8

Advanced C++ Aspects of SAX and DOM

PREVIOUS CHAPTERS HAVE DISCUSSED THE THEORY and practice of C++ SAX and C++ DOM. Several issues regarding the structure and usage of these XML manipulation mechanisms were discussed. However, some important C++-specific issues such as memory management and char types were not discussed.

This chapter addresses the following key comparative (and cooperative) issues that touch both C++ SAX and C++ DOM:

- How to decide whether to go with DOM or SAX
- Techniques to integrate SAX and DOM in C++
- Memory management (and C++ garbage collection) in C++ SAX and C++ DOM
- Unicode character encodings and their C++ representation (and the choices used in the major parsers)

C++ SAX Versus C++ DOM

One of the most recurrent and often oversimplified questions when developing XML applications is which criteria to use when deciding between SAX and DOM.

The most common answer to this question goes along the following lines: SAX is generally faster and requires less memory, because the footprint of creating new objects is not present. On the other hand, DOM provides random access to any point of the tree after the document has been read. There is basically a trade-off between memory consumption and fast multiple data accesses after parsing.

Even though the preceding sentences are the truth and nothing but the truth, they are far from being the whole truth. They are adequate for an entry-level discussion, because they address the nature of the interfaces, but they are unfit for our purposes, because they don't take into consideration the design of the applications in which these technologies will be used.

There are two dimensions to the decision of SAX versus DOM: One concerns performance, and the other focuses on system design quality. The following sections explain these issues in light of some facts.

Performance

When reading and processing XML files, you can measure performance considerations in two dimensions: time and memory. Let's examine the consumption patterns for these resources in both SAX and DOM.

Reading Time

The time to actually read the document using either SAX or DOM grows in linear proportion to the size of the XML source, as shown in Figure 8.1. However, DOM's line is much steeper than that of SAX, because all the objects for the document must be allocated (and eventually deleted), whereas in SAX, only simple strings are allocated and passed.

Script for Calling Counting Programs

Figure 8.1 depicts the time taken by different versions of an element counter to handle long XML files. (There are four element-counter programs: SAX and DOM implementations using C++, and SAX and DOM implementations using Java.) The files have the following structure:

```
<tests>
<test anAttribute='avalue'>Some content</test>
<!-- ...repeat n times -->
<test anAttribute='avalue'>Some content</test>
</tests>
```

In this case, the test element gets repeated 5,000, 10,000,...,150,000 times.

The Perl script used to generate the files and call the counting programs is provided on the CD.

Even though the test of passing these big files and recording the time taken to process them is not a rigorous benchmark, it achieves the main goal: to give you a realistic idea of the growth of processing time in heavy-duty situations.

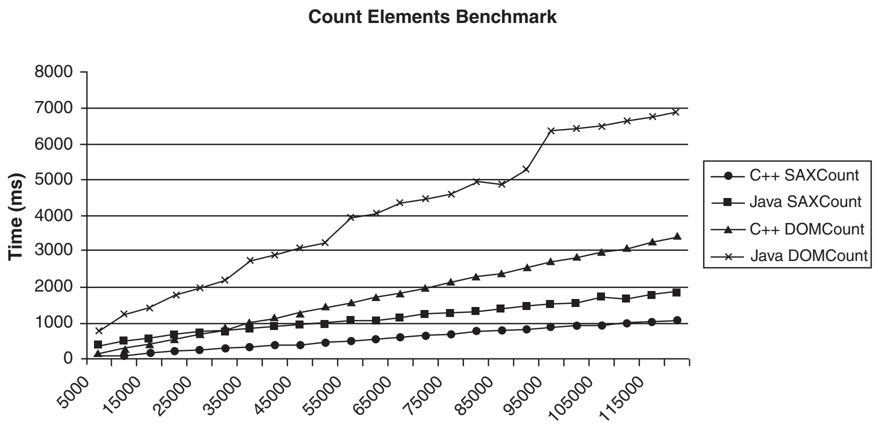


Figure 8.1 Parsing large numbers of elements.

Processing/Access Time

From the parsing time perspective, SAX seems to be a better approach than DOM. On the other hand, searches over the in-memory DOM tree are much faster than rereading the document. It is also important to mention that efficient random accesses to particular nodes of the document after it has been read are possible only using DOM (or an object model of your own). This was showcased in Chapter 7, "Advanced C++ DOM Manipulation," when you created an associative container to point to particular nodes in the tree, resulting in very fast access to elements.

If repeated access to arbitrary nodes is needed, an in-memory object model is always preferable to dealing with just events. However, this doesn't mean that "random access" equals "DOM," as you will see in the section "It's All About Design." (After all, you can always construct your own model, and sometimes you should.)

Memory Consumption

DOM parsers create a whole object model automatically whenever they parse an XML document, whereas SAX merely holds simple strings for a few moments each time it calls a handler. Naturally, DOM consumes more memory than SAX. However, some details must be considered.

Old quick formulas for DOM memory consumption were very popular at the beginning of XML and still are in some introductory literature. They stated things such as, "The amount of memory used will be approximately 7 to 10 times the size of the document." These guidelines become more obsolete every day as DOM implementations get smarter by means of patterns such as "flyweight" (factorize data repeated in many objects and represent it only once) and "proxy" (wait to create an object until it is really needed).

Even though the old formulas might be less of a guideline nowadays, one thing is for sure: memory consumption in a DOM implementation is dependent on the size of the file, usually O(*n*). SAX memory consumption is largely a constant, independent of the document's size.

The C++ Factor in Memory Consumption

One very appealing reason to stay with C++ in SAX/DOM programs is the excellent compromise between resource consumption and features. Consider the following anecdote:

While I was testing the benchmark script in the preceding section with a Pentium III 966Mhz computer running Windows 2000/XP with 256MB of RAM, the Java version of the DOMCount program consistently died with `java.lang.OutOfMemory,` except with files beyond 135,000 nodes. The C++ program performed fine (and significantly faster) for all test cases.

The programming effort to manipulate either version is virtually identical because of the C++ automatic garbage collection implemented in the Xerces toolkit (see the later section "Memory Management in C++ DOM").

This is just one of many examples of the pleasant (and sometimes antihype) facts you will encounter when developing C++ XML programs.

It's All About Design

Even though performance issues are important, the main factors to decide between DOM and SAX remain a matter of design.

Table 8.1 enumerates the most common types of C++ XML modules and their constraints and provides guidelines (not a recipe, of course) for the SAX versus DOM decision.

Table 8.1 **Types of Problems and Their Implementation Strategies in C++ XML**

Problem	Considerations	Implementation Strategy	Example
Construct numerous domain-specific objects based on a known vocabulary.	An object model is needed, but it must be one that is meaningful and rich for the program's problem, not a view of XML.	Use SAX handlers (or any event-oriented API, for that matter) as object builders, creating a hierarchy that is semantically rich for your program. Do not fall into the temptation of the "ready-made" hierarchy provided by DOM, because it is a terrible model for your problem and thus is a hard-to-maintain and inexpressive solution.	`syntaxTree` (see Chapter 3)

Problem	Considerations	Implementation Strategy	Example
Construct parallel representations of arbitrary XML documents (without knowing in advance their particular vocabulary).	This problem is faced by most general-purpose XML tools. There is no domain-specific knowledge, but there is sometimes the need to augment the DOM functionality.	Use DOM to represent the in-memory XML, and use Visitors over it in order to add the functionality required.	XML editor (see Chapters 6 and 7)
Manipulate another application's exported XML model.	In this case, apparently no decision needs to be made, because the external program exports a view of the XML (generally as a DOM tree).	In reality, there are two ways of adding functionality over the exported model: Work over it as it is originally provided (create Visitors over the DOM, as just discussed), or adapt the exported model so that it can be perceived as SAX events and work with it in an event-driven way (mainly for the reuse of existing SAX code).	Case 1: XPath programs for IE5.5 (see Chapter 10) Case 2: SAX wrapper for DOM (see the next section of this chapter)
Filter XML input into an XML output.	Because multiple independent outputs may be connected, parallelism becomes a consideration. The partial output of a filter must be available as the input of the next one, before it finishes processing the whole document.	Because SAX provides natural interfaces for filter development, and also because of the parallelism contract, the best approach is to implement this type of program in terms of SAX.	SAXTrimmer (see Chapter 4)
Create XML-based extension languages for C++ applications.	Extension languages (languages for the power user to manipulate and extend C++ programs) are a special case of constructing custom-made hierarchies	Use SAX to create hierarchies or lists of commands (objects that encapsulate an action and provide an "undo" interface).	All the examples in Chapter 13 (Linux and COM+)

Problem	Considerations	Implementation Strategy	Example
	based on XML input. The main difference between these and the general case is that they must provide self-contained behavior, not only data.		
Implement an XML view of other data sources.	When wrapping another data source (such as a database table) as XML, the decision between SAX and DOM must be driven by what your client expects. Chapter 15 expands on this, particularly for the problem of relational databases.	No inherently better approach. Totally driven by client preferences.	SAX view of databases (see Chapter 15) DOM view of databases (see Chapter 15)

About Patterns

The idea of this section is to create a good mental framework of the scenarios for the decision between C++ SAX and DOM. A related subject is that of design patterns. Even though you probably recognize parts of patterns in the list above, this list is not a patterns discussion. For more about XML patterns, refer to the following resources written by me:

- *XML Developer's Guide* (McGraw-Hill, 2000, ISBN 00702126485)
- *XML Patterns, Part I* (`www.xml.com/pub/a/2000/01/19/feature/index.html`, 1999)
- *XML Patterns, Part II* (`www.xml.com/pub/a/2000/02/16/feature/index.html`, 2000)
- *XMLable Pattern* (`www.thefaactory.com/ta/poa.ps`, Arciniegas and Casallas, 1998 [provided on the CD])

C++ SAX Plus C++ DOM

The preceding sections discussed how to decide between SAX and DOM. This complementary section discusses the opposite problem: how to glue them together for the sake of reuse.

The notion of adapters is a key topic in object-oriented software engineering. Adapters are objects whose only purpose is to provide the interface that a client expects while delegating the actual work to an underlying, preexisting object that would otherwise be unfit to use.

The problem of gluing together SAX and DOM interfaces is common whenever preexisting pieces of software written for one model must be adapted to be used with the other.

DOMAsSAX Adapter

Listing 8.1 is the source code of the DOMAsSAX adapter. It behaves like any other SAX source, except that its primary content is a DOM document (thus making the process invisible to the client). The edition shown here is based on the Linux version of Xerces 1.4.

DOMAsSAX as COM

DOMAsSAX is implemented in Listing 8.1. as a Linux program mainly because I try to keep platform variety in the examples. The code has also been ported to Windows and is available on the CD.

For a COM version that implements similar functionality, please refer to the MSDN code center at

`http://msdn.microsoft.com/code/default.asp?URL=/code/topic.asp?URL=/msdn-files/028/000/072/topic.xml`

and download “XML Provider consumer” under the “XML DOM” item. This URL is not very likely to change, but if it does, this book’s Web site will provide the new location.

Listing 8.1 **DOMAsSAX.h**

```
#ifndef DOMAsSAX_H
#define DOMAsSAX_H

#ifdef DO_DEBUG
#define DEBUG(x) cout << x
#else
#define DEBUG(x)
#endif

//---------------------------------------------------------------------------
// Includes and Constants
//---------------------------------------------------------------------------
#include <sax2/SAX2XMLReader.hpp>
#include <parsers/SAX2XMLReaderImpl.hpp>
```

continues

Listing 8.1 **Continued**

```
#include <util/PlatformUtils.hpp>
#include <util/XMLString.hpp>
#include <util/XMLUniDefs.hpp>

// Exceptions
#include <util/TranscodingException.hpp>
#include <util/IOException.hpp>
#include <dom/DOM_DOMException.hpp>
#include <dom/DOM.hpp>

#include <string.h>
#include <iostream.h>
//---------------------------------------------------------------------------
// Class definition
//---------------------------------------------------------------------------
/**
   Adapt a DOM Tree so it can be used as SAX events.
   @see DOMAsSAX
*/

class DOMAsSAX : public SAX2XMLReaderImpl

{
 public:
  /**
  Default constructor.
   */
  DOMAsSAX();

  /**
      Recursively traverse the given DOM node generating SAX events.
      Avoids multiple entrance
      @param n the node
   */
  void parse(const DOM_Node n);


 protected:
  /**
      Recursively traverse the given DOM node generating SAX events.
      @param n the node
   */
  void scanDOM(const DOM_Node n);


 private:
  bool fParseInProgress;
};
#endif DOMAsSAX_H
```

Inheritance Versus Delegation in Adapters

You might notice an apparently small difference between the code shown here and the code provided on the CD. Here, the DOMAsSAX class inherits from `SAX2XMLReaderImpl` (the Xerces implementation of a SAX 2 parser), whereas the DOMAsSAX on the CD inherits directly from the pure virtual `SAX2XMLReader`.

Deriving from the implementation seems very convenient, because much behavior is readily available, and there is no need to go through the pain of defining all the pure virtual methods specified in `SAX2XMLReader`. However, it is a less-than-ideal choice from a design point of view, because it forces an unnecessary coupling between classes that conceptually should not have anything in common except their base class. (This is similar to the awkwardness of being adopted by your brother.)

A much more sensible approach in this case is the one taken in the complete code on the CD: Derive DOMAsSAX from the `SAX2XMLReader` interface, include a reference to a `SAX2XMLReaderImpl` member, and delegate functionality as needed.

Even though these two approaches might pay your bills, it is much better to ask your brother for a favor than to force him to adopt you as a son.

Listing 8.2 shows the implementation file for the DOMAsSAX program.

Listing 8.2 **DOMAsSAX.cpp**

```
#include "DOMAsSAX.h"


DOMAsSAX::DOMAsSAX() : SAX2XMLReaderImpl(),
                    fParseInProgress(false)
{

}

void DOMAsSAX::parse(const DOM_Node n)
{
    // Avoid multiple entrance
    if (fParseInProgress)
        ThrowXML(IOException, XMLExcepts::Gen_ParseInProgress);

    try
    {
        fParseInProgress = true;
        scanDOM(n);
        fParseInProgress = false;
    }

    catch (...)
    {
        fParseInProgress = false;
        throw;
    }
}
```

continues

Listing 8.2 **Continued**

```
void DOMAsSAX::scanDOM(DOM_Node n)
{
  DOMString   nodeName = n.getNodeName();
  DOMString   nodeValue = n.getNodeValue();
  unsigned long len = nodeValue.length();

  // For the sake of readability
#define nn nodeName.rawBuffer()
#define nv nodeValue.rawBuffer()

  switch (n.getNodeType())
  {

  case DOM_Node::DOCUMENT_NODE :
    {

      startDocument();
      DOM_Node child = n.getFirstChild();
      while( child != 0) {
      scanDOM(child);
      child = child.getNextSibling();
      }
      break;
    }

  case DOM_Node::PROCESSING_INSTRUCTION_NODE :
    {
      docPI(nn,nv);
      break;
    }

  // ... and so on for each type of node.

  default:
    DEBUG("Unrecognized node type = " << (long)n.getNodeType());
  }
/**
What is happening on the base class:
void SAX2XMLReaderImpl::docPI(  const   XMLCh* const    target
                        , const XMLCh* const    data)
{
    // Just map to the SAX document handler
    if (fDocHandler)
        fDocHandler->processingInstruction(target, data);

    //
    //  If there are any installed advanced handlers, let's call them
    //  with this info.
    //
    for (unsigned int index = 0; index < fAdvDHCount; index++)
        fAdvDHList[index]->docPI(target, data);
}
**/
}
```

Naturally, it is worth mentioning that the converse case of this code, a SAXAsDOM adapter, is also possible. Such an exercise in SAX handlers is not included, but you should be able to implement it after reading Chapter 4, "SAX C++," Chapter 5, "SAX C++ 2.0 and Advanced Techniques," and this chapter.

Memory Management in C++ SAX

An always-important issue in C++ programs is memory management. Happily, the rules for memory responsibilities in all major C++ XML toolkits are quite well-defined. The following are the rules and guidelines for memory manipulation in C++ SAX.

String Allocation and Release

C++ XML toolkits (MSXML 3.0, Xerces 1.4, Unicorn 1.0) are responsible for creating and destroying the strings they pass to the calling process. Other design issues might change in future versions of these tools, but deleting strings passed to your handler is something that will never be required.

Naturally, copies you make of the strings received in handlers are your responsibility and should be explicitly deallocated.

Strings in MSXML are represented as `wchar` arrays for the character data, plus a separate integer indicating their length. Strings in Xerces are represented as arrays of `XMLCh` and an unsigned integer for their length. In either case, the strings are not guaranteed to be zero-terminated and should never be read beyond the specified length.

Another Comparison: Features Recognized by Each Parser

Before I end all SAX-specific discussions in this chapter, you might want to check out the following information, which compares the features recognized by MSXML and Xerces. (See Chapter 5 for the definition of *feature* in SAX2.)

The following are the features that are handled and recognized by the MSXML `SAXXMLReader`:

- `http://xml.org/sax/features/namespaces`
- `http://xml.org/sax/features/namespace-prefixes`
- `http://xml.org/sax/features/external-general-entities`
- `http://xml.org/sax/features/external-parameter-entities`
- `http://xml.org/sax/features/validation`
- `normalize-line-breaks`
- `server-http-request`

The following are the features that are handled and recognized by the Xerces `SAX2XMLReaderImpl`:

- `http://xml.org/sax/features/namespaces`
- `http://xml.org/sax/features/namespace-prefixes`
- `http://xml.org/sax/features/validation`
- `http://apache.org/xml/features/validation/dynamic`
- `http://xml.org/sax/features/validation feature (default)`
- `http://apache.org/xml/features/validation/reuse-validator`

Memory Management in C++ DOM

DOM, having a much bigger memory footprint than SAX, must pay even closer attention to memory issues.

The idea of automatic memory management has been introduced in all major DOM implementations, and the notion of garbage collection by reference count in C++ is used in Xerces (and recent versions of MSXML).

Garbage collection by reference count is simple: As long as a node is accessible, either directly or via some other node, it stays in memory. As soon as the number of references to a node decreases to zero, the node is ready for automatic collection and is deleted. The most common case for automatic collection happens whenever a local `DOM_Node` variable goes out of scope, as shown in Listing 8.3.

Listing 8.3 **Garbage Collection and Scope**

```
void found(const DOM_Document& doc)
{
      DOM_Element a = doc.createElement("founder");
      a.setAttribute("surname",
                       "Durden");
      DOM_Element b = doc.createElement("company");
      a.setAttribute("name",
                       "The Paper Street Soap Company");
        d.appendChild(b);
} // a gets destroyed, as it goes out of scope
  // b stays, as it is accessible via doc
```

One important consequence of automatic garbage collection as implemented by Xerces is that the application is no longer allowed to create objects in the heap with new objects as it sees fit. Instead, the factory view of creation defined by the DOM interfaces (all the `create` methods in `DOM_Document`) is enforced, and all new nodes must be created either by a factory method in `DOM_Document` or by cloning, as shown in Listing 8.4.

Listing 8.4 **Cloning Versus the Factory Method**

```
void extend(const DOM_Document& doc)
{
      // Factory method in DOM_Document
      DOM_Element a = d.createElement("worker");
      a.setAttribute("name",
                       "Tyler");
      // Cloning
      // Note that the cloneNode has a boolean parameter. This indicates
      // whether a deep cloning should be performed (i.e. copy character
      // data) or not.
      DOM_Element b = a.cloneNode(false);

}
```

Given the conditions just mentioned, there is a risk of relaxing too much and becoming less than careful with memory and performance issues. In particular, it is not uncommon to see beginners passing and returning huge DOM objects by value. As an experienced and well-behaved C++ XML developer, you must avoid such situations and use references whenever possible.

Character Encodings in C++

One very important aspect of C++ XML manipulation is a seemingly basic concept: characters. Characters, despite their ubiquitous use, are a surprisingly deep subject and deserve some careful attention. This section explains characters under C++ and their use in MSXML and Xerces.

Even though you can safely and profitably use the APIs without knowing much about Unicode or `wchar_t`, it is recommended that you read this section, because you can make performance and elegance improvements to your programs if you understand the inner workings of their character representation.

The following sections explain the relationship between characters, character codes, character encodings, and glyphs.

Characters

Characters are abstract entities. They are defined (in the Unicode Standard) as "the smallest components of a written language that have semantic value." Abstract as they may be, you can assign a description to each character (pretty much the same process as naming "the number one"). A character description contains the character's name and other useful information, such as related characters or voicing. For example, the following entry has the character "v" in the Unicode Standard:

```
028C v LATIN SMALL LETTER V WITH HOOK
       = LATIN SMALL LETTER SCRIPT V
       voiced labiodental approximate
       -> 01B2 V latin capital letter v with hook
       -> 03C5 v greek small letter upsilon
```

Character Codes

Just as you can assign a description to each character, you can assign it a unique universal value. This is the primary goal of the Unicode Standard: provide a 16-bit character code for each character in the world (this provides for 65,535 codes).

Character codes in Unicode are identified by a four-position hexadecimal number (four groups of 4 bits), such as the 028C in the example in the preceding section. (Now you know why you can insert Unicode characters in HTML documents using a reference such as `&#x028C;`.) In everyday use, people commonly call Unicode character codes Unicode characters. In print, it is also common to refer to a character with the notation U+*code*, such as U+0424.

Unicode character codes are organized in *blocks*—useful contiguous ranges of related characters. The following is the list of the first few blocks (the complete list is provided on the CD):

```
Start Code; End Code; Block Name
0000; 007F; Basic Latin
0080; 00FF; Latin-1 Supplement
0100; 017F; Latin Extended-A
0180; 024F; Latin Extended-B
0250; 02AF; IPA Extensions
02B0; 02FF; Spacing Modifier Letters
0300; 036F; Combining Diacritical Marks
0370; 03FF; Greek
0400; 04FF; Cyrillic
0530; 058F; Armenian
0590; 05FF; Hebrew
0600; 06FF; Arabic
0700; 074F; Syriac
0780; 07BF; Thaana
0900; 097F; Devanagari
0980; 09FF; Bengali
...   ...   ...
```

For the sake of extensibility, two surrogate blocks have been defined—D800-DBFF and DC00-DFFF. The idea is to expand the number of available points by identifying some Unicode characters as the combination of two codes: one in the surrogate block, and one outside it. These characters are currently not in use, and furthermore, they are illegal in XML, so there is no need to pay too much attention to them.

Character Encodings

As a C++ programmer, you are not as surprised as others when somebody mentions that there are numerous ways to encode that 16-bit character value.

For instance, you might choose to use an unsigned `long int`, thus allowing the representation of each character in only one allocation unit. Also, you might recognize that the first block in Unicode coincides with ASCII and that a more-compact representation of English can be made: Use only 1 byte for Basic Latin and multiple bites for other characters.

Different considerations such as these (in terms of performance and aesthetics) have led to a myriad of machine character representations or encodings. The two most popular (and the only two required to be understood by every XML parser) are UTF-16 (the uniform 16-bit storage model described later) and UTF-8 (a multibyte representation suitable for ASCII and European characters but very expensive for Asian).

Glyphs

We have gone from ideas (characters) to internal computer representation (encodings). Now it is time to present the encoded values to humans.

Characters are represented for humans as *glyphs*. A single character may have many glyphs, and more than two characters may be composed of only one glyph—such as when extension or composition characters such as an acute accent are applied, as shown in Figure 8.2.

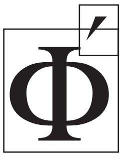
Character Description	Character Code	Encoded Character (UTF-8)	Glyph
Cyrillic Capital Letter ef Combining acute accent (modifier character)	U+0424 U+0301	d0a4 cc81 3c2f 623e (hex representation)	

Figure 8.2 From characters to glyphs.

Characters are rendered by a displaying engine using fonts and typesetting rules. (This might seem obvious for English, but displaying languages such as Kannada is a challenge.)

Representing UTF-16 and UTF-8 in C++

A UTF-8 character is represented as 1 to 3 bytes according to the following rules:

- Characters in the range U+0000 to U+007F in 1 byte: [0][bits 0 to 6]
- Characters in the range U+0080 to U+07FF in 2 bytes: [110][bits 7 to 10]10[bits 0 to 6]
- Characters in the range U+0800 to U+FFFF in 3 bytes: [1110][bits 12 to 15][10][bits 6 to 11][10][bits 0 to 5]

Even though both UTF-16 and UTF-8 are required to be read by a parser, all parsers represent their characters internally in a uniform way (normally, UTF-16).

UTF-16 is often represented by either a `wchar_t` (a wide char) or an unsigned integer. When using wide characters, you have three basic alternatives for string representation:

- Use an elegant implementation of strings defined by the C++ standard `basic_string`
- Rely on wide char C-style functions
- Use a custom-made string class

The `basic_string` class relies on the character representation definition given by the template `char_traits`, shown in Listing 8.5, to provide an efficient definition of a string.

Listing 8.5 **Char Traits**

```
struct char_traits<E> {
       //All the information needed by basic_string to elegantly and
       //efficiently implement a string made of this type of characters
    typedef E char_type;
    typedef T1 int_type;
    typedef T2 pos_type;
    typedef T3 off_type;
    typedef T4 state_type;
    static void assign(E& x, const E& y);
    static E *assign(E *x, size_t n, const E& y);
    static bool eq(const E& x, const E& y);
    static bool lt(const E& x, const E& y);
    static int compare(const E *x, const E *y, size_t n);
    static size_t length(const E *x);
    static E *copy(E *x, const E *y, size_t n);
    static E *move(E *x, const E *y, size_t n);
    static const E *find(const E *x, size_t n, const E& y);
    static E to_char_type(const int_type& ch);
    static int_type to_int_type(const E& c);
    static bool eq_int_type(const int_type& ch1, const int_type& ch2);
    static int_type eof();
    static int_type not_eof(const int_type& ch);
    };
```

This is elegant and convenient, but is unsupported by some compilers. None of the current major parsers rely on `basic_string`.

C-style functions (such as `strlen`) are available for wide characters in the library `<cwchar>`. MSXML 3.0, for the sake of efficiency, chooses to represent every string as an array of `wchar`s and an integer for length. The usage of C-style functions for wide characters is thus common in projects using MSXML.

Finally, custom-made strings are also an option, especially when you're dealing with automatic garbage collection and other assorted trickery. This is more akin to the style chosen by Xerces. It defines its own `XMLCh` character type as an unsigned short (in most platforms) but also defines a series of manipulation methods around it in the form of its `util` and `transcoding` classes.

`DOMString` is implemented as a black box Unicode string that can be transformed into `XMLCh*` by means of the method `rawBuffer`.

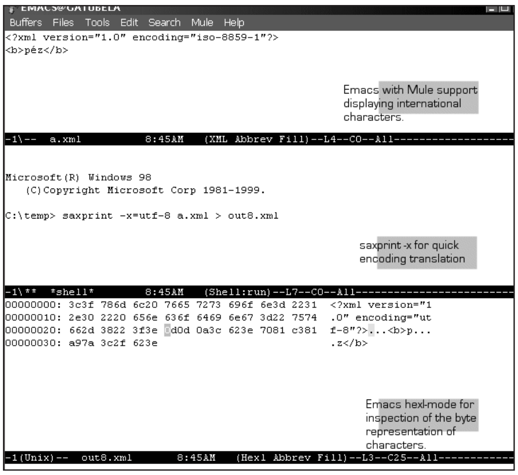
Practical C++ Tools for Translation and Display

Here are the four most useful tools I normally use for character display and transformation:

- emacs using mule and the UTF package for display (included with installation instructions on the CD).
- emacs using hexl mode for the inspection of particular codes for characters.

- saxprint with the `-x` switch. This Xerces utility produces a document using the specified encoding.
- The Unicode 3.0 Standard Book. This is a great tool and a total aesthetic pleasure. It contains sample glyphs for every code in the standard, plus detailed information on its implementation.

Figure 8.3 shows these tools in action.


EMACS@GATUBELA
Buffers Files Tools Edit Search Mule Help
Emacs with Mule support displaying international characters.
saxprint -x for quick encoding translation
Emacs hexl-mode for inspection of the byte representation of characters.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<b>péz</b>

-1\--  a.xml         8:45AM   (XML Abbrev Fill)--L4--C0--All------------------

Microsoft(R) Windows 98
   (C)Copyright Microsoft Corp 1981-1999.

C:\temp> saxprint -x=utf-8 a.xml > out8.xml

-1\**  *shell*       8:45AM   (Shell:run)--L7--C0--All------------------------
00000000: 3c3f 786d 6c20 7665 7273 696f 6e3d 2231  <?xml version="1
00000010: 2e30 2220 656e 636f 6469 6e67 3d22 7574  .0" encoding="ut
00000020: 662d 3822 3f3e 0d0d 0a3c 623e 7081 c381  f-8"?>...<b>p...
00000030: a97a 3c2f 623e                           .z</b>

-1(Unix)--  out8.xml      8:45AM   (Hexl Abbrev Fill)--L3--C25--All-----------
```

Figure 8.3 Tools for translation and display.

Programmatic Translation of UTF-16 and UTF-8 in C++

Even though the practical tools just discussed will prove very useful, sometimes you need programmatic access to conversions. The Listing 8.6 code snippet is based on the open-source program Cvtutf7 by Mark E. Davis and David Goldsmith. It shows how to translate from UTF-16 to UTF-8.

Listing 8.6 **Programmatic Conversion of UTF-16 to UTF-8**

```
typedef unsigned short     UTF16;
typedef unsigned char      UTF8;

// This code is embedded only as a guide to the look and feel of these
// transformations. Pointers to the commented code are given in the CD
// section for this chapter.
void ConvertUTF8toUTF16 (
            UTF8** sourceStart, UTF8* sourceEnd,
            UTF16** targetStart, const UTF16* targetEnd)
{
```

continues

Listing 8.6 **Continued**

```
      ConversionResult result = ok;
      register UTF8* source = *sourceStart;
      register UTF16* target = *targetStart;
      while (source < sourceEnd) {
            register UCS4 ch = 0;
            register unsigned short extraBytesToWrite =
            bytesFromUTF8[*source];
            if (source + extraBytesToWrite > sourceEnd) {
                  result = sourceExhausted; break;
            };
            switch(extraBytesToWrite) {
                  case 5:     ch += *source++; ch <<= 6;
                  case 4:     ch += *source++; ch <<= 6;
                  case 3:     ch += *source++; ch <<= 6;
                  case 2:     ch += *source++; ch <<= 6;
                  case 1:     ch += *source++; ch <<= 6;
                  case 0:     ch += *source++;
            };
            ch -= offsetsFromUTF8[extraBytesToWrite];

            if (target >= targetEnd) {
                  result = targetExhausted; break;
            };
            if (ch <= kMaximumUCS2) {
                  *target++ = ch;
            } else if (ch > kMaximumUTF16) {
                  *target++ = kReplacementCharacter;
            } else {
                  if (target + 1 >= targetEnd) {
                        result = targetExhausted; break;
                  };
                  ch -= halfBase;
                  *target++ = (ch >> halfShift) + kSurrogateHighStart;
                  *target++ = (ch & halfMask) + kSurrogateLowStart;
            };
      };
      *sourceStart = source;
      *targetStart = target;
      return result;
}
```

Summary

This chapter dealt with C++-specific issues, going from the highest-level design decision, SAX versus DOM, to the most basic and specific of concepts: characters.

Toolkits (MSXML and Xerces), paradigms (C++ DOM and SAX), techniques (filters versus builders), and implementations (`basic_string` versus `XMLCh*`, garbage collection versus manual management), were compared and unified to give you a better set of conceptual and physical tools to create advanced and efficient C++ XML applications.

This chapter concludes Part II of this book. The next chapters build on the concepts shown here in order to show programmatic applications in C++ of all the major XML-related technologies, such as XML Schema, XPointer, and XSLT.