C++/Tree Mapping

Getting Started Guide

This document is available in the following formats: XHTML, PDF, and PostScript.

Table of Contents

Preface

About This Document

The goal of this document is to provide you with an understanding of the C++/Tree programming model and allow you to efficiently evaluate XSD against your project's technical requirements. As such, this document is intended for C++ developers and software architects who are looking for an XML processing solution. For a more in-depth description of the C++/Tree mapping refer to the C++/Tree Mapping User Manual.

Prior experience with XML and C++ is required to understand this document. Basic understanding of XML Schema is advantageous but not expected or required.

More Information

Beyond this guide, you may also find the following sources of information useful:

- C++/Tree Mapping User Manual
- C++/Tree Mapping Customization Guide
- C++/Tree Mapping Frequently Asked Questions (FAQ)
- XSD Compiler Command Line Manual
- The `examples/cxx/tree/` directory in the XSD distribution contains a collection of examples and a README file with an overview of each example.
- The `README` file in the XSD distribution explains how to compile the examples on various platforms.
- The xsd-users mailing list is the place to ask technical questions about XSD and the C++/Parser mapping. Furthermore, the archives may already have answers to some of your questions.

1 Introduction

Welcome to CodeSynthesis XSD and the C++/Tree mapping. XSD is a cross-platform W3C XML Schema to C++ data binding compiler. C++/Tree is a W3C XML Schema to C++ mapping that represents the data stored in XML as a statically-typed, vocabulary-specific object model.

1.1 Mapping Overview

Based on a formal description of an XML vocabulary (schema), the C++/Tree mapping produces a tree-like data structure suitable for in-memory processing. The core of the mapping consists of C++ classes that constitute the object model and are derived from types defined in XML Schema as well as XML parsing and serialization code.

Besides the core features, C++/Tree provide a number of additional mapping elements that can be useful in some applications. These include serialization and extraction to/from formats others than XML, such as unstructured text (useful for debugging) and binary representations such as XDR and CDR for high-speed data processing as well as automatic documentation generation. The C++/Tree mapping also provides a wide range of mechanisms for controlling and customizing the generated code.

A typical application that uses C++/Tree for XML processing usually performs the following three steps: it first reads (parses) an XML document to an in-memory object model, it then performs some useful computations on that object model which may involve modification of the model, and finally it may write (serialize) the modified object model back to XML.

The next chapter presents a simple application that performs these three steps. The following chapters show how to use the C++/Tree mapping in more detail.

1.2 Benefits

Traditional XML access APIs such as Document Object Model (DOM) or Simple API for XML (SAX) have a number of drawbacks that make them less suitable for creating robust and maintainable XML processing applications. These drawbacks include:

- Generic representation of XML in terms of elements, attributes, and text forces an application developer to write a substantial amount of bridging code that identifies and transforms pieces of information encoded in XML to a representation more suitable for consumption by the application logic.
- String-based flow control defers error detection to runtime. It also reduces code readability and maintainability.
- Lack of type safety because the data is represented as text.
- Resulting applications are hard to debug, change, and maintain.

In contrast, statically-typed, vocabulary-specific object model produced by the C++/Tree mapping allows you to operate in your domain terms instead of the generic elements, attributes, and text. Static typing helps catch errors at compile-time rather than at run-time. Automatic code generation frees you for more interesting tasks (such as doing something useful with the information stored in the XML documents) and minimizes the effort needed to adapt your applications to changes in the document structure. To summarize, the C++/Tree object model has the following key advantages over generic XML access APIs:

- **Ease of use.** The generated code hides all the complexity associated with parsing and serializing XML. This includes navigating the structure and converting between the text representation and data types suitable for manipulation by the application logic.
- **Natural representation.** The object representation allows you to access the XML data using your domain vocabulary instead of generic elements, attributes, and text.

- **Concise code.** With the object representation the application implementation is simpler and thus easier to read and understand.
- **Safety.** The generated object model is statically typed and uses functions instead of strings to access the information. This helps catch programming errors at compile-time rather than at runtime.
- **Maintainability.** Automatic code generation minimizes the effort needed to adapt the application to changes in the document structure. With static typing, the C++ compiler can pin-point the places in the client code that need to be changed.
- **Compatibility.** Sequences of elements are represented in the object model as containers conforming to the standard C++ sequence requirements. This makes it possible to use standard C++ algorithms on the object representation and frees you from learning yet another container interface, as is the case with DOM.
- **Efficiency.** If the application makes repetitive use of the data extracted from XML, then the C++/Tree object model is more efficient because the navigation is performed using function calls rather than string comparisons and the XML data is extracted only once. Furthermore, the runtime memory usage is reduced due to more efficient data storage (for instance, storing numeric data as integers instead of strings) as well as the static knowledge of cardinality constraints.

2 Hello World Example

In this chapter we will examine how to parse, access, modify, and serialize a very simple XML document using the XSD-generated C++/Tree object model. The code presented in this chapter is based on the `hello` example which can be found in the `examples/cxx/tree/` directory of the XSD distribution.

2.1 Writing XML Document and Schema

First, we need to get an idea about the structure of the XML documents we are going to process. Our `hello.xml`, for example, could look like this:

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

  <greeting>Hello</greeting>

  <name>sun</name>
  <name>moon</name>
  <name>world</name>

</hello>
```

Then we can write a description of the above XML in the XML Schema language and save it into `hello.xsd`:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="hello_t">
    <xs:sequence>
      <xs:element name="greeting" type="xs:string"/>
      <xs:element name="name" type="xs:string" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="hello" type="hello_t"/>

</xs:schema>
```

Even if you are not familiar with XML Schema, it should be easy to connect declarations in `hello.xsd` to elements in `hello.xml`. The `hello_t` type is defined as a sequence of the nested `greeting` and `name` elements. Note that the term sequence in XML Schema means that elements should appear in a particular order as opposed to appearing multiple times. The `name` element has its `maxOccurs` property set to `unbounded` which means it can appear multiple times in an XML document. Finally, the globally-defined `hello` element prescribes the root element for our vocabulary. For an easily-approachable introduction to XML Schema refer to XML Schema Part 0: Primer.

The above schema is a specification of our XML vocabulary; it tells everybody what valid documents of our XML-based language should look like. We can also update our `hello.xml` to include the information about the schema so that XML parsers can validate our document:

```
<?xml version="1.0"?>
<hello xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="hello.xsd">

  <greeting>Hello</greeting>

  <name>sun</name>
  <name>moon</name>
  <name>world</name>

</hello>
```

The next step is to compile the schema to generate the object model and parsing functions.

2.2 Translating Schema to C++

Now we are ready to translate our `hello.xsd` to C++. To do this we invoke the XSD compiler from a terminal (UNIX) or a command prompt (Windows):

```
$ xsd cxx-tree hello.xsd
```

The XSD compiler produces two C++ files: `hello.hxx` and `hello.cxx`. The following code fragment is taken from `hello.hxx`; it should give you an idea about what gets generated:

```
class hello_t
{
public:
  // greeting
  //
  typedef xml_schema::string greeting_type;

  const greeting_type&
  greeting () const;

  greeting_type&
  greeting ();

  void
  greeting (const greeting_type& x);

  // name
  //
  typedef xml_schema::string name_type;
  typedef xsd::sequence<name_type> name_sequence;
  typedef name_sequence::iterator name_iterator;
  typedef name_sequence::const_iterator name_const_iterator;

  const name_sequence&
  name () const;

  name_sequence&
  name ();

  void
  name (const name_sequence& s);

  // Constructor.
  //
  hello_t (const greeting_type&);

  ...

};
```

```
std::auto_ptr<hello_t>
hello (const std::string& uri);

std::auto_ptr<hello_t>
hello (std::istream&);
```

The `hello_t` C++ class corresponds to the `hello_t` XML Schema type. For each element in this type a set of C++ type definitions as well as accessor and modifier functions are generated inside the `hello_t` class. Note that the type definitions and member functions for the `greeting` and `name` elements are different because of the cardinality differences between these two elements (`greeting` is a required single element and `name` is a sequence of elements).

The `xml_schema::string` type used in the type definitions is a C++ class provided by the XSD runtime that corresponds to built-in XML Schema type `string`. The `xml_schema::string` is based on `std::string` and can be used as such. Similarly, the `sequence` class template that is used in the `name_sequence` type definition is based on and has the same interface as `std::vector`. The mapping between the built-in XML Schema types and C++ types is described in more detail in Section 4.5, "Mapping for the Built-in XML Schema Types". The `hello_t` class also includes a constructor with an initializer for the required `greeting` element as its argument.

The `hello` overloaded global functions correspond to the `hello` global element in XML Schema. A global element in XML Schema is a valid document root. By default XSD generated a set of parsing functions for each global element defined in XML Schema (this can be overridden with the `--root-element-*` options). Parsing functions return a dynamically allocated object model as an automatic pointer. The actual pointer used depends on the C++ standard selected. For C++98 it is `std::auto_ptr` as shown above. For C++11 it is `std::unique_ptr`. For example, if we modify our XSD compiler invocation to select C++11:

```
$ xsd cxx-tree --std c++11 hello.xsd
```

Then the parsing function signatures will become:

```
std::unique_ptr<hello_t>
hello (const std::string& uri);

std::unique_ptr<hello_t>
hello (std::istream&);
```

For more information on parsing functions see Chapter 5, "Parsing".

2.3 Implementing Application Logic

At this point we have all the parts we need to do something useful with the information stored in our XML document:

```
#include <iostream>
#include "hello.hxx"

using namespace std;

int
main (int argc, char* argv[])
{
  try
  {
    auto_ptr<hello_t> h (hello (argv[1]));

    for (hello_t::name_const_iterator i (h->name ().begin ());
         i != h->name ().end ();
         ++i)
    {
      cerr << h->greeting () << ", " << *i << "!" << endl;
    }
  }
  catch (const xml_schema::exception& e)
  {
    cerr << e << endl;
    return 1;
  }
}
```

The first part of our application calls one of the parsing functions to parser an XML file specified in the command line. We then use the returned object model to iterate over names and print a greeting line for each of them. Finally, we catch and print the `xml_schema::exception` exception in case something goes wrong. This exception is the root of the exception hierarchy used by the XSD-generated code.

2.4 Compiling and Running

After saving our application from the previous section in `driver.cxx`, we are ready to compile our first program and run it on the test XML document. On a UNIX system this can be done with the following commands:

```
$ c++ -I.../libxsd -c driver.cxx hello.cxx
$ c++ -o driver driver.o hello.o -lxerces-c
$ ./driver hello.xml
Hello, sun!
Hello, moon!
Hello, world!
```

Here `.../libxsd` represents the path to the `libxsd` directory in the XSD distribution. Note also that we are required to link our application with the Xerces-C++ library because the generated code uses it as the underlying XML parser.

2.5 Adding Serialization

While parsing and accessing the XML data may be everything you need, there are applications that require creating new or modifying existing XML documents. By default XSD does not produce serialization code. We will need to request it with the `--generate-serialization` options:

```
$ xsd cxx-tree --generate-serialization hello.xsd
```

If we now examine the generated `hello.hxx` file, we will find a set of overloaded serialization functions, including the following version:

```
void
hello (std::ostream&,
       const hello_t&,
       const xml_schema::namespace_infomap& =
         xml_schema::namespace_infomap ());
```

Just like with parsing functions, XSD generates serialization functions for each global element unless instructed otherwise with one of the `--root-element-*` options. For more information on serialization functions see Chapter 6, "Serialization".

We first examine an application that modifies an existing object model and serializes it back to XML:

```
#include <iostream>
#include "hello.hxx"

using namespace std;

int
main (int argc, char* argv[])
{
  try
  {
    auto_ptr<hello_t> h (hello (argv[1]));
```

```
    // Change the greeting phrase.
    //
    h->greeting ("Hi");

    // Add another entry to the name sequence.
    //
    h->name ().push_back ("mars");

    // Serialize the modified object model to XML.
    //
    xml_schema::namespace_infomap map;
    map[""].name = "";
    map[""].schema = "hello.xsd";

    hello (cout, *h, map);
  }
  catch (const xml_schema::exception& e)
  {
    cerr << e << endl;
    return 1;
  }
}
```

First, our application parses an XML document and obtains its object model as in the previous example. Then it changes the greeting string and adds another entry to the list of names. Finally, it serializes the object model back to XML by calling the serialization function.

The first argument we pass to the serialization function is `cout` which results in the XML being written to the standard output for us to inspect. We could have also written the result to a file or memory buffer by creating an instance of `std::ofstream` or `std::ostringstream` and passing it instead of `cout`. The second argument is the object model we want to serialize. The final argument is an optional namespace information map for our vocabulary. It captures information such as namespaces, namespace prefixes to which they should be mapped, and schemas associated with these namespaces. If we don't provide this argument then generic namespace prefixes (`p1`, `p2`, etc.) will be automatically assigned to XML namespaces and no schema information will be added to the resulting document (see Chapter 6, "Serialization" for details). In our case, the prefix (map key) and namespace name are empty because our vocabulary does not use XML namespaces.

If we now compile and run this application we will see the output as shown in the following listing:

```
<?xml version="1.0"?>
<hello xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="hello.xsd">

  <greeting>Hi</greeting>
```

```
  <name>sun</name>
  <name>moon</name>
  <name>world</name>
  <name>mars</name>

</hello>
```

We can also create and serialize an object model from scratch as shown in the following example:

```
#include <iostream>
#include <fstream>
#include "hello.hxx"

using namespace std;

int
main (int argc, char* argv[])
{
  try
  {
    hello_t h ("Hi");

    hello_t::name_sequence& ns (h.name ());

    ns.push_back ("Jane");
    ns.push_back ("John");

    // Serialize the object model to XML.
    //
    xml_schema::namespace_infomap map;
    map[""].name = "";
    map[""].schema = "hello.xsd";

    std::ofstream ofs (argv[1]);
    hello (ofs, h, map);
  }
  catch (const xml_schema::exception& e)
  {
    cerr << e << endl;
    return 1;
  }
}
```

In this example we used the generated constructor to create an instance of type `hello_t`. To reduce typing, we obtained a reference to the name sequence which we then used to add a few names. The serialization part is identical to the previous example except this time we are writing to a file. If we compile and run this program, it produces the following XML file:

```
<?xml version="1.0"?>
<hello xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="hello.xsd">

  <greeting>Hi</greeting>

  <name>Jane</name>
  <name>John</name>

</hello>
```

2.6 Selecting Naming Convention

By default XSD uses the so-called K&R (Kernighan and Ritchie) identifier naming convention in the generated code. In this convention both type and function names are in lower case and words are separated by underscores. If your application code or schemas use a different notation, you may want to change the naming convention used in the generated code for consistency. XSD supports a set of widely-used naming conventions that you can select with the `--type-naming` and `--function-naming` options. You can also further refine one of the predefined conventions or create a completely custom naming scheme by using the `--*-regex` options.

As an example, let's assume that our "Hello World" application uses the so-called upper-camel-case naming convention for types (that is, each word in a type name is capitalized) and the K&R convention for function names. Since K&R is the default convention for both type and function names, we only need to change the type naming scheme:

```
$ xsd cxx-tree --type-naming ucc hello.xsd
```

The `ucc` argument to the `--type-naming` options stands for upper-camel-case. If we now examine the generated `hello.hxx`, we will see the following changes compared to the declarations shown in the previous sections:

```
class Hello_t
{
public:
  // greeting
  //
  typedef xml_schema::String GreetingType;

  const GreetingType&
  greeting () const;

  GreetingType&
  greeting ();

  void
  greeting (const GreetingType& x);
```

```
  // name
  //
  typedef xml_schema::String NameType;
  typedef xsd::sequence<NameType> NameSequence;
  typedef NameSequence::iterator NameIterator;
  typedef NameSequence::const_iterator NameConstIterator;

  const NameSequence&
  name () const;

  NameSequence&
  name ();

  void
  name (const NameSequence& s);

  // Constructor.
  //
  Hello_t (const GreetingType&);

  ...

};

std::auto_ptr<Hello_t>
hello (const std::string& uri);

std::auto_ptr<Hello_t>
hello (std::istream&);
```

Notice that the type names in the `xml_schema` namespace, for example `xml_schema::String`, now also use the upper-camel-case naming convention. The only thing that we may be unhappy about in the above code is the `_t` suffix in `Hello_t`. If we are not in a position to change the schema, we can *touch-up* the `ucc` convention with a custom translation rule using the `--type-regex` option:

```
$ xsd cxx-tree --type-naming ucc --type-regex '/ (.+)_t/\u$1/' hello.xsd
```

This results in the following changes to the generated code:

```
class Hello
{
public:
  // greeting
  //
  typedef xml_schema::String GreetingType;

  const GreetingType&
  greeting () const;
```

```
  GreetingType&
  greeting ();

  void
  greeting (const GreetingType& x);

  // name
  //
  typedef xml_schema::String NameType;
  typedef xsd::sequence<NameType> NameSequence;
  typedef NameSequence::iterator NameIterator;
  typedef NameSequence::const_iterator NameConstIterator;

  const NameSequence&
  name () const;

  NameSequence&
  name ();

  void
  name (const NameSequence& s);

  // Constructor.
  //
  Hello (const GreetingType&);

  ...

};

std::auto_ptr<Hello>
hello (const std::string& uri);

std::auto_ptr<Hello>
hello (std::istream&);
```

For more detailed information on the `--type-naming`, `--function-naming`, `--type-regex`, and other `--*-regex` options refer to the NAMING CONVENTION section in the XSD Compiler Command Line Manual.

2.7 Generating Documentation

While our object model is quite simple, real-world vocabularies can be quite complex with hundreds of types, elements, and attributes. For such vocabularies figuring out which types provide which member functions by studying the generated source code or schemas can be a daunting task. To provide application developers with a more accessible way of understanding the generated object models, the XSD compiler can be instructed to produce source code with documentation comments in the Doxygen format. Then the source code can be processed with the

Doxygen documentation system to extract this information and produce documentation in various formats.

In this section we will see how to generate documentation for our "Hello World" vocabulary. To showcase the full power of the XSD documentation facilities, we will first document our schema. The XSD compiler will then transfer this information from the schema to the generated code and then to the object model documentation. Note that the documentation in the schema is not required for XSD to generate useful documentation. Below you will find our `hello.xsd` with added documentation:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="hello_t">

    <xs:annotation>
      <xs:documentation>
        The hello_t type consists of a greeting phrase and a
        collection of names to which this greeting applies.
      </xs:documentation>
    </xs:annotation>

    <xs:sequence>

      <xs:element name="greeting" type="xs:string">
        <xs:annotation>
          <xs:documentation>
            The greeting element contains the greeting phrase
            for this hello object.
          </xs:documentation>
        </xs:annotation>
      </xs:element>

      <xs:element name="name" type="xs:string" maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>
            The name elements contains names to be greeted.
          </xs:documentation>
        </xs:annotation>
      </xs:element>

    </xs:sequence>
  </xs:complexType>

  <xs:element name="hello" type="hello_t">
    <xs:annotation>
      <xs:documentation>
        The hello element is a root of the Hello XML vocabulary.
        Every conforming document should start with this element.
      </xs:documentation>
```

```
    </xs:annotation>
  </xs:element>

</xs:schema>
```

The first step in obtaining the documentation is to recompile our schema with the `--generate-doxygen` option:

```
$ xsd cxx-tree --generate-serialization --generate-doxygen hello.xsd
```

Now the generated `hello.hxx` file contains comments in the Doxygen format. The next step is to process this file with the Doxygen documentation system. If your project does not use Doxygen then you first need to create a configuration file for your project:

```
$ doxygen -g hello.doxygen
```

You only need to perform this step once. Now we can generate the documentation by executing the following command in the directory with the generated source code:

```
$ doxygen hello.doxygen
```

While the generated documentation can be useful as is, we can go one step further and link (using the Doxygen tags mechanism) the documentation for our object model with the documentation for the XSD runtime library which defines C++ classes for the built-in XML Schema types. This way we can seamlessly browse between documentation for the `hello_t` class which is generated by the XSD compiler and the `xml_schema::string` class which is defined in the XSD runtime library. The Doxygen configuration file for the XSD runtime is provided with the XSD distribution.

You can view the result of the steps described in this section on the Hello Example Documentation page.

3 Overall Mapping Configuration

The C++/Tree mapping has a number of configuration parameters that determine the overall properties and behavior of the generated code. Configuration parameters are specified with the XSD command line options. This chapter describes configuration aspects that are most commonly encountered by application developers. These include: the C++ standard, the character type that is used by the generated code, handling of vocabularies that use XML Schema polymorphism, XML Schema to C++ namespace mapping, and thread safety. For more ways to configure the generated code refer to the XSD Compiler Command Line Manual.

3.1 C++ Standard

The C++/Tree mapping provides support for ISO/IEC C++ 1998/2003 (C++98) and ISO/IEC C++ 2011 (C++11). To select the C++ standard for the generated code we use the `--std` XSD compiler command line option. While the majority of the examples in this guide use C++98, support for the new functionality and library components introduced in C++11 are discussed throughout the document.

3.2 Character Type and Encoding

The C++/Tree mapping has built-in support for two character types: `char` and `wchar_t`. You can select the character type with the `--char-type` command line option. The default character type is `char`. The character type affects all string and string-based types that are used in the mapping. These include the string-based built-in XML Schema types, exception types, stream types, etc.

Another aspect of the mapping that depends on the character type is character encoding. For the `char` character type the default encoding is UTF-8. Other supported encodings are ISO-8859-1, Xerces-C++ Local Code Page (LPC), as well as custom encodings. You can select which encoding should be used in the object model with the `--char-encoding` command line option.

For the `wchar_t` character type the encoding is automatically selected between UTF-16 and UTF-32/UCS-4 depending on the size of the `wchar_t` type. On some platforms (for example, Windows with Visual C++ and AIX with IBM XL C++) `wchar_t` is 2 bytes long. For these platforms the encoding is UTF-16. On other platforms `wchar_t` is 4 bytes long and UTF-32/UCS-4 is used.

Note also that the character encoding that is used in the object model is independent of the encodings used in input and output XML. In fact, all three (object mode, input XML, and output XML) can have different encodings.

3.3 Support for Polymorphism

By default XSD generates non-polymorphic code. If your vocabulary uses XML Schema polymorphism in the form of `xsi:type` and/or substitution groups, then you will need to compile your schemas with the `--generate-polymorphic` option to produce polymorphism-aware code. For more information on working with polymorphic object models, refer to Section 2.11, "Mapping for `xsi:type` and Substitution Groups" in the C++/Tree Mapping User Manual.

3.4 Namespace Mapping

XSD maps XML namespaces specified in the `targetNamespace` attribute in XML Schema to one or more nested C++ namespaces. By default, a namespace URI is mapped to a sequence of C++ namespace names by removing the protocol and host parts and splitting the rest into a sequence of names with `'/'` as the name separator.

The default mapping of namespace URIs to C++ namespaces can be altered using the `--namespace-map` and `--namespace-regex` compiler options. For example, to map namespace URI `http://www.codesynthesis.com/my` to C++ namespace `cs::my`, we can use the following option:

```
--namespace-map http://www.codesynthesis.com/my=cs::my
```

A vocabulary without a namespace is mapped to the global scope. This also can be altered with the above options by using an empty name for the XML namespace:

```
--namespace-map =cs
```

3.5 Thread Safety

XSD-generated code is thread-safe in the sense that you can use different instantiations of the object model in several threads concurrently. This is possible due to the generated code not relying on any writable global variables. If you need to share the same object between several threads then you will need to provide some form of synchronization. One approach would be to use the generated code customization mechanisms to embed synchronization primitives into the generated C++ classes. For more information on generated code customization refer to the C++/Tree Mapping Customization Guide.

If you also would like to call parsing and/or serialization functions from several threads potentially concurrently, then you will need to make sure the Xerces-C++ runtime is initialized and terminated only once. The easiest way to do this is to initialize/terminate Xerces-C++ from `main()` when there are no threads yet/anymore:

```
#include <xercesc/util/PlatformUtils.hpp>

int
main ()
{
  xercesc::XMLPlatformUtils::Initialize ();

  {
    // Start/terminate threads and parse/serialize here.
```

```
  }

  xercesc::XMLPlatformUtils::Terminate ();
}
```

Because you initialize the Xerces-C++ runtime yourself you should also pass the `xml_schema::flags::dont_initialize` flag to parsing and serialization functions. See Chapter 5, "Parsing" and Chapter 6, "Serialization" for more information.

4 Working with Object Models

As we have seen in the previous chapters, the XSD compiler generates a C++ class for each type defined in XML Schema. Together these classes constitute an object model for an XML vocabulary. In this chapter we will take a closer look at different elements that comprise an object model class as well as how to create, access, and modify object models.

In this and subsequent chapters we will use the following schema that describes a collection of person records. We save it in `people.xsd`:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:simpleType name="gender_t">
    <xs:restriction base="xs:string">
      <xs:enumeration value="male"/>
      <xs:enumeration value="female"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="person_t">
    <xs:sequence>
      <xs:element name="first-name" type="xs:string"/>
      <xs:element name="middle-name" type="xs:string" minOccurs="0"/>
      <xs:element name="last-name" type="xs:string"/>
      <xs:element name="gender" type="gender_t"/>
      <xs:element name="age" type="xs:short"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:unsignedInt" use="required"/>
  </xs:complexType>

  <xs:complexType name="people_t">
    <xs:sequence>
      <xs:element name="person" type="person_t" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
```

```
  <xs:element name="people" type="people_t"/>

</xs:schema>
```

A sample XML instance to go along with this schema is saved in `people.xml`:

```
<?xml version="1.0"?>
<people xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="people.xsd">

  <person id="1">
    <first-name>John</first-name>
    <last-name>Doe</last-name>
    <gender>male</gender>
    <age>32</age>
  </person>

  <person id="2">
    <first-name>Jane</first-name>
    <middle-name>Mary</middle-name>
    <last-name>Doe</last-name>
    <gender>female</gender>
    <age>28</age>
  </person>

</people>
```

Compiling `people.xsd` with the XSD compiler results in three generated C++ classes: `gender_t`, `person_t`, and `people_t`. The `gender_t` class is modelled after the C++ `enum` type. Its definition is presented below:

```
class gender_t: public xml_schema::string
{
public:
  enum value
  {
    male,
    female
  };

  gender_t (value);
  gender_t (const xml_schema::string&);

  gender_t&
  operator= (value);

  operator value () const;
};
```

The following listing shows how we can use this type:

```
gender_t m (gender_t::male);
gender_t f ("female");

if (m == "female" || f == gender_t::male)
{
  ...
}

switch (m)
{
case gender_t::male:
  {
    ...
  }
case gender_t::female:
  {
    ...
  }
}
```

The other two classes will be examined in detail in the subsequent sections.

4.1 Attribute and Element Cardinalities

As we have seen in the previous chapters, XSD generates a different set of type definitions and member functions for elements with different cardinalities. The C++/Tree mapping divides all the possible element and attribute cardinalities into three cardinality classes: *one*, *optional*, and *sequence*.

The *one* cardinality class covers all elements that should occur exactly once as well as required attributes. In our example, the `first-name`, `last-name`, `gender`, and `age` elements as well as the `id` attribute belong to this cardinality class. The following code fragment shows type definitions as well as the accessor and modifier functions that are generated for the `gender` element in the `person_t` class:

```
class person_t
{
  // gender
  //
  typedef gender_t gender_type;

  const gender_type&
  gender () const;

  gender_type&
  gender ();
```

```
  void
  gender (const gender_type&);
};
```

The `gender_type` type is an alias for the element's type. The first two accessor functions return read-only (constant) and read-write references to the element's value, respectively. The modifier function sets the new value for the element.

The *optional* cardinality class covers all elements that can occur zero or one time as well as optional attributes. In our example, the `middle-name` element belongs to this cardinality class. The following code fragment shows the type definitions as well as the accessor and modifier functions that are generated for this element in the `person_t` class:

```
class person_t
{
  // middle-name
  //
  typedef xml_schema::string middle_name_type;
  typedef xsd::optional<middle_name_type> middle_name_optional;

  const middle_name_optional&
  middle_name () const;

  middle_name_optional&
  middle_name ();

  void
  middle_name (const middle_name_type&);

  void
  middle_name (const middle_name_optional&);
};
```

As with the `gender` element, `middle_name_type` is an alias for the element's type. The `middle_name_optional` type is a container for the element's optional value. It can be queried for the presence of the value using the `present()` function. The value itself can be retrieved using the `get()` accessor and set using the `set()` modifier. The container can be reverted to the value not present state with the call to the `reset()` function. The following example shows how we can use this container:

```
person_t::middle_name_optional n ("John");

if (n.present ())
{
  cout << n.get () << endl;
}

n.set ("Jane");
n.reset ();
```

Unlike the *one* cardinality class, the accessor functions for the *optional* class return read-only (constant) and read-write references to the container instead of the element's value directly. The modifier functions set the new value for the element.

Finally, the *sequence* cardinality class covers all elements that can occur more than once. In our example, the `person` element in the `people_t` type belongs to this cardinality class. The following code fragment shows the type definitions as well as the accessor and modifier functions that are generated for this element in the `people_t` class:

```
class people_t
{
  // person
  //
  typedef person_t person_type;
  typedef xsd::sequence<person_type> person_sequence;
  typedef person_sequence::iterator person_iterator;
  typedef person_sequence::const_iterator person_const_iterator;

  const person_sequence&
  person () const;

  person_sequence&
  person ();

  void
  person (const person_sequence&);
};
```

Identical to the other cardinality classes, `person_type` is an alias for the element's type. The `person_sequence` type is a sequence container for the element's values. It is based on and has the same interface as `std::vector` and therefore can be used in similar ways. The `person_iterator` and `person_const_iterator` types are read-only (constant) and read-write iterators for the `person_sequence` container.

Similar to the *optional* cardinality class, the accessor functions for the *sequence* class return read-only (constant) and read-write references to the sequence container. The modifier functions copies the entries from the passed sequence.

C++/Tree is a "flattening" mapping in a sense that many levels of nested compositors (`choice` and `sequence`), all potentially with their own cardinalities, are in the end mapped to a flat set of elements with one of the three cardinality classes discussed above. While this results in a simple and easy to use API for most types, in certain cases, the order of elements in the actual XML documents is not preserved once parsed into the object model. To overcome this limitation we can mark certain schema types, for which content order is not sufficiently preserved, as ordered. For more information on this functionality refer to Section 2.8.4, "Element Order" in the C++/Tree Mapping User Manual.

For complex schemas with many levels of nested compositors (`choice` and `sequence`) it can also be hard to deduce the cardinality class of a particular element. The generated Doxygen documentation can greatly help with this task. For each element and attribute the documentation clearly identifies its cardinality class. Alternatively, you can study the generated header files to find out the cardinality class of a particular attribute or element.

In the next sections we will examine how to access and modify information stored in an object model using accessor and modifier functions described in this section.

4.2 Accessing the Object Model

In this section we will learn how to get to the information stored in the object model for our person records vocabulary. The following application accesses and prints the contents of the `people.xml` file:

```
#include <iostream>
#include "people.hxx"

using namespace std;

int
main ()
{
  auto_ptr<people_t> ppl (people ("people.xml"));

  // Iterate over individual person records.
  //
  people_t::person_sequence& ps (ppl->person ());

  for (people_t::person_iterator i (ps.begin ()); i != ps.end (); ++i)
  {
    person_t& p (*i);

    // Print names: first-name and last-name are required elements,
    // middle-name is optional.
    //
    cout << "name:   " << p.first_name () << " ";
```

```
    if (p.middle_name ().present ())
      cout << p.middle_name ().get () << " ";

    cout << p.last_name () << endl;

    // Print gender, age, and id which are all required.
    //
    cout << "gender: " << p.gender () << endl
         << "age:    " << p.age () << endl
         << "id:     " << p.id () << endl
         << endl;
  }
}
```

This code shows common patterns of accessing elements and attributes with different cardinality classes. For the sequence element (`person` in `people_t`) we first obtain a reference to the container and then iterate over individual records. The values of elements and attributes with the *one* cardinality class (`first-name`, `last-name`, `gender`, `age`, and `id`) can be obtained directly by calling the corresponding accessor functions. For the optional element `middle-name` we first check if the value is present and only then call `get()` to retrieve it.

Note that when we want to reduce typing by creating a variable representing a fragment of the object model that we are currently working with (`ps` and `p` above), we obtain a reference to that fragment instead of making a potentially expensive copy. This is generally a good rule to follow when creating high-performance applications.

If we run the above application on our sample `people.xml`, the output looks as follows:

```
name:   John Doe
gender: male
age:    32
id:     1

name:   Jane Mary Doe
gender: female
age:    28
id:     2
```

4.3 Modifying the Object Model

In this section we will learn how to modify the information stored in the object model for our person records vocabulary. The following application changes the contents of the `people.xml` file:

```
#include <iostream>
#include "people.hxx"

using namespace std;
```

```
int
main ()
{
  auto_ptr<people_t> ppl (people ("people.xml"));

  // Iterate over individual person records and increment
  // the age.
  //
  people_t::person_sequence& ps (ppl->person ());

  for (people_t::person_iterator i (ps.begin ()); i != ps.end (); ++i)
  {
    // Alternative way: i->age ()++;
    //
    i->age (i->age () + 1);
  }

  // Add middle-name to the first record and remove it from
  // the second.
  //
  person_t& john (ps[0]);
  person_t& jane (ps[1]);

  john.middle_name ("Mary");
  jane.middle_name ().reset ();

  // Add another John record.
  //
  ps.push_back (john);

  // Serialize the modified object model to XML.
  //
  xml_schema::namespace_infomap map;
  map[""].name = "";
  map[""].schema = "people.xsd";

  people (cout, *ppl, map);
}
```

The first modification the above application performs is iterating over person records and incrementing the age value. This code fragment shows how to modify the value of a required attribute or element. The next modification shows how to set a new value for the optional `middle-name` element as well as clear its value. Finally the example adds a copy of the John Doe record to the `person` element sequence.

Note that in this case using references for the `ps`, `john`, and `jane` variables is no longer a performance improvement but a requirement for the application to function correctly. If we hadn't used references, all our changes would have been made on copies without affecting the

object model.

If we run the above application on our sample `people.xml`, the output looks as follows:

```
<?xml version="1.0"?>
<people xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="people.xsd">

  <person id="1">
    <first-name>John</first-name>
    <middle-name>Mary</middle-name>
    <last-name>Doe</last-name>
    <gender>male</gender>
    <age>33</age>
  </person>

  <person id="2">
    <first-name>Jane</first-name>
    <last-name>Doe</last-name>
    <gender>female</gender>
    <age>29</age>
  </person>

  <person id="1">
    <first-name>John</first-name>
    <middle-name>Mary</middle-name>
    <last-name>Doe</last-name>
    <gender>male</gender>
    <age>33</age>
  </person>

</people>
```

4.4 Creating the Object Model from Scratch

In this section we will learn how to create a new object model for our person records vocabulary. The following application recreates the content of the original `people.xml` file:

```
#include <iostream>
#include "people.hxx"

using namespace std;

int
main ()
{
  people_t ppl;
  people_t::person_sequence& ps (ppl.person ());

  // Add the John Doe record.
```

```
  //
  ps.push_back (
    person_t ("John",         // first-name
              "Doe",          // last-name
              gender_t::male, // gender
              32,             // age
              1));

  // Add the Jane Doe record.
  //
  ps.push_back (
    person_t ("Jane",           // first-name
              "Doe",            // last-name
              gender_t::female, // gender
              28,               // age
              2));              // id

  // Add middle name to the Jane Doe record.
  //
  person_t& jane (ps.back ());
  jane.middle_name ("Mary");

  // Serialize the object model to XML.
  //
  xml_schema::namespace_infomap map;
  map[""].name = "";
  map[""].schema = "people.xsd";

  people (cout, ppl, map);
}
```

The only new part in the above application is the calls to the `people_t` and `person_t` constructors. As a general rule, for each C++ class XSD generates a constructor with initializers for each element and attribute belonging to the *one* cardinality class. For our vocabulary, the following constructors are generated:

```
class person_t
{
  person_t (const first_name_type&,
            const last_name_type&,
            const gender_type&,
            const age_type&,
            const id_type&);
};

class people_t
{
  people_t ();
};
```

Note also that we set the `middle-name` element on the Jane Doe record by obtaining a reference to that record in the object model and setting the `middle-name` value on it. This is a general rule that should be followed in order to obtain the best performance: if possible, direct modifications to the object model should be preferred to modifications on temporaries with subsequent copying. The following code fragment shows a semantically equivalent but slightly slower version:

```
// Add the Jane Doe record.
//
person_t jane ("Jane",           // first-name
               "Doe",            // last-name
               gender_t::female, // gender
               28,               // age
               2);               // id

jane.middle_name ("Mary");

ps.push_back (jane);
```

We can also go one step further to reduce copying and improve the performance of our application by using the non-copying `push_back()` function which assumes ownership of the passed objects:

```
// Add the John Doe record. C++98 version.
//
auto_ptr<person_t> john_p (
  new person_t ("John",           // first-name
                "Doe",            // last-name
                gender_t::male,   // gender
                32,               // age
                1));
ps.push_back (john_p); // assumes ownership

// Add the Jane Doe record. C++11 version
//
unique_ptr<person_t> jane_p (
  new person_t ("Jane",           // first-name
                "Doe",            // last-name
                gender_t::female, // gender
                28,               // age
                2));              // id
ps.push_back (std::move (jane_p)); // assumes ownership
```

For more information on the non-copying modifier functions refer to Section 2.8, "Mapping for Local Elements and Attributes" in the C++/Tree Mapping User Manual. The above application produces the following output:

```
<?xml version="1.0" ?>
<people xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="people.xsd">

  <person id="1">
    <first-name>John</first-name>
    <last-name>Doe</last-name>
    <gender>male</gender>
    <age>32</age>
  </person>

  <person id="2">
    <first-name>Jane</first-name>
    <middle-name>Mary</middle-name>
    <last-name>Doe</last-name>
    <gender>female</gender>
    <age>28</age>
  </person>

</people>
```

4.5 Mapping for the Built-in XML Schema Types

Our person record vocabulary uses several built-in XML Schema types: `string`, `short`, and `unsignedInt`. Until now we haven't talked about the mapping of built-in XML Schema types to C++ types and how to work with them. This section provides an overview of the built-in types. For more detailed information refer to Section 2.5, "Mapping for Built-in Data Types" in the C++/Tree Mapping User Manual.

In XML Schema, built-in types are defined in the XML Schema namespace. By default, the C++/Tree mapping maps this namespace to C++ namespace `xml_schema` (this mapping can be altered with the `--namespace-map` option). The following table summarizes the mapping of XML Schema built-in types to C++ types:

XML Schema type	Alias in the `xml_schema` namespace	C++ type
fixed-length integral types		
`byte`	`byte`	`signed char`
`unsignedByte`	`unsigned_byte`	`unsigned char`
`short`	`short_`	`short`
`unsignedShort`	`unsigned_short`	`unsigned short`
`int`	`int_`	`int`

`unsignedInt`	`unsigned_int`	`unsigned int`
`long`	`long_`	`long long`
`unsignedLong`	`unsigned_long`	`unsigned long long`
arbitrary-length integral types		
`integer`	`integer`	`long long`
`nonPositiveInteger`	`non_positive_integer`	`long long`
`nonNegativeInteger`	`non_negative_integer`	`unsigned long long`
`positiveInteger`	`positive_integer`	`unsigned long long`
`negativeInteger`	`negative_integer`	`long long`
boolean types		
`boolean`	`boolean`	`bool`
fixed-precision floating-point types		
`float`	`float_`	`float`
`double`	`double_`	`double`
arbitrary-precision floating-point types		
`decimal`	`decimal`	`double`
string types		
`string`	`string`	type derived from `std::basic_string`
`normalizedString`	`normalized_string`	type derived from `string`
`token`	`token`	type derived from `normalized_string`
`Name`	`name`	type derived from `token`
`NMTOKEN`	`nmtoken`	type derived from `token`
`NMTOKENS`	`nmtokens`	type derived from `sequence<nmtoken>`
`NCName`	`ncname`	type derived from `name`
`language`	`language`	type derived from `token`
qualified name		
`QName`	`qname`	`xml_schema::qname`
ID/IDREF types		
`ID`	`id`	type derived from `ncname`

IDREF	idref	type derived from ncname
IDREFS	idrefs	type derived from sequence<idref>
URI types		
anyURI	uri	type derived from std::basic_string
binary types		
base64Binary	base64_binary	xml_schema::base64_binary
hexBinary	hex_binary	xml_schema::hex_binary
date/time types		
date	date	xml_schema::date
dateTime	date_time	xml_schema::date_time
duration	duration	xml_schema::duration
gDay	gday	xml_schema::gday
gMonth	gmonth	xml_schema::gmonth
gMonthDay	gmonth_day	xml_schema::gmonth_day
gYear	gyear	xml_schema::gyear
gYearMonth	gyear_month	xml_schema::gyear_month
time	time	xml_schema::time
entity types		
ENTITY	entity	type derived from name
ENTITIES	entities	type derived from sequence<entity>

As you can see from the table above a number of built-in XML Schema types are mapped to fundamental C++ types such as `int` or `bool`. All string-based XML Schema types are mapped to C++ types that are derived from either `std::string` or `std::wstring`, depending on the character type selected. For access and modification purposes these types can be treated as `std::string`. A number of built-in types, such as `qname`, the binary types, and the date/time types do not have suitable fundamental or standard C++ types to map to. As a result, these types are implemented from scratch in the XSD runtime. For more information on their interfaces refer to Section 2.5, "Mapping for Built-in Data Types" in the C++/Tree Mapping User Manual.

5 Parsing

We have already seen how to parse XML to an object model in this guide before. In this chapter we will discuss the parsing topic in more detail.

By default, the C++/Tree mapping provides a total of 14 overloaded parsing functions. They differ in the input methods used to read XML as well as the error reporting mechanisms. It is also possible to generate types for root elements instead of parsing and serialization functions. This may be useful if your XML vocabulary has multiple root elements. For more information on element types refer to Section 2.9, "Mapping for Global Elements" in the C++/Tree Mapping User Manual.

In this section we will discuss the most commonly used versions of the parsing functions. For a comprehensive description of parsing refer to Chapter 3, "Parsing" in the C++/Tree Mapping User Manual. For the `people` global element from our person record vocabulary, we will concentrate on the following three parsing functions:

```
std::[auto|unique]_ptr<people_t>
people (const std::string& uri,
        xml_schema::flags f = 0,
        const xml_schema::properties& p = xml_schema::properties ());

std::[auto|unique]_ptr<people_t>
people (std::istream& is,
        xml_schema::flags f = 0,
        const xml_schema::properties& p = xml_schema::properties ());

std::[auto|unique]_ptr<people_t>
people (std::istream& is,
        const std::string& resource_id,
        xml_schema::flags f = 0,
        const xml_schema::properties& p = ::xml_schema::properties ());
```

The first function parses a local file or a URI. We have already used this parsing function in the previous chapters. The second and third functions read XML from a standard input stream. The last function also requires a resource id. This id is used to identify the XML document being parser in diagnostics messages as well as to resolve relative paths to other documents (for example, schemas) that might be referenced from the XML document.

The last two arguments to all three parsing functions are parsing flags and properties. The flags argument provides a number of ways to fine-tune the parsing process. The properties argument allows to pass additional information to the parsing functions. We will use these two arguments in Section 5.1, "XML Schema Validation and Searching" below. All three functions return the object model as either `std::auto_ptr` (C++98) or `std::unique_ptr` (C++11), depending on the C++ standard selected (`--std` XSD compiler option). The following example shows how we can use the above parsing functions:

```
using std::auto_ptr;

// Parse a local file or URI.
//
auto_ptr<people_t> p1 (people ("people.xml"));
auto_ptr<people_t> p2 (people ("http://example.com/people.xml"));

// Parse a local file via ifstream.
//
std::ifstream ifs ("people.xml");
auto_ptr<people_t> p3 (people (ifs, "people.xml"));

// Parse an XML string.
//
std::string str ("..."); // XML in a string.
std::istringstream iss (str);
auto_ptr<people_t> p4 (people (iss));
```

5.1 XML Schema Validation and Searching

The C++/Tree mapping relies on the underlying Xerces-C++ XML parser for full XML document validation. The XML Schema validation is enabled by default and can be disabled by passing the `xml_schema::flags::dont_validate` flag to the parsing functions, for example:

```
auto_ptr<people_t> p (
  people ("people.xml", xml_schema::flags::dont_validate));
```

Even when XML Schema validation is disabled, the generated code still performs a number of checks to prevent construction of an inconsistent object model (for example, an object model with missing required attributes or elements).

When XML Schema validation is enabled, the XML parser needs to locate a schema to validate against. There are several methods to provide the schema location information to the parser. The easiest and most commonly used method is to specify schema locations in the XML document itself with the `schemaLocation` or `noNamespaceSchemaLocation` attributes, for example:

```
<?xml version="1.0" ?>
<people xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="people.xsd"
        xsi:schemaLocation="http://www.w3.org/XML/1998/namespace xml.xsd">
```

As you might have noticed, we used this method in all the sample XML documents presented in this guide up until now. Note that the schema locations specified with these two attributes are relative to the document's path unless they are absolute URIs (that is start with `http://`, `file://`, etc.). In particular, if you specify just file names as your schema locations, as we did above, then the schemas should reside in the same directory as the XML document itself.

Another method of providing the schema location information is via the `xml_schema::properties` argument, as shown in the following example:

```
xml_schema::properties props;
props.no_namespace_schema_location ("people.xsd");
props.schema_location ("http://www.w3.org/XML/1998/namespace", "xml.xsd");

auto_ptr<people_t> p (people ("people.xml", 0, props));
```

The schema locations provided with this method overrides those specified in the XML document. As with the previous method, the schema locations specified this way are relative to the document's path unless they are absolute URIs. In particular, if you want to use local schemas that are not related to the document being parsed, then you will need to use the `file://` URI. The following example shows how to use schemas that reside in the current working directory:

```
#include <unistd.h> // getcwd
#include <limits.h> // PATH_MAX

char cwd[PATH_MAX];
if (getcwd (cwd, PATH_MAX) == 0)
{
  // Buffer too small?
}

xml_schema::properties props;

props.no_namespace_schema_location (
  "file:///" + std::string (cwd) + "/people.xsd");

props.schema_location (
  "http://www.w3.org/XML/1998/namespace",
  "file:///" + std::string (cwd) + "/xml.xsd");

auto_ptr<people_t> p (people ("people.xml", 0, props));
```

A third method is the most useful if you are planning to parse several XML documents of the same vocabulary. In that case it may be beneficial to pre-parse and cache the schemas in the XML parser which can then be used to parse all documents without re-parsing the schemas. For more information on this method refer to the `caching` example in the `examples/cxx/tree/` directory of the XSD distribution. It is also possible to convert the schemas into a pre-compiled binary representation and embed this representation directly into the application executable. With this approach your application can perform XML Schema validation without depending on any external schema files. For more information on how to achieve this refer to the `embedded` example in the `examples/cxx/tree/` directory of the XSD distribution.

When the XML parser cannot locate a schema for the XML document, the validation fails and XML document elements and attributes for which schema definitions could not be located are reported in the diagnostics. For example, if we remove the `noNamespaceSchemaLocation` attribute in `people.xml` from the previous chapter, then we will get the following diagnostics if we try to parse this file with validation enabled:

```
people.xml:2:63 error: no declaration found for element 'people'
people.xml:4:18 error: no declaration found for element 'person'
people.xml:4:18 error: attribute 'id' is not declared for element 'person'
people.xml:5:17 error: no declaration found for element 'first-name'
people.xml:6:18 error: no declaration found for element 'middle-name'
people.xml:7:16 error: no declaration found for element 'last-name'
people.xml:8:13 error: no declaration found for element 'gender'
people.xml:9:10 error: no declaration found for element 'age'
```

5.2 Error Handling

The parsing functions offer a number of ways to handle error conditions with the C++ exceptions being the most commonly used mechanism. All C++/Tree exceptions derive from common base `xml_schema::exception` which in turn derives from `std::exception`. The easiest way to uniformly handle all possible C++/Tree exceptions and print detailed information about the error is to catch and print `xml_schema::exception`, as shown in the following example:

```
try
{
  auto_ptr<people_t> p (people ("people.xml"));
}
catch (const xml_schema::exception& e)
{
  cerr << e << endl;
}
```

Each individual C++/Tree exception also allows you to obtain error details programmatically. For example, the `xml_schema::parsing` exception is thrown when the XML parsing and validation in the underlying XML parser fails. It encapsulates various diagnostics information such as the file name, line and column numbers, as well as the error or warning message for each entry. For more information about this and other exceptions that can be thrown during parsing, refer to Section 3.3, "Error Handling" in the C++/Tree Mapping User Manual.

Note that if you are parsing `std::istream` on which exceptions are not enabled, then you will need to check the stream state after the call to the parsing function in order to detect any possible stream failures, for example:

```
std::ifstream ifs ("people.xml");

if (ifs.fail ())
{
```

```
  cerr << "people.xml: unable to open" << endl;
  return 1;
}

auto_ptr<people_t> p (people (ifs, "people.xml"));

if (ifs.fail ())
{
  cerr << "people.xml: read error" << endl;
  return 1;
}
```

The above example can be rewritten to use exceptions as shown below:

```
try
{
  std::ifstream ifs;
  ifs.exceptions (std::ifstream::badbit | std::ifstream::failbit);
  ifs.open ("people.xml");

  auto_ptr<people_t> p (people (ifs, "people.xml"));
}
catch (const std::ifstream::failure&)
{
  cerr << "people.xml: unable to open or read error" << endl;
  return 1;
}
```

6 Serialization

We have already seen how to serialize an object model back to XML in this guide before. In this chapter we will discuss the serialization topic in more detail.

By default, the C++/Tree mapping provides a total of 8 overloaded serialization functions. They differ in the output methods used to write XML as well as the error reporting mechanisms. It is also possible to generate types for root elements instead of parsing and serialization functions. This may be useful if your XML vocabulary has multiple root elements. For more information on element types refer to Section 2.9, "Mapping for Global Elements" in the C++/Tree Mapping User Manual.

In this section we will discuss the most commonly used version of serialization functions. For a comprehensive description of serialization refer to Chapter 4, "Serialization" in the C++/Tree Mapping User Manual. For the `people` global element from our person record vocabulary, we will concentrate on the following serialization function:

```
void
people (std::ostream& os,
        const people_t& x,
        const xml_schema::namespace_infomap& map =
          xml_schema::namespace_infomap (),
        const std::string& encoding = "UTF-8",
        xml_schema::flags f = 0);
```

This function serializes the object model passed as the second argument to the standard output stream passed as the first argument. The third argument is a namespace information map which we will discuss in more detail in the next section. The fourth argument is a character encoding that the resulting XML document should be in. Possible valid values for this argument are "US-ASCII", "ISO8859-1", "UTF-8", "UTF-16BE", "UTF-16LE", "UCS-4BE", and "UCS-4LE". Finally, the flags argument allows fine-tuning of the serialization process. The following example shows how we can use the above serialization function:

```
people_t& p = ...

xml_schema::namespace_infomap map;
map[""].schema = "people.xsd";

// Serialize to stdout.
//
people (std::cout, p, map);

// Serialize to a file.
//
std::ofstream ofs ("people.xml");
people (ofs, p, map);

// Serialize to a string.
//
std::ostringstream oss;
people (oss, p, map);
std::string xml (oss.str ());
```

6.1 Namespace and Schema Information

While XML serialization can be done just from the object model alone, it is often desirable to assign meaningful prefixes to XML namespaces used in the vocabulary as well as to provide the schema location information. This is accomplished by passing the namespace information map to the serialization function. The key in this map is a namespace prefix that should be assigned to an XML namespace specified in the `name` variable of the map value. You can also assign an optional schema location for this namespace in the `schema` variable. Based on each key-value entry in this map, the serialization function adds two attributes to the resulting XML document: the namespace-prefix mapping attribute and schema location attribute. The empty prefix indicates that the namespace should be mapped without a prefix. For example, the following map:

```
xml_schema::namespace_infomap map;

map[""].name = "http://www.example.com/example";
map[""].schema = "example.xsd";

map["x"].name = "http://www.w3.org/XML/1998/namespace";
map["x"].schema = "xml.xsd";
```

Results in the following XML document:

```
<?xml version="1.0" ?>
<example
  xmlns="http://www.example.com/example"
  xmlns:x="http://www.w3.org/XML/1998/namespace"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.example.com/example example.xsd
                      http://www.w3.org/XML/1998/namespace xml.xsd">
```

The empty namespace indicates that the vocabulary has no target namespace. For example, the following map results in only the `noNamespaceSchemaLocation` attribute being added:

```
xml_schema::namespace_infomap map;

map[""].name = "";
map[""].schema = "example.xsd";
```

6.2 Error Handling

Similar to the parsing functions, the serialization functions offer a number of ways to handle error conditions with the C++ exceptions being the most commonly used mechanisms. As with parsing, the easiest way to uniformly handle all possible serialization exceptions and print detailed information about the error is to catch and print `xml_schema::exception`:

```
try
{
  people_t& p = ...

  xml_schema::namespace_infomap map;
  map[""].schema = "people.xsd";

  people (std::cout, p, map));
}
catch (const xml_schema::exception& e)
{
  cerr << e << endl;
}
```

The most commonly encountered serialization exception is `xml_schema::serialization`. It is thrown when the XML serialization in the underlying XML writer fails. It encapsulates various diagnostics information such as the file name, line and column numbers, as well as the error or warning message for each entry. For more information about this and other exceptions that can be thrown during serialization, refer to Section 4.4, "Error Handling" in the C++/Tree Mapping User Manual.

Note that if you are serializing to `std::ostream` on which exceptions are not enabled, then you will need to check the stream state after the call to the serialization function in order to detect any possible stream failures, for example:

```
std::ofstream ofs ("people.xml");

if (ofs.fail ())
{
  cerr << "people.xml: unable to open" << endl;
  return 1;
}

people (ofs, p, map));

if (ofs.fail ())
{
  cerr << "people.xml: write error" << endl;
  return 1;
}
```

The above example can be rewritten to use exceptions as shown below:

```
try
{
  std::ofstream ofs;
  ofs.exceptions (std::ofstream::badbit | std::ofstream::failbit);
  ofs.open ("people.xml");

  people (ofs, p, map));
}
catch (const std::ofstream::failure&)
{
  cerr << "people.xml: unable to open or write error" << endl;
  return 1;
}
```