Make the most of Xerces-C++, Part 1

A parsing how-to for C++ programmers

Rick Parrish (rfmobile@swbell.net), Consultant

Summary: This two-part article offers an introduction to the Xerces-C++ XML library. Part 1 explains how to link the library into applications written in Linux and Windows. Ample code demonstrates parsing with the SAX API, and a sample application shows you how to create a bar graph in ASCII art. In Part 2, I'll demonstrate how to load, manipulate, or synthesize a DOM document, and you'll see how to create the same bar graph 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.

Date: 13 Aug 2003
Level: Intermediate
Also available in: Japanese

Xerces-C++ is a very robust XML parser that offers validation, plus SAX and DOM APIs. XML validation is well supported for a Document Type Definition (DTD), and essentially complete open-standards support for W3C XML Schema was added in December 2001.

Xerces-C++: a capsule bio

Other articles in this series

- Make the most of Xerces-C++, Part 2: A DOM implementation

Xerces-C++ originated as the XML4C project at IBM. XML4C was a companion project to XML4J, which likewise was the origins of Xerces-J -- the Java implementation. IBM released the source for both projects to the Apache Software Foundation, where they were renamed Xerces-C++ and Xerces-J, respectively. These two are core projects of the Apache XML group. (If you see "Xerces-C" instead of "Xerces-C++", it's the same thing; the project was written in C++ from the start.)

The XML4C project continues at IBM, based on Xerces-C++. XML4C's distinguishing merit relative to Xerces-C++ is better out-of-the-box support for a huge number of international character encodings in the version that I explored (see Resources).

Validation

The two principal means of specifying the structure of an XML document are the DTD and W3C XML Schema, with DTD being the much older of the two. XML Schema is basically a DTD expressed as XML. Xerces-C++ offers great out-of-the-box validation capabilities for ensuring that an XML document conforms to a DTD.

Licensing

Xerces-C++ is made available under the terms of the Apache Software License (see Resources), which happens to be one of the more readable open-source licenses around. It compares very well to the BSD license. Essentially, you can use Xerces-C++ in your (or your company's) software royalty free at the mere expense of disclosing to your customers and users that your software includes Apache code, and including the proper copyright notice. Check the Web page for the exact text of the license.

SAX: the event API model

SAX, as you may know, is an event-oriented programming API for parsing XML documents. A parsing engine consumes XML sequential data and makes callbacks into the application as it discovers the structure of the incoming XML data. These callbacks are referred to as event handlers. SAX is actually two APIs: SAX 1.0 is the original, and SAX 2.0 is the current revised specification. The two are similar, but different enough that most applications based on SAX 1.0 break when they are moved to the newer specification.

The SAX API specification was moved to SourceForge as a project of its own (see Resources). The SAX examples I give later in this article make use of SAX 2.0.

DOM: the Document Object Model

Unlike SAX, the DOM API permits editing and saving an XML document back to a file or stream. It also permits programmatically constructing a new XML document from scratch. The reason for this is that DOM provides an in-memory model for the document. You can traverse the document tree, prune nodes, or graft on new ones.

The tech wrecks

DOM is a family of W3C technical recommendations affectionately called *tech wrecks*. DOM has three levels, with Levels 1 and 2 at full technical recommendation status and Level 3 at working draft status.

The DOM Level 1 Core defines most of what is needed for basic XML functionality: the ability to construct a representation of an XML document. The `DOMString` type is explicitly specified to consist of wide UTF-16 characters. Level 1 goes on to define the interfaces for programmatically interacting with the various pieces of a DOM tree. Serialization of XML is intentionally omitted from Level 1. Just beyond the Level 1 core is the DOM Level 1 HTML definition. This area attempts to resolve DOM Level 1 core with the earlier Dynamic HTML object model (loosely referred to as Level 0).

The DOM Level 2 adds namespaces, events, and iterators, plus view and stylesheet support. You need DOM Level 2 for some applications: For instance, assigning an XML Schema to a namespace is essential for applications like RDF, where XML tags come from different schemas and the chance for a name collision is high. Level 2 adds a pair of `createDocument` methods to the `DOMImplementation` interface. One of the examples will show why this is important. Just when you thought you were safe from the callbacks and event handlers found in SAX, here they are again in the `Event` interface. Unlike the SAX events, which are for parsing, DOM events can reflect user interactions with a document as well as changes to a live document. DOM events that reflect the change in the structure of a document are called **mutation events**. `TreeWalkers` and `NodeIterators` enhance DOM tree traversal. Programs can inspect style information through the `StyleSheet` interface. Finally, view support allows an XML application to examine a document in both original and stylesheet rendered forms. These before and after views are called the **document** and **abstract** views.

DOM Level 3 Core adds the `getInterface` method to the `DOMImplementation` interface. In a Level 3 document, you can specify the document's character encoding or set some of its basic XML declarations like `version` and `standalone`. Level 2 doesn't permit moving DOM nodes from one document to another. Level 3 drops this limitation. Level 3 adds *user data* -- extra application data that can be optionally attached to any node. Level 3 has a number of other advanced features, but the W3C committee is still working on the Level 3 drafts. Check Resources for a link to read up on the committee's progress.

Download and install

You can download Xerces-C++ as a zipped tarball or a precompiled binary (see Resources). Script users accessing the library through Perl, Python, VBScript, or JavaScript can download the binary for their platform to get a jumpstart on installation. C++ programmers will most likely prefer to go with building their own binaries from the source tarball. The building instructions on the Apache XML group Web site are well written; a little farther on in this article I discuss a couple of subtle issues that I have discovered -- a pthreads linking problem and a fix for potential memory leaks on Windows platforms. Part 2 will include a tip for specifying a DOCTYPE in the SVG example. If you want to build the library as you read this, look at the Xerces build documentation found on the Apache site (see Resources) first and then come back here to read about linking Xerces to your own applications.

You can download the tarball and work offline (with a laptop, for example). The full HTML documentation is included in the tarball, so you don't need to keep referring back to the Web site for the instructions.

Building for Win32

The steps for installing the software on Visual Studio dot-NET or Win64 are nearly identical to these steps for building on Win32.

1. Unzip and untar the Xerces source tarball to a working directory. Xerces-C++ has its own directory structure, so you should make sure you preserve relative path names during this step.
2. Using Windows Explorer or your favorite file manager, drill down to the `\xerces-c-src_2_3_0\Projects\Win32\VC6\xerces-all\` folder and click the `xerces-all.dsw` workspace file to launch Microsoft Developer Studio.
   Note: These instructions assume that you're building Win32 applications in Visual Studio 6. For Visual Studio dot-NET or Win64 applications, repeat steps 1 and 2 in the Win64 or VC7 variants of the directory.
3. From Developer Studio, make XercesLib the current active project and press F7 to build the DLL. On last year's hardware this takes a minute or two.
4. Add a path to the Xerces header files into your project. (Applications wanting to link against Xerces-C++ need to either include the XercesLib DSP project file in their workspace or add the LIB file in their project file to permit linking.) Select **Project>Settings** to bring up the project settings dialog box. Select **All Configurations** from the **Settings** combo box, click the C++ tab, select the **Preprocessor**

category, and add the Xerces include path (something like `\xerces-c-sr2_2_0\src`) to the **Additional include directories** text box.
5. If you have added the XercesLib DSP to your workspace, remember to mark your own project as dependent upon the XercesLib project; otherwise, you will be greeted with link errors.
6. Create a stub C++ source file that does nothing but contain a line that reads `#include <xercesc/sax/HandlerBase.hpp>`. If you can compile this one-line C++ file, your include paths are probably right. Save your workspace after doing that. To run and debug your application, place a copy of the Xerces DLL in the working directory.

Building for Linux

Build the Xerces-C++ shared library by following the thorough instructions in the `doc/html` folder. The commands below illustrate how to build the Xerces-C++ library from the zipped source. This assumes that the `xerces-c-src_2_3_0.tar.gz` file is present in a directory like `/home/user`. Whatever directory you choose should match the `XERCESCROOT` variable; the `configure` script requires it.

```
# cd /home/user
# gunzip xerces-c-src_2_3_0.tar.gz
# tar -xvf xerces-c-src_2_3_0.tar
# export XERCESCROOT=/home/user/xerces-c-src_2_3_0
# cd $(XERCESCROOT)/src/xercesc
# ./configure
# make all
```

For the rest of this example, I'll assume the source tree is under the `/home/user/xerces-c-src_2_3_0` directory. If all goes well, the shared library should appear in the `lib` folder. If you have problems, review the build instructions in the `/doc/html` folder. At this point, you can either copy the library (and symlinks) to `/usr/lib` or define the appropriate environment variable so that the loader can locate your newly-compiled library.

The easy way to test out your new library is to build and run one of the samples:

```
# export XERCESCROOT=/home/user/xerces-c-src_2_3_0
# cd $(XERCESCROOT)/samples
# ./configure
# make all
```

I tripped over a small problem building one of the samples on a fresh installation of Slackware Linux 9.0. The linker complained of some missing pthreads-related exports. I edited the `Makefile.in` file to include a reference to `-lpthread` and ran `configure` again. The second time around, typing `make all` worked.

Once you know the library works, you can start your own Xerces-C++ project. Use the `-I` compiler option to help the compiler locate the Xerces header files. Use the `-L` and `-l` linker options to help the linker locate the Xerces-C++ library. Listing 1 gives you a working minimal makefile to get started.

Listing 1. A minimal makefile

```
                APP = example
XERCES = /home/user/xerces-c-src_2_3_0
INCS = ${XERCES}/src

${APP} :: ${APP}.cpp
        ${CC} -lxerces-c-src_2_3_0 -I${INCS} ${APP}.cpp -o ${APP}
```

The command to kick off Listing 1 is `make` or `gmake`. You can change the `APP` variable to whatever source file suits you. The examples in this article use similar makefiles.

Xerces C++ added C++ namespace support (not to be confused with XML namespaces) as of Version 2.2.0. If you have code that works on 2.1.0 and you'd like to take advantage of the newer version, add the following three lines to your code, just after including the Xerces C++ headers.

Listing 2. Xerces C++ namespace support

```
                #ifdef XERCES_CPP_NAMESPACE_USE
XERCES_CPP_NAMESPACE_USE
#endif
```

You could, of course, just prefix all of your Xerces-C++ objects with the `XERCES_CPP_NAMESPACE::` namespace.

The sample application

To keep things interesting as I explain the basics of using Xerces-C++, I'm going to create a simple bar graph using XML as the data format. To dodge the cross-platform bullet of platform GUI specifics, I'm doing the bar graph using ASCII art. This is, after all, an article on XML and not GTK, OpenGL, or Direct-X. If you are interested in using an XML representation of graphical data, look at SVG and SMIL (see Resources). The DOM example that I describe in Part 2 outputs SVG. I'll start with the simple text-only app.

Listing 3 is the DTD for the data. Next I'll construct a program to load the data, determine what scale to use, and then actually plot the data to the screen.

Listing 3. DTD for sample application data

```
                APP = example
<?xml version="1.0" ?>
<!ELEMENT figures (PCDATA) >
<!ATTLIST figures type (sales | inventory | labor) >
<!ATTLIST figures value CDATA >
<!ELEMENT department (figures*) >
<!ATTLIST department name CDATA>
<!ELEMENT corporate (department*) >
<!ATTLIST corporate name CDATA >
```

Listing 4 shows a sampling of what the data might look like.

Listing 4. Sample input XML data

```
                APP = example
<?xml version="1.0" ?>
<corporate name="Big Biz">
<department name="North">
<figures type="sales" value="125000.00"/>
<figures type="inventory" value="90000.00"/>
<figures type="labor" value="110000.00">estimated</figures>
</department>
<department name="South">
<figures type="sales" value="980000.00"/>
<figures type="inventory" value="110000.00"/>
<figures type="labor" value="115000.00">estimated</figures>
</department>
<department name="East">
<figures type="sales" value="210000.00"/>
<figures type="inventory" value="80000.00"/>
<figures type="labor" value="95000.00">estimated</figures>
</department>
<department name="West">
<figures type="sales" value="160000.00"/>
<figures type="inventory" value="75000.00"/>
<figures type="labor" value="130000.00">estimated</figures>
</department>
<department name="Central">
<figures type="sales" value="723000.00"/>
<figures type="inventory" value="11000.00"/>
<figures type="labor" value="221000.00">estimated</figures>
</department>
</corporate>
```

SAX2 implementation

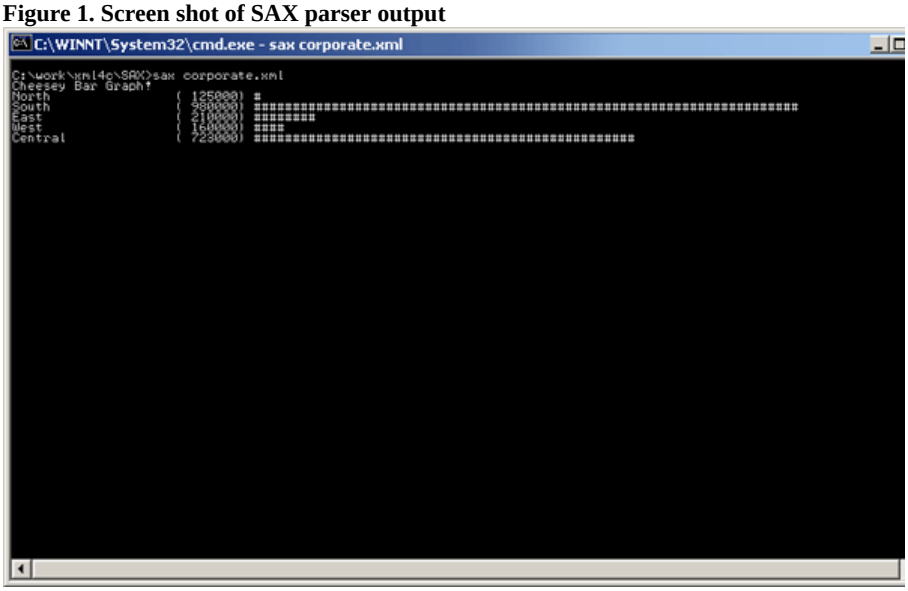
Listing 5 is a baseline SAX implementation. This isn't a complete program because it is missing the handler implementation, but it does show what exactly is needed to put the framework into place. The calls to `XMLPlatformUtils:Initialize()` and `XMLPlatformUtils::Terminate()` are very important. The library guards against applications that fail to initialize the library properly by throwing an exception.

To make the program in Listing 5 a complete application, you need to add the event-handler class in Listing 6. SAX2 comes with a default event-handler class called `DefaultHandler`, defined in the C++ header file of the same name. The default handler does nothing -- it is just a stub implementation -- but it is complete, and so I'm using it here as a base class for the graphing event-handler class.

This file in Listing 7 is the actual implementation of the event-handler class in Listing 6. While the rest of the program is pretty much just boilerplate code to get the SAX2 parser running, the part in Listing 7 defines the application's personality.

Xerces-C++ uses `XMLCh` as a typedef'd character representation. On some platforms it is compatible with the C type `wchar_t`, which is usually two -- but sometimes four -- bytes wide. Because of that possibility, the docs discourage the practice of interchanging `wchar_t` and `XMLCh`. You can get away with it on some platforms, but it will break on others. Xerces-C++ uses this larger character representation to exchange text as UTF-16 as opposed to UTF-8 or ISO-8859. To debug this program, I'm using the `XMLString::transcode` function to convert the wide character strings for display on a console, as shown in Figure 1.

Figure 1. Screen shot of SAX parser output



I discovered a problem using the Xerces internal string class on Microsoft Windows. The comments in `XMLString.hpp` require the caller of `replicate` and other similar functions to release the memory returned. The problem comes from linking your application against the Xerces-C++ library as a DLL. The strings are allocated from the DLL's local heap. If both your application and the XercesLib DLL use the exact same C runtime (CRT) library DLL, then all is well. If, however, your program uses the single-threaded CRT and XercesLib uses the multithreaded CRT, DLL problems happen. When your program attempts to release the string memory, the C runtime notices that the memory did not come from your application's local heap. For debug builds it throws an exception, but for release builds it may silently leak memory. The sample programs found in earlier versions of Xerces (like 1_5_1) avoided this by simply not releasing the memory.

My fix for this was to add a pair of static discard functions to the `XMLString` class. Because the string memory is released by code executing inside the DLL, the correct local heap is used, and no debug assertion results. I was pleased to see that Xerces developer Tinny Ng added this to the `XMLString` class and went a step further to null the string pointer (see Resources). The other nice feature of this is that programmers don't need to worry about how the implementation of `XMLString` allocates memory. Instead of guessing whether they should be using `delete[]` or `free`, they can just call `XMLString::release`. You can, of course, just make sure the CRT that your application expects is the same as the CRT used by the XercesLib DLL.

What's next?

Here in Part 1, you've seen how to link the Xerces-C++ XML library into applications written in Linux and Windows, and I've demonstrated parsing with the SAX API by creating a bar graph in ASCII art. In Part 2, I'll show you how to load, manipulate, or synthesize a DOM document, and create the same bar graph using Scalable Vector Graphics (SVG).

Download

Name	Size	Download method
x-xercc/xml4c.zip		HTTP

Information about download methods

Resources

- Download the source code and figures for this article.
- In "Make the most of Xerces-C++, Part 2" by the author, learn 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) (developerWorks, August 2003).
- 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 you're 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).
- Read "Serialize XML data" by IBMer and Xerces developer Tinny Ng (*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 a 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.

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.

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

A DOM implementation

Rick Parrish (rfmobile@swbell.net), Consultant

Summary: 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.

Date: 15 Aug 2003
Level: Intermediate
PDF: A4 and Letter (232 KB | 11 pages)Get Adobe® Reader®
Also available in: Japanese

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.

Other articles in this series

- Make the most of Xerces-C++, Part 1: A parsing how-to for C++ programmers

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.

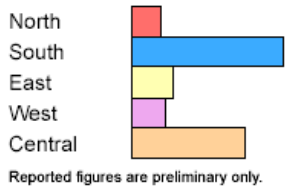
Listing 16. Sample XML/SVG output

```
<?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.

Download

Name	Size	Download method
x-xercc/xml4c.zip		HTTP

Information about download methods

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 (developerWorks, August 2003).
- 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).