C++/Parser 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++/Parser 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. 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:

- XSD Compiler Command Line Manual
- The `examples/cxx/parser/` 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++/Parser mapping. XSD is a cross-platform W3C XML Schema to C++ data binding compiler. C++/Parser is a W3C XML Schema to C++ mapping that represents an XML vocabulary as a set of parser skeletons which you can implement to perform XML processing as required by your application logic.

1.1 Mapping Overview

The C++/Parser mapping provides event-driven, stream-oriented XML parsing, XML Schema validation, and C++ data binding. It was specifically designed and optimized for high performance and small footprint. Based on the static analysis of the schemas, XSD generates compact, highly-optimized hierarchical state machines that combine data extraction, validation, and even dispatching in a single step. As a result, the generated code is typically 2-10 times faster than general-purpose validating XML parsers while maintaining the lowest static and dynamic memory footprints.

To speed up application development, the C++/Parser mapping can be instructed to generate sample parser implementations and a test driver which can then be filled with the application logic code. The mapping also provides a wide range of mechanisms for controlling and customizing the generated code.

The next chapter shows how to create a simple application that uses the C++/Parser mapping to parse, validate, and extract data from a simple XML document. The following chapters show how to use the C++/Parser 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 all information is represented as text.
- Resulting applications are hard to debug, change, and maintain.

In contrast, statically-typed, vocabulary-specific parser skeletons produced by the C++/Parser mapping allow 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++/Parser mapping has the following key advantages over generic XML access APIs:

- **Ease of use.** The generated code hides all the complexity associated with recreating the document structure, maintaining the dispatch state, and converting the data from the text representation to data types suitable for manipulation by the application logic. Parser templates also provide a convenient mechanism for building custom in-memory representations.
- **Natural representation.** The generated parser skeletons implement parser hooks as virtual functions with names corresponding to elements and attributes in XML. As a result, you process the XML data using your domain vocabulary instead of generic elements, attributes, and text.
- **Concise code.** With separate parser skeleton for each XML Schema type, the application logic implementation is simpler and thus easier to read and understand.

- **Safety.** The XML data is delivered to parser hooks as statically typed objects. The parser hooks themselves are virtual functions. 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.
- **Efficiency.** The generated parser skeletons combine data extraction, validation, and even dispatching in a single step. This makes them much more efficient than traditional architectures with separate stages for validation and data extraction/dispatch.

2 Hello World Example

In this chapter we will examine how to parse a very simple XML document using the XSD-generated C++/Parser skeletons. The code presented in this chapter is based on the `hello` example which can be found in the `examples/cxx/parser/` 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>earth</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">
    <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"/>

</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` 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. The next step is to compile this 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++ parser skeletons. To do this we invoke the XSD compiler from a terminal (UNIX) or a command prompt (Windows):

```
$ xsd cxx-parser --xml-parser expat hello.xsd
```

The `--xml-parser` option indicates that we want to use Expat as the underlying XML parser (see Section 5.2, "Underlying XML Parser"). The XSD compiler produces two C++ files: `hello-pskel.hxx` and `hello-pskel.cxx`. The following code fragment is taken from `hello-pskel.hxx`; it should give you an idea about what gets generated:

```
class hello_pskel
{
public:
  // Parser hooks. Override them in your implementation.
  //
  virtual void
  pre ();

  virtual void
  greeting (const std::string&);

  virtual void
  name (const std::string&);

  virtual void
  post_hello ();
```

```
  // Parser construction API.
  //
  void
  greeting_parser (xml_schema::string_pskel&);

  void
  name_parser (xml_schema::string_pskel&);

  void
  parsers (xml_schema::string_pskel& /* greeting */,
           xml_schema::string_pskel& /* name */);

private:
  ...
};
```

The first four member functions shown above are called parser hooks. You would normally override them in your implementation of the parser to do something useful. Let's go through all of them one by one.

The `pre()` function is an initialization hook. It is called when a new element of type `hello` is about to be parsed. You would normally use this function to allocate a new instance of the resulting type or clear accumulators that are used to gather information during parsing. The default implementation of this function does nothing.

The `post_hello()` function is a finalization hook. Its name is constructed by adding the parser skeleton name to the `post_` prefix. The finalization hook is called when parsing of the element is complete and the result, if any, should be returned. Note that in our case the return type of `post_hello()` is `void` which means there is nothing to return. More on parser return types later.

You may be wondering why the finalization hook is called `post_hello()` instead of `post()` just like `pre()`. The reason for this is that finalization hooks can have different return types and result in function signature clashes across inheritance hierarchies. To prevent this the signatures of finalization hooks are made unique by adding the type name to their names.

The `greeting()` and `name()` functions are called when the `greeting` and `name` elements have been parsed, respectively. Their arguments are of type `std::string` and contain the data extracted from XML.

The last three functions are for connecting parsers to each other. For example, there is a predefined parser for built-in XML Schema type `string` in the XSD runtime. We will be using it to parse the contents of `greeting` and `name` elements, as shown in the next section.

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. The first step is to implement the parser:

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

class hello_pimpl: hello_pskel
{
public:
  virtual void
  greeting (const std::string& g)
  {
    greeting_ = g;
  }

  virtual void
  name (const std::string& n)
  {
    std::cout << greeting_ << ", " << n << "!" << std::endl;
  }

private:
  std::string greeting_;
};
```

We left both `pre()` and `post_hello()` with the default implementations; we don’t have anything to initialize or return. The rest is pretty straightforward: we store the greeting in a member variable and later, when parsing names, use it to say hello.

An observant reader my ask what happens if the `name` element comes before `greeting`? Don’t we need to make sure `greeting_` was initialized and report an error otherwise? The answer is no, we don’t have to do any of this. The `hello_pskel` parser skeleton performs validation of XML according to the schema from which it was generated. As a result, it will check the order of the `greeting` and `name` elements and report an error if it is violated.

Now it is time to put this parser implementation to work:

```
using namespace std;

int
main (int argc, char* argv[])
{
  try
  {
    // Construct the parser.
    //
```

```
    xml_schema::string_pimpl string_p;
    hello_pimpl hello_p;

    hello_p.greeting_parser (string_p);
    hello_p.name_parser (string_p);

    // Parse the XML instance.
    //
    xml_schema::document doc_p (hello_p, "hello");

    hello_p.pre ();
    doc_p.parse (argv[1]);
    hello_p.post_hello ();
  }
  catch (const xml_schema::exception& e)
  {
    cerr << e << endl;
    return 1;
  }
}
```

The first part of this code snippet instantiates individual parsers and assembles them into a complete vocabulary parser. `xml_schema::string_pimpl` is an implementation of a parser for built-in XML Schema type `string`. It is provided by the XSD runtime along with parsers for other built-in types (for more information on the built-in parsers see Chapter 6, "Built-In XML Schema Type Parsers"). We use `string_pimpl` to parse the `greeting` and `name` elements as indicated by the calls to `greeting_parser()` and `name_parser()`.

Then we instantiate a document parser (`doc_p`). The first argument to its constructor is the parser for the root element (`hello_p` in our case). The second argument is the root element name.

The final piece is the calls to `pre()`, `parse()`, and `post_hello()`. The call to `parse()` perform the actual XML parsing while `pre()` and `post_hello()` make sure that the parser for the root element can perform proper initialization and cleanup.

While our parser implementation and test driver are pretty small and easy to write by hand, for bigger XML vocabularies it can be a substantial effort. To help with this task XSD can automatically generate sample parser implementations and a test driver from your schema. You can request generation of a sample implementation with empty function bodies by specifying the `--generate-noop-impl` option. Or you can generate a sample implementation that prints the data store in XML by using the `--generate-print-impl` option. To request generation of a test driver you can use the `--generate-test-driver` option. For more information on these options refer to the XSD Compiler Command Line Manual. The `'generated'` example in the XSD distribution shows the sample implementation generation feature in action.

2.4 Compiling and Running

After saving all the parts from the previous section in `driver.cxx`, we are ready to compile our first application 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-pskel.cxx
$ c++ -o driver driver.o hello-pskel.o -lexpat
$ ./driver hello.xml
Hello, sun!
Hello, moon!
Hello, world!
```

Here `.../libxsd` represents the path to the `libxsd` directory in the XSD distribution. We can also test the error handling. To test XML well-formedness checking, we can try to parse `hello-pskel.hxx`:

```
$ ./driver hello-pskel.hxx
hello-pskel.hxx:1:0: not well-formed (invalid token)
```

We can also try to parse a valid XML but not from our vocabulary, for example `hello.xsd`:

```
$ ./driver hello.xsd
hello.xsd:2:0: expected element ’hello’ instead of
’http://www.w3.org/2001/XMLSchema#schema’
```

3 Parser Skeletons

As we have seen in the previous chapter, the XSD compiler generates a parser skeleton class for each type defined in XML Schema. In this chapter we will take a closer look at different functions that comprise a parser skeleton as well as the way to connect our implementations of these parser skeletons to create a complete parser.

In this and subsequent chapters we will use the following more realistic XML Schema definition 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">
    <xs:restriction base="xs:string">
      <xs:enumeration value="male"/>
      <xs:enumeration value="female"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="person">
```

```
    <xs:sequence>
      <xs:element name="first-name" type="xs:string"/>
      <xs:element name="last-name" type="xs:string"/>
      <xs:element name="gender" type="gender"/>
      <xs:element name="age" type="xs:short"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="people">
    <xs:sequence>
      <xs:element name="person" type="person" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

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

</xs:schema>
```

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

```
<?xml version="1.0"?>
<people>
  <person>
    <first-name>John</first-name>
    <last-name>Doe</last-name>
    <gender>male</gender>
    <age>32</age>
  </person>
  <person>
    <first-name>Jane</first-name>
    <last-name>Doe</last-name>
    <gender>female</gender>
    <age>28</age>
  </person>
</people>
```

Compiling `people.xsd` with the XSD compiler results in three parser skeletons being generated: `gender_pskel`, `person_pskel`, and `people_pskel`. We are going to examine and implement each of them in the subsequent sections.

3.1 Implementing the Gender Parser

The generated `gender` parser skeleton looks like this:

```
class gender_pskel: public virtual xml_schema::string_pskel
{
public:
  // Parser hooks. Override them in your implementation.
  //
```

```
  virtual void
  pre ();

  virtual void
  post_gender ();
};
```

Notice that gender_pskel inherits from xml_schema::string_skel which is a parser skeleton for the built-in XML Schema type string and is predefined in the XSD runtime library. This is an example of the general rule that parser skeletons follow: if a type in XML Schema inherits from another then there will be an equivalent inheritance between the corresponding parser skeleton classes.

The pre and post_gender hooks should look familiar from the previous chapter. Let's now implement the parser. Our implementation will simply print the gender to cout:

```
class gender_pimpl: gender_pskel, xml_schema::string_pimpl
{
public:
  virtual void
  post_gender ()
  {
    std::string s = post_string ();
    cout << "gender: " << s << endl;
  }
};
```

While the code is quite short, there is a lot going on. First, notice that we are inheriting from gender_pskel *and* from xml_schema::string_pimpl. We've encountered xml_schema::string_pimpl already; it is an implementation of the xml_schema::string_pskel parser skeleton for built-in XML Schema type string.

This is another common theme in the C++/Parser programming model: reusing implementations of the base parsers in the derived ones with the C++ mixin idiom. In our case, string_pimpl will do all the dirty work of extracting the data and we can just get it at the end with the call to post_string.

In case you are curious, here is what xml_schema::string_pskel and xml_schema::string_pimpl look like:

```
namespace xml_schema
{
  class string_pskel: virtual xml_schema::simple_content
  {
  public:
    virtual std::string
    post_string () = 0;
  };
```

```
  class string_pimpl: virtual xml_schema::string_pskel
  {
  public:
    virtual void
    _pre ();

    virtual void
    _characters (const xml_schema::ro_string&);

    virtual std::string
    post_string ();

  protected:
    std::string str_;
  };
}
```

There are three new pieces in this code that we haven't seen yet. Those are the `simple_content` class and the `_pre()` and `_characters()` functions. The `simple_content` class is defined in the XSD runtime and is a base class for all parser skeletons that conform to the simple content model in XML Schema. Types with the simple content model cannot have nested elements—only text and attributes. There is also the `complex_content` class which corresponds to the complex content mode (types with nested elements, e.g., `person` from `people.xsd`).

The `_pre()` function is a parser hook. Remember we talked about `pre()` and `post_*()` hooks in the previous chapter? There are actually two more hooks with similar roles: `_pre()` and `_post ()`. As a result, each parser skeleton has four special hooks:

```
virtual void
pre ();

virtual void
_pre ();

virtual void
_post ();

virtual void
post_name ();
```

`pre()` and `_pre()` are initialization hooks. They get called in that order before a new instance of the type is about to be parsed. The difference between `pre()` and `_pre()` is conventional: `pre()` is intended to be completely overridden by a derived parser. The derived parser can also override `_pre()` but has to always call the original version. This allows you to partition initialization into customizable and required parts.

Similarly, `_post()` and `post_name()` are finalization hooks with exactly the same semantics: `post_name()` can be completely overridden by the derived parser while the original `_post()` should always be called.

The final bit we need to discuss in this section is the `_characters()` function. As you might have guessed, it is also a hook. A low-level one that delivers raw character content for the type being parsed. You will seldom need to use this hook directly. Using implementations for built-in parsers provided by the XSD runtime is usually a simpler and more convenient alternative.

At this point you might be wondering why some `post_*` hooks, for example `post_string`, return some data while others, for example `post_gender`, have `void` as a return type. This is a valid concern and it will be addressed in the next chapter.

3.2 Implementing the Person Parser

The generated `person_pskel` parser skeleton looks like this:

```
class person_pskel: public virtual xml_schema::complex_content
{
public:
  // Parser hooks. Override them in your implementation.
  //
  virtual void
  pre ();

  virtual void
  first_name (const std::string&);

  virtual void
  last_name (const std::string&);

  virtual void
  gender ();

  virtual void
  age (short);

  virtual void
  post_person ();

  // Parser construction API.
  //
  void
  first_name_parser (xml_schema::string_pskel&);

  void
  last_name_parser (xml_schema::string_pskel&);
```

```
  void
  gender_parser (gender_pskel&);

  void
  age_parser (xml_schema::short_pskel&);

  void
  parsers (xml_schema::string_pskel& /* first-name */,
           xml_schema::string_pskel& /* last-name */,
           gender_pskel&             /* gender */,
           xml_schema::short_pskel&  /* age */);
};
```

As you can see, we have a parser hook for each of the nested elements found in the `person` XML Schema type. The implementation of this parser is straightforward:

```
class person_pimpl: person_pskel
{
public:
  virtual void
  first_name (const std::string& n)
  {
    cout << "first: " << f << endl;
  }

  virtual void
  last_name (const std::string& l)
  {
    cout << "last: " << l << endl;
  }

  virtual void
  age (short a)
  {
    cout << "age: " << a << endl;
  }
};
```

Notice that we didn’t override the `gender()` hook because all the printing is done by `gender_pimpl`.

3.3 Implementing the People Parser

The generated `people_pskel` parser skeleton looks like this:

```
class people_pskel: public virtual xml_schema::complex_content
{
public:
  // Parser hooks. Override them in your implementation.
```

```
  //
  virtual void
  pre ();

  virtual void
  person ();

  virtual void
  post_people ();

  // Parser construction API.
  //
  void
  person_parser (person_pskel&);

  void
  parsers (person_pskel& /* person */);
};
```

The `person` hook will be called after parsing each `person` element. While `person_pimpl` does all the printing, one useful thing we can do in this hook is to print an extra newline after each person record so that our output is more readable:

```
class people_pimpl: people_pskel
{
public:
  virtual void
  person ()
  {
    cout << endl;
  }
};
```

Now it is time to put everything together.

3.4 Connecting the Parsers Together

At this point we have all the individual parsers implemented and can proceed to assembling them into a complete parser for our XML vocabulary. The first step is to instantiate all the individual parsers that we will need:

```
xml_schema::short_pimpl short_p;
xml_schema::string_pimpl string_p;

gender_pimpl gender_p;
person_pimpl person_p;
people_pimpl people_p;
```

Notice that our schema uses two built-in XML Schema types: `string` for the `first-name` and `last-name` elements as well as `short` for `age`. We will use predefined parsers that come with the XSD runtime to handle these types. The next step is to connect all the individual parsers. We do this with the help of functions defined in the parser skeletons and marked with the "Parser Construction API" comment. One way to do it is to connect each individual parser by calling the `*_parser()` functions:

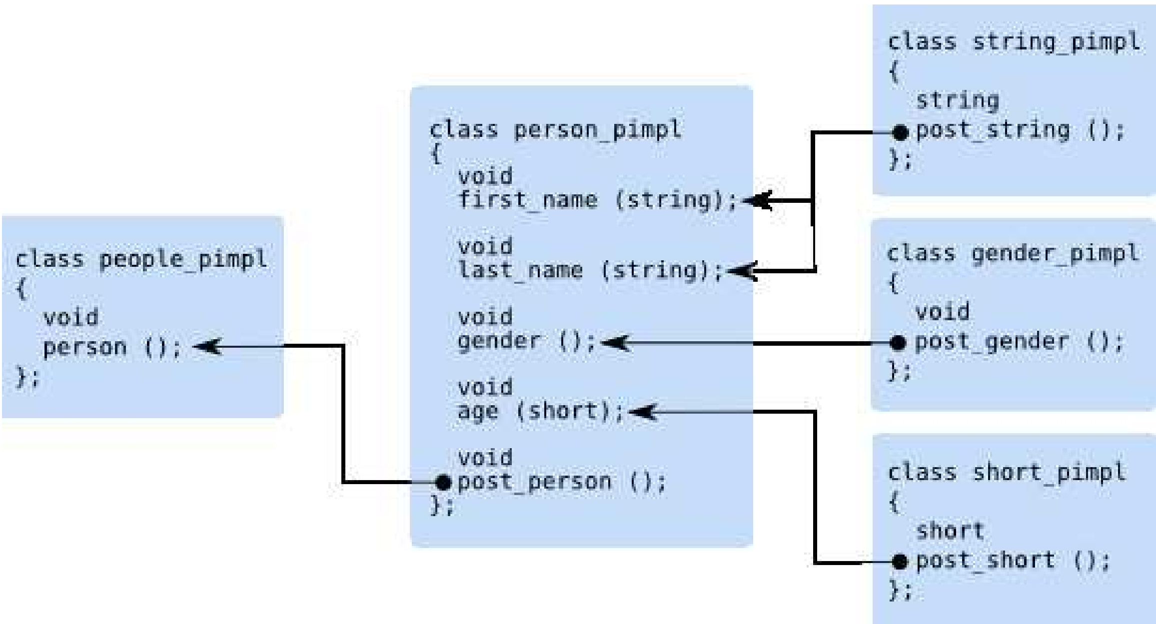
```
person_p.first_name_parsers (string_p);
person_p.last_name_parsers (string_p);
person_p.gender (gender_p);
person_p.age (short_p);

people_p.person_parser (person_p);
```

You might be wondering what happens if you do not provide a parser by not calling one of the `*_parser()` functions. In that case the corresponding XML content will be skipped, including validation. This is an efficient way to ignore parts of the document that you are not interested in.

An alternative, shorter, way to connect the parsers is by using the `parsers()` functions which connects all the parsers for a given type at once:

```
person_p.parsers (string_p, string_p, gender_p, short_p);
people_p.parsers (person_p);
```

The following figure illustrates the resulting connections.



The last step is the invocation of the parser on our sample XML instance:

```
xml_schema::document doc_p (people_p, "people");

people_p.pre ();
doc_p.parse ("people.xml");
people_p.post_people ();
```

Let's consider `xml_schema::document` in more detail. While the exact definition of this class varies depending on the underlying parser selected, here is the common part:

```
namespace xml_schema
{
  class document
  {
  public:
    document (xml_schema::parser_base&,
              const std::string& root_element_name);

    document (xml_schema::parser_base&,
              const std::string& root_element_namespace,
              const std::string& root_element_name);

    void
    parse (const std::string& file);

    void
    parse (std::istream&);

    ...

  };
}
```

`xml_schema::document` is a root parser for the vocabulary. The first argument to its constructors is the parser for the type of the root element (`people_impl` in our case). Because a type parser is only concerned with the element's content and not with the element's name, we need to specify the root element name somewhere. That's what is passed as the second and third arguments to the `document`'s constructors.

There are also two overloaded `parse()` function defined in the `document` class (there are actually more but other are specific to the underlying XML parser). The first version parses a local file identified by a name. The second version reads the data from an input stream. For more information on the `xml_schema::document` class refer to Chapter 7, "Document Object and Error Handling".

Let’s now consider a step-by-step list of actions that happen as we parse through people.xml. The content of people.xml is repeated below for convenience.

```
<?xml version="1.0"?>
<people>
  <person>
    <first-name>John</first-name>
    <last-name>Doe</last-name>
    <gender>male</gender>
    <age>32</age>
  </person>
  <person>
    <first-name>Jane</first-name>
    <last-name>Doe</last-name>
    <gender>female</gender>
    <age>28</age>
  </person>
</people>
```

1. people_p.pre() is called from main(). We did not provide any implementation for this hook so this call is a no-op.
2. doc_p.parse("people.xml") is called from main(). The parser opens the file and starts parsing its content.
3. The parser encounters the root element. doc_p verifies that the root element is correct and calls _pre() on people_p which is also a no-op. Parsing is now delegated to people_p.
4. The parser encounters the person element. people_p determines that person_p is responsible for parsing this element. pre() and _pre() hooks are called on person_p. Parsing is now delegated to person_p.
5. The parser encounters the first-name element. person_p determines that string_p is responsible for parsing this element. pre() and _pre() hooks are called on string_p. Parsing is now delegated to string_p.
6. The parser encounters character content consisting of "John". The _characters hook is called on string_p.
7. The parser encounters the end of first-name element. The _post() and post_string() hooks are called on string_p. The first_name() hook is called on person_p with the return value of post_string(). The first_name() implementation prints "first: John" to cout. Parsing is now returned to person_p.
8. Steps analogous to 5-7 are performed for the last-name, gender, and age elements.
9. The parser encounters the end of person element. The _post() and post_person() hooks are called on person_p. The person() hook is called on people_p. The person() implementation prints a new line to cout. Parsing is now returned to people_p.
10. Steps 4-9 are performed for the second person element.
11. The parser encounters the end of people element. The _post() hook is called on

`people_p`. The `doc_p.parse("people.xml")` call returns to `main()`.

12. `people_p.post_people()` is called from `main()` which is a no-op.

4 Type Maps

There are many useful things you can do inside parser hooks as they are right now. There are, however, times when you want to propagate some information from one parser to another or to the caller of the parser. One common task that would greatly benefit from such a possibility is building a tree-like in-memory object model of the data stored in XML. During execution, each individual sub-parser would create a sub-tree and return it to its *parent* parser which can then incorporate this sub-tree into the whole tree.

In this chapter we will discuss the mechanisms offered by the C++/Parser mapping for returning information from individual parsers and see how to use them to build an object model of our people vocabulary.

4.1 Object Model

An object model for our person record example could look like this (saved in the `people.hxx` file):

```
#include <string>
#include <vector>

enum gender
{
  male,
  female
};

class person
{
public:
  person (const std::string& first,
          const std::string& last,
          ::gender gender,
          short age)
    : first_ (first), last_ (last),
      gender_ (gender), age_ (age)
  {
  }

  const std::string&
  first () const
  {
    return first_;
  }
```

```
  const std::string&
  last () const
  {
    return last_;
  }

  ::gender
  gender () const
  {
    return gender_;
  }

  short
  age () const
  {
    return age_;
  }

private:
  std::string first_;
  std::string last_;
  ::gender gender_;
  short age_;
};

typedef std::vector<person> people;
```

While it is clear which parser is responsible for which part of the object model, it is not exactly clear how, for example, `gender_pimpl` will deliver `gender` to `person_pimpl`. You might have noticed that `string_pimpl` manages to deliver its value to the `first_name` hook of `person_pimpl`. Let's see how we can utilize the same mechanism to propagate our own data.

There is a way to tell the XSD compiler that you want to exchange data between parsers. More precisely, for each type defined in XML Schema, you can tell the compiler two things. First, the return type of the `post_*` hook in the parser skeleton generated for this type. And, second, the argument type for hooks corresponding to elements and attributes of this type. For example, for XML Schema type `gender` we can specify the return type for `post_gender` in the `gender_pskel` skeleton and an argument type for the `gender` hook in the `person_pskel` skeleton. As you might have guessed, the generated code will then pass the return value from the `post_*` hook as an argument to the element or attribute hook.

The way to tell the XSD compiler about these XML Schema to C++ mappings is with type map files. Here is a simple type map for the gender example from the previous paragraph:

```
include "people.hxx";
gender ::gender ::gender;
```

The first line indicates that the generated code must include `parser.hxx` in order to get the definition for the `gender` type. The second line specifies that both argument and return types for the `gender` XML Schema type should be the `::gender` C++ enum. The next section will describe the type map format in detail. We save this type map in `people.map` and then translate our schemas with the `--type-map` option to let the XSD compiler know about our type map:

```
$ xsd cxx-parser --type-map people.map people.xsd
```

If we now look at the generated `people-pskel.hxx`, we will see the following changes in the `gender_pskel` and `person_pskel` skeletons:

```
#include "people.hxx"

class gender_pskel: public virtual xml_schema::string_pskel
{
  virtual ::gender
  post_gender () = 0;

  ...
};

class person_pskel: public virtual xml_schema::complex_content
{
  virtual void
  gender (::gender);

  ...
};
```

Notice that `#include "people.hxx"` was added to the generated header file from the type map to provide the definition for the `gender` enum.

4.2 Type Map File Format

Type map files are used to define a mapping between XML Schema and C++ types. The compiler uses this information to determine the return types of `post_*` hooks in parser skeletons corresponding to XML Schema types as well as argument types for hooks corresponding to elements and attributes of these types.

The compiler has a set of built-in mapping rules that map built-in XML Schema types to suitable C++ types (discussed below) and all other types to `void`. By providing type maps you can override these built-in rules. The format of the type map file is presented below:

```
namespace <schema-namespace> [<cxx-namespace>]
{
  (include <file-name>;)*
  ([type] <schema-type> <cxx-ret-type> [<cxx-arg-type>];)*
}
```

Both *<schema-namespace>* and *<schema-type>* are regex patterns while *<cxx-namespace>*, *<cxx-ret-type>*, and *<cxx-arg-type>* are regex pattern substitutions. All names can be optionally enclosed in " ", for example, to include white-spaces.

<schema-namespace> determines XML Schema namespace. Optional *<cxx-namespace>* is prefixed to every C++ type name in this namespace declaration. *<cxx-ret-type>* is a C++ type name that is used as a return type for the post_* hook. Optional *<cxx-arg-type>* is an argument type for hooks corresponding to elements and attributes of this type. If not specified, it defaults to const *<cxx-ret-type>*&. *<file-name>* is a file name either in the " " or < > format and is added with the #include directive to the generated code. For example:

```
namespace http://www.example.com/xmlns/my my
{
  include "my.hxx";

  apple apple;
  orange orange_t* orange_t*;
}
```

In the example above, for the http://www.example.com/xmlns/my#orange XML Schema type, the my::orange_t* C++ type will be used as both return and argument types.

Several namespace declarations can be specified in a single file. The namespace declaration can also be completely omitted to map types in a schema without a namespace. For instance:

```
include "my.hxx";
apple apple;

namespace http://www.example.com/xmlns/my
{
  orange "const orange_t*" "const orange_t*";
}
```

The compiler has a number of built-in mapping rules that can be presented as the following map files. The string-based XML Schema built-in types are mapped to either **std::string** or **std::wstring** depending on the character type selected (see Section 5.1, "Character Type" for more information).

```
namespace http://www.w3.org/2001/XMLSchema
{
  boolean bool bool;
```

```
byte "signed char" "signed char";
unsignedByte "unsigned char" "unsigned char";

short short short;
unsignedShort "unsigned short" "unsigned short";

int int int;
unsignedInt "unsigned int" "unsigned int";

long "long long" "long long";
unsignedLong "unsigned long long" "unsigned long long";

integer "long long" "long long";

negativeInteger "long long" "long long";
nonPositiveInteger "long long" "long long";

positiveInteger "unsigned long long" "unsigned long long";
nonNegativeInteger "unsigned long long" "unsigned long long";

float float float;
double double double;
decimal double double;

string std::string;
normalizedString std::string;
token std::string;
Name std::string;
NMTOKEN std::string;
NCName std::string;
ID std::string;
IDREF std::string;
language std::string;
anyURI std::string;

NMTOKENS xml_schema::string_sequence;
IDREFS xml_schema::string_sequence;

QName xml_schema::qname;

base64Binary std::auto_ptr<xml_schema::buffer>
             std::auto_ptr<xml_schema::buffer>;
hexBinary std::auto_ptr<xml_schema::buffer>
          std::auto_ptr<xml_schema::buffer>;

date xml_schema::date;
dateTime xml_schema::date_time;
duration xml_schema::duration;
gDay xml_schema::gday;
gMonth xml_schema::gmonth;
gMonthDay xml_schema::gmonth_day;
```

```
  gYear xml_schema::gyear;
  gYearMonth xml_schema::gyear_month;
  time xml_schema::time;
}
```

For more information about the mapping of built-in XML Schema types to C++ types refer to Chapter 6, "Built-In XML Schema Type Parsers". The last predefined rule maps anything that wasn't mapped by previous rules to `void`:

```
namespace .*
{
  .* void void;
}
```

When you provide your own type maps with the `--type-map`, they are evaluated before any of the built-in rules. This allows you to selectively override any of the built-in rules. Note also that if you change the mapping of a built-in XML Schema type then it becomes your responsibility to provide the corresponding parser skeleton and implementation in the `xml_schema` namespace. You can include the custom definitions into the generated header file using the `--hxx-prologue-*` options.

4.3 Parser Implementations

With the knowledge from the previous section, we can proceed with creating a type map that maps types in the `people.xsd` schema to our object model classes in `people.hxx`. In fact, we already have the beginning of our type map file in `people.map`. Let's extend it with the rest of the types:

```
include "people.hxx";

gender ::gender ::gender;
person ::person;
people ::people;
```

A few things to note about this type map. We did not provide the argument types for `person` and `people` because the default constant reference is exactly what we need. We also did not provide any mappings for the built-in XML Schema types `string` and `short` because they are handled by the built-in rules and we are happy with the result. Note also that all C++ types are fully qualified. This is done to avoid potential name conflicts in the generated code. Now we can recompile our schema and move on to implementing the parsers:

```
$ xsd cxx-parser --xml-parser expat --type-map people.map people.xsd
```

Here is the implementation of our three parsers in full. One way to save typing when implementing your own parsers is to open the generated code and copy the signatures of parser hooks into your code. Or you could always auto generate the sample implementations and fill them with your code.

```
#include "people-pskel.hxx"

class gender_pimpl: gender_pskel, xml_schema::string_pimpl
{
public:
  virtual ::gender
  post_gender ()
  {
    return post_string () == "male" ? male : female;
  }
};

class person_pimpl: person_pskel
{
public:
  virtual void
  first_name (const std::string& f)
  {
    first_ = f;
  }

  virtual void
  last_name (const std::string& l)
  {
    last_ = l;
  }

  virtual void
  gender (::gender g)
  {
    gender_ = g;
  }

  virtual void
  age (short a)
  {
    age_ = a;
  }

  virtual ::person
  post_person ()
  {
    return ::person (first_, last_, gender_, age_);
  }

private:
  std::string first_;
  std::string last_;
  ::gender gender_;
```

```
  short age_;
};

class people_pimpl: people_pskel
{
public:
  virtual void
  person (const ::person& p)
  {
    people_.push_back (p);
  }

  virtual ::people
  post_people ()
  {
    ::people r;
    r.swap (people_);
    return r;
  }

private:
  ::people people_;
};
```

This code fragment should look familiar by now. Just note that all the `post_*` hooks now have return types instead of `void`. Here is the implementation of the test driver for this example:

```
#include <iostream>

using namespace std;

int
main (int argc, char* argv[])
{
  // Construct the parser.
  //
  xml_schema::short_pimpl short_p;
  xml_schema::string_pimpl string_p;

  gender_pimpl gender_p;
  person_pimpl person_p;
  people_pimpl people_p;

  person_p.parsers (string_p, string_p, gender_p, short_p);
  people_p.parsers (person_p);

  // Parse the document to obtain the object model.
  //
  xml_schema::document doc_p (people_p, "people");

  people_p.pre ();
```

```
  doc_p.parse (argv[1]);
  people ppl = people_p.post_people ();

  // Print the object model.
  //
  for (people::iterator i (ppl.begin ()); i != ppl.end (); ++i)
  {
    cout << "first:  " << i->first () << endl
         << "last:   " << i->last () << endl
         << "gender: " << (i->gender () == male ? "male" : "female") << endl
         << "age:    " << i->age () << endl
         << endl;
  }
}
```

The parser creation and assembly part is exactly the same as in the previous chapter. The parsing part is a bit different: `post_people` now has a return value which is the complete object model. We store it in the `ppl` variable. The last bit of the code simply iterates over the `people` vector and prints the information for each person. We save the last two code fragments to `driver.cxx` and proceed to compile and test our new application:

```
$ c++ -I.../libxsd -c driver.cxx people-pskel.cxx
$ c++ -o driver driver.o people-pskel.o -lexpat
$ ./driver people.xml
first:  John
last:   Doe
gender: male
age:    32

first:  Jane
last:   Doe
gender: female
age:    28
```

5 Mapping Configuration

The C++/Parser 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 and include the character type that is used by the generated code, the underlying XML parser, and whether the XML Schema validation is performed in the generated code. This chapter describes these three configuration parameters in more detail. For more ways to configure the generated code refer to the XSD Compiler Command Line Manual.

5.1 Character Type

The C++/Parser 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 string-based built-in XML Schema types are returned as either `std::string` or `std::wstring` depending on the character type selected.

Another aspect of the mapping that depends on the character type is character encoding. For the `char` character type the encoding is UTF-8. 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 (e.g., 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.

5.2 Underlying XML Parser

The C++/Parser mapping can be used with either Xerces-C++ or Expat as the underlying XML parser. You can select the XML parser with the `--xml-parser` command line option. Valid values for this option are `xerces` and `expat`. The default XML parser is Xerces-C++.

The generated code is identical for both parsers except for the `xml_schema::document` object in which some of the `parse()` functions are parser-specific as described in Chapter 7, "Document Object and Error Handling".

5.3 XML Schema Validation

The C++/Parser mapping provides support for validating a commonly-used subset of W3C XML Schema in the generated code. For the list of supported XML Schema constructs refer to Appendix A, "Supported XML Schema Constructs".

By default validation in the generated code is disabled if the underlying XML parser is validating (Xerces-C++) and enabled otherwise (Expat). See Section 5.2, "Underlying XML Parser" for more information about the underlying XML parser. You can override the default behavior with the `--generate-validation` and `--suppress-validation` command line options.

6 Built-In XML Schema Type Parsers

The XSD runtime provides parser implementations for all built-in XML Schema types as summarized in the following table. Note that some parsers return either `std::string` or `std::wstring` depending on the character type selected.

XML Schema type	Parser implementation in the `xml_schema` namespace	Parser return type
anyType and anySimpleType types		
`anyType`	`any_type_pimpl`	`void`
`anySimpleType`	`any_simple_type_pimpl`	`void`
fixed-length integral types		
`byte`	`byte_pimpl`	`signed char`
`unsignedByte`	`unsigned_byte_pimpl`	`unsigned char`
`short`	`short_pimpl`	`short`
`unsignedShort`	`unsigned_short_pimpl`	`unsigned short`
`int`	`int_pimpl`	`int`
`unsignedInt`	`unsigned_int_pimpl`	`unsigned int`
`long`	`long_pimpl`	`long long`
`unsignedLong`	`unsigned_long_pimpl`	`unsigned long long`
arbitrary-length integral types		
`integer`	`integer_pimpl`	`long long`
`nonPositiveInteger`	`non_positive_integer_pimpl`	`long long`
`nonNegativeInteger`	`non_negative_integer_pimpl`	`unsigned long long`
`positiveInteger`	`positive_integer_pimpl`	`unsigned long long`
`negativeInteger`	`negative_integer_pimpl`	`long long`
boolean types		
`boolean`	`boolean_pimpl`	`bool`
fixed-precision floating-point types		
`float`	`float_pimpl`	`float`
`double`	`double_pimpl`	`double`
arbitrary-precision floating-point types		
`decimal`	`decimal_pimpl`	`double`
string-based types		
`string`	`string_pimpl`	`std::string` or `std::wstring`
`normalizedString`	`normalized_string_pimpl`	`std::string` or `std::wstring`
`token`	`token_pimpl`	`std::string` or `std::wstring`
`Name`	`name_pimpl`	`std::string` or `std::wstring`
`NMTOKEN`	`nmtoken_pimpl`	`std::string` or `std::wstring`

NCName	ncname_pimpl	std::string or std::wstring
language	language_pimpl	std::string or std::wstring
qualified name		
QName	qname_pimpl	xml_schema::qname Section 6.1, "QName Parser"
ID/IDREF types		
ID	id_pimpl	std::string or std::wstring
IDREF	idref_pimpl	std::string or std::wstring
list types		
NMTOKENS	nmtokens_pimpl	xml_schema::string_sequence Section 6.2, "NMTOKENS and IDREFS Parsers"
IDREFS	idrefs_pimpl	xml_schema::string_sequence Section 6.2, "NMTOKENS and IDREFS Parsers"
URI types		
anyURI	uri_pimpl	std::string or std::wstring
binary types		
base64Binary	base64_binary_pimpl	std::auto_ptr<xml_schema::buffer> Section 6.3, "base64Binary and hexBinary Parsers"
hexBinary	hex_binary_pimpl	std::auto_ptr<xml_schema::buffer> Section 6.3, "base64Binary and hexBinary Parsers"
date/time types		
date	date_pimpl	xml_schema::date Section 6.4, "date Parser"
dateTime	date_time_pimpl	xml_schema::date_time Section 6.5, "dateTime Parser"
duration	duration_pimpl	xml_schema::duration Section 6.6, "duration Parser"
gDay	gday_pimpl	xml_schema::gday Section 6.7, "gDay Parser"
gMonth	gmonth_pimpl	xml_schema::gmonth Section 6.8, "gMonth Parser"
gMonthDay	gmonth_day_pimpl	xml_schema::gmonth_day Section 6.9, "gMonthDay Parser"
gYear	gyear_pimpl	xml_schema::gyear Section 6.10, "gYear Parser"

gYearMonth	gyear_month_pimpl	xml_schema::gyear_month Section 6.11, "gYearMonth Parser"
time	time_pimpl	xml_schema::time Section 6.12, "time Parser"

6.1 QName Parser

The return type of the qname_pimpl parser implementation is xml_schema::qname which represents an XML qualified name. Its interface is presented below. Note that the std::string type in the interface becomes std::wstring if the selected character type is wchar_t.

```
namespace xml_schema
{
  class qname
  {
  public:
    explicit
    qname (const std::string& name);
    qname (const std::string& prefix, const std::string& name);

    const std::string&
    prefix () const;

    void
    prefix (const std::string&);

    const std::string&
    name () const;

    void
    name (const std::string&);
  };

  bool
  operator== (const qname&, const qname&);

  bool
  operator!= (const qname&, const qname&);
}
```

6.2 NMTOKENS and IDREFS Parsers

The return type of the nmtokens_pimpl and idrefs_pimpl parser implementations is xml_schema::string_sequence which represents a sequence of strings. Its interface is presented below. Note that the std::string type in the interface becomes std::wstring if the selected character type is wchar_t.

```
namespace xml_schema
{
  class string_sequence: public std::vector<std::string>
  {
  public:
    string_sequence ();

    explicit
    string_sequence (std::vector<std::string>::size_type n,
                     const std::string& x = std::string ());

    template <typename I>
    string_sequence (const I& begin, const I& end);
  };

  bool
  operator== (const string_sequence&, const string_sequence&);

  bool
  operator!= (const string_sequence&, const string_sequence&);
}
```

6.3 `base64Binary` and `hexBinary` Parsers

The return type of the `base64_binary_pimpl` and `hex_binary_pimpl` parser implementations is `std::auto_ptr<xml_schema::buffer>`. The `xml_schema::buffer` type represents a binary buffer and its interface is presented below.

```
namespace xml_schema
{
  class buffer
  {
  public:
    typedef std::size_t size_t;

    class bounds {}; // Out of bounds exception.

  public:
    explicit
    buffer (size_t size = 0);
    buffer (size_t size, size_t capacity);
    buffer (const void* data, size_t size);
    buffer (const void* data, size_t size, size_t capacity);
    buffer (void* data,
            size_t size,
            size_t capacity,
            bool assume_ownership);

  public:
    buffer (const buffer&);
```

```
    buffer&
    operator= (const buffer&);

    void
    swap (buffer&);

  public:
    size_t
    capacity () const;

    bool
    capacity (size_t);

  public:
    size_t
    size () const;

    bool
    size (size_t);

  public:
    const char*
    data () const;

    char*
    data ();

    const char*
    begin () const;

    char*
    begin ();

    const char*
    end () const;

    char*
    end ();
  };

  bool
  operator== (const buffer&, const buffer&);

  bool
  operator!= (const buffer&, const buffer&);
}
```

If the assume_ownership argument to the constructor is true, the instance assumes the ownership of the memory block pointed to by the data argument and will eventually release it by calling operator delete(). The capacity() and size() modifier functions return true if the underlying buffer has moved.

The bounds exception is thrown if the constructor arguments violate the (size <= capacity) constraint.

6.4 date Parser

The return type of the date_pimpl parser implementation is xml_schema::date which represents year, day, and month with an optional time zone. Its interface is presented below. Note that the std::string type in the interface becomes std::wstring if the selected character type is wchar_t. The unspecified time zone is indicated by an empty string.

```
namespace xml_schema
{
  class date
  {
  public:
    date (int year, unsigned short month, unsigned short day);
    date (int year, unsigned short month, unsigned short day,
          const std::string& zone);

    int
    year () const;

    void
    year (int);

    unsigned short
    month () const;

    void
    month (unsigned short);

    unsigned short
    day () const;

    void
    day (unsigned short);

    const std::string&
    zone () const;

    void
    zone (const std::string&);
  };
```

```
  bool
  operator== (const date&, const date&);

  bool
  operator!= (const date&, const date&);
}
```

6.5 `dateTime` Parser

The return type of the `date_time_pimpl` parser implementation is `xml_schema::date_time` which represents year, month, day, hours, minutes, and seconds with an optional time zone. Its interface is presented below. Note that the `std::string` type in the interface becomes `std::wstring` if the selected character type is `wchar_t`. The unspecified time zone is indicated by an empty string.

```
namespace xml_schema
{
  class date_time
  {
  public:
    date_time (int year, unsigned short month, unsigned short day,
               unsigned short hours, unsigned short minutes,
               double seconds);

    date_time (int year, unsigned short month, unsigned short day,
               unsigned short hours, unsigned short minutes,
               double seconds, const std::string& zone);

    int
    year () const;

    void
    year (int);

    unsigned short
    month () const;

    void
    month (unsigned short);

    unsigned short
    day () const;

    void
    day (unsigned short);

    unsigned short
    hours () const;
```

```
    void
    hours (unsigned short);

    unsigned short
    minutes () const;

    void
    minutes (unsigned short);

    double
    seconds () const;

    void
    seconds (double);

    const std::string&
    zone () const;

    void
    zone (const std::string&);
  };

  bool
  operator== (const date_time&, const date_time&);

  bool
  operator!= (const date_time&, const date_time&);
}
```

6.6 `duration` Parser

The return type of the `duration_pimpl` parser implementation is `xml_schema::duration` which represents a potentially negative duration in the form of years, months, days, hours, minutes, and seconds. Its interface is presented below.

```
namespace xml_schema
{
  class duration
  {
  public:
    duration (bool negative,
              unsigned int years, unsigned int months, unsigned int days,
              unsigned int hours, unsigned int minutes, double seconds);

    bool
    negative () const;

    void
    negative (bool);
```

```
    unsigned int
    years () const;

    void
    years (unsigned int);

    unsigned int
    months () const;

    void
    months (unsigned int);

    unsigned int
    days () const;

    void
    days (unsigned int);

    unsigned int
    hours () const;

    void
    hours (unsigned int);

    unsigned int
    minutes () const;

    void
    minutes (unsigned int);

    double
    seconds () const;

    void
    seconds (double);
  };

  bool
  operator== (const duration&, const duration&);

  bool
  operator!= (const duration&, const duration&);
}
```

6.7 gDay Parser

The return type of the gday_pimpl parser implementation is xml_schema::gday which represents a day of the month with an optional time zone. Its interface is presented below. Note that the std::string type in the interface becomes std::wstring if the selected character type is wchar_t. The unspecified time zone is indicated by an empty string.

```
namespace xml_schema
{
  class gday
  {
  public:
    explicit
    gday (unsigned short day);
    gday (unsigned short day, const std::string& zone);

    unsigned short
    day () const;

    void
    day (unsigned short);

    const std::string&
    zone () const;

    void
    zone (const std::string&);
  };

  bool
  operator== (const gday&, const gday&);

  bool
  operator!= (const gday&, const gday&);
}
```

6.8 gMonth Parser

The return type of the gmonth_pimpl parser implementation is xml_schema::gmonth which represents a month of the year with an optional time zone. Its interface is presented below. Note that the std::string type in the interface becomes std::wstring if the selected character type is wchar_t. The unspecified time zone is indicated by an empty string.

```
namespace xml_schema
{
  class gmonth
  {
  public:
```

```
    explicit
    gmonth (unsigned short month);
    gmonth (unsigned short month, const std::string& zone);

    unsigned short
    month () const;

    void
    month (unsigned short);

    const std::string&
    zone () const;

    void
    zone (const std::string&);
  };

  bool
  operator== (const gmonth&, const gmonth&);

  bool
  operator!= (const gmonth&, const gmonth&);
}
```

6.9 `gMonthDay` Parser

The return type of the `gmonth_day_pimpl` parser implementation is `xml_schema::gmonth_day` which represents day and month of the year with an optional time zone. Its interface is presented below. Note that the `std::string` type in the interface becomes `std::wstring` if the selected character type is `wchar_t`. The unspecified time zone is indicated by an empty string.

```
namespace xml_schema
{
  class gmonth_day
  {
  public:
    gmonth_day (unsigned short month, unsigned short day);
    gmonth_day (unsigned short month, unsigned short day,
                const std::string& zone);

    unsigned short
    month () const;

    void
    month (unsigned short);

    unsigned short
    day () const;
```

```
    void
    day (unsigned short);

    const std::string&
    zone () const;

    void
    zone (const std::string&);
  };

  bool
  operator== (const gmonth_day&, const gmonth_day&);

  bool
  operator!= (const gmonth_day&, const gmonth_day&);
}
```

6.10 gYear Parser

The return type of the `gyear_pimpl` parser implementation is `xml_schema::gyear` which represents a year with an optional time zone. Its interface is presented below. Note that the `std::string` type in the interface becomes `std::wstring` if the selected character type is `wchar_t`. The unspecified time zone is indicated by an empty string.

```
namespace xml_schema
{
  class gyear
  {
  public:
    explicit
    gyear (int year);
    gyear (int year, const std::string& zone);

    int
    year () const;

    void
    year (int);

    const std::string&
    zone () const;

    void
    zone (const std::string&);
  };

  bool
  operator== (const gyear&, const gyear&);
```

```
  bool
  operator!= (const gyear&, const gyear&);
}
```

6.11 gYearMonth Parser

The return type of the `gyear_month_pimpl` parser implementation is `xml_schema::gyear_month` which represents year and month with an optional time zone. Its interface is presented below. Note that the `std::string` type in the interface becomes `std::wstring` if the selected character type is `wchar_t`. The unspecified time zone is indicated by an empty string.

```
namespace xml_schema
{
  class gyear_month
  {
  public:
    gyear_month (int year, unsigned short month);
    gyear_month (int year, unsigned short month,
                 const std::string& zone);

    int
    year () const;

    void
    year (int);

    unsigned short
    month () const;

    void
    month (unsigned short);

    const std::string&
    zone () const;

    void
    zone (const std::string&);
  };

  bool
  operator== (const gyear_month&, const gyear_month&);

  bool
  operator!= (const gyear_month&, const gyear_month&);
}
```

6.12 `time` Parser

The return type of the `time_pimpl` parser implementation is `xml_schema::time` which represents hours, minutes, and seconds with an optional time zone. Its interface is presented below. Note that the `std::string` type in the interface becomes `std::wstring` if the selected character type is `wchar_t`. The unspecified time zone is indicated by an empty string.

```
namespace xml_schema
{
  class time
  {
  public:
    time (unsigned short hours, unsigned short minutes, double seconds);
    time (unsigned short hours, unsigned short minutes, double seconds,
          const std::string& zone);

    unsigned short
    hours () const;

    void
    hours (unsigned short);

    unsigned short
    minutes () const;

    void
    minutes (unsigned short);

    double
    seconds () const;

    void
    seconds (double);

    const std::string&
    zone () const;

    void
    zone (const std::string&);
  };

  bool
  operator== (const time&, const time&);

  bool
  operator!= (const time&, const time&);
}
```

7 Document Object and Error Handling

In this chapter we will discuss the xml_schema::document type as well as the error handling mechanisms provided by the mapping in more detail. As mentioned in Section 3.4, "Connecting the Parsers Together", the interface of xml_schema::document depends on the underlying XML parser selected (Section 5.2, "Underlying XML Parser"). The following sections describe the document type interface for Xerces-C++ and Expat as underlying parsers.

7.1 Xerces-C++ Document Object

When Xerces-C++ is used as the underlying XML parser, the document type has the following interface. Note that if the character type is wchar_t, then the string type in the interface becomes std::wstring (see Section 5.1, "Character Type").

```
namespace xml_schema
{
  class parser_base;
  class error_handler;

  struct flags
  {
    // Do not validate XML documents with the Xerces-C++ validator.
    //
    static const unsigned long dont_validate;

    // Do not initialize the Xerces-C++ runtime.
    //
    static const unsigned long dont_initialize;
  };

  class properties
  {
  public:
    // Add a location for a schema with a target namespace.
    //
    void
    schema_location (const std::string& namespace_,
                     const std::string& location);

    // Add a location for a schema without a target namespace.
    //
    void
    no_namespace_schema_location (const std::string& location);
  };

  class document
  {
  public:
```

```
  document (parser_base& root,
            const std::string& root_element_name);

  document (parser_base& root,
            const std::string& root_element_namespace,
            const std::string& root_element_name);

public:
  // Parse URI or a local file.
  //
  void
  parse (const std::string& uri,
         flags = 0,
         const properties& = properties ());

  // Parse URI or a local file with a user-provided error_handler
  // object.
  //
  void
  parse (const std::string& uri,
         error_handler&,
         flags = 0,
         const properties& = properties ());

  // Parse URI or a local file with a user-provided ErrorHandler
  // object. Note that you must initialize the Xerces-C++ runtime
  // before calling this function.
  //
  void
  parse (const std::string& uri,
         xercesc::ErrorHandler&,
         flags = 0,
         const properties& = properties ());

  // Parse URI or a local file using a user-provided SAX2XMLReader
  // object. Note that you must initialize the Xerces-C++ runtime
  // before calling this function.
  //
  void
  parse (const std::string& uri,
         xercesc::SAX2XMLReader&,
         flags = 0,
         const properties& = properties ());

public:
  // Parse std::istream.
  //
  void
  parse (std::istream&,
         flags = 0,
         const properties& = properties ());
```

```
    // Parse std::istream with a user-provided error_handler object.
    //
    void
    parse (std::istream&,
           error_handler&,
           flags = 0,
           const properties& = properties ());


    // Parse std::istream with a user-provided ErrorHandler object.
    // Note that you must initialize the Xerces-C++ runtime before
    // calling this function.
    //
    void
    parse (std::istream&,
           xercesc::ErrorHandler&,
           flags = 0,
           const properties& = properties ());


    // Parse std::istream using a user-provided SAX2XMLReader object.
    // Note that you must initialize the Xerces-C++ runtime before
    // calling this function.
    //
    void
    parse (std::istream&,
           xercesc::SAX2XMLReader&,
           flags = 0,
           const properties& = properties ());

  public:
    // Parse std::istream with a system id.
    //
    void
    parse (std::istream&,
           const std::string& system_id,
           flags = 0,
           const properties& = properties ());


    // Parse std::istream with a system id and a user-provided
    // error_handler object.
    //
    void
    parse (std::istream&,
           const std::string& system_id,
           error_handler&,
           flags = 0,
           const properties& = properties ());
```

```
  // Parse std::istream with a system id and a user-provided
  // ErrorHandler object. Note that you must initialize the
  // Xerces-C++ runtime before calling this function.
  //
  void
  parse (std::istream&,
         const std::string& system_id,
         xercesc::ErrorHandler&,
         flags = 0,
         const properties& = properties ());


  // Parse std::istream with a system id using a user-provided
  // SAX2XMLReader object. Note that you must initialize the
  // Xerces-C++ runtime before calling this function.
  //
  void
  parse (std::istream&,
         const std::string& system_id,
         xercesc::SAX2XMLReader&,
         flags = 0,
         const properties& = properties ());



public:
  // Parse std::istream with system and public ids.
  //
  void
  parse (std::istream&,
         const std::string& system_id,
         const std::string& public_id,
         flags = 0,
         const properties& = properties ());


  // Parse std::istream with system and public ids and a user-provided
  // error_handler object.
  //
  void
  parse (std::istream&,
         const std::string& system_id,
         const std::string& public_id,
         error_handler&,
         flags = 0,
         const properties& = properties ());


  // Parse std::istream with system and public ids and a user-provided
  // ErrorHandler object. Note that you must initialize the Xerces-C++
```

```
    // runtime before calling this function.
    //
    void
    parse (std::istream&,
           const std::string& system_id,
           const std::string& public_id,
           xercesc::ErrorHandler&,
           flags = 0,
           const properties& = properties ());


    // Parse std::istream with system and public ids using a user-
    // provided SAX2XMLReader object. Note that you must initialize
    // the Xerces-C++ runtime before calling this function.
    //
    void
    parse (std::istream&,
           const std::string& system_id,
           const std::string& public_id,
           xercesc::SAX2XMLReader&,
           flags = 0,
           const properties& = properties ());


  public:
    // Parse InputSource. Note that you must initialize the Xerces-C++
    // runtime before calling this function.
    //
    void
    parse (const xercesc::InputSource&,
           flags = 0,
           const properties& = properties ());


    // Parse InputSource with a user-provided error_handler object.
    // Note that you must initialize the Xerces-C++ runtime before
    // calling this function.
    //
    void
    parse (const xercesc::InputSource&,
           error_handler&,
           flags = 0,
           const properties& = properties ());


    // Parse InputSource with a user-provided ErrorHandler object.
    // Note that you must initialize the Xerces-C++ runtime before
    // calling this function.
    //
    void
    parse (const xercesc::InputSource&,
```

```
            xercesc::ErrorHandler&,
            flags = 0,
            const properties& = properties ());


    // Parse InputSource using a user-provided SAX2XMLReader object.
    // Note that you must initialize the Xerces-C++ runtime before
    // calling this function.
    //
    void
    parse (const xercesc::InputSource&,
           xercesc::SAX2XMLReader&,
           flags = 0,
           const properties& = properties ());
  };
}
```

The `document` object is a root parser for the vocabulary. The first argument to its constructors is the parser for the type of the root element. The `parser_base` class is the base type for all parser skeletons. The second and third arguments to the `document`'s constructors are the root element's name and namespace.

The rest of the `document` interface consists of overloaded `parse()` functions. The last two arguments in each of these functions are `flags` and `properties`. The `flags` argument allows you to modify the default behavior of the parsing functions. The `properties` argument allows you to override the schema location attributes specified in XML documents. Note that the schema location paths are relative to an XML document unless they are complete URIs. For example if you want to use a local schema file then you will need to use a URI in the form `file:///absolute/path/to/your/schema`.

A number of overloaded `parse()` functions have the `system_id` and `public_id` arguments. The system id is a *system* identifier of the resources being parsed (e.g., URI or a full file path). The public id is a *public* identifier of the resource (e.g., an application-specific name or a relative file path). The system id is used to resolve relative paths (e.g., schema paths). In diagnostics messages the public id is used if it is available. Otherwise the system id is used.

The error handling mechanisms employed by the `document` object are described in Section 7.3, "Error Handling".

7.2 Expat Document Object

When Expat is used as the underlying XML parser, the `document` type has the following interface. Note that if the character type is `wchar_t`, then the string type in the interface becomes `std::wstring` (see Section 5.1, "Character Type").

```
namespace xml_schema
{
  class parser_base;
  class error_handler;

  struct document
  {
  public:
    document (parser_base&,
              const std::string& root_element_name);

    document (parser_base&,
              const std::string& root_element_namespace,
              const std::string& root_element_name);

  public:
    // Parse a local file. The file is accessed with std::ifstream
    // in binary mode. The std::ios_base::failure exception is used
    // to report io errors (badbit and failbit).
    void
    parse (const std::string& file);

    // Parse a local file with a user-provided error_handler
    // object. The file is accessed with std::ifstream in binary
    // mode. The std::ios_base::failure exception is used to report
    // io errors (badbit and failbit).
    //
    void
    parse (const std::string& file, error_handler&);

  public:
    // Parse std::istream.
    //
    void
    parse (std::istream&);

    // Parse std::istream with a user-provided error_handler object.
    //
    void
    parse (std::istream&, error_handler&);

    // Parse std::istream with a system id.
    //
    void
    parse (std::istream&, const std::string& system_id);

    // Parse std::istream with a system id and a user-provided
    // error_handler object.
    //
    void
    parse (std::istream&,
```

```
          const std::string& system_id,
          error_handler&);

    // Parse std::istream with system and public ids.
    //
    void
    parse (std::istream&,
           const std::string& system_id,
           const std::string& public_id);

    // Parse std::istream with system and public ids and a user-provided
    // error_handler object.
    //
    void
    parse (std::istream&,
           const std::string& system_id,
           const std::string& public_id,
           error_handler&);

  public:
    // Parse a chunk of input. You can call these functions multiple
    // times with the last call having the last argument true.
    //
    void
    parse (const void* data, std::size_t size, bool last);

    void
    parse (const void* data, std::size_t size, bool last,
           error_handler&);

    void
    parse (const void* data, std::size_t size, bool last,
           const std::string& system_id);

    void
    parse (const void* data, std::size_t size, bool last,
           const std::string& system_id,
           error_handler&);

    void
    parse (const void* data, std::size_t size, bool last,
           const std::string& system_id,
           const std::string& public_id);

    void
    parse (const void* data, std::size_t size, bool last,
           const std::string& system_id,
           const std::string& public_id,
           error_handler&);

  public:
```

```
    // Low-level Expat-specific parsing API.
    //
    void
    parse_begin (XML_Parser);

    void
    parse_end ();
  };
}
```

The `document` object is a root parser for the vocabulary. The first argument to its constructors is the parser for the type of the root element. The `parser_base` class is the base type for all parser skeletons. The second and third arguments to the `document`'s constructors are the root element's name and namespace.

A number of overloaded `parse()` functions have the `system_id` and `public_id` arguments. The system id is a *system* identifier of the resources being parsed (e.g., URI or a full file path). The public id is a *public* identifier of the resource (e.g., an application-specific name or a relative file path). The system id is used to resolve relative paths. In diagnostics messages the public id is used if it is available. Otherwise the system id is used.

The `parse_begin()` and `parse_end()` functions present a low-level, Expat-specific parsing API for maximum control. A typical use case would look like this (pseudo-code):

```
xxx_pimpl root_p;
document doc_p (root_p, "root");

root_p.pre ();
doc_p.parse_begin (xml_parser);

while (more_data_to_parse)
{
  // Call XML_Parse or XML_ParseBuffer.
  // Handle XML well-formedness errors if any.
}

doc_p.parse_end ();
result_type result (root_p.post_xxx ());
```

Note that if your vocabulary use XML namespaces, the `XML_ParserCreateNS()` functions should be used to create the XML parser. Space (`XML_Char (' ')`) should be used as a separator (the second argument to `XML_ParserCreateNS()`).

The error handling mechanisms employed by the `document` object are described in Section 7.3, "Error Handling".

7.3 Error Handling

There are three categories of errors that can result from running a parser on an XML document: System, XML, and Application. The System category contains memory allocation and file/stream operation errors. The XML category covers XML parsing and well-formedness checking as well as XML Schema validation errors. Finally, the Application category is for application logic errors that you may want to propagate from parser implementations to the caller of the parser.

The System errors are mapped to the standard exceptions. The out of memory condition is indicated by throwing an instance of `std::bad_alloc`. The stream operation errors are reported either by throwing an instance of `std::ios_base::failure` if exceptions are enabled or by setting the stream state.

Note that if you are parsing `std::istream` on which exceptions are not enabled, then you will need to check the stream state before calling the `post` hook, as shown in the following example:

```
int
main (int argc, char* argv[])
{
  ...

  std::ifstream ifs (argv[1]);

  if (ifs.fail ())
  {
    cerr << argv[1] << ": unable to open" << endl;
    return 1;
  }

  root_p.pre ();
  doc_p.parse (ifs);

  if (ifs.fail ())
  {
    cerr << argv[1] << ": io failure" << endl;
    return 1;
  }

  result_type result (root_p.post_xxx ());
}
```

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

```
int
main (int argc, char* argv[])
{
  try
  {
```

```
    ...

    std::ifstream ifs;
    ifs.exceptions (std::ifstream::badbit | std::ifstream::failbit);
    ifs.open (argv[1]);

    root_p.pre ();
    doc_p.parse (ifs);
    result_type result (root_p.post_xxx ());
  }
  catch (const std::ifstream::failure&)
  {
    cerr << argv[1] << ": unable to open or io failure" << endl;
    return 1;
  }
}
```

For reporting Application errors from parsing hooks, you can throw any exceptions of your choice. They are propagated to the caller of the parser without any alterations.

The XML errors can be reported either by throwing the `xml_schema::parsing` exception or by a callback to the `xml_schema::error_handler` object (and `xercesc::ErrorHandler` object in case of Xerces-C++).

The `xml_schema::parsing` exception contains a list of warnings and errors that were accumulated during parsing. Note that this exception is thrown only if there was an error. This makes it impossible to obtain warnings from an otherwise successful parsing using this mechanism. The following listing shows the definition of `xml_schema::parsing` exception. Note that if the character type is `wchar_t`, then the string type and output stream type in the definition become `std::wstring` and `std::wostream`, respectively (see Section 5.1, "Character Type").

```
namespace xml_schema
{
  class exception: public std::exception
  {
  protected:
    virtual void
    print (std::ostream&) const = 0;
  };

  inline std::ostream&
  operator<< (std::ostream& os, const exception& e)
  {
    e.print (os);
    return os;
  }


  struct severity
```

```
{
  enum value
  {
    warning,
    error
  };
};


class error
{
public:
  error (xml_schema::severity,
         const std::string& id,
         unsigned long line,
         unsigned long column,
         const std::string& message);

  xml_schema::severity
  severity () const;

  const std::string&
  id () const;

  unsigned long
  line () const;

  unsigned long
  column () const;

  const std::string&
  message () const;
};

std::ostream&
operator<< (std::ostream&, const error&);


struct diagnostics: std::vector<error>
{
};

std::ostream&
operator<< (std::ostream&, const diagnostics&);


struct parsing: exception
{
  parsing ();
  parsing (const xml_schema::diagnostics&);
```

```
    const xml_schema::diagnostics&
    diagnostics () const;

    virtual const char*
    what () const throw ();

  protected:
    virtual void
    print (std::ostream&) const;
  };
}
```

The following example shows how we can catch and print this exception. The code will print diagnostics messages one per line in case of an error.

```
int
main (int argc, char* argv[])
{
  try
  {
    // Parse.
  }
  catch (const xml_schema::parsing& e)
  {
    cerr << e << endl;
    return 1;
  }
}
```

With the `error_handler` approach the diagnostics messages are delivered as parsing progresses. The following listing presents the definition of the `error_handler` interface. Note that if the character type is `wchar_t`, then the string type in the interface becomes `std::wstring` (see Section 5.1, "Character Type").

```
namespace xml_schema
{
  class error_handler
  {
  public:
    struct severity
    {
      enum value
      {
        warning,
        error,
        fatal
      };
    };

    virtual bool
```

```
    handle (const std::string& id,
            unsigned long line,
            unsigned long column,
            severity,
            const std::string& message) = 0;
  };
}
```

The return value of the `handle()` function indicates whether parsing should continue if possible. The error with the fatal severity level terminates the parsing process regardless of the returned value. At the end of the parsing process with an error that was reported via the `error_handler` object, an empty `xml_schema::parsing` exception is thrown to indicate the failure to the caller. You can alter this behavior by throwing your own exception from the `handle()` function.

Appendix A — Supported XML Schema Constructs

The C++/Parser mapping supports validation of the following W3C XML Schema constructs in the generated code.

Construct	Notes
Structure	
element	
attribute	
any	
anyAttribute	
all	
sequence	
choice	
complex type, empty content	
complex type, mixed content	
complex type, simple content extension	
complex type, simple content restriction	Simple type facets are not validated.
complex type, complex content extension	

complex type, complex content restriction	
list	
Datatypes	
byte	
unsignedByte	
short	
unsignedShort	
int	
unsignedInt	
long	
unsignedLong	
integer	
nonPositiveInteger	
nonNegativeInteger	
positiveInteger	
negativeInteger	
boolean	
float	
double	
decimal	
string	
normalizedString	
token	
Name	
NMTOKEN	
NCName	
language	

anyURI	
ID	Identity constraint is not enforced.
IDREF	Identity constraint is not enforced.
NMTOKENS	
IDREFS	Identity constraint is not enforced.
QName	
base64Binary	
hexBinary	
date	
dateTime	
duration	
gDay	
gMonth	
gMonthDay	
gYear	
gYearMonth	
time	