Embedded C++/Serializer Mapping

Getting Started Guide

Table of Contents

Preface

About This Document

The goal of this document is to provide you with an understanding of the C++/Serializer 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/serializer/` 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++/Serializer mapping. Furthermore, the archives may already have answers to some of your questions.

1 Introduction

Welcome to CodeSynthesis XSD/e and the Embedded C++/Serializer mapping. XSD/e is a validating XML parser/serializer generator for mobile and embedded systems. Embedded C++/Serializer is a W3C XML Schema to C++ mapping that represents an XML vocabulary as a set of serializer skeletons which you can implement to perform XML serialization as required by your application logic.

1.1 Mapping Overview

The Embedded C++/Serializer mapping provides event-driven, stream-oriented XML serialization, 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 serializers are 2-10 times faster than general-purpose validating XML serializers while at the same time maintaining extremely low static and dynamic memory footprints. For example, a validating serializer executable can be as small as 60KB 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.

The next chapter shows how to create a simple application that uses the Embedded C++/Serializer mapping to validate and serialize simple data to an XML document. The following chapters describe the Embedded C++/Serializer mapping in more detail.

1.2 Benefits

Traditional XML serialization APIs such as Document Object Model (DOM) or XML Writer 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 produced by the application logic to the text encoding used in XML.
- Resulting applications are hard to debug, change, and maintain.

In contrast, statically-typed, vocabulary-specific serializer skeletons produced by the Embedded C++/Serializer mapping use native data types (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 serializer skeletons allow you to operate in your domain terms instead of the generic elements, attributes, and text. Automatic code generation frees you for more interesting tasks (such as doing something useful with the information that needs to be stored in XML) and minimizes the effort needed to adapt your applications to changes in the document structure. To summarize, the C++/Serializer mapping has the following key advantages over generic XML serialization APIs:

- **Ease of use.** The generated code hides all the complexity associated with recreating the document structure, maintaining the state, and converting the data from types suitable for manipulation by the application logic to the text representation used in XML.
- **Natural representation.** The generated serializer skeletons implement serializer callbacks as virtual functions with names corresponding to elements and attributes in XML. As a result, you serialize the data using your domain vocabulary instead of generic elements, attributes, and text.
- **Concise code.** With a separate serializer skeleton for each XML Schema type, the applica-

tion implementation is simpler and thus easier to read and understand.
- **Safety.** The data is passed by serializer callbacks as statically typed objects. The serializer 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 serializer skeletons use native data types and combine validation and data-to-text conversion in a single step. This makes them much more efficient than traditional architectures with separate stages for validation and data conversion.

2 Hello World Example

In this chapter we will examine how to create a very simple XML document using the XSD/e-generated C++/Serializer skeletons. All the code presented in this chapter is based on the `hello` example which can be found in the `examples/cxx/serializer/` directory of the XSD/e distribution.

2.1 Writing Schema Definition

First, we need to get an idea about the structure of the XML document that we are going to create. The sample XML that we will try to produce with our Hello application looks like this:

```
<hello>

  <greeting>Hello</greeting>

  <name>sun</name>
  <name>earth</name>
  <name>world</name>

</hello>
```

Now we can create an XML Schema definition that describes the above XML and save it into `hello.xsd`:

```
<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 the sample XML document above. The `hello` type is defined as a sequence of the nested `greeting` and `name` elements. Note that the term sequence in XML Schema means that elements should appear in a particular order as opposed to appearing multiple times. The `name` element has its `maxOccurs` property set to `unbounded` which means it can appear multiple times in an XML document. Finally, the globally-defined `hello` element prescribes the root element for our vocabulary. For an easily-approachable introduction to XML Schema refer to XML Schema Part 0: Primer.

The above schema is a specification of our 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++ serializer skeletons.

2.2 Translating Schema Definition to C++

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

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

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

```
class hello_sskel
{
public:
  // Serializer callbacks. Override them in your implementation.
  //
  virtual void
  pre ();

  virtual std::string
  greeting () = 0;

  virtual bool
  name_next () = 0;

  virtual std::string
  name () = 0;

  virtual void
  post ();

  // Serializer construction API.
  //
  void
  greeting_serializer (xml_schema::string_sskel&);

  void
  name_serializer (xml_schema::string_sskel&);

  void
  serializers (xml_schema::string_sskel& /* greeting */,
               xml_schema::string_sskel& /* name */);

private:
  ...
};
```

The first five member functions shown above are called serializer callbacks. You would normally override them in your implementation of the serializer. 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 serialized. You would normally use this function to initialize data structures, such as iterators, which will be used during serialization. As we will see in subsequent chapters, there is also a way to pass an argument to this function which may be useful if you are serializing an in-memory data structure to XML. The default implementation of the initialization callback does nothing.

The `post()` function is a finalization callback. It is called when serialization of the element is completed. If necessary, you can use this function to perform cleanups of data structures initialized in `pre()` or during serialization. The default implementation of the finalization callback also does nothing.

The `greeting()` and `name()` functions are called when the `greeting` and `name` elements are about to be serialized and the values for these elements need to be provided. Because the `name` element can be repeated several times (note the `maxOccurs="unbounded"` attribute in the schema), the serializer skeleton also has the `name_next()` function which is called before `name()` to check if another `name` element needs to be serialized.

The last three functions are for connecting serializers to each other. For example, there is a predefined serializer for built-in XML Schema type `string` in the XSD/e runtime. We will be using it to serialize the values 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 create our sample XML document. The first step is to implement the serializer:

```
#include <string>
#include <vector>
#include "hello-sskel.hxx"

struct hello_simpl: hello_sskel
{
  hello_simpl ()
  {
    names_.push_back ("sun");
    names_.push_back ("moon");
    names_.push_back ("world");
  }

  virtual void
  pre ()
  {
    i_ = names_.begin ();
  }

  virtual std::string
  greeting ()
  {
    return "Hello";
  }

  virtual bool
  name_next ()
  {
    return i_ != names_.end ();
  }

  virtual std::string
  name ()
  {
    return *i_++;
  }

private:
  typedef std::vector<std::string> names;

  names names_;
  names::iterator i_;
};
```

We use the `hello_simpl`'s constructor to initialize a vector of names. Then, in the `pre()` initialization callback, we initialize an iterator to point to the beginning of the names vector. The `greeting` callback simply returns the string representing our greeting. The `name_next()` callback checks if we reached the end of the names vector and returns `false` if that's the case. The `name()` callback returns the next name from the names vector and advances the iterator. Note that `name()` is not called if `name_next()` returned false. Finally, we left `post()` with the default implementations since we don't have anything to cleanup.

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

```
#include <iostream>

using namespace std;

int
main ()
{
  try
  {
    // Construct the serializer.
    //
    xml_schema::string_simpl string_s;
    hello_simpl hello_s;

    hello_s.greeting_serializer (string_s);
    hello_s.name_serializer (string_s);

    // Create the XML document.
    //
    xml_schema::document_simpl doc_s (hello_s, "hello");

    hello_s.pre ();
    doc_s.serialize (cout);
    hello_s.post ();
  }
  catch (const xml_schema::serializer_exception& e)
  {
    cerr << "error: " << e.text () << endl;
    return 1;
  }
}
```

The first part of this code snippet instantiates individual serializers and assembles them into a complete vocabulary serializer. `xml_schema::string_simpl` is an implementation of a serializer for built-in XML Schema type `string`. It is provided by the XSD/e runtime along with serializers for other built-in types (for more information on the built-in serializers see Chapter 7, "Built-In XML Schema Type Serializers"). We use `string_simpl` to serialize the `greeting` and `name` elements as indicated by the calls to `greeting_serializer()` and `name_serializer()`.

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

The final piece is the calls to `pre()`, `serialize()`, and `post()`. The call to `serialize()` performs the actual XML serialization with the result written to `std::cout`. The calls to `pre()` and `post()` make sure that the serializer for the root element can perform proper initialization and cleanup.

2.4 Compiling and Running

After saving all the parts from the previous section in `driver.cxx`, we are ready to compile and run our first application. On UNIX this can be done with the following commands:

```
$ c++ -I.../libxsde -c driver.cxx hello-sskel.cxx
$ c++ -o driver driver.o hello-sskel.o .../libxsde/xsde/libxsde.a
$ ./driver
<hello>
  <greeting>Hello</greeting>
  <name>sun</name>
  <name>moon</name>
  <name>world</name>
</hello>
```

Here `.../libxsde` represents the path to the `libxsde` directory in the XSD/e distribution.

We can also test XML Schema validation. What can "forget" to add any names to the vector so that `name_next()` returns `false` on the first call:

```
struct hello_simpl: hello_sskel
{
  hello_simpl ()
  {
    /*
    names_.push_back ("sun");
    names_.push_back ("moon");
    names_.push_back ("world");
    */
  }
  ...
};
```

This will violate our vocabulary specification which requires at least one `name` element to be present. If we make the above change and recompile our application, we will get the following output:

```
$ ./driver
error: expected element not encountered
```

3 Serializer Skeletons

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

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

```
<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 could look like this:

```
<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 serializer skeletons being generated: `gender_sskel`, `person_sskel`, and `people_sskel`. We are going to examine and implement each of them in the subsequent sections.

In the previous chapter we used pre-initialized, static data to create an XML document. In this chapter we will use the standard input (`std::cin`) as the source of data. This approach reflects a common design theme where the data to be serialized is computed on the fly instead of being stored in, for example, an in-memory object model. The next chapter will examine mechanisms provided by the C++/Serializer mapping for serializing in-memory object models.

3.1 Implementing the Gender Serializer

The generated `gender_sskel` serializer skeleton looks like this:

```
class gender_sskel: public virtual xml_schema::string_sskel
{
public:
  // Serializer callbacks. Override them in your implementation.
  //
  virtual void
  pre ();

  virtual void
  post ();
};
```

Notice that `gender_sskel` inherits from `xml_schema::string_sskel` which is a serializer 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 serializer skeletons follow: if a type in XML Schema inherits from another then there will be an equivalent inheritance between the corresponding serializer skeleton classes.

The `pre` and `post` callbacks should look familiar from the previous chapter. Let’s now implement this serializer. Our implementation will simply query the gender value from the standard input stream (`std::cin`):

```
#include <string>
#include <iostream>

using namespace std;

class gender_simpl: public gender_sskel,
                    public xml_schema::string_simpl
{
public:
  virtual void
  pre ()
  {
    string g;
    cerr << "gender (male/female): ";
    getline (cin, g);
    string_simpl::pre (g);
  }
};
```

While the code is quite short, there is a lot going on. First, notice that we are inheriting from `gender_sskel` *and* from `xml_schema::string_simpl`. We’ve encountered `xml_schema::string_simpl` already; it is an implementation of the `xml_schema::string_sskel` serializer skeleton for built-in XML Schema type `string`.

This is another common theme in the C++/Serializer programming model: reusing implementations of the base serializers in the derived ones with the C++ mixin idiom. In our case, `string_simpl` will do all the dirty work of serializing the data which we pass to it with the call to `string_simpl::pre()`.

In case you are curious, here are the definitions for `xml_schema::string_sskel` and `xml_schema::string_simpl`:

```
namespace xml_schema
{
  class string_sskel: public virtual serializer_simple_content
  {
  public:
    virtual void
    pre (const std::string&) = 0;
  };

  class string_simpl: public virtual string_sskel
  {
  public:
    virtual void
    pre (const std::string&);

    virtual void
    _serialize_content ();
  protected:
    std::string value_;
  };
}
```

There are two new pieces in this code that we haven't seen yet. Those are the `xml_schema::serializer_simple_content` class and the `_serialize_content()` function. The `serializer_simple_content` class is defined in the XSD/e runtime and is a base class for all serializer 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 `xml_schema::serializer_complex_content` class which corresponds to the complex content mode (types with nested elements, for example, `person` from `people.xsd`).

The `_serialize_content()` function is a low-level serializer callback that is called to perform actual content serialization (that is to output text or nested elements). There is also the `_serialize_attributes()` callback which is called to serialize attributes in complex types. You will seldom need to use these callbacks directly. Using implementations for the built-in serializers provided by the XSD/e runtime is usually a simpler and more convenient alternative.

Another bit of information that is useful to know about is the `_pre()` and `_post` serialization callbacks. Remember we talked about `pre()` and `post()` callbacks in the previous chapter? The `_pre()` and `_post` have very similar but somewhat different roles. As a result, each serializer skeleton has four special callbacks:

```
  virtual void
  pre ();

  virtual void
  _pre ();

  virtual void
  _post ();

  virtual void
  post ();
```

`pre()` and `_pre()` are initialization callbacks. They get called in that order before a new instance of the type is about to be serialized. The difference between `pre()` and `_pre()` is conventional: `pre()` can be completely overridden by a derived serializer. The derived serializer 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()` are finalization callbacks with exactly the same semantics: `post()` can be completely overridden by the derived serializer while the original `_post()` should always be called.

At this point you might be wondering why some `pre()` callbacks, for example `string_sskel::pre()`, have an argument with which they receive the data they need to serialize while others, for example `gender_sskel::pre()`, have no such argument. This is a valid concern and it will be addressed in the next chapter.

3.2 Implementing the Person Serializer

The generated `person_sskel` serializer skeleton looks like this:

```
class person_sskel: public virtual xml_schema::serializer_complex_content
{
public:
  // Serializer callbacks. Override them in your implementation.
  //
  virtual void
  pre ();

  virtual std::string
  first_name () = 0;

  virtual std::string
  last_name () = 0;

  virtual void
  gender ();

  virtual short
  age () = 0;

  virtual void
  post ();

  // Serializer construction API.
  //
  void
  first_name_serializer (xml_schema::string_sskel&);

  void
  last_name_serializer (xml_schema::string_sskel&);

  void
  gender_serializer (gender_sskel&);

  void
  age_serializer (xml_schema::short_sskel&);
  void
  serializers (xml_schema::string_sskel& /* first-name */,
               xml_schema::string_sskel& /* last-name */,
               gender_sskel&             /* gender */,
               xml_schema::short_sskel&  /* age */);
};
```

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

```
class person_simpl: public person_sskel
{
public:
  virtual string
  first_name ()
  {
    string fn;
    cerr << "first name: ";
    getline (cin, fn);
    return fn;
  }

  virtual std::string
  last_name ()
  {
    string ln;
    cerr << "last name: ";
    getline (cin, ln);
    return ln;
  }

  virtual short
  age ()
  {
    short a;
    cerr << "age: ";
    cin >> a;
    return a;
  }
};
```

Notice that we didn’t need to override the `gender()` callback because all the work is done by `gender_simpl`.

3.3 Implementing the People Serializer

The generated `people_sskel` serializer skeleton looks like this:

```
class people_sskel: public virtual xml_schema::serializer_complex_content
{
public:
  // Serializer callbacks. Override them in your implementation.
  //
  virtual void
  pre ();

  virtual bool
  person_next () = 0;

  virtual void
  person ();

  virtual void
  post ();

  // Serializer construction API.
  //
  void
  person_serializer (person_sskel&);

  void
  serializers (person_sskel& /* person */);
};
```

The `person_next()` callback will be called before serializing each `person` element. Our implementation of `person_next()` asks the user whether to serialize another person record:

```
class people_simpl: public people_sskel
{
public:
  virtual bool
  person_next ()
  {
    string s;
    cerr << "serialize another person record (y/n): ";
    cin >> ws; // Skip leading whitespaces.
    getline (cin, s);
    return s == "y";
  }
};
```

Now it is time to put everything together.

3.4 Connecting the Serializers Together

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

```
xml_schema::short_simpl short_s;
xml_schema::string_simpl string_s;

gender_simpl gender_s;
person_simpl person_s;
people_simpl people_s;
```

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 serializers that come with the XSD/e runtime to serialize these types. The next step is to connect all the individual serializers. We do this with the help of functions defined in the serializer skeletons and marked with the "Serializer Construction API" comment. One way to do it is to connect each individual serializers by calling the `*_serializer()` functions:

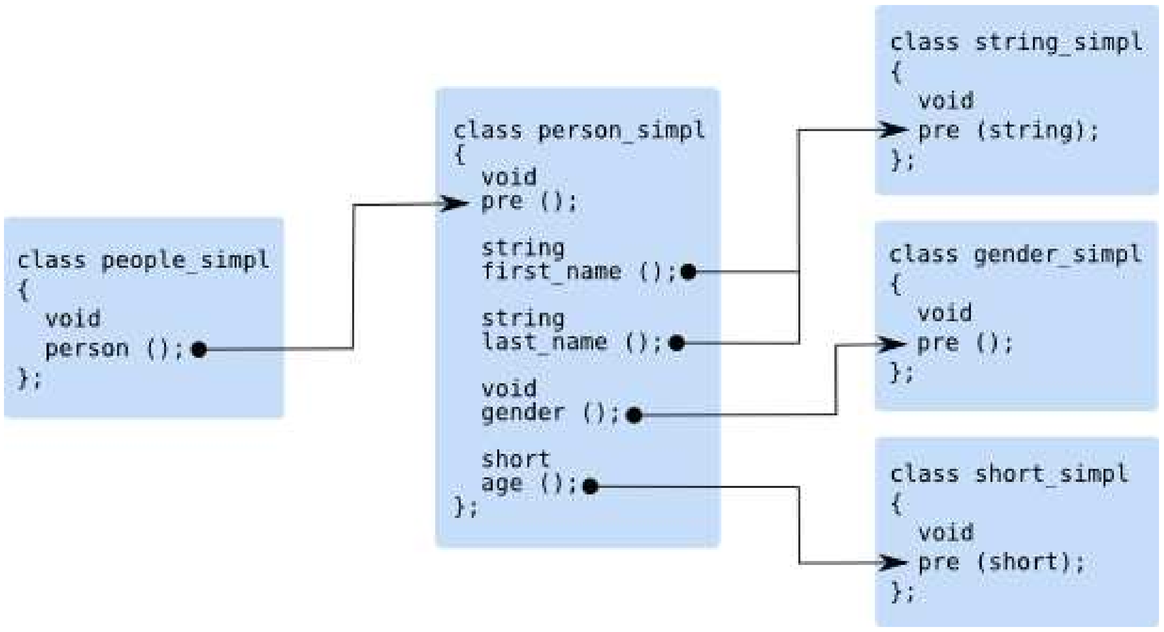
```
person_s.first_name_serializer (string_s);
person_s.last_name_serializer (string_s);
person_s.gender_serializer (gender_s);
person_s.age_serializer (short_s);

people_s.person_serializer (person_s);
```

You might be wondering what happens if you do not provide a serializer by not calling one of the `*_serializer()` functions. In that case the corresponding XML fragment will be skipped.

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

```
person_s.serializers (string_s, string_s, gender_s, short_s);
people_s.serializers (person_s);
```

The following figure illustrates the resulting connections. Notice the correspondence between return types of element callbacks and argument types of the `pre()` functions that are connected by the arrows.



The last step is the construction of the document serializer and invocation of the complete serializer to produce an XML document:

```
xml_schema::document_simpl doc_s (people_s, "people");

std::ostringstream os;

people_s.pre ();
doc_s.serialize (os);
people_s.post ();

cout << os.str ();
```

Note that we first serialize the document into an `std::ostringstream` object and then write the result to the standard output stream. This is done to prevent the input prompts and output XML from interleaving. However, writing XML directly to `std::cout` in this example is a great way to observe the moments in the XML document construction process at which serializer callbacks are being called.

Let’s consider `xml_schema::document_simpl` 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_simpl
  {
  public:
    document_simpl (xml_schema::serializer_base&,
                    const std::string& root_element_name);

    document_simpl (xml_schema::serializer_base&,
                    const std::string& root_element_namespace,
                    const std::string& root_element_name);

    void
    serialize (std::ostream&);
  };
}
```

`xml_schema::document_simpl` is a root serializer for the vocabulary. The first argument to its constructors is the serializer for the type of the root element (`people_simpl` in our case). Because a type serializer 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_simpl`'s constructors.

There is also a number of overloaded `serialize()` function defined in the `document_simpl` class. At the moment we are only interested in the version that writes XML to a standard output stream. For more information on the `xml_schema::document_simpl` class refer to Chapter 8, "Document Serializer and Error Handling".

Let's now consider a step-by-step list of actions that happen as we serialize the following sample XML document:

```
<people>
  <person>
    <first-name>John</first-name>
    <last-name>Doe</last-name>
    <gender>male</gender>
    <age>32</age>
  </person>
</people>
```

1. `people_s.pre()` is called from `main()`. We did not provide any implementation for this callback so this call is a no-op.
2. `doc_s.serialize(os)` is called from `main()`. The document serializer writes out the `<people>` opening tag and calls `_pre()` on the root element type serializer (`people_s`) which is also a no-op. Serialization is delegated to `people_s`.
3. The `people_s` serializer calls `person_next()` to determine if another `person` element needs to be serialized. Our implementation ask the user (who answers `"y"`) and returns `true`.
4. The `people_s` serializer calls `person()` which is a no-op. It then calls `pre()` on `person_s` (no-op), writes out the `<person>` opening tag, and calls `_pre()` on `person_s` (no-op). Serialization is delegated to `person_s`.
5. The `person_s` serializer calls `first_name()` that returns a first name which it asks the

user to enter. `person_s` then calls `pre()` on `string_s` and passes the name returned from `first_name()` as its argument. It then writes out the `<first-name>` opening tag and calls `_pre()` on `string_s`. Serialization is delegated to `string_s`.

6. The `_serialize_content()` callback is called on `string_s` which writes out the string passed to it in the `pre()` call.
7. Control is returned to `person_s` which calls `_post()` on `string_s`, writes out the `</first-name>` closing tag, and calls `post()` on `string_s`.
8. Steps analogous to 5-7 are performed for the `last-name`, `gender`, and `age` elements.
9. Control is returned to `people_s` which calls `_post()` on `person_s` (no-op), writes out the `</person>` closing tag, and calls `post()` on `person_s` (no-op).
10. The `people_s` serializer calls `person_next()` to determine if another `person` element needs to be serialized. Our implementation ask the user (who answers `"n"`) and returns `false`.
11. Control is returned to `doc_s` which calls `_post()` on `people_s` (no-op) and writes out the `</people>` closing tag.
12. Control is returned to `main()` which calls `post()` on `people_s` (no-op).

4 Type Maps

There are many useful things you can do inside serializer callbacks as they are right now. There are, however, times when you want to propagate some information from one serializer to another or from the caller of the serializer. One common task that would greatly benefit from such a possibility is serializing a tree-like in-memory object model to XML. During execution, each individual serializer would be responsible for disaggregating and serializing a specific portion of the tree and delegating the rest to its sub-serializers.

In this chapter we will discuss the mechanisms offered by the C++/Serializer mapping for passing information between individual serializers and see how to use them to serialize a sample object model for 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 serializer is responsible for which part of the object model, it is not exactly clear how, for example, `person_simpl` will pass `gender` to `gender_simpl`. You might have noticed that `string_simpl` manages to receive its value from the `first_name()` callback. 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 serializers. More precisely, for each type defined in XML Schema, you can tell the compiler two things. First, the argument type of the `pre()` callback in the serializer skeleton generated for this type. And, second, the return type for callbacks corresponding to elements and attributes of this type. For example, for XML Schema type `gender` we can specify the argument type for `pre()` in the `gender_sskel` skeleton and the return type for the `gender()` callback in the `person_sskel` skeleton. As you might have guessed, the generated code will then pass the return value from an element or attribute callback (`person_sskel::gender()` in our case) to the `pre()` callback of the corresponding serializer skeleton (`gender_sskel::pre()` in our case).

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 more 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-serializer --type-map people.map people.xsd
```

If we now look at the generated `people-sskel.hxx`, we will see the following changes in the `gender_sskel` and `person_sskel` skeletons:

```
#include "people.hxx"

class gender_sskel: public virtual xml_schema::string_sskel
{
  virtual void
  pre (::gender) = 0;

  ...
};

class person_sskel: public virtual xml_schema::serializer_complex_content
{
  virtual ::gender
  gender () = 0;

  ...
};
```

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 argument types of `pre ()` callbacks in serializer skeletons corresponding to XML Schema types as well as return 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 element and attribute callbacks corresponding to this schema type. Optional <*`cxx-arg-type`*> is an argument type for the `pre()` callback in the serializer skeleton for this schema type. If not specified, it defaults to `const` <*`cxx-ret-type`*>&. <*`file-name`*> is a file name either in the `" "` or `< >` format and is added with the `#include` directive to the generated code. For example:

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

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

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

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

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

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

The compiler has a number of 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;

  base64Binary "const xml_schema::buffer*"
               "const xml_schema::buffer*";

  hexBinary "const xml_schema::buffer*"
            "const 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 6.1, "Standard Template Library"), the following mapping is used for the string-based XML Schema built-in types:

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

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

  QName xml_schema::qname;

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

Otherwise, a C string-based mapping is used:

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

  QName "const xml_schema::qname*"
        "const xml_schema::qname*";
  NMTOKENS "const xml_schema::string_sequence*"
           "const xml_schema::string_sequence*";

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

For more information about the mapping of the built-in XML Schema types to C++ types refer to Chapter 7, "Built-In XML Schema Type Serializers". 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 serializer 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 Serializer 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 "const ::person&" "const ::person&";
people "const ::people&" "const ::people&";
```

A few things to note about this type map. We decided to pass the `person` and `people` objects by constant references in order to avoid unnecessary copying. We can do this because we know that our object model is present for the duration of serialization. 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 serializers:

```
$ xsde cxx-serializer --type-map people.map people.xsd
```

Here is the implementation of our three serializers in full. One way to save typing when implementing your own serializers is to open the generated code and copy the signatures of serializer callbacks into your code.

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

const char* gender_strings[] = {"male", "female"};

class gender_simpl: public gender_sskel,
                    public xml_schema::string_simpl
{
public:
  virtual void
  pre (gender g)
  {
    string_simpl::pre (gender_strings[g]);
  }
};

class person_simpl: public person_sskel
{
public:
  virtual void
  pre (const person& p)
  {
    p_ = &p;
  }

  virtual std::string
  first_name ()
  {
    return p_->first ();
  }

  virtual std::string
  last_name ()
  {
    return p_->last ();
  }

  virtual ::gender
  gender ()
  {
    return p_->gender ();
  }

  virtual short
  age ()
  {
    return p_->age ();
  }

private:
  const person* p_;
};

class people_simpl: public people_sskel
{
public:
  virtual void
  pre (const people& p)
  {
    p_ = &p;
    i_ = p_->begin ();
  }

  virtual bool
  person_next ()
  {
    return i_ != p_->end ();
  }

  virtual const ::person&
  person ()
  {
    return *i_++;
  }

private:
  const people* p_;
  people::const_iterator i_;
};
```

This code fragment should look familiar by now. Just note that all the `pre()` callbacks now have arguments. Here is the implementation of the test driver for this example:

```
#include <iostream>

using namespace std;

int
main ()
{
  // Create a sample object model.
  //
  people ppl;

  ppl.push_back (person ("John", "Doe", male, 32));
  ppl.push_back (person ("Jane", "Doe", female, 28));
  // Construct the serializer.
  //
  xml_schema::short_simpl short_s;
  xml_schema::string_simpl string_s;

  gender_simpl gender_s;
  person_simpl person_s;
  people_simpl people_s;

  person_s.serializers (string_s, string_s, gender_s, short_s);
  people_s.serializers (person_s);

  // Create the XML document.
  //
  xml_schema::document_simpl doc_s (people_s, "people");

  people_s.pre (ppl);
  doc_s.serialize (cout);
  people_s.post ();
}
```

The serializer creation and assembly part is exactly the same as in the previous chapter. The serialization part is a bit different: `people_simpl::pre()` now has an argument which is the complete object model. Also we write the resulting XML directly to the standard output stream instead of first storing it in a string. We can now 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-sskel.cxx
$ c++ -o driver driver.o people-sskel.o .../libxsde/xsde/libxsde.a
$ ./driver
<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>
```

5 Serializer Callbacks

In previous chapters we have learned that for each attribute and element in a schema type there is a callback in a serializer skeleton with the same name and which optionally returns this element's or attribute's value. We've also seen that elements that can appear multiple times (`maxOccurs="unbounded"`) have an additional serializer callback in the form:

```
virtual bool
<name>_next ();
```

Where `<name>` stands for the element's name. In this chapter we will discuss other additional serializer callbacks that are generated for certain XML Schema constructs. We will also learn that besides elements and attributes, serializer callback can be generated for the `all`, `choice`, and `sequence` compositors as well as the `any` and `anyAttribute` wildcards.

When additional serializer callback are generated for elements and attributes, their names are derived from element's and attribute's names. Compositors and wildcards, on the other hand, do not have names and as a result the serializer callback names for these constructs are based on synthesized names in the form: `all` for the `all` compositor, `sequence`, `sequence1`, etc., for the `sequence` compositors, `choice`, `choice1`, etc., for the `choice` compositors, `any`, `any1`, etc., for the `any` wildcards, and `any_attribute`, `any_attribute1`, etc., for the `anyAttribute` wildcards. For example:

```
<xs:complexType name="coordinates">
  <xs:sequence maxOccurs="unbounded">
    <xs:element name="lat" type="xs:float"/>
    <xs:element name="lon" type="xs:float"/>
  </xs:sequence>
</xs:complexType>
```

The above schema fragment, when compiled, results in the following serializer skeleton:

```
class coordinates_sskel: public xml_schema::serializer_complex_content
{
public:
  virtual void
  pre ();

  virtual bool
  sequence_next ();

  virtual float
  lan () = 0;

  virtual float
  lon () = 0;
  virtual void
  post ();

  ...
};
```

5.1 Optional Callback

For elements, compositors, and element wildcards with the minimal occurrence constraint equals `0` (`minOccurs="0"`) and the maximum occurrence constraint equals `1` (`maxOccurs="1"`) as well as for optional attributes, the optional callback is generated in the form:

```
virtual bool
<name>_present ();
```

This callback is called before any other callbacks for this schema construct and if it returns `false` no further callback calls corresponding to this construct are made and the corresponding XML fragment is omitted. For example:

```
<xs:complexType name="name">
  <xs:sequence minOccurs="0">
    <xs:element name="first" type="xs:string"/>
    <xs:element name="initial" type="xs:string" minOccurs="0"/>
    <xs:element name="last" type="xs:string"/>
  </xs:sequence>
  <xs:attribute name="lang" type="xs:language"/>
</xs:complexType>
```

The above schema fragment, when compiled, results in the following serializer skeleton:

```
class name_sskel: public xml_schema::serializer_complex_content
{
public:
  virtual void
  pre ();

  virtual bool
  lang_present ();

  virtual std::string
  lang () = 0;

  virtual bool
  sequence_present ();

  virtual std::string
  first () = 0;

  virtual bool
  initial_present ();

  virtual std::string
  initial () = 0;

  virtual std::string
  last () = 0;

  virtual void
  post ();

  ...
};
```

5.2 Sequence Callback

For elements, compositors, and element wildcards with the the maximum occurrence constraint greater than `1` (for example, `maxOccurs="unbounded"`) the sequence callback is generated in the form:

```
virtual bool
<name>_next ();
```

This callback is called before each new item of the sequence is about to be serialized. Returning `false` from this callback indicates that no more items in the sequence need to be serialized. For example:

```
<xs:complexType name="names">
  <xs:sequence maxOccurs="unbounded">
    <xs:element name="first" type="xs:string"/>
    <xs:element name="last" type="xs:string"/>
    <xs:element name="pseudonym" type="xs:string" maxOccurs="3"/>
  </xs:sequence>
</xs:complexType>
```

The above schema fragment, when compiled, results in the following serializer skeleton:

```
class names_sskel: public xml_schema::serializer_complex_content
{
public:
  virtual void
  pre ();

  virtual bool
  sequence_next () = 0;

  virtual std::string
  first () = 0;
  virtual std::string
  last () = 0;

  virtual bool
  pseudonym_next () = 0;

  virtual std::string
  pseudonym () = 0;

  virtual void
  post ();
};
```

5.3 Choice Callback

The choice compositor allows an XML document to contain one of several element or compositor options. In the Embedded C++/Serializer mapping, these options are called *choice arms* and are identified by the *arm tags*. For example:

```
<xs:complexType name="name">
  <xs:choice>
    <xs:element name="full-name" type="xs:string"/>
    <xs:sequence>
      <xs:element name="first-name" type="xs:string"/>
      <xs:element name="last-name" type="xs:string"/>
    </xs:sequence>
  </xs:choice>
</xs:complexType>
```

The above schema fragment, when compiled, results in the following serializer skeleton:

```
class name_sskel: public xml_schema::serializer_complex_content
{
public:
  virtual void
  pre ();

  enum choice_arm_tag
  {
    full_name_tag,
    sequence_tag
  };

  virtual choice_arm_tag
  choice_arm () = 0;

  virtual std::string
  full_name () = 0;

  virtual std::string
  first_name () = 0;

  virtual std::string
  last_name () = 0;

  virtual void
  post ();
};
```

The arm tags enum name (`choice_arm_tag` above) is derived from the choice compositor name (that is, `choice`, `choice1`, etc.) by adding the `_arm_tag` suffix. The tag names themselves are derived from the corresponding elements, compositors, or element wildcards.

The choice compositor callback has a name in the form `choice_tag()` (or `choice1_tag()`, etc., for subsequent `choice` compositors in the type). It returns the arm tag which identifies the choice arm that should be serialized. For example, if a `name_sskel` implementation returns `full_name_tag` from the `choice_arm()` callback, then the first choice arm is chosen and the `full_name()` callback is then called. Otherwise the `first_name` and `last_name()` callbacks are called.

5.4 Element Wildcard Callbacks

An element wildcard allows an arbitrary element from the specified namespace list to be present in an XML instance. Element wildcards can have the same cardinality constraints as elements and, as as a result, the optional or sequence callbacks can be generated. For example:

```
<xs:complexType name="name">
  <xs:sequence>
    <xs:element name="first" type="xs:string"/>
    <xs:element name="last" type="xs:string"/>
    <xs:any namespace="##other" processContents="skip" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

The above schema fragment, when compiled, results in the following serializer skeleton:

```
class name_sskel: public xml_schema::serializer_complex_content
{
public:
  virtual void
  pre ();

  virtual std::string
  first () = 0;

  virtual std::string
  last () = 0;
  virtual bool
  any_present ();

  virtual void
  any (std::string& ns, std::string& name);

  virtual void
  serialize_any ();

  virtual void
  post ();
};
```

The `any()` callback is called to obtain the element name and namespace. If validation is enabled, the namespace is checked against the allowed list. Then an element with these name and namespace is created and the `serialize_any()` callback is called to allow you to serialize the element's attributes and content. There are two common ways to serialize a wildcard element. The first approach is to use a serializer implementation. This approach is shown in the `wildcard` example which is part of the XSD/e distribution. The other approach is to use the low-level XML serialization API that is available to every serializer implementation via the `xml_schema::serializer_base` base serializer:

```
namespace xml_schema
{
  class serializer_base
  {
  public:
    void
    _start_element (const char* name);

    void
    _start_element (const char* ns, const char* name);

    void
    _end_element ();

    void
    _start_attribute (const char* name);

    void
    _start_attribute (const char* ns, const char* name);

    void
    _end_attribute ();

    void
    _attribute (const char* name, const char* value);

    void
    _attribute (const char* ns, const char* name, const char* value);
    void
    _characters (const char*);

    void
    _characters (const char*, size_t);

    void
    _declare_namespace (const char* ns, const char* prefix);

    void
    _declare_default_namespace (const char* ns);

    void
    _clear_default_namespace ();
  };
}
```

The following example shows how we could implement the `name_sskel` skeleton using this approach:

```
class name_simpl: public name_sskel
{
public:
  virtual std::string
  first ()
  {
    return "John";
  }

  virtual ::std::string
  last ()
  {
    return "Doe";
  }

  virtual bool
  any_present ()
  {
    return true;
  }

  virtual void
  any (std::string& ns, std::string& name)
  {
    ns = "http://www.example.com/extension";
    name = "pseudonyms";
  }

  virtual void
  serialize_any ()
  {
    _attribute ("id", "jd");

    _start_element ("pseudonym");
    _characters ("Johnny Doer");
    _end_element ();

    _start_element ("pseudonym");
    _characters ("Johnty Doo");
    _end_element ();
  }
};
```

5.5 Attribute Wildcard Callbacks

An attribute wildcard allows an arbitrary number of attributes from the specified namespace list to be present in an XML instance. As a result, the serializer callbacks for an attribute wildcard resemble those of an element with `maxOccurs="unbounded"`. For example:

```
<xs:complexType name="name">
  <xs:sequence>
    <xs:element name="first" type="xs:string"/>
    <xs:element name="last" type="xs:string"/>
  </xs:sequence>
  <xs:anyAttribute namespace="##any" processContents="skip"/>
</xs:complexType>
```

The above schema fragment, when compiled, results in the following serializer skeleton:

```
class name_sskel: public xml_schema::serializer_complex_content
{
public:
  virtual void
  pre ();

  virtual bool
  any_attribute_next ();

  virtual void
  any_attribute (std::string& ns, std::string& name);

  virtual void
  serialize_any_attribute ();

  virtual std::string
  first () = 0;

  virtual std::string
  last () = 0;
  virtual void
  post ();
};
```

Every time the `any_attribute_next()` callback returns `true`, `any_attribute()` is called to obtain the attribute name and namespace. If validation is enabled, the namespace is checked against the allowed list. Then an attribute with these name and namespace is created and the `serialize_any_attribute()` callback is called to allow you to write the attribute value, for example using one of the serializer implementations (see the `wildcard` example on how to do it) or the low-level `_characters()` function (for more information about the low-level XML serialization API see the previous section). The following example show how we could implement the `name_sskel` skeleton using the latter approach:

```
class name_simpl: public name_sskel
{
public:
  virtual void
  pre ()
  {
    id_written_ = false;
  }

  virtual bool
  any_attribute_next ()
  {
    return !id_written_;
  }

  virtual void
  any_attribute (std::string& ns, std::string& name)
  {
    ns = "";
    name = "id";
  }

  virtual void
  serialize_any_attribute ()
  {
    _characters ("jd");
    id_written_ = true;
  }

  virtual std::string
  first ()
  {
    return "John";
  }

  virtual ::std::string
  last ()
  {
    return "Doe";
  }

private:
  bool id_written_;
};
```

6 Mapping Configuration

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

In order to enable or disable a particular feature, the corresponding 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.

The Embedded C++/Serializer mapping always expects character data supplied by the application to be in the UTF-8 encoding. The underlying XML serializer used by the Embedded C++/Serializer mapping produces the resulting XML in the UTF-8 encoding as well.

6.1 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 `const 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 the `first_name()` and `last_name()` callbacks from the person record example.

```
class person_sskel
{
public:
  virtual const char*
  first_name ();

  virtual const char*
  last_name ();

  ...
};
```

When STL is disabled, the serializer implementations for the string-based built-in XML Schema types can be instructed to release the string after serialization using operator `delete[]`. For more information on how to do this refer to Section 7.2, "String-Based Type Serializers".

6.2 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 `serialize()` function in the `xml_schema::document_simpl` class become unavailable:

```
void
serialize (std::ostream&);
```

See Section 8.1, "Document Serializer" for more information.

6.3 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 8.3, "Error Codes".

6.4 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 serialization performance and reduce footprint in cases when the data being serialized is known to be valid.

6.5 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.

6.6 A Minimal Example

The following example is a re-implementation of the person records example presented in Chapter 4, "Type Maps". It is intended to work without STL, iostream, and C++ exceptions. It can be found in the `examples/cxx/serializer/minimal/` directory of the XSD/e distribution. The `people.xsd` schema is compiled with the `--no-stl`, `--no-iostream`, and `--no-exceptions` options. The object model types in `people.hxx` have also been reimplemented in order not to use STL types:

```
#include <stddef.h> // size_t

enum gender
{
  male,
  female
};

struct person
{
  const char* first_name_;
  const char* last_name_;
  gender gender_;
  unsigned short age_;
};

struct people
{
  person* people_;
  size_t size_;
};
```

The following listing presents the implementation of serializer skeletons and the test driver in full:

```
#include <stdio.h>
#include "people-sskel.hxx"

const char* gender_strings[] = {"male", "female"};

class gender_simpl: public gender_sskel,
                    public xml_schema::string_simpl
{
public:
  virtual void
  pre (gender g)
  {
    string_simpl::pre (gender_strings[g]);
  }
};

class person_simpl: public person_sskel
{
public:
  virtual void
  pre (const person& p)
  {
    person_ = &p;
  }

  virtual const char*
  first_name ()
  {
    return person_->first_name_;
  }

  virtual const char*
  last_name ()
  {
    return person_->last_name_;
  }

  virtual ::gender
  gender ()
  {
    return person_->gender_;
  }

  virtual unsigned short
  age ()
  {
    return person_->age_;
  }

private:
  const person* person_;
};

class people_simpl: public people_sskel
{
public:
  virtual void
  pre (const people& p)
  {
    i_ = 0;
    people_ = &p;
  }

  virtual bool
  person_next ()
  {
    return i_ < people_->size_;
  }

  virtual const ::person&
  person ()
  {
    return people_->people_[i_++];
  }

private:
  size_t i_;
  const people* people_;
};

class writer: public xml_schema::writer
{
public:
  virtual bool
  write (const char* s, size_t n)
  {
    return fwrite (s, n, 1, stdout) == 1;
  }

  virtual bool
  flush ()
  {
    return fflush (stdout) == 0;
  }
};

int
main ()
{
  // Create a sample person list.
  //
  people p;

  p.size_ = 2;
  p.people_ = new person[p.size_];

  if (p.people_ == 0)
  {
    fprintf (stderr, "error: no memory\n");
    return 1;
  }

  p.people_[0].first_name_ = "John";
  p.people_[0].last_name_ = "Doe";
  p.people_[0].gender_ = male;
```

```
p.people_[0].age_ = 32;

p.people_[1].first_name_ = "Jane";
p.people_[1].last_name_ = "Doe";
p.people_[1].gender_ = female;
p.people_[1].age_ = 28;

// Construct the serializer.
//
xml_schema::unsigned_short_simpl unsigned_short_s;
xml_schema::string_simpl string_s;

gender_simpl gender_s;
person_simpl person_s;
people_simpl people_s;

person_s.serializers (string_s, string_s, gender_s, unsigned_short_s);
people_s.serializers (person_s);

// Serialize.
//
typedef xml_schema::serializer_error error;

error e;
writer w;

do
{
  xml_schema::document_simpl doc_s (people_s, "people");

  if (e = doc_s._error ())
    break;

  people_s.pre (p);

  if (e = people_s._error ())
    break;

  doc_s.serialize (w);

  if (e = doc_s._error ())
    break;

  people_s.post ();

  e = people_s._error ();

} while (false);

delete[] p.people_;
```

```
  // Handle errors.
  //
  if (e)
  {
    switch (e.type ())
    {
    case error::sys:
      {
        fprintf (stderr, "error: %s\n", e.sys_text ());
        break;
      }
    case error::xml:
      {
        fprintf (stderr, "error: %s\n", e.xml_text ());
        break;
      }
    case error::schema:
      {
        fprintf (stderr, "error: %s\n", e.schema_text ());
        break;
      }
    case error::app:
      {
        fprintf (stderr, "application error: %d\n", e.app_code ());
        break;
      }
    default:
      break;
    }

    return 1;
  }

  return 0;
}
```

7 Built-In XML Schema Type Serializers

The XSD/e runtime provides serializer 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 serializer implementations.

XML Schema type	Serializer implementation in the `xml_schema` namespace	Serializer argument type
anyType and anySimpleType types		
`anyType`	`any_type_simpl`	`void`

`anySimpleType`	`any_simple_type_simpl`	`void`
fixed-length integral types		
`byte`	`byte_simpl`	`signed char`
`unsignedByte`	`unsigned_byte_simpl`	`unsigned char`
`short`	`short_simpl`	`short`
`unsignedShort`	`unsigned_short_simpl`	`unsigned short`
`int`	`int_simpl`	`int`
`unsignedInt`	`unsigned_int_simpl`	`unsigned int`
`long`	`long_simpl`	`long long` or `long` Section 6.5, "64-bit Integer Type"
`unsignedLong`	`unsigned_long_simpl`	`unsigned long long` or `unsigned long` Section 6.5, "64-bit Integer Type"
arbitrary-length integral types		
`integer`	`integer_simpl`	`long`
`nonPositiveInteger`	`non_positive_integer_simpl`	`long`
`nonNegativeInteger`	`non_negative_integer_simpl`	`unsigned long`
`positiveInteger`	`positive_integer_simpl`	`unsigned long`
`negativeInteger`	`negative_integer_simpl`	`long`
boolean types		
`boolean`	`boolean_simpl`	`bool`
fixed-precision floating-point types		
`float`	`float_simpl`	`float` Section 7.1, "Floating-Point Type Serializers"
`double`	`double_simpl`	`double` Section 7.1, "Floating-Point Type Serializers"
arbitrary-precision floating-point types		
`decimal`	`decimal_simpl`	`double` Section 7.1, "Floating-Point Type Serializers"
string-based types		

`string`	`string_simpl`	`const std::string&` or `const char*` Section 7.2, "String-Based Type Serializers"
`normalizedString`	`normalized_string_simpl`	`const std::string&` or `const char*` Section 7.2, "String-Based Type Serializers"
`token`	`token_simpl`	`const std::string&` or `const char*` Section 7.2, "String-Based Type Serializers"
`Name`	`name_simpl`	`const std::string&` or `const char*` Section 7.2, "String-Based Type Serializers"
`NMTOKEN`	`nmtoken_simpl`	`const std::string&` or `const char*` Section 7.2, "String-Based Type Serializers"
`NCName`	`ncname_simpl`	`const std::string&` or `const char*` Section 7.2, "String-Based Type Serializers"
`language`	`language_simpl`	`const std::string&` or `const char*` Section 7.2, "String-Based Type Serializers"
qualified name		
`QName`	`qname_simpl`	`const xml_schema::qname&` or `const xml_schema::qname*` Section 7.3, "`QName` Serializer"
ID/IDREF types		
`ID`	`id_simpl`	`const std::string&` or `const char*` Section 7.2, "String-Based Type Serializers"
`IDREF`	`idref_simpl`	`const std::string&` or `const char*` Section 7.2, "String-Based Type Serializers"

list types		
`NMTOKENS`	`nmtokens_simpl`	`const xml_schema::string_sequence&` or `const xml_schema::string_sequence*` Section 7.4, "`NMTOKENS` and `IDREFS` Serializers"
`IDREFS`	`idrefs_simpl`	`const xml_schema::string_sequence&` or `const xml_schema::string_sequence*` Section 7.4, "`NMTOKENS` and `IDREFS` Serializers"
URI types		
`anyURI`	`uri_simpl`	`const std::string&` or `const char*` Section 7.2, "String-Based Type Serializers"
binary types		
`base64Binary`	`base64_binary_simpl`	`const xml_schema::buffer*` Section 7.5, "`base64Binary` and `hexBinary` Serializers"
`hexBinary`	`hex_binary_simpl`	`const xml_schema::buffer*` Section 7.5, "`base64Binary` and `hexBinary` Serializers"
date/time types		
`date`	`date_simpl`	`const xml_schema::date&` Section 7.7, "`date` Serializer"
`dateTime`	`date_time_simpl`	`const xml_schema::date_time&` Section 7.8, "`dateTime` Serializer"
`duration`	`duration_simpl`	`const xml_schema::duration&` Section 7.9, "`duration` Serializer"
`gDay`	`gday_simpl`	`const xml_schema::gday&` Section 7.10, "`gDay` Serializer"
`gMonth`	`gmonth_simpl`	`const xml_schema::gmonth&` Section 7.11, "`gMonth` Serializer"

gMonthDay	gmonth_day_simpl	const xml_schema::gmonth_day& Section 7.12, "gMonthDay Serializer"
gYear	gyear_simpl	const xml_schema::gyear& Section 7.13, "gYear Serializer"
gYearMonth	gyear_month_simpl	const xml_schema::gyear_month& Section 7.14, "gYearMonth Serializer"
time	time_simpl	const xml_schema::time& Section 7.15, "time Serializer"

7.1 Floating-Point Type Serializers

The serializer implementations for the `float`, `double`, and `decimal` built-in XML Schema types allow you to specify the resulting notation (fixed or scientific) as well as precision. This is done by passing the corresponding arguments to their constructors:

```
namespace xml_schema
{
  class float_simpl: public float_sskel
  {
    enum notation
    {
      notation_auto,
      notation_fixed,
      notation_scientific
    };

    float_simpl (notation = notation_auto,
                 unsigned int precision = FLT_DIG);

    virtual void
    pre (float);

    ...
  };

  class double_simpl: public double_sskel
  {
    enum notation
    {
      notation_auto,
      notation_fixed,
      notation_scientific
    };

    double_simpl (notation = notation_auto,
                  unsigned int precision = DBL_DIG);

    virtual void
    pre (double);

    ...
  };

  class decimal_simpl: public decimal_sskel
  {
    decimal_simpl (unsigned int precision = DBL_DIG);

    virtual void
    pre (double);

    ...
  };
}
```

By default the notation for the `float` and `double` types is automatically selected to produce the shortest representation. Note that the `decimal` values are always serialized in the fixed-point notation.

7.2 String-Based Type Serializers

When STL is enabled (Section 6.1, "Standard Template Library"), the serializer argument type for the `string`, `normalizedString`, `token`, `Name`, `NMTOKEN`, `NCName`, `ID`, `IDREF`, `language`, and `anyURI` built-in XML Schema types is `const std::string&`. When STL is disabled, the value is passed as a constant C-string: `const char*`. In this case, you can also instruct the serializer implementations for string-based types to release the string with operator `delete[]` by passing `true` to their constructors. For instance, using the person records example from the previous chapter:

```
class person_simpl: public person_sskel
{
public:
  virtual const char*
  first_name ()
  {
    char* r = new char[5];
    strcpy (r, "John");
    return r;
  }

  virtual const char*
  last_name ()
  {
    char* r = new char[4];
    strcpy (r, "Doe");
    return r;
  }

  ...
};

int
main ()
{
  // Construct the serializer.
  //
  xml_schema::unsigned_short_simpl unsigned_short_s;
  xml_schema::string_simpl string_s (true); // Release the string passed.

  gender_simpl gender_s;
  person_simpl person_s;
  people_simpl people_s;

  person_s.serializers (string_s, string_s, gender_s, unsigned_short_s);

  ...
}
```

7.3 `QName` Serializer

The argument type of the `qname_simpl` serializer implementation is either `const xml_schema::qname&` when STL is enabled (Section 6.1, "Standard Template Library") or `const xml_schema::qname*` when STL is disabled. The `qname` class represents an XML qualified name. When the argument type is `const xml_schema::qname*`, you can optionally instruct the serializer to release the `qname` object with operator `delete` by passing `true` to its constructor.

With STL enabled, the `qname` type has the following interface:

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

    void
    swap (qname&);

    const std::string&
    prefix () const;
    std::string&
    prefix ();

    void
    prefix (const std::string&);

    const std::string&
    name () const;

    std::string&
    name ();

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

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

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

When STL is disabled and C++ exceptions are enabled (Section 6.3, "C++ Exceptions"), the `qname` type has the following interface:

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

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

    void
    swap (qname&);

  private:
    qname (const qname&);

    qname&
    operator= (const qname&);

  public:
    char*
    prefix ();

    const char*
    prefix () const;

    void
    prefix (const char*);

    char*
    detach_prefix ();

    void
    attach_prefix (char*);

  public:
    char*
    name ();

    const char*
    name () const;

    void
    name (const char*);

    char*
    detach_name ();

    void
    attach_name (char*);
  };

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

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

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

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

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

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

    void
    swap (qname&);

  private:
    qname (const qname&);

    qname&
    operator= (const qname&);

  public:
    char*
    prefix ();

    const char*
    prefix () const;

    error
    prefix (const char*);

    char*
    detach_prefix ();

    void
    attach_prefix (char*);

  public:
    char*
    name ();

    const char*
    name () const;

    error
    name (const char*);
    char*
    detach_name ();

    void
    attach_name (char*);
  };

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

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

7.4 `NMTOKENS` and `IDREFS` Serializers

The argument type of the `nmtokens_simpl` and `idrefs_simpl` serializer implementations is either `const xml_schema::string_sequence&` when STL is enabled (Section 6.1, "Standard Template Library") or `const xml_schema::string_sequence*` when STL is disabled. When the argument type is `const xml_schema::string_sequence*`, you can optionally instruct these serializers to release the `string_sequence` object with operator `delete` by passing `true` to their constructors.

With STL enabled, the `string_sequence` type has the following interface:

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

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

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

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

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

When STL is disabled and C++ exceptions are enabled (Section 6.3, "C++ Exceptions"), the `string_sequence` type has the following interface:

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

    string_sequence ();

    void
    swap (string_sequence&);

  private:
    string_sequence (string_sequence&);

    string_sequence&
    operator= (string_sequence&);

  public:
    iterator
    begin ();

    const_iterator
    begin () const;

    iterator
    end ();

    const_iterator
    end () const;

    char*
    front ();

    const char*
    front () const;

    char*
    back ();

    const char*
    back () const;

    char*
    operator[] (size_t);

    const char*
    operator[] (size_t) const;

  public:
    bool
    empty () const;

    size_t
    size () const;

 public:
    void
    push_back (const char*);

    void
    push_back_attach (char*);

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

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

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

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

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

```
namespace xml_schema
{
  class string_sequence
  {
  public:
    enum error
    {
      error_none,
      error_no_memory
    };
    typedef char** iterator;
    typedef const char* const* const_iterator;

    string_sequence ();

    void
    swap (string_sequence&);

  private:
    string_sequence (string_sequence&);

    string_sequence&
    operator= (string_sequence&);

  public:
    iterator
    begin ();

    const_iterator
    begin () const;

    iterator
    end ();

    const_iterator
    end () const;

    char*
    front ();

    const char*
    front () const;

    char*
    back ();

    const char*
    back () const;

    char*
    operator[] (size_t);

    const char*
    operator[] (size_t) const;

  public:
    bool
    empty () const;

    size_t
    size () const;
  public:
    error
    push_back (const char*);

    error
    push_back_attach (char*);

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

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

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

7.5 `base64Binary` and `hexBinary` Serializers

The argument type of the `base64_binary_simpl` and `hex_binary_simpl` serializer implementations is `const xml_schema::buffer*`. You can optionally instruct these serializers to release the `buffer` object with operator `delete` by passing `true` to their constructors. With C++ exceptions enabled (Section 6.3, "C++ Exceptions"), the `buffer` type has the following interface:

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

  public:
    buffer ();

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

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

  private:
    buffer (const buffer&);

    buffer&
    operator= (const buffer&);

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

    void*
    detach ();

    void
    swap (buffer&);

  public:
    size_t
    capacity () const;

    bool
    capacity (size_t);

  public:
    size_t
    size () const;

    bool
    size (size_t);

  public:
    const char*
    data () const;

    char*
    data ();

    const char*
    begin () const;

    char*
    begin ();

    const char*
    end () const;

    char*
    end ();
  };

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

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

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

The `capacity()` and `size()` modifier functions return `true` if the underlying buffer has moved. The `bounds` exception is thrown if the constructor or `attach()` member function arguments violate the `(size <= capacity)` constraint.

If 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&);
}
```

7.6 Time Zone Representation

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

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

    bool
    zone () const;

    void
    zone_reset ();

    short
    zone_hours () const;

    void
    zone_hours (short);

    short
    zone_minutes () const;

    void
    zone_minutes (short);
  };

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

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

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

7.7 `date` Serializer

The argument type of the `date_simpl` serializer implementation is `const xml_schema::date&`. The `date` class represents year, day, and month with an optional time zone. Its interface is presented below. For more information on the base `xml_schema::time_zone` class refer to Section 7.6, "Time Zone Representation".

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

    int
    year () const;

    void
    year (int);

    unsigned short
    month () const;

    void
    month (unsigned short);

    unsigned short
    day () const;

    void
    day (unsigned short);
  };

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

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

7.8 `dateTime` Serializer

The argument type of the `date_time_simpl` serializer implementation is `const xml_schema::date_time&`. The `date_time` class represents year, month, day, hours, minutes, and seconds with an optional time zone. Its interface is presented below. For more information on the base `xml_schema::time_zone` class refer to Section 7.6, "Time Zone Representation".

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

    date_time (int year, unsigned short month, unsigned short day,
               unsigned short hours, unsigned short minutes,
               double seconds, short zone_hours, short zone_minutes);

    int
    year () const;

    void
    year (int);

    unsigned short
    month () const;

    void
    month (unsigned short);

    unsigned short
    day () const;

    void
    day (unsigned short);

    unsigned short
    hours () const;

    void
    hours (unsigned short);

    unsigned short
    minutes () const;

    void
    minutes (unsigned short);

    double
    seconds () const;

    void
    seconds (double);
  };
  bool
  operator== (const date_time&, const date_time&);

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

7.9 `duration` Serializer

The argument type of the `duration_simpl` serializer implementation is `const xml_schema::duration&`. The `duration` class represents a potentially negative duration in the form of years, months, days, hours, minutes, and seconds. Its interface is presented below.

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

    bool
    negative () const;

    void
    negative (bool);

    unsigned int
    years () const;

    void
    years (unsigned int);

    unsigned int
    months () const;

    void
    months (unsigned int);

    unsigned int
    days () const;

    void
    days (unsigned int);

    unsigned int
    hours () const;
    void
    hours (unsigned int);

    unsigned int
    minutes () const;

    void
    minutes (unsigned int);

    double
    seconds () const;

    void
    seconds (double);
  };

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

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

7.10 `gDay` Serializer

The argument type of the `gday_simpl` serializer implementation is `const xml_schema::gday&` The `gday` class 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 7.6, "Time Zone Representation".

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

    unsigned short
    day () const;

    void
    day (unsigned short);
  };

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

7.11 `gMonth` Serializer

The argument type of the `gmonth_simpl` serializer implementation is `const xml_schema::gmonth&`. The `gmonth` class 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 7.6, "Time Zone Representation".

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

    unsigned short
    month () const;

    void
    month (unsigned short);
  };

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

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

7.12 `gMonthDay` Serializer

The argument type of the `gmonth_day_simpl` serializer implementation is `const xml_schema::gmonth_day&`. The `gmonth_day` class represents day and month of the year with an optional time zone. Its interface is presented below. For more information on the base `xml_schema::time_zone` class refer to Section 7.6, "Time Zone Representation".

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

    unsigned short
    month () const;

    void
    month (unsigned short);

    unsigned short
    day () const;

    void
    day (unsigned short);
  };

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

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

7.13 `gYear` Serializer

The argument type of the `gyear_simpl` serializer implementation is `const xml_schema::gyear&`. The `gyear` class 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 7.6, "Time Zone Representation".

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

    int
    year () const;

    void
    year (int);
  };

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

7.14 `gYearMonth` Serializer

The argument type of the `gyear_month_simpl` serializer implementation is `const xml_schema::gyear_month&`. The `gyear_month` class represents year and month with an optional time zone. Its interface is presented below. For more information on the base `xml_schema::time_zone` class refer to Section 7.6, "Time Zone Representation".

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

    int
    year () const;

    void
    year (int);

    unsigned short
    month () const;

    void
    month (unsigned short);
  };

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

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

7.15 `time` Serializer

The argument type of the `time_simpl` serializer implementation is `const xml_schema::time&`. The `time` class 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 7.6, "Time Zone Representation".

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

    unsigned short
    hours () const;

    void
    hours (unsigned short);

    unsigned short
    minutes () const;

    void
    minutes (unsigned short);

    double
    seconds () const;

    void
    seconds (double);
  };

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

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

8 Document Serializer and Error Handling

In this chapter we will discuss the `xml_schema::document_simpl` type as well as the error handling mechanisms provided by the mapping in more detail.

There are four categories of errors that can result from running a serializer to produce an XML instance: system, xml, schema, and application. The system category contains memory allocation and input/output operation errors. The xml category is for XML serialization 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 serializer implementations to the caller of the serializer.

The C++/Serializer 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 6.3, "C++ Exceptions".

8.1 Document Serializer

The `xml_schema::document_simpl` serializer is a root serializer for the vocabulary. As mentioned in Section 3.4, "Connecting the Serializer Together", its interface varies depending on the mapping configuration (Chapter 6, "Mapping Configuration"). When STL, C++ exceptions, and the iostream library are enabled, the `xml_schema::document_simpl` class has the following interface:

```
namespace xml_schema
{
  class serializer_base;

  class writer
  {
  public:
    // The first write function is called to write a ’\0’-terminated
    // string. Its default implementation calls the second versions:
    // write (s, strlen (s)). These functions use exceptions to
    // indicate a write failure.
    //
    virtual void
    write (const char* s);

    virtual void
    write (const char* s, size_t n) = 0;

    virtual void
    flush () = 0;
  };

  class document_simpl
  {
  public:
    document_simpl (serializer_base&,
                    const char* root_element_name);

    document_simpl (serializer_base&,
                    const char* root_element_namespace,
                    const char* root_element_name);

    document_simpl (serializer_base&,
                    const std::string& root_element_name);

    document_simpl (serializer_base&,
                    const std::string& root_element_namespace,
                    const std::string& root_element_name);

  public:
    void
    add_prefix (const char* prefix, const char* namespace_);

    void
    add_default_prefix (const char* namespace_);

    void
    add_schema (const char* namespace_, const char* location);

    void
    add_no_namespace_schema (const char* location);

    void
    add_prefix (const std::string& prefix,
                const std::string& namespace_);

    void
    add_default_prefix (const std::string& namespace_);

    void
    add_schema (const std::string& namespace_,
                const std::string& location);

    void
    add_no_namespace_schema (const std::string& location);

  public:
    // Serialize to std::ostream. The std::ios_base::failure
    // exception is used to report io errors (badbit and failbit)
    // if C++ exceptions are enabled. Otherwise error codes are
    // used.
    //
    void
    serialize (std::ostream&);

  public:
    // Serialize by calling writer::write() and writer::flush() to
    // output XML.
    //
    void
    serialize (writer&);

    // Serialize by calling the write and flush functions. If the
    // unbounded write function is not provided, the bounded version
    // is called: write_bound_func (s, strlen (s)). user_data is
    // passed as a first argument to these functions. These functions
    // use exceptions to indicate a write failure.
```

```
    //
    typedef void (*write_func) (void*, const char*);
    typedef void (*write_bound_func) (void*, const char*, size_t);
    typedef void (*flush_func) (void*);

    void
    serialize (write_bound_func, flush_func, void* user_data);

    void
    serialize (write_func, write_bound_func, flush_func, void* user_data);

  public:
    // Low-level, genx-specific serialization. With this method
    // it is your responsibility to call genxStartDoc*() and
    // genxEndDocument().
    //
    void
    serialize (genxWriter);
  };
}
```

When the use of STL is disabled, the constructors, as well as the `add_prefix()` and `add_schema()` functions that use `std::string` in their signatures are not available. When the use of iostream is disabled, the `serialize()` functions that serializes to `std::ostream` is not available.

When C++ exceptions are disabled, the `write()` and `flush()` virtual functions in the `writer` interface as well as `write_func`, `write_bound_func`, and `flush_func` function pointers use `bool` return type for error reporting. These functions should return `true` if the operation was successful and `false` otherwise. The relevant parts in the `writer` and `document_simpl` interfaces change as follows:

```
namespace xml_schema
{
  class serializer_base;

  class writer
  {
  public:
    // The first write function is called to write a ’\0’-terminated
    // string. Its default implementation calls the second versions:
    // write (s, strlen (s)). These functions return true if the
    // operation was successful and false otherwise.
    //
    // indicate a write failure.
    //
    virtual bool
    write (const char* s);
    virtual bool
    write (const char* s, size_t n) = 0;

    virtual bool
    flush () = 0;
  };

  class document_simpl
  {
    ...

    // Serialize by calling the write and flush functions. If the
    // unbounded write function is not provided, the bounded version
    // is called: write_bound_func (s, strlen (s)). user_data is
    // passed as a first argument to these functions. These functions
    // return true if the operation was successful and false otherwise.
    //
    typedef bool (*write_func) (void*, const char*);
    typedef bool (*write_bound_func) (void*, const char*, size_t);
    typedef bool (*flush_func) (void*);

    ...

  public:
    const serializer_error&
    _error () const;
  };
}
```

For more information on error handling with C++ exceptions and error codes see Section 8.2, "Exceptions" and Section 8.3, "Error Codes" below.

The first argument to all overloaded constructors is the serializer for the type of the root element. The `serializer_base` class is the base type for all serializer skeletons. The second and third arguments to the `document_simpl`'s constructors are the root element's name and namespace.

The `add_prefix()` and `add_default_prefix()` functions allow you to establish custom prefixes for XML namespaces. If none is provided, and namespaces are used by your vocabulary, the serializer will automatically assign namespace prefixes in an implementation-specific manner. For example:

```
xml_schema::document_simpl doc_s (
  root_s,
  "http://www.example.com/example",
  "root");

doc_s.add_prefix ("ex", "http://www.example.com/example");
```

The resulting XML will have the following namespace declaration:

```
<ex:root xmlns:ex="http://www.example.com/example" ...>
  ...
</ex:root>
```

Similarly, the `add_schema()` and `add_no_namespace_schema()` functions allow you to embed schema location information for a particular namespace into resulting XML. The schema location information is placed into the `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` attributes. For example:

```
xml_schema::document_simpl doc_s (
  root_s,
  "http://www.example.com/example",
  "root");

doc_s.add_prefix ("ex", "http://www.example.com/example");
doc_s.add_schema ("http://www.example.com/example", "example.xsd");
```

The resulting XML will have the following namespace declaration:

```
<ex:root
  xmlns:ex="http://www.example.com/example"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.example.com/example example.xsd" ...>
  ...
</ex:root>
```

8.2 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::serializer_xml` and `xml_schema::serializer_schema` exceptions, respectively. These two exceptions derive from `xml_schema::serializer_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::serializer_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 serializer_exception: public std::exception
  {
```

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

    ...
  };

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


  typedef <implementation-details> serializer_xml_error;

  class serializer_xml: public serializer_exception
  {
  public:
    serializer_xml_error
    code () const;

    virtual const char*
    text () const;

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

    ...
  };


  typedef <implementation-details> serializer_schema_error;

  class serializer_schema: public serializer_exception
  {
  public:
    serializer_schema_error
    code () const;

    virtual const char*
    text () const;

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

    ...
  };
}
```

The `serializer_xml_error` and `serializer_schema_error` are implementation-specific error code types. The `operator<<` defined for the `serializer_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 ()
{
  try
  {
    // Serialize.
  }
  catch (const xml_schema::serializer_exception& e)
  {
    cout << "error: " << e.text () << endl;
    return 1;
  }
}
```

Finally, for reporting application errors from serializer callbacks, you can throw any exceptions of your choice. They are propagated to the caller of the serializer without any alterations.

8.3 Error Codes

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

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

To handle all possible error conditions, you will need to obtain the error status after calls to: the `document_simpl`'s constructor (it performs memory allocations which may fail), calls to `add_prefix()` and `add_schema()` functions if any, the call to the root serializer `pre()` callback, the call to the `serialize()` function, and, finally, the call to the root serializer `post()` callback. The definition of `xml_schema::serializer_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> serializer_xml_error;
  typedef <implementation-details> serializer_schema_error;

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

    error_type
    type () const;

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

    // system
    //
    sys_error
    sys_code () const;

    const char*
    sys_text () const;

    // xml
    //
    serializer_xml_error
    xml_code () const;

    const char*
    xml_text () const;
    // schema
    //
    serializer_schema_error
    schema_code () const;

    const char*
    schema_text () const;

    // app
    //
    int
    app_code () const;

    ...
  };
}
```

The `serializer_xml_error` and `serializer_schema_error` are implementation-specific error code types. The `serializer_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, "Serializer Skeletons".

```
int
main ()
{
  // Construct the serializer.
  //
  xml_schema::short_simpl short_s;
  xml_schema::string_simpl string_s;

  gender_simpl gender_s;
  person_simpl person_s;
  people_simpl people_s;

  person_s.serializers (string_s, string_s, gender_s, short_s);
  people_s.serializers (person_s);

  // Serialize.
  //
  using xml_schema::serializer_error;
  serializer_error e;

  do
  {
    xml_schema::document_simpl doc_s (people_s, "people");
    if (e = doc_s._error ())
      break;

    people_s.pre ();
    if (e = people_s._error ())
      break;

    doc_s.serialize (cout);
    if (e = doc_s._error ())
      break;

    people_s.post ();
    e = people_s._error ();

  } while (false);

  // Handle errors.
  //
  if (e)
  {
    switch (e.type ())
    {
    case serializer_error::sys:
      {
        cerr << "system error: " << e.sys_text () << endl;
        break;
      }
    case serializer_error::xml:
      {
        cerr << "xml error: " << e.xml_text () << endl;
        break;
      }
    case serializer_error::schema:
      {
        cerr << "schema error: " << e.schema_text () << endl;
        break;
      }
    case serializer_error::app:
      {
        cerr << "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 serializer 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_simpl: public person_sskel
{
public:
  virtual short
  age ()
  {
    short a = ...;

    if (a < 18)
      _app_error (1);

    return a;
  }
};
```

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

```
class person_simpl: public person_sskel
{
public:
  virtual const char*
  first_name ()
  {
    char* r = new char[5];

    if (r == 0)
    {
      _sys_error (xml_schema::sys_error::no_memory);
      return 0;
    }

    strcpy (r, "John");
    return r;
  }
};
```

Appendix A — Supported XML Schema Constructs

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

Construct	Notes
Structure	
element	

attribute	
any	
anyAttribute	
all	
sequence	
choice	
complex type, empty content	
complex type, mixed content	
complex type, simple content extension	
complex type, simple content restriction	Simple type facets are not validated.
complex type, complex content extension	
complex type, complex content restriction	
list	
Datatypes	
byte	
unsignedByte	
short	
unsignedShort	
int	
unsignedInt	
long	
unsignedLong	
integer	
nonPositiveInteger	
nonNegativeInteger	
positiveInteger	
negativeInteger	

boolean	
float	
double	
decimal	
string	
normalizedString	
token	
Name	
NMTOKEN	
NCName	
language	
anyURI	
ID	Identity constraint is not enforced.
IDREF	Identity constraint is not enforced.
NMTOKENS	
IDREFS	Identity constraint is not enforced.
QName	
base64Binary	
hexBinary	
date	
dateTime	
duration	
gDay	
gMonth	
gMonthDay	
gYear	
gYearMonth	

time	