NAME (TITLE, SUBTITLE?)>
<!ELEMENT RELEASE (STUDIO, DATE, LENGTH)>
<!ELEMENT RATING (#PCDATA)>
<!ELEMENT STAR (#PCDATA)>
<!ELEMENT DIRECTOR (#PCDATA)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT SUBTITLE (#PCDATA)>
<!ELEMENT STUDIO (#PCDATA)>
<!ELEMENT DATE (#PCDATA)>
<!ELEMENT LENGTH (#PCDATA)>
]>
```

Listing 2 | Here's a sample Document Type Declaration (DTD) for our movie XML document. A DTD defines what tags are valid for a particular document, how often they occur, and where they can occur.

XML is the ideal open format for new applications to share structured data and communicate with one another over the network.

Each of these parties would create XML-aware Web services. You could submit the order in an XML format to the order fulfillment house, which would in turn parse the order, verify that the items were in stock, and return the results. You would then display the availability of the various products to the end user. Once the order is finalized, you could submit the payment details to the clearinghouse to verify availability of funds. You continue this process throughout the order's life cycle, to share required information between the parties.

An application or component using XML, as in our shopping site example, is both a consumer and a producer of XML content. DOM defines an API for creating, traversing, and manipulating XML documents. DOM is language-independent but has been mapped to C++ and other languages, such as Java.

This next discussion assumes you have installed the XML4C libraries and added the \include and \lib paths to the Visual C++ directories options (see the sidebar, "Choosing an XML Parser"). The sample code available for download from the *VCDJ* Web site (see the Go Online box for details) includes a simple Win32 console application project called MyParser that I'll discuss to demonstrate some of the API functions you can use with DOM.

Choosing an XML Parser

You can choose from a number of C++ XML libraries that work on Win32 platforms and conform to the DOM 1.0 standard. I have evaluated the libraries from these vendors:

- Microsoft XML Parser (MSXML), included with IE5
- Oracle XML Parser for C++ version 2
- IBM alphaWorks XML for C++ Parser (XML4C) version 3.0.1

All the evaluated libraries support both nonvalidating and validating modes. In validating mode, the parser verifies the XML document is in compliance with its DTD. For example, if you parse Listing 1 including the DTD from Listing 2, and insert <MUSIC>David Shire</MUSIC> as an additional element in the document, the validating parser informs you that the <MUSIC> element is not defined and therefore invalid. In nonvalidating mode, the parser verifies only that the XML is "well-formed"—meaning that every beginning tag, there is also an ending tag.

My coding examples are based on the IBM libraries (XML4C). I found them to be the most mature of the three APIs, with support for exception handling and consistent interfaces. Because all three support DOM, however, they have similar properties and methods.

You must include these XML4C-specific headers at the top of the MyParser.cpp file:

```
#include <dom/DOM.hpp>
#include <dom/DOM_Node.hpp>
#include <parsers/DOMParser.hpp>
#include <util/XMLException.hpp>
#include <util/PlatformUtils.hpp>
```

Once you include the appropriate headers, initialize the library by calling XMLPlatformUtils::Initialize() at the beginning of the main() function. Your code must catch exceptions when calling this method and many other methods in the library, so wrap the call to this method in a try/catch block that handles an XMLException exception type.

Then define a DOMParser object to prepare to parse an XML file. You can pass a true value to this object's setDoValidation() method—this requires the parser to verify that the XML data conforms to its DTD. Once you're ready to parse the file, execute the DOMParser object's parse() method, passing in the filename as a parameter. If the parse() method finds an error while parsing the file, it throws an exception.

Traversing a Document Structure

When the parser completes its operation for Listing 1, it will have built a linked tree-like structure of nodes (see Figure 1). You can get a reference to the top document node by calling the getDocument() method of the DOMParser object. When you have this document object of type DOM_Node, you can use several different methods of the DOM_Node object to traverse the document structure. DOM_Node uses these methods to get information about related nodes: getChildNodes(),

VC++ 6.0 | Walk the Node Tree

```
void print_NonEmptyTextElements(DOM_Node& mynode)
{
  switch (mynode.getNodeType())
  {
    case DOM_Node::TEXT_NODE:
      if (!DOMStrOnlyWhiteSpc(
      mynode.getNodeValue()))
      {
        DOMString myfelabel;
        FullElementLabel(mynode,
        myfelabel, ".");
        char *label = myfelabel.transcode();
        char *content = mynode.getNodeValue().transcode();
        cout << label << ": " << content << endl;
        delete label;
        delete content;
      }
      break;
    case DOM_Node::DOCUMENT_NODE:
    case DOM_Node::ELEMENT_NODE:
      DOM_Node mychildnode =
      mynode.getFirstChild();
      while( mychildnode != 0)
      {
        print_NonEmptyTextElements(mychildnode);
        mychildnode = mychildnode.getNextSibling();
      }
      break;
  }
}
```

Listing 3 | Here's how to traverse an XML document using a recursive function to walk the node tree. In the process, you can display the contents of all nonempty text nodes.

Building Trees

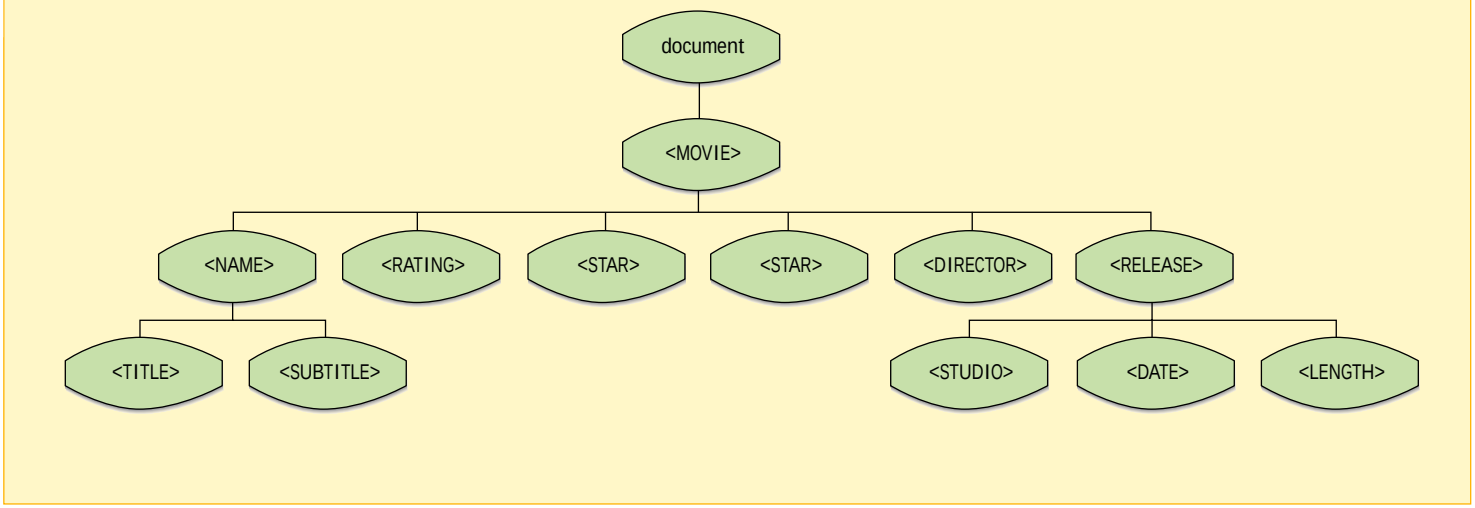

Figure 1 | When the DOMParser object completes its operation for Listing 1, it will have built a linked tree-like structure of nodes. Here's the memory tree structure of our sample XML document.

getFirstChild(), getLastChild(), getPreviousSibling(), and getNextSibling(). When you're on a particular node, you can get information on its name, value, and type by using the getNodeName(), getNodeValue(), and getNodeType() methods, respectively. Some of the node types you'll use include DOCUMENT_NODE, ELEMENT_NODE, and TEXT_NODE.

The simplest way to traverse an entire document structure is through a recursive function. In the MyParser application, you can use the methods discussed so far to recursively traverse the document structure and display the contents of all nonempty text nodes. You can create a print_NonEmptyTextElements() function that gets passed to the top document node (see Listing 3).

Also create a recursive helper function—FullElementLabel()—for print_NonEmptyTextElements(). This function displays a label for each nonempty text node that includes its path to the root of the document structure. Use the getParentNode() method of DOM_Node to travel up the structure to get all the necessary information (see Listing 4).

Executing the code in Listing 3 and Listing 4 on our sample XML file yields this result:

```
.MOVIE.NAME.TITLE: 2010
.MOVIE.NAME.SUBTITLE: The Year We Make Contact
.MOVIE.RATING: PG
.MOVIE.STAR: Roy Scheider
.MOVIE.STAR: John Lithgow
.MOVIE.DIRECTOR: Peter Hyams
.MOVIE.RELEASE.STUDIO: MGM
.MOVIE.RELEASE.DATE: 1984
.MOVIE.RELEASE.LENGTH: 116 minutes
```

With these two examples, you should have a good feel for how to "walk around" a document structure and retrieve information.

While traversing a document structure, you can change it by using other methods of the DOM_Node object. You can insert nodes, replace existing nodes, and remove nodes using insertBefore(), replaceChild(), removeChild(), and appendChild(). Try copying and modifying the example function in Listing 3 to incorporate some of these methods. You can also use the setNodeValue() method to change the contents of an existing node.

In addition to changing a document structure, you can also create a new one. The DOM_Document class is derived from DOM_Node, which is a special type of node for the root of the document structure. Other methods you can use to create new documents and content include createDocument(), createEntity(), createElement(), createTextNode(), and createComment(). Try creating a new document in memory and adding some additional nodes; then, use the existing code from Listing 3 to display the contents.

You should now have a good basic understanding of what an XML document looks like and why you would want to use this format in your C++ applications. You should also have a good feel for the DOM API and its associated classes, properties, and methods. You can apply this technology in many different areas to solve problems requiring a universal data format for sharing structured data. From e-commerce and e-business applications to traditional client/server applications, XML is here to stay. VCDJ

Links

- XML 1.0 Standard: www.w3.org/TR/REC-xml
- DOM 1.0 Standard: www.w3.org/TR/PR-DOM-Level-1
- IBM alphaWorks XML for C++: www.alphaworks.ibm.com/tech/xml4c
- Microsoft's XML Developer Center: http://msdn.microsoft.com/xml/default.asp
- Oracle TechNet: http://technet.oracle.com/

VC++ 6.0 | Get to the Root

```
oid FullElementLabel(DOM_Node& mynode,
  DOMString& mylabel, const DOMString
  myseperator)
{
  if (!mynode.getParentNode().isNull())
      FullElementLabel(mynode.getParentNode(),
    mylabel, myseperator);

  if (mynode.getNodeType() ==
    DOM_Node::ELEMENT_NODE)
  {
    mylabel.appendData(myseperator);
    mylabel.appendData(mynode.getNodeName());
  }
}
```

Listing 4 | Display the element path of a node by traveling up the node tree to the root node. Use the FullElementLabel recursive helper function and the getParentNode method to accomplish this.

About the Author

Brian Holman is a seasoned computer scientist, author, and information systems professional. He works as e-business and directory architect for Novell Inc.'s IT department. Reach him by e-mail at me@brianholman.com.

Go Online

Use the following DevX Locator+ codes at www.vcdj.com to go directly to these related resources.

VC0007 Download all the code for this issue of *VCDJ*. Free online registration is required.

VC0007MT Download the code for this article separately. This article's code includes the MyParser.cpp XML parser, along with some XML files and a preconfigured VC++ 6.0 project. Free online registration is required.

VC0007MT_T Read this article online. DevX Premier Club membership is required.

Not a member yet? Go to www.devx.com to sign up for the free Registered Level or subscribe to the Premier Club.