Embedded 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/e against your project's technical requirements. As such, this document is intended for embedded C++ developers and software architects who are looking for an embedded 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/e Compiler Command Line Manual
- The `INSTALL` file in the XSD/e distribution provides build instructions for various platforms.
- The `examples/cxx/parser/` directory in the XSD/e distribution contains a collection of examples and a README file with an overview of each example.
- The xsde-users mailing list is the place to ask technical questions about XSD/e and the Embedded C++/Parser mapping. Furthermore the archives may already have answers to some of your questions.

1 Introduction

Welcome to CodeSynthesis XSD/e and the Embedded C++/Parser mapping. XSD/e is a validating XML parser generator for mobile and embedded systems. Embedded 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 Embedded C++/Parser mapping provides event-driven, stream-oriented XML parsing, XML Schema validation, and C++ data binding. It was specifically designed and optimized for mobile and embedded systems where hardware constrains require high efficiency and economical use of resources. As a result, the generated parsers are 2-10 times faster than general-purpose validating XML parsers while at the same time maintaining extremely low static and dynamic memory footprints. For example, a validating parser executable can be as small as 120KB in size. The size can be further reduced by disabling support for XML Schema validation.

The generated code and the runtime library are also highly-portable and, in their minimal configuration, can be used without STL, RTTI, iostream, C++ exceptions, or C++ templates.

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 Embedded C++/Parser mapping to parse, validate, and extract data from a simple XML instance document. The following chapters describe the Embedded 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) as well as general-purpose XML Schema validators have a number of drawbacks that make them less suitable for creating mobile and embedded XML processing applications. These drawbacks include:

- Text-based representation results in inefficient use of resources.
- Extra validation code that is not used by the application.
- 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 Embedded C++/Parser mapping use native data representations (e.g., integers are passed as integers, not as text) and include validation code only for XML Schema constructs that are used in the application. This results in efficient use of resources and compact object code.

Furthermore, the parser skeletons 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 skeletons 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 use native data representations and 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 Parser

In this chapter we will examine how to parse a very simple XML document using the XSD/e-generated C++/Parser skeletons. All 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/e distribution.

2.1 Writing XML Document and Schema Definition

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 Schema definition for the above instance 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 the XML Schema language, 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 `unbounde` 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 vocabulary; it tells everybody what valid XML instances of our vocabulary should look like. The next step is to compile this schema to generate C++ parser skeletons.

2.2 Translating Schema Definition to C++

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

```
$ xsde cxx-parser hello.xsd
```

The XSD/e 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/e 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 XML documents. 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 << argv[1] << ":" << e.line () << ":" << e.column ()
         << ": " << e.text () << 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/e 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/e can automatically generate sample parser implementations and a test driver from your schemas. 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/e Compiler Command Line Manual. The `'generated'` example in the XSD/e 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 UNIX this can be done with the following commands:

```
$ c++ -I.../libxsde -c driver.cxx hello-pskel.cxx
$ c++ -o driver driver.o hello-pskel.o .../libxsde/xsde/libxsde.a
$ ./driver hello.xml
Hello, sun!
Hello, moon!
Hello, world!
```

Here `.../libxsde` represents the path to the `libxsde` directory in the XSD/e 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:2:57: unexpected element encountered
```

3 Parser Skeletons

As we have seen in the previous chapter, the XSD/e 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 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/e 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_pskel` which is a parser skeleton for the built-in XML Schema type `string` and is predefined in the XSD/e 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 are the definitions for `xml_schema::string_pskel` and `xml_schema::string_pimpl`:

```
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/e 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/e 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/e 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:

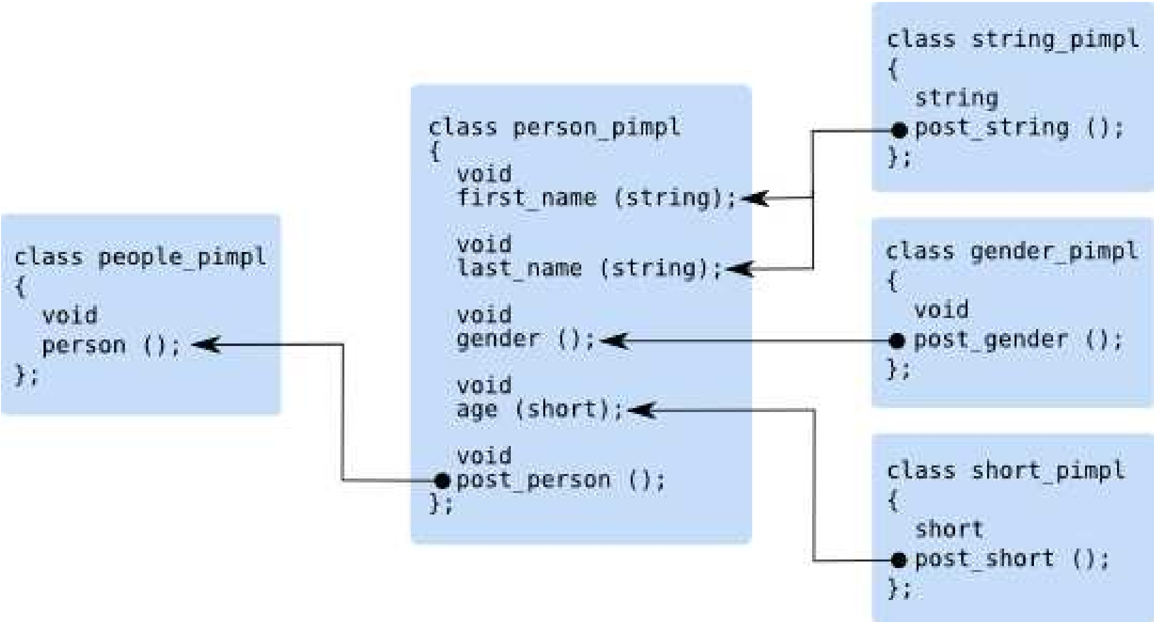
```
person_p.first_name_parsers (string_p);
person_p.last_name_parsers (string_p);
person_p.gender (gender_p);
person_p.age (short_p);

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 mapping configuration, here is the part relevant to our example:

```
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&);
```

```
    void
    parse (const void* data, size_t size, bool last);
  };
}
```

`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_pimpl` 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 three overloaded `parse()` function defined in the `document` class. The first version parses a local file identified by a name. The second version reads the data from an input stream. The last version allows you to parse the data directly from a buffer, one chunk at a time. You can call this function multiple times with the final call having the `last` argument set to true. For more information on the `xml_schema::document` class refer to Chapter 7, "Document Parser 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/e 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/e 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/e compiler know about our type map:

```
$ xsde 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:

```
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;

  negativeInteger long long;
  nonPositiveInteger long long;

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

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

  base64Binary xml_schema::buffer* xml_schema::buffer*;
  hexBinary xml_schema::buffer* 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;
}
```

If STL is enabled (Section 5.1, "Standard Template Library"), the following mapping is used for the string-based XML Schema built-in types:

```
namespace http://www.w3.org/2001/XMLSchema
{
  include <string>;

  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;

  QName xml_schema::qname;

  NMTOKENS xml_schema::string_sequence;
  IDREFS xml_schema::string_sequence;
}
```

Otherwise, a C string-based mapping is used:

```
namespace http://www.w3.org/2001/XMLSchema
{
  string char* char*;
  normalizedString char* char*;
  token char* char*;
  Name char* char*;
  NMTOKEN char* char*;
  NCName char* char*;
  ID char* char*;
  IDREF char* char*;
  language char* char*;
  anyURI char* char*;

  QName xml_schema::qname* xml_schema::qname*;

  NMTOKENS xml_schema::string_sequence*
```

```
           xml_schema::string_sequence*;

  IDREFS xml_schema::string_sequence*
         xml_schema::string_sequence*;
}
```

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:

```
$ xsde cxx-parser --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.../libxsde -c driver.cxx people-pskel.cxx
$ c++ -o driver driver.o people-pskel.o .../libxsde/xsde/libxsde.a
$ ./driver people.xml
first:  John
last:   Doe
gender: male
age:    32

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

5 Mapping Configuration

The Embedded C++/Parser mapping has a number of configuration parameters that determine the overall properties and behavior of the generated code, such as the use of Standard Template Library (STL), Input/Output Stream Library (iostream), C++ exceptions, XML Schema validation, and 64-bit integer types. Previous chapters assumed that all these features were enabled. This chapter will discuss the changes in the Embedded C++/Parser programming model when these optional features are disabled. A complete example that uses the minimal mapping configuration is presented at the end of this chapter.

In order to enable or disable a particular feature, the corresponding parameter should be set accordingly in the XSD/e runtime library as well as specified during schema compilation with the XSD/e command line options as described in the XSD/e Compiler Command Line Manual.

While the XML documents can use various encodings, the Embedded C++/Parser mapping always delivers character data to the application in the UTF-8 encoding. The underlying XML parser used by the Embedded C++/Parser mapping includes built-in support for XML documents encoded in UTF-8, UTF-16, ISO-8859-1, and US-ASCII. Other encodings can be supported by providing application-specific decoder functions.

5.1 Standard Template Library

To disable the use of STL you will need to configure the XSD/e runtime without support for STL as well as pass the `--no-stl` option to the XSD/e compiler when translating your schemas. When STL is disabled, all string-based XML Schema types are mapped to C-style `char*` instead of `std::string`, as described in Section 4.2, "Type Map File Format". The following code fragment shows changes in the signatures of `first_name()` and `last_name()` hooks from the person record example.

```
class person_pskel
{
public:
  virtual void
  first_name (char*);

  virtual void
  last_name (char*);

  ...
};
```

Note that it is your responsibility to eventually release the memory associated with these strings using operator `delete[]`.

5.2 Input/Output Stream Library

To disable the use of iostream you will need to configure the XSD/e runtime library without support for iostream as well as pass the `--no-iostream` option to the XSD/e compiler when translating your schemas. When iostream is disabled, the following two `parse()` functions become unavailable:

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

  void
  parse (std::istream&);
```

Leaving you with only one function in the form:

```
void
parse (const void* data, size_t size, bool last);
```

See Section 7.1, "Document Parser" for more information on the semantics of these functions.

5.3 C++ Exceptions

To disable the use of C++ exceptions, you will need to configure the XSD/e runtime without support for exceptions as well as pass the `--no-exceptions` option to the XSD/e compiler when translating your schemas. When C++ Exceptions are disabled, the error conditions are indicated with error codes instead of exceptions, as described in Section 7.3, "Error Codes".

5.4 XML Schema Validation

To disable support for XML Schema validation, you will need to configure the XSD/e runtime accordingly as well as pass the `--suppress-validation` option to the XSD/e compiler when translating your schemas. Disabling XML Schema validation allows to further increase the parsing performance and reduce footprint in cases when XML instances are known to be valid.

5.5 64-bit Integer Type

By default the 64-bit `long` and `unsignedLong` XML Schema built-in types are mapped to the 64-bit `long long` and `unsigned long long` fundamental C++ types. To disable the use of these types in the mapping you will need to configure the XSD/e runtime accordingly as well as pass the `--no-long-long` option to the XSD/e compiler when translating your schemas. When the use of 64-bit integral C++ types is disabled the `long` and `unsignedLong` XML Schema built-in types are mapped to `long` and `unsigned long` fundamental C++ types.

5.6 A Minimal Example

The following example is a re-implementation of the person records example presented in Chapter 3, "Parser Skeletons". It is intended to work without STL, iostream, and C++ exceptions. It can be found in the `examples/cxx/parser/minimal/` directory of the XSD/e distribution. The `people.xsd` schema is compiled with the `--no-stl`, `--no-iostream`, and `--no-exceptions` options. The following listing presents the implementation of parser skeletons and the test driver in full.

```
#include <stdio.h>

#include "people-pskel.hxx"

class gender_pimpl: gender_pskel, xml_schema::string_pimpl
```

```
{
public:
  virtual void
  post_gender ()
  {
    char* s = post_string ();
    printf ("gender: %s\n", s);
    delete[] s;
  }
};

class person_pimpl: person_pskel
{
public:
  virtual void
  first_name (char* n)
  {
    printf ("first: %s\n", n);
    delete[] n;
  }

  virtual void
  last_name (char* n)
  {
    printf ("last: %s\n", n);
    delete[] n;
  }

  virtual void
  age (short a)
  {
    printf ("age: %hd\n", a);
  }
};

class people_pimpl: people_pskel
{
public:
  virtual void
  person ()
  {
    // Add an extra newline after each person record.
    //
    printf ("\n");
  }
};

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);

// Open the file.
//
FILE* f = fopen (argv[1], "rb");

if (f == 0)
{
  fprintf (stderr, "%s: unable to open\n", argv[1]);
  return 1;
}

// Parse.
//
using xml_schema::error;
error e;
bool io_error = false;

do
{
  xml_schema::document doc_p (people_p, "people");
  if (e = doc_p.error ())
    break;

  people_p.pre ();
  if (e = people_p.error ())
    break;

  char buf[4096];
  do
  {
    size_t s = fread (buf, 1, sizeof (buf), f);

    if (s != sizeof (buf) && ferror (f))
    {
      io_error = true;
      break;
    }

    doc_p.parse (buf, s, feof (f) != 0);
    e = doc_p.error ();
```

```
    } while (!e && !feof (f));

    if (io_error || e)
      break;

    people_p.post_people ();
    e = people_p.error ();

  } while (false);

  fclose (f);

  // Handle errors.
  //

  if (io_error)
  {
    fprintf (stderr, "%s: read failure\n", argv[1]);
    return 1;
  }

  if (e)
  {
    switch (e.type ())
    {
    case error::sys:
      {
        fprintf (stderr, "%s: %s\n", argv[1], e.sys_text ());
        break;
      }
    case error::xml:
      {
        fprintf (stderr, "%s:%lu:%lu: %s\n",
                 argv[1], e.line (), e.column (), e.xml_text ());
        break;
      }
    case error::schema:
      {
        fprintf (stderr, "%s:%lu:%lu: %s\n",
                 argv[1], e.line (), e.column (), e.schema_text ());
        break;
      }
    case error::app:
      {
        fprintf (stderr, "%s:%lu:%lu: application error %d\n",
                 argv[1], e.line (), e.column (), e.app_code ());
        break;
      }
    default:
      break;
    }
```

```
    return 1;
  }
  return 0;
}
```

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. Declarations for these types are automatically included into each generated header file. As a result you don't need to include any headers to gain access to these parser implementations.

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` or `long` Section 5.5, "64-bit Integer Type"
`unsignedLong`	`unsigned_long_pimpl`	`unsigned long long` or `unsigned long` Section 5.5, "64-bit Integer Type"
arbitrary-length integral types		
`integer`	`integer_pimpl`	`long`
`nonPositiveInteger`	`non_positive_integer_pimpl`	`long`
`nonNegativeInteger`	`non_negative_integer_pimpl`	`unsigned long`
`positiveInteger`	`positive_integer_pimpl`	`unsigned long`
`negativeInteger`	`negative_integer_pimpl`	`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 `char*` Section 5.1, "Standard Template Library"
`normalizedString`	`normalized_string_pimpl`	`std::string` or `char*` Section 5.1, "Standard Template Library"
`token`	`token_pimpl`	`std::string` or `char*` Section 5.1, "Standard Template Library"
`Name`	`name_pimpl`	`std::string` or `char*` Section 5.1, "Standard Template Library"
`NMTOKEN`	`nmtoken_pimpl`	`std::string` or `char*` Section 5.1, "Standard Template Library"
`NCName`	`ncname_pimpl`	`std::string` or `char*` Section 5.1, "Standard Template Library"
`language`	`language_pimpl`	`std::string` or `char*` Section 5.1, "Standard Template Library"
qualified name		
`QName`	`qname_pimpl`	`xml_schema::qname` or `xml_schema::qname*` Section 6.1, "`QName` Parser"
ID/IDREF types		
`ID`	`id_pimpl`	`std::string` or `char*` Section 5.1, "Standard Template Library"
`IDREF`	`idref_pimpl`	`std::string` or `char*` Section 5.1, "Standard Template Library"
list types		

`NMTOKENS`	`nmtokens_pimpl`	`xml_schema::string_sequence`
		or
		`xml_schema::string_sequence*`
		Section 6.2, "`NMTOKENS` and `IDREFS` Parsers"

IDREFS	idrefs_pimpl	xml_schema::string_sequence or xml_schema::string_sequence* Section 6.2, "NMTOKENS and IDREFS Parsers"
URI types		
anyURI	uri_pimpl	std::string or char* Section 5.1, "Standard Template Library"
binary types		
base64Binary	base64_binary_pimpl	xml_schema::buffer* Section 6.3, "base64Binary and hexBinary Parsers"
hexBinary	hex_binary_pimpl	xml_schema::buffer* Section 6.3, "base64Binary and hexBinary Parsers"
date/time types		
date	date_pimpl	xml_schema::date Section 6.5, "date Parser"
dateTime	date_time_pimpl	xml_schema::date_time Section 6.6, "dateTime Parser"
duration	duration_pimpl	xml_schema::duration Section 6.7, "duration Parser"
gDay	gday_pimpl	xml_schema::gday Section 6.8, "gDay Parser"
gMonth	gmonth_pimpl	xml_schema::gmonth Section 6.9, "gMonth Parser"
gMonthDay	gmonth_day_pimpl	xml_schema::gmonth_day Section 6.10, "gMonthDay Parser"
gYear	gyear_pimpl	xml_schema::gyear Section 6.11, "gYear Parser"
gYearMonth	gyear_month_pimpl	xml_schema::gyear_month Section 6.12, "gYearMonth Parser"
time	time_pimpl	xml_schema::time Section 6.13, "time Parser"

6.1 `QName` Parser

The return type of the `qname_pimpl` parser implementation is either `xml_schema::qname` when STL is enabled (Section 5.1, "Standard Template Library") or `xml_schema::qname*` when STL is disabled. The `qname_pimpl` represents an XML qualified name. When the return type is `xml_schema::qname*`, the returned object is dynamically allocated with operator `new` and should be eventually deallocated with operator `delete`. With STL enabled, the `qname` type has the following interface:

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

    void
    swap (qname&);

    const std::string&
    prefix () const;

    std::string&
    prefix ();

    void
    prefix (const std::string&);

    const std::string&
    name () const;

    std::string&
    name ();

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

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

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

When STL is disabled and the C++ exceptions are enabled (Section 5.3, "C++ Exceptions"), the `qname` type has the following interface:

```
namespace xml_schema
{
  class qname
  {
  public:
    // The default c-tor creates an uninitialized qname. Use
    // modifiers to initialize it.
    //
    qname ();

    explicit
    qname (const char* name);
    qname (const char* prefix, const char* name);

    void
    swap (qname&);

  private:
    qname (const qname&);

    qname&
    operator= (const qname&);

  public:
    char*
    prefix ();

    const char*
    prefix () const;

    void
    prefix (const char*);

    char*
    detach_prefix ();

    void
    attach_prefix (char*);

  public:
    char*
    name ();

    const char*
    name () const;

    void
    name (const char*);
```

```
    char*
    detach_name ();

    void
    attach_name (char*);
  };

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

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

If you detach the underlying prefix or name strings, then they should be eventually deallocated with operator `delete[]`. If you attach the underlying prefix or name strings, then they should be allocated with operator `new char[]` and will be deallocated with operator `delete[]` by the `qname` object.

Finally, if both STL and the C++ exceptions are disabled, the `qname` type has the following interface:

```
namespace xml_schema
{
  class qname
  {
  public:
    enum error
    {
      error_none,
      error_no_memory
    };

    // The default c-tor creates an uninitialized qname. Use
    // modifiers to initialize it.
    //
    qname ();

    void
    swap (qname&);

  private:
    qname (const qname&);

    qname&
    operator= (const qname&);

  public:
    char*
```

```
    prefix ();

    const char*
    prefix () const;

    error
    prefix (const char*);

    char*
    detach_prefix ();

    void
    attach_prefix (char*);

  public:
    char*
    name ();

    const char*
    name () const;

    error
    name (const char*);

    char*
    detach_name ();

    void
    attach_name (char*);
  };

  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 either `xml_schema::string_sequence` when STL is enabled (Section 5.1, "Standard Template Library") or `xml_schema::string_sequence*` when STL is disabled. When the return type is `xml_schema::string_sequence*`, the returned object is dynamically allocated with operator `new` and should be eventually deallocated with operator `delete`. With STL enabled, the `string_sequence` type has the following interface:

```
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&);
}
```

When STL is disabled and the C++ exceptions are enabled (Section 5.3, "C++ Exceptions"), the `string_sequence` type has the following interface:

```
namespace xml_schema
{
  class string_sequence
  {
  public:
    typedef char** iterator;
    typedef const char* const* const_iterator;

    string_sequence ();

    void
    swap (string_sequence&);

  private:
    string_sequence (string_sequence&);

    string_sequence&
    operator= (string_sequence&);

  public:
    iterator
    begin ();

    const_iterator
    begin () const;
```

```
    iterator
    end ();

    const_iterator
    end () const;

    char*
    front ();

    const char*
    front () const;

    char*
    back ();

    const char*
    back () const;

    char*
    operator[] (size_t);

    const char*
    operator[] (size_t) const;

  public:
    bool
    empty () const;

    size_t
    size () const;

 public:
    void
    push_back (const char*);

    void
    push_back_attach (char*);

    // Detach a string from the sequence at a given position.
    // The string pointer at this position in the sequence is
    // set to 0.
    //
    char*
    detach (iterator);
  };

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

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

If you detach the underlying element string, then it should be eventually deallocated with operator `delete[]`. If you attach the underlying element string, then it should be allocated with operator `new char[]` and will be deallocated with operator `delete[]` by the `string_sequence` object.

Finally, if both STL and the C++ exceptions are disabled, the `string_sequence` type has the following interface:

```
namespace xml_schema
{
  class string_sequence
  {
  public:
    enum error
    {
      error_none,
      error_no_memory
    };

    typedef char** iterator;
    typedef const char* const* const_iterator;

    string_sequence ();

    void
    swap (string_sequence&);

  private:
    string_sequence (string_sequence&);

    string_sequence&
    operator= (string_sequence&);

  public:
    iterator
    begin ();

    const_iterator
    begin () const;

    iterator
    end ();

    const_iterator
    end () const;
```

```
    char*
    front ();

    const char*
    front () const;

    char*
    back ();

    const char*
    back () const;

    char*
    operator[] (size_t);

    const char*
    operator[] (size_t) const;

  public:
    bool
    empty () const;

    size_t
    size () const;

  public:
    error
    push_back (const char*);

    error
    push_back_attach (char*);

    // Detach a string from the sequence at the given position.
    // The string pointer at this position in the sequence is
    // set to 0.
    //
    char*
    detach (iterator);
  };

  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 `xml_schema::buffer*`. The returned object is dynamically allocated with operator `new` and should be eventually deallocated with operator `delete`. With the C++ exceptions enabled (Section 5.3, "C++ Exceptions"), the `buffer` type has the following interface:

```
namespace xml_schema
{
  class buffer
  {
  public:
    class bounds {}; // Out of bounds exception.

  public:
    buffer ();

    explicit
    buffer (size_t size);
    buffer (size_t size, size_t capacity);
    buffer (const void* data, size_t size);
    buffer (const void* data, size_t size, size_t capacity);

    enum ownership_value { assume_ownership };

    // This constructor assumes ownership of the memory passed.
    //
    buffer (void* data, size_t size, size_t capacity, ownership_value);

  private:
    buffer (const buffer&);

    buffer&
    operator= (const buffer&);

  public:
    void
    attach (void* data, size_t size, size_t capacity);

    void*
    detach ();

    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&);
}
```

The last constructor and the `attach()` member function make the `buffer` instance assume the ownership of the memory block pointed to by the `data` argument and eventually release it by calling `operator delete()`. The `detach()` member function detaches and returns the underlying memory block which should be eventually released 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 or `attach()` member function arguments violate the `(size <= capacity)` constraint.

If the C++ exceptions are disabled, the `buffer` type has the following interface:

```
namespace xml_schema
{
  class buffer
  {
  public:
    enum error
    {
      error_none,
      error_bounds,
      error_no_memory
    };

    buffer ();

  private:
    buffer (const buffer&);

    buffer&
    operator= (const buffer&);

  public:
    error
    attach (void* data, size_t size, size_t capacity);

    void*
    detach ();

    void
    swap (buffer&);

  public:
    size_t
    capacity () const;

    error
    capacity (size_t);

    error
    capacity (size_t, bool& moved);

  public:
    size_t
    size () const;

    error
    size (size_t);

    error
    size (size_t, bool& moved);

  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&);
}
```

6.4 Time Zone Representation

The `date`, `dateTime`, `gDay`, `gMonth`, `gMonthDay`, `gYear`, `gYearMonth`, and `time` XML Schema built-in types all include an optional time zone component. The following `xml_schema::time_zone` base class is used to represent this information:

```
namespace xml_schema
{
  class time_zone
  {
  public:
    time_zone ();
    time_zone (short hours, short minutes);

    bool
    zone () const;

    void
    zone_reset ();

    short
    zone_hours () const;

    void
```

```
    zone_hours (short);

    short
    zone_minutes () const;

    void
    zone_minutes (short);
  };

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

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

The `zone()` accessor function returns `true` if the time zone is specified. The `zone_reset()` modifier function resets the time zone object to the *not specified* state. If the time zone offset is negative then both hours and minutes components are represented as negative integers.

6.5 `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. For more information on the base `xml_schema::time_zone` class refer to Section 6.4, "Time Zone Representation".

```
namespace xml_schema
{
  class date: public time_zone
  {
  public:
    date (int year, unsigned short month, unsigned short day);
    date (int year, unsigned short month, unsigned short day,
          short zone_hours, short zone_minutes);

    int
    year () const;

    void
    year (int);

    unsigned short
    month () const;

    void
    month (unsigned short);

    unsigned short
```

```
    day () const;

    void
    day (unsigned short);
  };

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

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

6.6 `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. For more information on the base `xml_schema::time_zone` class refer to Section 6.4, "Time Zone Representation".

```
namespace xml_schema
{
  class date_time: public time_zone
  {
  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, short zone_hours, short zone_minutes);

    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);
  };

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

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

6.7 `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&);
}
```

6.8 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. For more information on the base `xml_schema::time_zone` class refer to Section 6.4, "Time Zone Representation".

```
namespace xml_schema
{
  class gday: public time_zone
  {
  public:
    explicit
    gday (unsigned short day);
    gday (unsigned short day, short zone_hours, short zone_minutes);

    unsigned short
    day () const;

    void
    day (unsigned short);
  };

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

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

6.9 `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. For more information on the base `xml_schema::time_zone` class refer to Section 6.4, "Time Zone Representation".

```
namespace xml_schema
{
  class gmonth: public time_zone
  {
  public:
    explicit
    gmonth (unsigned short month);
    gmonth (unsigned short month,
            short zone_hours, short zone_minutes);

    unsigned short
    month () const;

    void
    month (unsigned short);
  };

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

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

6.10 `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. For more information on the base `xml_schema::time_zone` class refer to Section 6.4, "Time Zone Representation".

```
namespace xml_schema
{
  class gmonth_day: public time_zone
  {
  public:
    gmonth_day (unsigned short month, unsigned short day);
    gmonth_day (unsigned short month, unsigned short day,
                short zone_hours, short zone_minutes);

    unsigned short
    month () const;

    void
    month (unsigned short);

    unsigned short
    day () const;

    void
    day (unsigned short);
  };

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

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

6.11 `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. For more information on the base `xml_schema::time_zone` class refer to Section 6.4, "Time Zone Representation".

```
namespace xml_schema
{
  class gyear: public time_zone
  {
  public:
    explicit
    gyear (int year);
    gyear (int year, short zone_hours, short zone_minutes);

    int
    year () const;

    void
    year (int);
  };

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

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

6.12 `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. For more information on the base `xml_schema::time_zone` class refer to Section 6.4, "Time Zone Representation".

```
namespace xml_schema
{
  class gyear_month: public time_zone
  {
  public:
    gyear_month (int year, unsigned short month);
    gyear_month (int year, unsigned short month,
                 short zone_hours, short zone_minutes);

    int
    year () const;

    void
    year (int);

    unsigned short
    month () const;

    void
    month (unsigned short);
```

```
  };

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

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

6.13 `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. For more information on the base `xml_schema::time_zone` class refer to Section 6.4, "Time Zone Representation".

```
namespace xml_schema
{
  class time: public time_zone
  {
  public:
    time (unsigned short hours, unsigned short minutes, double seconds);
    time (unsigned short hours, unsigned short minutes, double seconds,
          short zone_hours, short zone_minutes);

    unsigned short
    hours () const;

    void
    hours (unsigned short);

    unsigned short
    minutes () const;

    void
    minutes (unsigned short);

    double
    seconds () const;

    void
    seconds (double);
  };

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

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

7 Document Parser 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.

There are four categories of errors that can result from running a parser on an XML instance: system, xml, schema, and application. The system category contains memory allocation and file/stream operation errors. The xml category is for XML parsing and well-formedness checking errors. Similarly, the schema category is for 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 C++/Parser mapping supports two methods of report errors: using C++ exceptions and with error codes. The method used depends on whether or not you have configured the XSD/e runtime and the generated code with C++ exceptions enabled, as described in Section 5.3, "C++ Exceptions".

7.1 Document Parser

The `xml_schema::document` parser is a root parser for the vocabulary. As mentioned in Section 3.4, "Connecting the Parsers Together", its interface varies depending on the mapping configuration (Chapter 5, "Mapping Configuration"). When STL and the iostream library are enabled, the `xml_schema::document` class has the following interface:

```
namespace xml_schema
{
  class parser_base;

  class document
  {
  public:
    document (parser_base&,
              const char* root_element_name);

    document (parser_base&,
              const char* root_element_namespace,
              const char* root_element_name);

    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. std::ios_base::failure exception is used to
    // report io errors (badbit and failbit) if exceptions are
    // enabled. Otherwise error codes are used.
    //
    void
    parse (const char* file);

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

    // Parse std::istream. std::ios_base::failure exception is used
    // to report io errors (badbit and failbit) if exceptions are
    // enabled. Otherwise error codes are used.
    //
    void
    parse (std::istream&);

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

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

    void
    parse_end ();
  };
}
```

When the use STL is disabled, the constructors and the `parse()` function that use `std::string` in their signatures are not available. When the use of iostream is disabled, the `parse()` functions that parse a local file and `std::istream` are not available.

The first argument to all overloaded 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 `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_stuff_to_parse)
{
   // Call XML_Parse or XML_ParseBuffer:
   //
   if (XML_Parse (...) != XML_STATUS_ERROR)
     break;
}

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()`). Furthermore, if XML_Parse or XML_ParseBuffer fail, call `parse_end()` to determine the error which is indicated either via exception or set as an error code.

The error handling mechanisms employed by the `document` parser are described in Section 7.2, "Exceptions" and Section 7.3, "Error Codes".

7.2 Exceptions

When C++ exceptions are used for error reporting, 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 by throwing an instance of `std::ios_base::failure`.

The xml and schema errors are reported by throwing the `xml_schema::xml` and `xml_schema::schema` exceptions, respectively. These two exceptions derive from `xml_schema::exception` which, in turn, derives from `std::exception`. As a result, you can handle any error from these two categories by either catching `std::exception`, `xml_schema::exception`, or individual exceptions. The further down the hierarchy you go the more detailed error information is available to you. The following listing shows the definitions of these exceptions:

```
namespace xml_schema
{
  class exception: std::exception
  {
  public:
    unsigned long
    line () const;

    unsigned long
    column () const;
```

```
    virtual const char*
    text () const = 0;

    ...
  };

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


  typedef <implementation-details> xml_error;

  class xml: exception
  {
  public:
    xml_error
    code () const;

    virtual const char*
    text () const;

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

    ...
  };


  typedef <implementation-details> schema_error;

  class schema: exception
  {
  public:
    schema_error
    code () const;

    virtual const char*
    text () const;

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

    ...
  };
}
```

The `xml_error` and `schema_error` are implementation-specific error code types. The `operator<<` defined for the `exception` class simply prints the error description as returned by the `text()` function. The following example shows how we can catch these exceptions:

```
int
main (int argc, char* argv[])
{
  try
  {
    // Parse argv[1].
  }
  catch (const xml_schema::exception& e)
  {
    cout << argv[1] << ":" << e.line () << ":" << e.column ()
         << ": error: " << e.text () << endl;
    return 1;
  }
}
```

Finally, 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.

7.3 Error Codes

When C++ exceptions are not available, error codes are used to report error conditions. Each parser skeleton and the root document object have the following member function for querying the error status:

```
xml_schema::error
error () const;
```

To handle all possible error conditions, you will need to obtain the error status after calls to: the document constructor (it does some memory allocations which may fail), the root parser `pre()` hook, each call to the `parse()` function, and, finally, the call to the root parser `post_*` hook. The definition of `xml_schema::error` class is presented below:

```
namespace xml_schema
{
  class sys_error
  {
  public:
    enum value
    {
      none,
      no_memory,
      open_failed,
      read_failed
    };

    sys_error (value);

    operator value () const;
```

```
  static const char*
  text (value);

  ...
};

typedef <implementation-details> xml_error;
typedef <implementation-details> schema_error;

class error
{
public:
  enum error_type
  {
    none,
    sys,
    xml,
    schema,
    app
  };

  error_type
  type () const;

  // Line and column are only available for xml, schema, and
  // app errors.
  //
  unsigned long
  line () const;

  unsigned long
  column () const;

  // Returns true if there is an error so that you can write
  // if (p.error ()) or if (error e = p.error ()).
  //
  typedef void (error::*bool_convertible) ();
  operator bool_convertible () const;

  // system
  //
  sys_error
  sys_code () const;

  const char*
  sys_text () const;

  // xml
  //
  xml_error
  xml_code () const;
```

```
    const char*
    xml_text () const;

    // schema
    //
    schema_error
    schema_code () const;

    const char*
    schema_text () const;

    // app
    //
    int
    app_code () const;

    ...
  };
}
```

The `xml_error` and `schema_error` are implementation-specific error code types. The `error` class incorporates four categories of errors which you can determine by calling the `type()` function. The following example shows how to handle error conditions with error codes. It is based on the person record example presented in Chapter 3, "Parser Skeletons".

```
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.
  //
  using xml_schema::error;
  error e;

  do
  {
    xml_schema::document doc_p (people_p, "people");
    if (e = doc_p.error ())
```

```
      break;

    people_p.pre ();
    if (e = people_p.error ())
      break;

    doc_p.parse (argv[1]);
    if (e = doc_p.error ())
      break;

    people_p.post_people ();
    e = people_p.error ();

  } while (false);

  // Handle errors.
  //
  if (e)
  {
    switch (e.type ())
    {
    case error::sys:
      {
        cerr << argv[1] << ": error: " << e.sys_text () << endl;
        break;
      }
    case error::xml:
      {
        cerr << argv[1] << ":" << e.line () << ":" << e.column ()
             << ": error: " << e.xml_text () << endl;
        break;
      }
    case error::schema:
      {
        cerr << argv[1] << ":" << e.line () << ":" << e.column ()
             << ": error: " << e.schema_text () << endl;
        break;
      }
    case error::app:
      {
        cerr << argv[1] << ":" << e.line () << ":" << e.column ()
             << ": application error " << e.app_code () << endl;
        break;
      }
    }
    return 1;
  }
}
```

The error type for application errors is `int` with the value `0` indicated the absence of error. You can set the application error by calling the `error()` function inside a parser hook implementation. For example, if it was invalid to have a person younger than 18 in our people catalog, then we could have implemented this check as follows:

```
class person_pimpl: person_pskel
{
public:
  virtual void
  age (short a)
  {
    if (a < 18)
      error (1);
  }
};
```

Appendix A — Supported XML Schema Constructs

The Embedded 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	