Make the most of Xerces-C++, Part 2

A DOM implementation

Level: Intermediate

Rick Parrish (rfmobile@swbell.net)
Consultant
15 Aug 2003

This two-part article offers an introduction to the Xerces-C++ XML library. Here in Part 2, Rick Parrish demonstrates how to load, manipulate, or synthesize a Document Object Model (DOM) document, and how to recreate the bar graph in Part 1 using Scalable Vector Graphics (SVG). C++ programmers who read these articles should be able to easily add XML parsing and processing capabilities to their applications.

Contents:

Subscriptions:

In Part 1, you saw how to link the library into applications written in Linux and Windows, and how to parse with the SAX API. A sample application showed you how to create a bar graph in ASCII art.

Up next is a description of DOM node organization, followed by loading and parsing to produce a DOM document from a file or stream, synthesis to programmatically produce a DOM document, and serialization or writing a DOM document as output to a file or stream.

If you have used Xerces-C++ prior to version 2.0 for DOM, beware! Things have changed. The most notable change is the renaming of most DOM class objects from a `DOM_` prefix to a `DOM` prefix, and a preference for passing pointers instead of references.

DOM node types

The fundamental data type for DOM is the `DOMNode` class. All DOM node objects are extensions of this base type. Table 1 shows the DOM node types. The first column is the enumerated type name returned by the `DOMNode::getNodeType()` method. The second column is the C++ class used to declare an instance of that particular node type. The third column shows the construction method used to actually produce such an instance.

Table 1. DOM node types

DOM Type	DOM Class	Construction method
DOMNode::TEXT_NODE	DOMText	createTextNode
DOMNode::PROCESSING_INSTRUCTION_NODE	DOMProcessingInstruction	createProcessingInstruction
DOMNode::DOCUMENT_NODE	DOMDocument	createDocument
DOMNode::ELEMENT_NODE	DOMElement	createElement
DOMNode::ATTRIBUTE	DOMAttr	createAttribute
DOMNode::ENTITY_REFERENCE_NODE	DOMEntityReference	createEntityReference
DOMNode::CDATA_SECTION_NODE	DOMCDATASection	createCDATASection
DOMNode::COMMENT_NODE	DOMComment	createComment
DOMNode::DOCUMENT_TYPE_NODE	DOMDocumentType	createDocumentType
DOMNode::ENTITY_NODE	DOMEntity	createEntity
DOMNode::NOTATION_NODE	DOMNotation	createNotation

Note that node type `XML_DECL_NODE` was replaced with `DOMDocument` methods for `get/setEncoding`, `get/setVersion`, and `get/setStandalone` for Xerces-C++ version 2.

All of the construction methods in Table 1 are available from the `DOMDocument` class. The construction method for creating a `DOMDocument` object avoids the chicken-and-egg issue by forcing you to go through a `DOMImplementation` instance. You can get the `DOMImplementation` instance by calling the `DOMImplementation::getImplementation()` method, which is declared as static. To create a document node, follow this two-step recipe:

```
DOMImplementation *pImpl = DOMImplementation::getImplementation();
DOMDocumentType* pDoctype = pImpl->createDocumentType(L"svg",
     NULL, L"svg-20000303-stylable.dtd");
```

```
pSVG = pImpl->createDocument(L"svg", L"svg", pDoctype);
```

DOM loading and parsing

The code in Listing 8 initializes the parser and loads an XML document as a DOM tree. It uses a `DOMParser` object to do all the work. When the parser's work is done, a call to `getDocument()` returns the resulting DOM tree.

The parser can throw one of three types of exceptions -- DOM, SAX, or XML -- so I've included stubbed exception handlers in Listing 8. An additional place to check for errors is the `DOMParser::getErrorCount` function.

Synthesis

The code sample in Listing 9 builds, or **synthesizes**, an XML document through calls to the DOM API. The document constructed happens to be a small XHTML page but could in fact be any XML. To prove that an XML document was in fact produced, the code shown dumps the XML document, tags and all, to the console. (I describe the additional code needed to display the document to the console later.)

As you look over the piece of code in Listing 9, notice how it uses the DOM document object to create the other nodes. Notice also how each node must be explicity attached to its parent node. Even the root node must be explicity attached to the document using the `appendChild` method.

Serializing

In a library that implements the DOM API, you would expect the ability to persist a DOM document to a file or stream to be built into the library. That functionality is described in the base classes `XMLFormatTarget` and `XMLWriter`. The implementations are tucked away in classes named `XMLWriter`, `LocalFileFormatTarget`, and `StdOutFormatTarget`.

The code in Listing 10 creates an `XMLFormatter` object for writing to a file or the standard output stream. The `XMLFormatter` object handles transcoding across character sets and also escaping text that might contain reserved XML characters. The `XMLWriter` object traverses the DOM tree, passing chunks of XML data off to the formatter for output. The downside to using the ad-hoc `dump_xml` to examine DOM content is the absence of line breaks between elements. Although not necessary -- or in some cases even desired -- in production data, the extra whitespace makes the XML much more readable during debugging sessions. This next listing is a two-line fix to the code above which adds just enough whitespace for readability, but without injecting text nodes into the DOM tree.

Listing 11. Pretty-printing XML

```
  // turn on serializer "pretty print" option

  if ( pSerializer->canSetFeature(XMLUni::fgDOMWRTFormatPrettyPrint, true) )
    pSerializer->setFeature(XMLUni::fgDOMWRTFormatPrettyPrint, true);
```

Keep the code in Listings 10 and 11 handy for debugging. Even if you don't need to create an XML document, you'll find the ability to take a snapshot of a working DOM subtree useful. If you take the time to download the sample code for this article (see Resources), you'll find a wide-character version for writing UTF-16 data to files. In earlier versions of Xerces-C++, I was suprised to discover that creating an `XMLFormatter` object for a "UTF-16" encoding fails, while creating one for "UTF-16 (LE)" succeeds. That has been fixed in version 2.

DOM traversal

The `DOMPrint` code in Listing 11 shows how to visit every node of a DOM tree. Listing 12 shows a different approach using an iterator and then a tree walker to accomplish the same objective. The node iterator code in Listing 12 assumes that the valid DOM node variable `root` already exists.

Listing 12. Creating an iterator to visit all text nodes

```
// create an iterator to visit all text nodes.
DOMNodeIterator iterator =
  doc.createNodeIterator(root, DOMNodeFilter::SHOW_TEXT, NULL, true);
// use the tree walker to print out the text nodes.
for ( current = iterator.nextNode(); current != 0; current = iterator.nextNode() )
  // note: this leaks memory!
  std::cout << current.getNodeValue().transcode();
std::cout << std::endl;
```

The example in Listing 12 just rips through the entire document picking out text nodes and displaying them. Note `wcout`, the wide-character version of `cout`. Listing 13 is the tree-walker code, which also assumes that a valid DOM node variable `root` already exists.

Listing 13. Creating a walker to visit all text nodes

```
// create a walker to visit all text nodes.
DOMTreeWalker walker =
  doc.createTreeWalker(root, DOMNodeFilter::SHOW_TEXT, NULL, true);
// use the tree walker to print out the text nodes.
for (DOMNode current = walker.nextNode(); current != 0; current = walker.nextNode() )
  // note: this leaks memory!
  std::cout << current.getNodeValue().transcode();
std::cout << std::endl;
```

The tree-walker example in Listing 13 functions just like the node iterator in this instance because it isn't using any of the additional features of a tree walker. When you first create the tree walker using `createTreeWalker`, the `getCurrentNode()` method returns the root node regardless of filter or to-show settings. Only after the first call to `nextNode()` does `getCurrentNode()` operate as expected.

DOM manipulation

The DOM API gives you the ability to act as a tree surgeon to clip, graft, and prune the nodes of a DOM tree. The methods for manipulating a DOM tree overlap with one used for synthesizing a DOM tree from scratch. Listing 14 gives a summary of the methods.

Listing 14. Summary of DOMNode methods

```
DOMNode cloneNode(bool deep) const;
DOMNode insertBefore(const DOMNode &newChild, const DOMNode &refChild);
DOMNode replaceChild(const DOMNode &newChild, const DOMNode &oldChild);
DOMNode removeChild(const DOMNode &oldChild);
DOMNode appendChild(const DOMNode &newChild);
DOMNode insertBefore(const DOMNode &newChild, const DOMNode &refChild);
DOMNode replaceChild(const DOMNode &newChild, const DOMNode &oldChild);
```

Node-specific creation methods like `createTextNode` and `createElement` are available only from a `DOMDocument` object. However, the `cloneNode` method can be used from any `DOMNode` object.

`DOMElement` nodes have a few additional methods for dealing with grafting and pruning attributes, shown in Listing 15.

Listing 15. Summary of DOMElement methods

```
void setAttribute(const DOMString &name, const DOMString &value);
DOMAttr setAttributeNode(DOMAttr newAttr);
void setAttributeNS(const DOMString &namespaceURI, const DOMString &qualifiedName,
     const DOMString &value);
DOMAttr removeAttributeNode(DOMAttr oldAttr);
void removeAttribute(const DOMString &name);
void removeAttributeNS(const DOMString &namespaceURI, const DOMString &localName);
```

Removing an attribute from an element that is bound to a DTD can sometimes cause an unexpected result for the unwary. If the DTD defines a default value for an attribute, that attribute appears in the DOM tree regardless of the original XML that produced it. If you use the DOM API to prune the attribute from the DOM tree, it is replaced with its default value. In other words, an attribute node that is a default value can't be removed!

Enough playing around; it's time to actually do something useful with all of this. While the SAX-based graph application is okay, it didn't exactly impress the business suits upstairs. Since you now have the DOM API at your disposal, you can generate XML as well as parse it. To enhance the visual appeal of the earlier bar chart, use DOM to produce a Scalable Vector Graphics (SVG) version suitable for display in an SVG viewer like the Adobe plug-in, the W3C browser Amaya, or the SVG build of Mozilla (including 1.0 and later).

Using the same XML data as the input, the output looks something like Listing 16.

Listing 16. Sample XML/SVG output

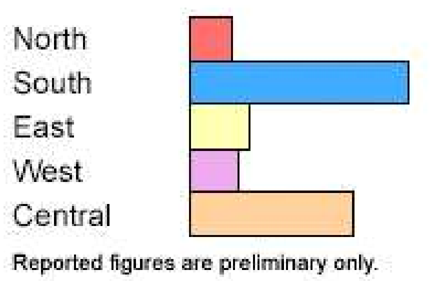
```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<!DOCTYPE svg SYSTEM "svg-20000303-stylable.dtd">
<svg width="320" height="200">
```

```
<g style="font-size:14">
  <rect x="100" y="20" style="stroke:node; fill:red" width="20" height="20"/>
  <text x="20" y="35">North</text>
</g>
<g style="font-size:14">
  <rect x="100" y="40" style="stroke:node; fill:blue" width="100" height="20"/>
  <text x="20" y="55">South</text>
</g>
<g style="font-size:14">
  <rect x="100" y="60" style="stroke:node; fill:yellow" width="27" height="20"/>
  <text x="20" y="75">East</text>
</g>
<g style="font-size:14">
  <rect x="100" y="80" style="stroke:node; fill:violet" width="23" height="20"/>
  <text x="20" y="95">West</text>
</g>
<g style="font-size:14">
  <rect x="100" y="100" style="stroke:node; fill:orange" width="75" height="20"/>
  <text x="20" y="115">Central</text>
</g>
<text x="20" y="135" style="font-size:10">Reported figures are preliminary only.</text>
</svg>
```

You could use SAX to implement this by printing out SVG-compatible XML tags. Notice the `"!DOCTYPE"` declaration included within the SVG output. The document type is an important clue to the SVG viewer as to which revision of the SVG technical recommendation is expected. With a subtle trick, you can get Xerces-C++ to include the DOCTYPE in the document output. Figure 2 shows what the SVG in Listing 16 looks like in a viewer.

Figure 2. Screen shot of SVG output



Next, Listing 17 shows the DOM code to pull in the XML source data to produce the SVG final result.

The trick to getting the `"!DOCTYPE"` declaration included with the SVG output document is to use `DOMDOMImplementation::createDocument()` to create the document instead of `DOMDocument::createDocument()`. The code for this is near the beginning of the doc2svg static function. Using `DOMDOMImplementation` gives you an opportunity to create a `DOMDocumentType` node for inclusion in the creation method. The `DOMDocument` version of this creation method does not offer a way to specify a document type.

You could create a `DOMDocumentType` node using the `DOMDocument::createDocumentType()` creation method. That method isn't useful here because it doesn't allow setting the DOCTYPE's system ID or public ID. One other subtle issue with this technique is that it creates the top-level root element node for you. That is why the code is able to call `getDocumentElement()` instead of explicitly creating and appending the root element to the document object.

Conclusion

In this article and its predecessor, Part 1, you've seen that the benefits of the Xerces-C++ library include open source, portability, easy licensing, and community support. You can dissect the library's operation by examining the source code. You can deploy to just about any platform that supports a C++ compiler. Windows programmers get a COM version usable from C++, Visual Basic, and even VBScript/JScript. The Apache license permits commercial Xerces-C++ use with a simple legal disclosure and disclaimer to users. You can share development-issue questions and answers with other programmers on the mailing lists. All of these advantages make Xerces-C++ an excellent choice for adding XML capabilities to your projects.

Resources

- Download the source code and figures for this article.
- In "Make the most of Xerces-C++, Part 1" by the author, learn to link the library into applications written in Linux and Windows, and to parse with the SAX API. A sample application shows you how to create a bar graph in ASCII art.
- Find out more about IBM's XML4C++ parser project, which is based on Xerces-C++, and available on *alphaWorks*.
- Download the Xerces-C++ XML parser library from the Apache site. While your there, you can also read the Xerces build documentation.
- Read the terms of the Apache Software License, which governs the use of Xerces-C++ in your applications.
- Read about or download the Apache XML Project's Xalan-C++, an XSLT transformation engine.
- See other Apache-sponsored XML projects.
- Read the SAX API specifications at SourceForge.
- Read up on SAX in a chapter excerpted from Benoit Marchal's book SAX, the Power API (*developerWorks*, August 2001) or take the basic tutorial, "Understanding SAX" (*developerWorks*, July 2003).
- Catch up on the DOM:
  - DOM Level 1
  - DOM Level 2 Core
  - DOM Level 2 Views
  - DOM Level 2 Events
  - DOM Level 2 Style
  - DOM Level 2 Traversal and Range
  - DOM Level 3 Core
  - DOM Level 3 Events
  - DOM Level 3 Validation
  - DOM Level 3 Abstract Schemas with Load and Save
  - For basic intro to the DOM, try the *developerWorks* tutorial, "Understanding DOM" (*developerWorks*, July 2003).
- Learn more about Scalable Vector Graphics with the *developerWorks* tutorials "Introduction to SVG" (February 2002) and "Interactive, dynamic SVG" (June 2003).
- Read about the SVG specification, at the W3C site.
- Read about animating SVG graphics on a timeline with SMIL at the W3C site and in the *developerWorks* article by Anne Zieger (September 2002).
- Find more XML resources on the *developerWorks* XML zone.
- IBM's DB2 database provides not only relational database storage, but also XML-related tools such as the DB2 XML Extender which provides a bridge between XML and relational systems. Visit the DB2 Developer Domain to learn more about DB2.
- Find out how you can become an IBM Certified Developer in XML and related technologies.

Download

Name	Size	Download method
x-xercc/xml4c.zip		**FTP**

Information about download methods

About the author

Rick Parrish writes software for a living but also lends a hand in several open-source projects. His current interests include 3D graphics and visualization, decoding digital radio signals, and creating a scriptable database framework for Mozilla. You can contact Rick at rfmobile@swbell.net.

Code PDF e-mail it!

Rate this article

This content was helpful to me:

Strongly disagree (1) Disagree (2) Neutral (3) Agree (4) Strongly agree (5)

Comments?

Submit feedback