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/serializer 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 constraints 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, and 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 (for example, 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 callbacks 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 a separate parser skeleton for each XML Schema type, the application implementation is simpler and thus easier to read and understand.
- **Safety.** The XML data is delivered to parser callbacks as statically typed objects. The parser callbacks 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 application 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 Example

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

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

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

  <greeting>Hello</greeting>

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

</hello>
```

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

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

  <xs:complexType name="hello">
    <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 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 callbacks. 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 callbacks. 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 callback. 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 callback. Its name is constructed by adding the parser skeleton name to the `post_` prefix. The finalization callback 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 callback is called `post_hello()` instead of `post()` just like `pre()`. The reason for this is that finalization callbacks can have different return types and result in function signature clashes across inheritance hierarchies. To prevent this, the signatures of finalization callbacks 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: public 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_pimpl doc_p (hello_p, "hello");

    hello_p.pre ();
    doc_p.parse (argv[1]);
    hello_p.post_hello ();
  }
  catch (const xml_schema::parser_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 the calls to `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 the 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 the 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 schema that describes a collection of person records. We save it in `people.xsd`:

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

  <xs:simpleType name="gender">
    <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_pskel` parser skeleton looks like this:

```
class gender_pskel: public xml_schema::string_pskel
{
public:
  gender_pskel (xml_schema::string_pskel* base_impl);

  // Parser callbacks. 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 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 `gender_pskel` class also declares a constructor which expects a pointer to the base parser skeleton. We will discuss the purpose of this constructor shortly.

The `pre()` and `post_gender()` callbacks 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: public gender_pskel
{
public:
  gender_pimpl ()
    : gender_pskel (&base_impl_)
  {
  }

  virtual void
  post_gender ()
  {
    std::string s = post_string ();
    cout << "gender: " << s << endl;
  }
```

```
private:
  xml_schema::string_pimpl base_impl_;
};
```

While the code is quite short, there is a lot going on. First, notice that we define a member variable `base_impl_` of type `xml_schema::string_pimpl` and then pass it to the `gender_pskel`'s constructor. We have 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`. By passing `base_impl_` to the `gender_pskel`'s constructor we provide an implementation for the part of the parser skeleton that is inherited from `string_pskel`.

This is another common theme in the C++/Parser programming model: reusing implementations of the base parsers in the derived ones. 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()`. For more information on parser implementation reuse refer to Section 5.6, "Parser Reuse".

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: public parser_simple_content
  {
  public:
    virtual std::string
    post_string () = 0;
  };

  class string_pimpl: public 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 `parser_simple_content` class and the `_pre()` and `_characters()` functions. The `parser_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 `parser_complex_content` class which corresponds to the complex content mode (types with nested elements, for example, `person` from `people.xsd`).

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

```
virtual void
pre ();

virtual void
_pre ();

virtual void
_post ();

virtual void
post_name ();
```

`pre()` and `_pre()` are initialization callbacks. 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()` can 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 callbacks 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 callback. A low-level one that delivers raw character content for the type being parsed. You will seldom need to use this callback directly. Using implementations for the 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_*()` callbacks, 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 xml_schema::parser_complex_content
{
public:
  // Parser callbacks. 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 callback for each of the nested elements found in the `person` XML Schema type. The implementation of this parser is straightforward:

```
class person_pimpl: public 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()` callback 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 xml_schema::parser_complex_content
{
public:
  // Parser callbacks. 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()` callback will be called after parsing each `person` element. While `person_pimpl` does all the printing, one useful thing we can do in this callback is to print an extra newline after each person record so that our output is more readable:

```
class people_pimpl: public 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 assemble 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:

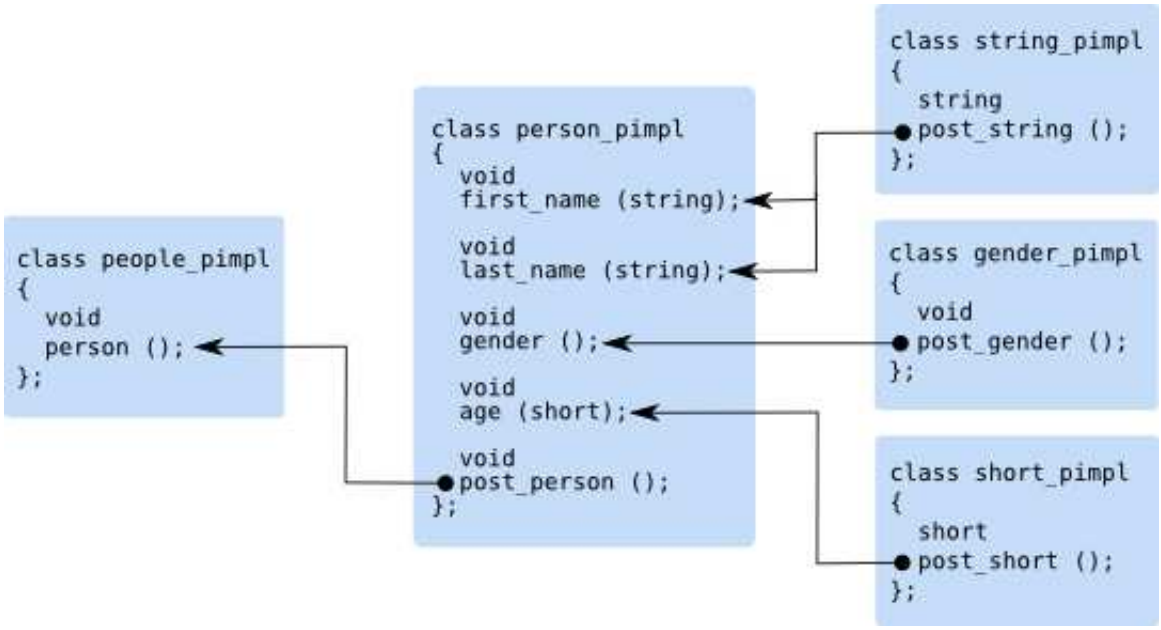
```
person_p.first_name_parser (string_p);
person_p.last_name_parser (string_p);
person_p.gender_parser (gender_p);
person_p.age_parser (short_p);

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. Notice the correspondence between return types of the `post_*()` functions and argument types of element callbacks that are connected by the arrows.



The last step is the construction of the document parser and invocation of the complete parser on our sample XML instance:

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

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

Let's consider `xml_schema::document_pimpl` 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_pimpl
  {
  public:
    document_pimpl (xml_schema::parser_base&,
                    const std::string& root_element_name);

    document_pimpl (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_pimpl` 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_pimpl`'s constructors.

There are also three overloaded `parse()` function defined in the `document_pimpl` 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_pimpl` 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 callback 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()` callbacks 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()` callbacks are called on `string_p`. Parsing is now delegated to `string_p`.
6. The parser encounters character content consisting of `"John"`. The `_characters()` callback is called on `string_p`.
7. The parser encounters the end of `first-name` element. The `_post()` and `post_string()` callbacks are called on `string_p`. The `first_name()` callback 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()` callbacks are called on `person_p`. The `person()` callback 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()` callback 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 callbacks 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()` callback 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_*()` callback in the parser skeleton generated for this type. And, second, the argument type for callbacks 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 the argument type for the `gender()` callback in the `person_pskel` skeleton. As you might have guessed, the generated code will then pass the return value from the `post_*()` callback as an argument to the element or attribute callback.

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` type from the previous paragraph.

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

The first line indicates that the generated code must include `people.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 (we use fully-qualified C++ names to avoid name clashes). 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 xml_schema::string_pskel
{
  virtual ::gender
  post_gender () = 0;

  ...
};

class person_pskel: public xml_schema::parser_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 return types of `post_*()` callbacks in parser skeletons corresponding to XML Schema types as well as argument types for callbacks corresponding to elements and attributes of these types.

The compiler has a set of predefined mapping rules that map the built-in XML Schema types to suitable C++ types (discussed below) and all other types to `void`. By providing your own type maps you can override these predefined 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_*()` callback. Optional *<cxx-arg-type>* is an argument type for callbacks corresponding to elements and attributes of this type. If *<cxx-arg-type>* is not specified, it defaults to *<cxx-ret-type>* if *<cxx-ret-type>* ends with `*` or `&` (that is, it is a pointer or a reference) and `const` *<cxx-ret-type>*`&` otherwise. *<file-name>* is a file name either in the `" "` or `< >` format and is added with the `#include` directive to the generated code.

The **#** character starts a comment that ends with a new line or end of file. To specify a name that contains **#** enclose it in **" "**. For example:

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

  # Pass apples by value.
  #
  apple apple;

  # Pass oranges as pointers.
  #
  orange 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*";
}
```

The compiler has a number of predefined mapping rules for the built-in XML Schema types which 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;

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

  base64Binary xml_schema::buffer*;
  hexBinary 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>;

  anySimpleType std::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;
}
```

Otherwise, a C string-based mapping is used:

```
namespace http://www.w3.org/2001/XMLSchema
{
  anySimpleType char*;

  string char*;
  normalizedString char*;
  token char*;
  Name char*;
  NMTOKEN char*;
  NCName char*;
  ID char*;
  IDREF char*;
  language char*;
  anyURI char*;

  QName xml_schema::qname*;
}
```

For more information about the mapping of the 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` option, they are evaluated first. This allows you to selectively override any of the predefined 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 built-in XML Schema types `string` and `short` because they are handled by the predefined 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 callbacks 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: public gender_pskel
{
public:
  gender_pimpl ()
    : gender_pskel (&base_impl_)
  {
  }

  virtual ::gender
  post_gender ()
  {
    return post_string () == "male" ? male : female;
  }

private:
  xml_schema::string_pimpl base_impl_;
};

class person_pimpl: public 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: public 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_*()` callbacks 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_pimpl 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, 64-bit integer types, parser implementation reuse styles, and support for XML Schema polymorphism. Previous chapters assumed that the use of STL, iostream, C++ exceptions, and XML Schema validation were enabled. This chapter will discuss the changes in the Embedded C++/Parser programming model that result from the changes to these configuration parameters. 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 configuration 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 same encoding. The application encoding can either be UTF-8 (default) or ISO-8859-1. To select a particular encoding, configure the XSD/e runtime library accordingly and pass the `--char-encoding` option to the XSD/e compiler when translating your schemas.

When using ISO-8859-1 as the application encoding, XML documents being parsed may contain characters with Unicode values greater than 0xFF which are unrepresentable in the ISO-8859-1 encoding. By default, in such situations parsing will terminate with an error. However, you can suppress the error by providing a replacement character that should be used instead of unrepresentable characters, for example:

```
xml_schema::iso8859_1::unrep_char ('?');
```

To revert to the default behavior, set the replacement character to `'\0'`.

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()` callbacks 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 in the `xml_schema::document_pimpl` class 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 where 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 Parser Reuse

When one type in XML Schema inherits from another, it is often desirable to be able to reuse the parser implementation corresponding to the base type in the parser implementation corresponding to the derived type. XSD/e provides support for two parser reuse styles: the so-called *mixin* (generated when the `--reuse-style-mixin` option is specified) and *tiein* (generated by default) styles.

The compiler can also be instructed not to generate any support for parser reuse with the `--reuse-style-none` option. This is mainly useful to further reduce the generated code size when your vocabulary does not use inheritance or when you plan to implement each parser from scratch. Note also that the XSD/e runtime should be configured in accordance with the parser reuse style used in the generated code. The remainder of this section discusses the mixin and tiein parser reuse styles in more detail.

To provide concrete examples for each reuse style we will use the following schema fragment:

```
<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="age" type="xs:short"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="emplyee">
  <complexContent>
    <extension base="person">
      <xs:sequence>
        <xs:element name="position" type="xs:string"/>
        <xs:element name="salary" type="xs:unsignedLong"/>
      </xs:sequence>
    </extension>
  </complexContent>
</xs:complexType>
```

The mixin parser reuse style uses the C++ mixin idiom that relies on multiple and virtual inheritance. Because virtual inheritance can result in a significant object code size increase, this reuse style should be considered when such an overhead is acceptable and/or the vocabulary consists of only a handful of types. When the mixin reuse style is used, the generated parser skeletons use virtual inheritance, for example:

```
class person_pskel: public virtual parser_complex_content
{
  ...
};

class employee_pskel: public virtual person_pskel
{
  ...
};
```

When you implement the base parser you also need to use virtual inheritance. The derived parser is implemented by inheriting from both the derived parser skeleton and the base parser implementation (that is, *mixing in* the base parser implementation), for example:

```
class person_pimpl: public virtual person_pskel
{
  ...
};

class employee_pimpl: public employee_pskel,
                      public person_pimpl
{
  ...
};
```

The tiein parser reuse style uses delegation and normally results in a significantly smaller object code while being almost as convenient to use as the mixin style. When the tiein reuse style is used, the generated derived parser skeleton declares a constructor which allows you to specify the implementation of the base parser:

```
class person_pskel: public parser_complex_content
{
  ...
};

class employee_pskel: public person_pskel
{
public:
  employee_pskel (person_pskel* base_impl)

  ...
};
```

If you pass the implementation of the base parser to this constructor then the generated code will transparently forward all the callbacks corresponding to the base parser skeleton to this implementation. You can also pass `0` to this constructor in which case you will need to implement the derived parser from scratch. The following example shows how we could implement the `person` and `employee` parsers using the tiein style:

```
class person_pimpl: public person_pskel
{
  ...
};

class employee_pimpl: public employee_pskel
{
public:
  employee_pimpl ()
    : employee_pskel (&base_impl_)
  {
  }

  ...
```

```
private:
  person_pimpl base_impl_;
};
```

Note that you cannot use the *tied in* base parser instance (`base_impl_` in the above code) for parsing anything except the derived type.

The ability to override the base parser callbacks in the derived parser is also available in the tiein style. For example, the following code fragment shows how we can override the `age()` callback if we didn't like the implementation provided by the base parser:

```
class employee_pimpl: public employee_pskel
{
public:
  employee_pimpl ()
    : employee_pskel (&base_impl_)
  {
  }

  virtual void
  age (short a)
  {
    ...
  }

  ...

private:
  person_pimpl base_impl_;
};
```

In the above example the `age` element will be handled by `emplyee_pimpl` while the `first-name` and `last-name` callbacks will still go to `base_impl_`.

It is also possible to inherit from the base parser implementation instead of declaring it as a member variable. This can be useful if you need to access protected members in the base implementation or need to override a virtual function that is not part of the parser skeleton interface. Note, however, that in this case you will need to resolve a number of ambiguities with explicit qualifications or using-declarations. For example:

```
class person_pimpl: public person_pskel
{
  ...
protected:
  virtual person*
  create ()
  {
    return new person ();
```

```
  }
};

class employee_pimpl: public employee_pskel,
                      public person_pimpl
{
public:
  employee_pimpl ()
    : employee_pskel (static_cast<person_pimpl*> (this))
  {
  }

  // Resolve ambiguities.
  //
  using emplyee_pskel::parsers;

  ...

protected:
  virtual employee*
  create ()
  {
    return new employee ();
  }
};
```

5.7 Support for Polymorphism

By default the XSD/e compiler generates non-polymorphic code. If your vocabulary uses XML Schema polymorphism in the form of `xsi:type` and/or substitution groups, then you will need to configure the XSD/e runtime with support for polymorphism, compile your schemas with the `--generate-polymorphic` option to produce polymorphism-aware code, as well as pass `true` as the last argument to the `xml_schema::document_pimpl`'s constructors. If some of your schemas do not require support for polymorphism then you can compile them with the `--runtime-polymorphic` option and still use the XSD/e runtime configured with polymorphism support.

When using the polymorphism-aware generated code, you can specify several parsers for a single element by passing a parser map instead of an individual parser to the parser connection function for the element. One of the parsers will then be looked up and used depending on the `xsi:type` attribute value or an element name from a substitution group. Consider the following schema as an example:

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

  <xs:complexType name="person">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
```

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

  <!-- substitution group root -->
  <xs:element name="person" type="person"/>

  <xs:complexType name="superman">
    <xs:complexContent>
      <xs:extension base="person">
        <xs:attribute name="can-fly" type="xs:boolean"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:element name="superman"
              type="superman"
              substitutionGroup="person"/>

  <xs:complexType name="batman">
    <xs:complexContent>
      <xs:extension base="superman">
        <xs:attribute name="wing-span" type="xs:unsignedInt"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:element name="batman"
              type="batman"
              substitutionGroup="superman"/>

  <xs:complexType name="supermen">
    <xs:sequence>
      <xs:element ref="person" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

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

</xs:schema>
```

Conforming XML documents can use the `superman` and `batman` types in place of the `person` type either by specifying the type with the `xsi:type` attributes or by using the elements from the substitution group, for instance:

```
<supermen xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <person>
    <name>John Doe</name>
  </person>

  <superman can-fly="false">
```

```
    <name>James "007" Bond</name>
  </superman>

  <superman can-fly="true" wing-span="10" xsi:type="batman">
    <name>Bruce Wayne</name>
  </superman>

</supermen>
```

To print the data stored in such XML documents we can implement the parsers as follows:

```
class person_pimpl: public person_pskel
{
public:
  virtual void
  pre ()
  {
    cout << "starting to parse person" << endl;
  }

  virtual void
  name (const std::string& v)
  {
    cout << "name: " << v << endl;
  }

  virtual void
  post_person ()
  {
    cout << "finished parsing person" << endl;
  }
};

class superman_pimpl: public superman_pskel
{
public:
  superman_pimpl ()
    : superman_pskel (&base_impl_)
  {
  }

  virtual void
  pre ()
  {
    cout << "starting to parse superman" << endl;
  }

  virtual void
  can_fly (bool v)
  {
    cout << "can-fly: " << v << endl;
```

```
  }

  virtual void
  post_person ()
  {
    post_superman ();
  }

  virtual void
  post_superman ()
  {
    cout << "finished parsing superman" << endl
  }

private:
  person_pimpl base_impl_;
};

class batman_pimpl: public batman_pskel
{
public:
  batman_pimpl ()
    : batman_pskel (&base_impl_)
  {
  }

  virtual void
  pre ()
  {
    cout << "starting to parse batman" << endl;
  }

  virtual void
  wing_span (unsigned int v)
  {
    cout << "wing-span: " << v << endl;
  }

  virtual void
  post_person ()
  {
    post_superman ();
  }

  virtual void
  post_superman ()
  {
    post_batman ();
  }

  virtual void
```

```
  post_batman ()
  {
    cout << "finished parsing batman" << endl;
  }

private:
  superman_pimpl base_impl_;
};
```

Note that because the derived type parsers (`superman_pskel` and `batman_pskel`) are called via the `person_pskel` interface, we have to override the `post_person()` virtual function in `superman_pimpl` and `batman_pimpl` to call `post_superman()` and the `post_superman()` virtual function in `batman_pimpl` to call `post_batman()` (when the mixin parser reuse style is used it is not necessary to override `post_person()` in `batman_pimpl` since the suitable implementation is inherited from `superman_pimpl`).

The following code fragment shows how to connect the parsers together. Notice that for the `person` element in the `supermen_p` parser we specify a parser map instead of a specific parser and we pass `true` as the last argument to the document parser constructor to indicate that we are parsing potentially-polymorphic XML documents:

```
int
main (int argc, char* argv[])
{
  // Construct the parser.
  //
  xml_schema::string_pimpl string_p;
  xml_schema::boolean_pimpl boolean_p;
  xml_schema::unsigned_int_pimpl unsigned_int_p;

  person_pimpl person_p;
  superman_pimpl superman_p;
  batman_pimpl batman_p;

  xml_schema::parser_map_impl person_map (5); // 5 hashtable buckets
  supermen_pimpl supermen_p;

  person_p.parsers (string_p);
  superman_p.parsers (string_p, boolean_p);
  batman_p.parsers (string_p, boolean_p, unsigned_int_p);

  // Here we are specifying several parsers that can be used to
  // parse the person element.
  //
  person_map.insert (person_p);
  person_map.insert (superman_p);
  person_map.insert (batman_p);

  supermen_p.person_parser (person_map);
```

```
  // Parse the XML document. The last argument to the document's
  // constructor indicates that we are parsing polymorphic XML
  // documents.
  //
  xml_schema::document_pimpl doc_p (supermen_p, "supermen", true);

  supermen_p.pre ();
  doc_p.parse (argv[1]);
  supermen_p.post_supermen ();
}
```

When polymorphism-aware code is generated, each element's `*_parser()` function is overloaded to also accept an object of the `xml_schema::parser_map` type. For example, the `supermen_pskel` class from the above example looks like this:

```
class supermen_pskel: public xml_schema::parser_complex_content
{
public:

  ...

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

  // Individual element parsers.
  //
  void
  person_parser (person_pskel&);

  void
  person_parser (xml_schema::parser_map&);

  ...
};
```

Note that you can specify both the individual (static) parser and the parser map. The individual parser will be used when the static element type and the dynamic type of the object being parsed are the same. This is the case, for example, when there is no `xsi:type` attribute and the element hasn't been substituted. Because the individual parser for an element is cached and no map lookup is necessary, it makes sense to specify both the individual parser and the parser map when most of the objects being parsed are of the static type and optimal performance is important. The following code fragment shows how to change the above example to set both the individual parser and the parser map:

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

  // Here we are specifying several parsers that can be used to
  // parse the person element.
  //
  person_map.insert (superman_p);
  person_map.insert (batman_p);

  supermen_p.person_parser (person_p);
  supermen_p.person_parser (person_map);

  ...
}
```

The `xml_schema::parser_map` interface and the `xml_schema::parser_map_impl` default implementation are presented below:

```
namespace xml_schema
{
  class parser_map
  {
  public:
    virtual parser_base*
    find (const char* type) const = 0;

    virtual void
    reset () const = 0;
  };

  class parser_map_impl: public parser_map
  {
  public:
    parser_map_impl (size_t buckets);

    void
    insert (parser_base&);

    virtual parser_base*
    find (const char* type) const;

    virtual void
    reset () const;

  private:
    parser_map_impl (const parser_map_impl&);

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

```
    ...
  };
}
```

The `type` argument in the `find()` virtual function is the type name and namespace from the xsi:type attribute (the namespace prefix is resolved to the actual XML namespace) or the type of an element from the substitution group in the form `"<name> <namespace>"` with the space and the namespace part absent if the type does not have a namespace. You can obtain a parser's dynamic type in the same format using the `_dynamic_type()` function. The static type can be obtained by calling the static `_static_type()` function, for example `person_pskel::_static_type()`. Both functions return a C string (`const char*`) which is valid for as long as the application is running. The `reset()` virtual function is used to reset the parsers contained in the map (as opposed to resetting or clearing the map itself). For more information on parser resetting refer to Section 7.4, "Reusing Parsers after an Error". The following example shows how we can implement our own parser map using `std::map`:

```
#include <map>
#include <string>

class parser_map: public xml_schema::parser_map
{
public:
 void
 insert (xml_schema::parser_base& p)
 {
   map_[p._dynamic_type ()] = &p;
 }

 virtual xml_schema::parser_base*
 find (const char* type) const
 {
   map::const_iterator i = map_.find (type);
   return i != map_.end () ? i->second : 0;
 }

 virtual void
 reset () const
 {
   for (map::const_iterator i (map_.begin ()), e (map_.end ());
        i != e; ++i)
   {
     xml_schema::parser_base* p = i->second;
     p->_reset ();
   }
 }
```

```
private:
  typedef std::map<std::string, xml_schema::parser_base*> map;
  map map_;
};
```

The XSD/e runtime provides the default implementation for the `xml_schema::parser_map` interface, `xml_schema::parser_map_impl`, which is a hashmap. It requires that you specify the number of buckets it will contain and it does not support automatic table resizing. To obtain good performance the elements to buckets ratio should be between 0.7 and 0.9. It is also recommended to use prime numbers for bucket counts: 53, 97, 193, 389, 769, 1543, 3079, 6151, 12289, 24593, 49157, 98317, 196613, 393241.

If C++ exceptions are disabled (Section 5.3, "C++ Exceptions"), the `xml_schema::parser_map_impl` class has the following additional error querying API. It can be used to detect the out of memory errors after calls to the `parser_map_impl`'s constructor and `insert()` function.

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

    error
    _error () const;

    ...
  };
}
```

To support polymorphic parsing the XSD/e runtime and generated code maintain a number of hashmaps that contain substitution and, if XML Schema validation is enabled (Section 5.4, "XML Schema Validation"), inheritance information. Because the number of elements in these hashmaps depends on the schemas being compiled and thus is fairly static, these hashmaps do not perform automatic table resizing and instead the number of buckets is specified when the XSD/e runtime is configured. To obtain good performance the elements to buckets ratio in these hashmaps should be between 0.7 and 0.9. The recommended way to ensure this range is to add diagnostics code to your application as shown in the following example:

```
int
main ()
{
  // Check that the load in substitution and inheritance hashmaps
```

```
  // is not too high.
  //
#ifndef NDEBUG
  float load = xml_schema::parser_smap_elements ();
  load /= xml_schema::parser_smap_buckets ();

  if (load > 0.8)
  {
    cerr << "substitution hashmap load is " << load << endl;
    cerr << "time to increase XSDE_PARSER_SMAP_BUCKETS" << endl;
  }

  load = xml_schema::parser_imap_elements ();
  load /= xml_schema::parser_imap_buckets ();

  if (load > 0.8)
  {
    cerr << "inheritance hashmap load is " << load << endl;
    cerr << "time to increase XSDE_PARSER_IMAP_BUCKETS" << endl;
  }
#endif

  ...
}
```

Most of the code presented in this section is taken from the `polymorphism` example which can be found in the `examples/cxx/parser/` directory of the XSD/e distribution. Handling of `xsi:type` and substitution groups when used on root elements requires a number of special actions as shown in the `polyroot` example.

5.8 Custom Allocators

By default the XSD/e runtime and generated code use the standard operators `new` and `delete` to manage dynamic memory. However, it is possible to instead use custom allocator functions provided by your application. To achieve this, configure the XSD/e runtime library to use custom allocator functions as well as pass the `--custom-allocator` option to the XSD/e compiler when translating your schemas. The signatures of the custom allocator functions that should be provided by your application are listed below. Their semantics should be equivalent to the standard C `malloc()`, `realloc()`, and `free()` functions.

```
extern "C" void*
xsde_alloc (size_t);

extern "C" void*
xsde_realloc (void*, size_t);

extern "C" void
xsde_free (void*);
```

Note also that when custom allocators are enabled, any dynamically-allocated object of which the XSD/e runtime or generated code assume ownership should be allocated using the custom allocation function. Similarly, if your application assumes ownership of any dynamically-allocated object returned by the XSD/e runtime or the generated code, then such an object should be disposed of using the custom deallocation function. To help with these tasks the generated `xml_schema` namespace defines the following two helper functions and, if C++ exceptions are enabled, automatic pointer class:

```
namespace xml_schema
{
  void*
  alloc (size_t);

  void
  free (void*);

  struct alloc_guard
  {
    alloc_guard (void*);
    ~alloc_guard ();

    void*
    get () const;

    void
    release ();

  private:
    ...
  };
}
```

If C++ exceptions are disabled, these functions are equivalent to `xsde_alloc()` and `xsde_free()`. If exceptions are enabled, `xml_schema::alloc()` throws `std::bad_alloc` on memory allocation failure.

The following code fragment shows how to create and destroy a dynamically-allocated object with custom allocators when C++ exceptions are disabled:

```
void* v = xml_schema::alloc (sizeof (type));

if (v == 0)
{
  // Handle out of memory condition.
}

type* x = new (v) type (1, 2);

...
```

```
if (x)
{
  x->~type ();
  xml_schema::free (x);
}
```

The equivalent code fragment for configurations with C++ exceptions enabled is shown below:

```
xml_schema::alloc_guard g (xml_schema::alloc (sizeof (type)));
type* x = new (g.get ()) type (1, 2);
g.release ();

...

if (x)
{
  x->~type ();
  xml_schema::free (x);
}
```

5.9 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: public gender_pskel
{
public:
  gender_pimpl ()
    : gender_pskel (&base_impl_)
  {
  }

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

```
private:
  xml_schema::string_pimpl base_impl_;
};

class person_pimpl: public 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: public 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.
//
typedef xml_schema::parser_error error;
error e;
bool io_error = false;

do
{
  xml_schema::document_pimpl 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/e 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`	`std::string` or `char*` Section 5.1, "Standard Template Library"
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*` 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 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 class represents an XML qualified name. When the return type is xml_schema::qname*, the returned object is dynamically allocated with operator new and should eventually be deallocated with operator delete. With STL enabled, the qname type has the following interface:

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

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

```
namespace xml_schema
{
  class qname
  {
  public:
```

```
  // The default constructor creates an uninitialized object.
  // Use modifiers to initialize it.
  //
  qname ();

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

  void
  swap (qname&);

private:
  qname (const qname&);

  qname&
  operator= (const qname&);

public:
  char*
  prefix ();

  const char*
  prefix () const;

  void
  prefix (char*);

  void
  prefix_copy (const char*);

  char*
  prefix_detach ();

public:
  char*
  name ();

  const char*
  name () const;

  void
  name (char*);

  void
  name_copy (const char*);

  char*
  name_detach ();
};
```

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

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

The modifier functions and constructors that have the `char*` argument assume ownership of the passed strings which should be allocated with operator `new char[]` and will be deallocated with operator `delete[]` by the `qname` object. If you detach the underlying prefix or name strings, then they should eventually be deallocated with operator `delete[]`.

Finally, if both STL and 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 constructor creates an uninitialized object.
    // Use modifiers to initialize it.
    //
    qname ();

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

    void
    swap (qname&);

  private:
    qname (const qname&);

    qname&
    operator= (const qname&);

  public:
    char*
    prefix ();

    const char*
    prefix () const;
```

```
    void
    prefix (char*);

    error
    prefix_copy (const char*);

    char*
    prefix_detach ();

  public:
    char*
    name ();

    const char*
    name () const;

    void
    name (char*);

    error
    name_copy (const char*);

    char*
    name_detach ();
  };

  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*`. The returned object is dynamically allocated with operator `new` and should eventually be deallocated with operator `delete`. With STL and C++ exceptions enabled (Section 5.1, "Standard Template Library", Section 5.3, "C++ Exceptions"), the `string_sequence` type has the following interface:

```
namespace xml_schema
{
  class string_sequence
  {
  public:
    typedef std::string         value_type;
    typedef std::string*        pointer;
```

```
    typedef const std::string*  const_pointer;
    typedef std::string&        reference;
    typedef const std::string&  const_reference;

    typedef size_t              size_type;
    typedef ptrdiff_t           difference_type;

    typedef std::string*        iterator;
    typedef const std::string*  const_iterator;

  public:
    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;

    std::string&
    front ();

    const std::string&
    front () const;

    std::string&
    back ();

    const std::string&
    back () const;

    std::string&
    operator[] (size_t);

    const std::string&
```

```
    operator[] (size_t) const;

  public:
    bool
    empty () const;

    size_t
    size () const;

    size_t
    capacity () const;

    size_t
    max_size () const;

  public:
    void
    clear ();

    void
    pop_back ();

    iterator
    erase (iterator);

    void
    push_back (const std::string&);

    iterator
    insert (iterator, const std::string&);

    void
    reserve (size_t);
  };

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

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

When STL is enabled and C++ exceptions are disabled, the signatures of the `push_back()`, `insert()`, and `reserve()` functions change as follows:

```
namespace xml_schema
{
  class string_sequence
  {
  public:
    enum error
```

```
    {
      error_none,
      error_no_memory
    };

    ...

  public:
    error
    push_back (const std::string&);

    error
    insert (iterator, const std::string&);

    error
    insert (iterator, const std::string&, iterator& result);

    error
    reserve (size_t);
  };
}
```

When STL is disabled and C++ exceptions are enabled, the `string_sequence` type has the following interface:

```
namespace xml_schema
{
  class string_sequence
  {
  public:
    typedef char*         value_type;
    typedef char**        pointer;
    typedef const char**  const_pointer;
    typedef char*         reference;
    typedef const char*   const_reference;

    typedef size_t        size_type;
    typedef ptrdiff_t     difference_type;

    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;

    size_t
    capacity () const;

    size_t
    max_size () const;

  public:
    void
    clear ();

    void
```

```
    pop_back ();

    iterator
    erase (iterator);

    void
    push_back (char*);

    void
    push_back_copy (const char*);

    iterator
    insert (iterator, char*);

    void
    reserve (size_t);

    // 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&);
}
```

The `push_back()` and `insert()` functions assume ownership of the passed string which should be allocated with operator `new char[]` and will be deallocated with operator `delete[]` by the `string_sequence` object. These two functions free the passed object if the reallocation of the underlying sequence buffer fails. The `push_back_copy()` function makes a copy of the passed string. If you detach the underlying element string, then it should eventually be deallocated with operator `delete[]`.

When both STL and C++ exceptions are disabled, the signatures of the `push_back()`, `push_back_copy()`, `insert()`, and `reserve()` functions change as follows:

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

```
      error_no_memory
    };

    ...

  public:
    error
    push_back (char*);

    error
    push_back_copy (const char*);

    error
    insert (iterator, char*);

    error
    insert (iterator, char*, iterator& result);

    error
    reserve (size_t);
  };
}
```

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 eventually be deallocated with operator `delete`. With 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 eventually be 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 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_present () 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_present()` 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 a year, a day, and a 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:
    // The default constructor creates an uninitialized object.
    // Use modifiers to initialize it.
    //
    date ();

    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 a year, a month, a 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:
    // The default constructor creates an uninitialized object.
    // Use modifiers to initialize it.
    //
    date_time ();

    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:
    // The default constructor creates an uninitialized object.
    // Use modifiers to initialize it.
    //
    duration ();

    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:
    // The default constructor creates an uninitialized object.
    // Use modifiers to initialize it.
    //
    gday ();

    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:
    // The default constructor creates an uninitialized object.
    // Use modifiers to initialize it.
    //
    gmonth ();

    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 a day and 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_day: public time_zone
  {
  public:
    // The default constructor creates an uninitialized object.
    // Use modifiers to initialize it.
    //
    gmonth_day ();

    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:
    // The default constructor creates an uninitialized object.
    // Use modifiers to initialize it.
    //
    gyear ();

    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 a year and a 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:
    // The default constructor creates an uninitialized object.
    // Use modifiers to initialize it.
    //
    gyear_month ();

    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:
    // The default constructor creates an uninitialized object.
    // Use modifiers to initialize it.
    //
```

```
    time ();

    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_pimpl` type, the error handling mechanisms provided by the mapping, as well as how to reuse a parser after an error has occurred.

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 reporting 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_pimpl` 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_pimpl` class has the following interface:

```
namespace xml_schema
{
  class parser_base;

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

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

    document_pimpl (parser_base&,
                    const std::string& root_element_name);

    document_pimpl (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) 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 of 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.

When support for XML Schema polymorphism is enabled, the overloaded `document_pimpl` constructors have additional arguments which control polymorphic parsing. For more information refer to Section 5.7, "Support for Polymorphism".

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_pimpl`'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_pimpl 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_pimpl` 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::parser_xml` and `xml_schema::parser_schema` exceptions, respectively. These two exceptions derive from `xml_schema::parser_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::parser_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 parser_exception: public std::exception
  {
  public:
    unsigned long
    line () const;

    unsigned long
    column () const;

    virtual const char*
    text () const = 0;

    ...
  };

  std::ostream&
```

```
  operator<< (std::ostream&, const parser_exception&);


  typedef <implementation-details> parser_xml_error;

  class parser_xml: public parser_exception
  {
  public:
    parser_xml_error
    code () const;

    virtual const char*
    text () const;

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

    ...
  };


  typedef <implementation-details> parser_schema_error;

  class parser_schema: public parser_exception
  {
  public:
    parser_schema_error
    code () const;

    virtual const char*
    text () const;

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

    ...
  };
}
```

The `parser_xml_error` and `parser_schema_error` are implementation-specific error code types. The `operator<<` defined for the `parser_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::parser_exception& e)
  {
    cout << argv[1] << ":" << e.line () << ":" << e.column ()
         << ": error: " << e.text () << endl;
    return 1;
  }
}
```

Finally, for reporting application errors from parsing callbacks, 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_pimpl` parser have the following member function for querying the error status:

```
xml_schema::parser_error
_error () const;
```

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

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

    sys_error (value);

    operator value () const;

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

```
  ...
};

typedef <implementation-details> parser_xml_error;
typedef <implementation-details> parser_schema_error;

class parser_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
  //
  parser_xml_error
  xml_code () const;

  const char*
  xml_text () const;
```

```
    // schema
    //
    parser_schema_error
    schema_code () const;

    const char*
    schema_text () const;

    // app
    //
    int
    app_code () const;

    ...
  };
}
```

The `parser_xml_error` and `parser_schema_error` are implementation-specific error code types. The `parser_error` class incorporates four categories of errors which you can query 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::parser_error;
  parser_error e;

  do
  {
    xml_schema::document_pimpl 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 parser_error::sys:
      {
        cerr << argv[1] << ": error: " << e.sys_text () << endl;
        break;
      }
    case parser_error::xml:
      {
        cerr << argv[1] << ":" << e.line () << ":" << e.column ()
             << ": error: " << e.xml_text () << endl;
        break;
      }
    case parser_error::schema:
      {
        cerr << argv[1] << ":" << e.line () << ":" << e.column ()
             << ": error: " << e.schema_text () << endl;
        break;
      }
    case parser_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 `_app_error()` function inside a parser callback. 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: public person_pskel
{
public:
  virtual void
  age (short a)
  {
    if (a < 18)
      _app_error (1);
  }

  ...
};
```

You can also set a system error by calling the `_sys_error()` function inside a parser callback. This function has one argument of type `xml_schema::sys_error` which was presented above. For example:

```
class person_pimpl: public person_pskel
{
public:
  virtual void
  pre ()
  {
    p_ = new person ();

    if (p_ == 0)
      _sys_error (xml_schema::sys_error::no_memory);
  }

  ...

private:
  person* p_;
};
```

7.4 Reusing Parsers after an Error

After a successful execution a parser returns into the initial state and can be used to parse another document without any extra actions. On the other hand, if an error occurred during parsing and you would like to reuse the parser to parse another document, you need to explicitly reset it into the initial state as shown in the following code fragment:

```
int
main ()
{
  ...
```

```
  std::vector<std::string> files = ...

  xml_schema::document_pimpl doc_p (people_p, "people");

  for (size_t i = 0; i < files.size (); ++i)
  {
    try
    {
      people_p.pre ();
      doc_p.parse (files[i]);
      people_p.post_people ();
    }
    catch (const xml_schema::parser_exception&)
    {
      doc_p.reset ();
    }
  }
}
```

If you do not need to reuse parsers after an error for example because your application terminates or you create a new parser instance in such situations, then you can avoid generating parser reset code by specifying the `--suppress-reset` XSD/e compiler option.

Your individual parser implementations may also require extra actions in order to bring them into a usable state after an error. To accomplish this you can override the `_reset()` virtual function as shown below. Notice that when you override the `_reset()` function in your implementation, you should always call the base skeleton version to allow it to reset its state:

```
class person_pimpl: public person_pskel
{
public:
  virtual void
  pre ()
  {
    p_ = new person ();
  }

  virtual void
  _reset ()
  {
    person_pskel::_reset ();
    delete p_;
    p_ = 0;
  }

  ...

private:
  person* p_;
};
```

Note also that the `_reset()` mechanism is used only when an error has occurred. To make sure that your parser implementations arrive at the initial state during successful execution, use the initialization (`pre()` and `_pre()`) and finalization (`post_*()` and `_post()`) callbacks.

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	
complex type, complex content extension	
complex type, complex content restriction	
list	
Facets	
length	String-based types.
minLength	String-based types.
maxLength	String-based types.
pattern	String-based types.

whiteSpace	String-based types.
enumeration	String-based types.
minExclusive	Integer and floating-point types.
minInclusive	Integer and floating-point types.
maxExclusive	Integer and floating-point types.
maxInclusive	Integer and floating-point types.
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	