# Embedded C++/Hybrid Mapping

# Getting Started Guide

Copyright © 2005-2011 CODE SYNTHESIS TOOLS CC

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, version 1.2; with no Invariant Sections, no Front-Cover Texts and no Back-Cover Texts.

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

# Table of Contents

Preface . . . . . 1
About This Document . . . . . 1
More Information . . . . . 1
1 Introduction . . . . . 1
1.1 Mapping Overview . . . . . 1
1.2 Benefits . . . . . 3
2 Hello World Example . . . . . 4
2.1 Writing XML Document and Schema . . . . . 4
2.2 Translating Schema to C++ . . . . . 5
2.3 Implementing Application Logic . . . . . 7
2.4 Compiling and Running . . . . . 8
2.5 Adding Serialization . . . . . 9
2.6 A Minimal Version . . . . . 12
3 Mapping Configuration . . . . . 17
3.1 Standard Template Library . . . . . 18
3.2 Input/Output Stream Library . . . . . 19
3.3 C++ Exceptions . . . . . 19
3.4 XML Schema Validation . . . . . 19
3.5 64-bit Integer Type . . . . . 20
3.6 Parser and Serializer Reuse . . . . . 20
3.7 Support for Polymorphism . . . . . 20
3.8 Custom Allocators . . . . . 23
4 Working with Object Models . . . . . 25
4.1 Namespaces . . . . . 29
4.2 Memory Management . . . . . 29
4.3 Enumerations . . . . . 32
4.4 Attributes and Elements . . . . . 33
4.5 Compositors . . . . . 47
4.6 Accessing the Object Model . . . . . 55
4.7 Modifying the Object Model . . . . . 56
4.8 Creating the Object Model from Scratch . . . . . 58
4.9 Customizing the Object Model . . . . . 60
4.10 Polymorphic Object Models . . . . . 67
5 Mapping for Built-In XML Schema Types . . . . . 72
5.1 Mapping for `QName` . . . . . 77
5.2 Mapping for `NMTOKENS` and `IDREFS` . . . . . 81
5.3 Mapping for `base64Binary` and `hexBinary` . . . . . 81
5.4 Time Zone Representation . . . . . 85
5.5 Mapping for `date` . . . . . 86
5.6 Mapping for `dateTime` . . . . . 87

5.7 Mapping for `duration` . . . . . 88
5.8 Mapping for `gDay` . . . . . 89
5.9 Mapping for `gMonth` . . . . . 90
5.10 Mapping for `gMonthDay` . . . . . 91
5.11 Mapping for `gYear` . . . . . 92
5.12 Mapping for `gYearMonth` . . . . . 92
5.13 Mapping for `time` . . . . . 93
5.14 Mapping for `anyType` . . . . . 94
6 Parsing and Serialization . . . . . 96
6.1 Customizing Parsers and Serializers . . . . . 99
7 Binary Representation . . . . . 104
7.1 CDR (Common Data Representation) . . . . . 105
7.2 XDR (eXternal Data Representation) . . . . . 106
7.3 Custom Representations . . . . . 108

# Preface

## About This Document

The goal of this document is to provide you with an understanding of the C++/Hybrid 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
- Embedded C++/Parser Mapping Getting Started Guide. The C++/Hybrid mapping uses C++/Parser for XML parsing.
- Embedded C++/Serializer Mapping Getting Started Guide. The C++/Hybrid mapping uses C++/Serializer for XML serialization.
- The `INSTALL` file in the XSD/e distribution provides build instructions for various platforms.
- The `examples/cxx/hybrid/` 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++/Hybrid mapping. Furthermore, the archives may already have answers to some of your questions.

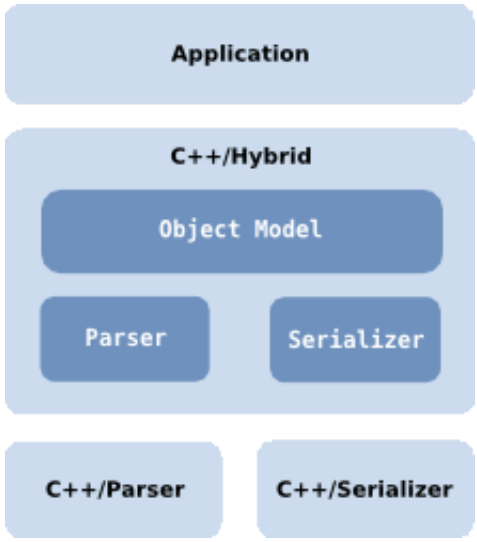
# 1 Introduction

Welcome to CodeSynthesis XSD/e and the Embedded C++/Hybrid mapping. XSD/e is a dependency-free W3C XML Schema to C++ compiler for mobile, embedded, and light-weight applications. Embedded C++/Hybrid is an XML Schema to C++ mapping that represents the data stored in XML as a light-weight, statically-typed, in-memory object model.

## 1.1 Mapping Overview

Based on a formal description of an XML vocabulary (schema), the C++/Hybrid mapping produces a tree-like data structure suitable for in-memory processing. The core of the mapping consists of C++ classes that constitute the object model and are derived from types defined in XML Schema. The C++/Hybrid mapping uses the APIs provided by the Embedded C++/Parser

and Embedded C++/Serializer mappings to perform validation and parsing of XML to the object model and validation and serialization of the object model to XML. The following diagram illustrates the high-level architecture of the C++/Hybrid mapping:




The use of well-defined APIs presented by the C++/Parser and C++/Serializer mappings for XML parsing and serialization allows a number of advanced techniques, for example, customization of parsing and serialization code, filtering of XML during parsing or object model during serialization, as well as the hybrid, partially event-driven, partially in-memory processing where the XML document is delivered to the application as parts of the object model. The last feature combines the ease and convenience of the in-memory processing model with the ability to minimize the use of RAM and process documents that would otherwise not fit into memory.

Besides reading from and writing to XML, the C++/Hybrid mapping also supports saving the object model to and loading it from a number of predefined as well as custom binary formats. Binary representations contain only the data without any meta information or markup. Consequently, saving to and loading from a binary format can be an order of magnitude faster as well as result in a much smaller application footprint compared to parsing and serializing the same data in XML. Furthermore, the resulting representation is normally several times smaller than the equivalent XML.

The Embedded C++/Hybrid mapping 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 parsing and serialization code is 2-10 times faster than general-purpose XML processors while at the same time maintaining extremely low static and dynamic memory footprints. For example, an executable that performs validating XML parsing and serialization can be as small as 150KB in size. The size can be further reduced by disabling support for parsing or serialization as well as 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 with the minimal use of C++ templates.

A typical application that uses the C++/Hybrid mapping for XML processing performs the following three steps: it first reads (parses) an XML document to an in-memory object model, it then performs some useful computations on that object model which may involve modification of the model, and finally it may write (serialize) the modified object model back to XML. The next chapter presents a simple application that performs these three steps. The following chapters describe the Embedded C++/Hybrid mapping in more detail.

## 1.2 Benefits

Traditional XML access APIs such as Document Object Model (DOM) or Simple API for XML (SAX) as well as general-purpose XML Schema validators have a number of drawbacks that make them less suitable for creating mobile and embedded XML processing applications. These drawbacks include:

- Generic representation of XML in terms of elements, attributes, and text forces an application developer to write a substantial amount of bridging code that identifies and transforms pieces of information encoded in XML to a representation more suitable for consumption by the application logic.
- String-based flow control defers error detection to runtime. It also reduces code readability and maintainability.
- Lack of type safety and inefficient use of resources due to the data being represented as text.
- Extra validation code that is not used by the application.
- Resulting applications are hard to debug, change, and maintain.

In contrast, a light-weight, statically-typed, vocabulary-specific object model produced by the Embedded C++/Hybrid mapping allows you to operate in your domain terms instead of the generic elements, attributes, and text. Native data types are used to store the XML data (for example, integers are stored as integers, not as text). Validation code is included only for XML Schema constructs that are used in the application. This results in efficient use of resources and compact object code.

Furthermore, static typing helps catch errors at compile-time rather than at run-time. Automatic code generation frees you for more interesting tasks (such as doing something useful with the information stored in the XML documents) and minimizes the effort needed to adapt your applications to changes in the document structure. To summarize, the C++/Hybrid object model has the following key advantages over generic XML access APIs:

- **Ease of use.** The generated code hides all the complexity associated with parsing and serializing XML. This includes navigating the structure and converting between the text representation and data types suitable for manipulation by the application logic.
- **Natural representation.** The object representation allows you to access the XML data using your domain vocabulary instead of generic elements, attributes, and text.
- **Concise code.** With the object representation the application implementation is simpler and thus easier to read and understand.
- **Safety.** The generated object model is statically typed and uses functions instead of strings to access the information. This helps catch programming errors at compile-time rather than at runtime.
- **Maintainability.** Automatic code generation minimizes the effort needed to adapt the application to changes in the document structure. With static typing, the C++ compiler can pin-point the places in the client code that need to be changed.
- **Efficiency.** If the application makes repetitive use of the data extracted from XML, then the C++/Hybrid object model is more efficient because the navigation is performed using function calls rather than string comparisons and the XML data is extracted only once. The runtime memory usage is also reduced due to more efficient data storage (for instance, storing numeric data as integers instead of strings) as well as the static knowledge of cardinality constraints.

  Furthermore, the generated XML parsing and serialization code combines validation and data-to-text conversion in a single step. This makes the generated code 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 parse, access, modify, and serialize a very simple XML document using the generated C++/Hybrid object model as well as the XML parser and serializer. The code presented in this chapter is based on the `hello` example which can be found in the `examples/cxx/hybrid/` directory of the XSD/e distribution.

## 2.1 Writing XML Document and Schema

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

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

  <greeting>Hello</greeting>

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

</hello>
```

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

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

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

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

</xs:schema>
```

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

The above schema is a specification of our XML vocabulary; it tells everybody what valid documents of our XML-based language should look like. The next step is to compile the schema to generate the object model and the parser.

## 2.2 Translating Schema to C++

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

```
$ xsde cxx-hybrid --generate-parser --generate-aggregate hello.xsd
```

This invocation of the XSD/e compiler produces three pairs of C++ files: `hello.hxx` and `hello.cxx`, `hello-pskel.hxx` and `hello-pskel.cxx`, as well as `hello-pimpl.hxx` and `hello-pimpl.cxx`. The first pair contains the object model classes. The second pair contains parser skeletons. Parser skeletons are generated by the C++/Parser mapping which is automatically invoked by C++/Hybrid. For now we can ignore parser skeletons except that we need to compile them and link the result to our application. The last pair of files contains parser implementations. They implement the parser skeletons to create and populate the object model types from XML data. The generation of parser skeletons and parser implementations is requested with the `--generate-parser` XSD/e compiler option.

You may be wondering what is the `--generate-aggregate` option for. This option instructs the XSD/e compiler to generate parser and, as we will see later, serializer aggregates. The generated parser implementation files mentioned above contain a separate parser implementation class for each type defined in XML Schema. These parser implementations need to be instantiated and connected before we can use them to parse an XML document. When you specify the `--generate-aggregate` option, the XSD/e compiler generates a class (in the parser implementation files), called parser aggregate, for each global element defined in the schema (you can also generate a parser aggregate for a type as well as control for which global elements parser aggregates are generated, see the XSD/e Compiler Command Line Manual for more information). A parser aggregate instantiates and connects all the necessary parser implementations needed to parse an XML document with a given root element. We will see how to use the parser aggregate for the `hello` root element in the next section.

The following code fragment is taken from `hello.hxx`; it shows what the C++ object model for our "Hello World" XML vocabulary looks like:

```
class hello
{
public:
  hello ();

  // greeting
  //
  const std::string&
  greeting () const;

  std::string&
  greeting ();

  void
  greeting (const std::string&);

  // name
  //
  typedef xml_schema::string_sequence name_sequence;
```

```
  typedef name_sequence::iterator name_iterator;
  typedef name_sequence::const_iterator name_const_iterator;

  const name_sequence&
  name () const;

  name_sequence&
  name ();

private:
  ...
};
```

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

It is also evident that the built-in XML Schema type `string` is mapped to `std::string`. The `string_sequence` class that is used in the `name_sequence` type definition has an interface similar to `std::vector`. The mapping between the built-in XML Schema types and C++ types is described in more detail in Chapter 5, "Mapping for Built-in XML Schema Types".

# 2.3 Implementing Application Logic

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

```
#include <iostream>

#include "hello.hxx"
#include "hello-pimpl.hxx"

using namespace std;

int
main (int argc, char* argv[])
{
  try
  {
    // Parse.
    //
    hello_paggr hello_p;
    xml_schema::document_pimpl doc_p (hello_p.root_parser (),
                                      hello_p.root_name ());
    hello_p.pre ();
    doc_p.parse (argv[1]);
```

```
    hello* h = hello_p.post ();

    // Print what we've got.
    //
    for (hello::name_const_iterator i = h->name ().begin ();
         i != h->name ().end ();
         ++i)
    {
      cout << h->greeting () << ", " << *i << "!" << endl;
    }

    delete h;
  }
  catch (const xml_schema::parser_exception& e)
  {
    cerr << argv[1] << ":" << e.line () << ":" << e.column ()
         << ": " << e.text () << endl;
    return 1;
  }
}
```

The first part of our application creates a document parser and parses the XML file specified in the command line to the object model. The `hello_paggr` class is the parser aggregate class we discussed earlier. Parsing is covered in more detail in Chapter 6, "Parsing and Serialization". The second part uses the returned object model to iterate over names and print a greeting line for each of them. We also catch and print the `xml_schema::parser_exception` exception in case something goes wrong.

## 2.4 Compiling and Running

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

```
$ c++ -I.../libxsde -c driver.cxx hello.cxx hello-pskel.cxx \
  hello-pimpl.cxx

$ c++ -o driver driver.o hello.o hello-pskel.o hello-pimpl.o \
  .../libxsde/xsde/libxsde.a

$ ./driver hello.xml
Hello, sun!
Hello, moon!
Hello, world!
```

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

We can also test the error handling. To test XML well-formedness checking, we can try to parse `hello.hxx`:

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

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

```
$ ./driver hello.xsd
hello.xsd:2:57: unexpected element encountered
```

## 2.5 Adding Serialization

While parsing and accessing the XML data may be everything you need, there are applications that require creating new or modifying existing XML documents. To request the generation of serialization support we will need to add the `--generate-serializer` option to our XSD/e compiler invocation:

```
$ xsde cxx-hybrid --generate-parser --generate-serializer \
  --generate-aggregate hello.xsd
```

This will result in two additional pairs of C++ files: `hello-sskel.hxx` and `hello-sskel.cxx`, as well as `hello-simpl.hxx` and `hello-simpl.cxx`. Similar to the parser files, the first pair contains serializer skeletons (generated by the C++/Serializer mapping) and the second pair contains serializer implementations as well as the serializer aggregate for the `hello` root element.

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

```
#include <iostream>

#include "hello.hxx"
#include "hello-pimpl.hxx"
#include "hello-simpl.hxx"

using namespace std;

int
main (int argc, char* argv[])
{
  try
  {
    // Parse.
    //
    hello_paggr hello_p;
    xml_schema::document_pimpl doc_p (hello_p.root_parser (),
                                      hello_p.root_name ());
```

```
    hello_p.pre ();
    doc_p.parse (argv[1]);
    hello* h = hello_p.post ();

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

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

    // Serialize the modified object model to XML.
    //
    hello_saggr hello_s;
    xml_schema::document_simpl doc_s (hello_s.root_serializer (),
                                      hello_s.root_name ());
    hello_s.pre (*h);
    doc_s.serialize (cout, xml_schema::document_simpl::pretty_print);
    hello_s.post ();

    delete h;
  }
  catch (const xml_schema::parser_exception& e)
  {
    cerr << argv[1] << ":" << e.line () << ":" << e.column ()
         << ": " << e.text () << endl;
    return 1;
  }
  catch (const xml_schema::serializer_exception& e)
  {
    cerr << "error: " << e.text () << endl;
    return 1;
  }
}
```

First, our application parses an XML document and obtains its object model as in the previous example. Then it changes the greeting string and adds another entry to the list of names. Finally, it creates a document serializer and serializes the object model back to XML. The `hello_saggr` class is the serializer aggregate class we discussed earlier.

The resulting XML is written to the standard output (`cout`) for us to inspect. We could have also written the result to a file or memory buffer by creating an instance of `std::ofstream` or `std::ostringstream` and passing it to `serialize()` instead of `cout`. The second argument in the call to `serialize()` is a flag that requests pretty-printing of the resulting XML document. You would normally specify this flag during testing to obtain easily-readable XML and remove it in production to get faster serialization and smaller documents. Serialization is covered in more detail in Chapter 6, "Parsing and Serialization".

If we now compile and run this application (don't forget to compile and link `hello-sskel.cxx` and `hello-simpl.cxx`), we will see the output as shown in the following listing:

```
<hello>
  <greeting>Hi</greeting>
  <name>sun</name>
  <name>moon</name>
  <name>world</name>
  <name>mars</name>
</hello>
```

We can also test XML Schema validation. We can "accidently" remove all the names from the object model by adding the following after: `push_back ("mars")`:

```
h->name ().clear ();
```

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 hello.xml
error: expected element not encountered
```

It is also possible to create and serialize an object model from scratch as shown in the following example. For this case we can remove the `--generate-parser` option since we don't need support for XML parsing.

```
#include <sstream>
#include <iostream>

#include "hello.hxx"
#include "hello-simpl.hxx"

using namespace std;

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

    hello::name_sequence& ns = h.name ();
    ns.push_back ("Jane");
    ns.push_back ("John");

    // Serialize the object model to XML.
```

```
    //
    hello_saggr hello_s;
    xml_schema::document_simpl doc_s (hello_s.root_serializer (),
                                      hello_s.root_name ());
    ostringstream ostr;

    hello_s.pre (h);
    doc_s.serialize (ostr, xml_schema::document_simpl::pretty_print);
    hello_s.post ();

    cout << ostr.str () << endl;
  }
  catch (const xml_schema::serializer_exception& e)
  {
    cerr << "error: " << e.text () << endl;
    return 1;
  }
}
```

In this example we used the generated default constructor to create an empty instance of type `hello`. We then set greeting and, to reduce typing, we obtained a reference to the name sequence which we used to add a few names. The serialization part is identical to the previous example except this time we first save the XML representation into a string. If we compile and run this program, it produces the following output:

```
<hello>
  <greeting>Hi</greeting>
  <name>Jane</name>
  <name>John</name>
</hello>
```

# 2.6 A Minimal Version

The previous sections showed a number of examples that relied on STL for strings, iostream of input/output and C++ exceptions for error handling. As was mentioned in the introduction and will be discussed in further detail in the next chapter, the C++/Hybrid mapping can be configured only to rely on the minimal subset of C++. In this section we will implement an example that parses, prints, modifies and serializes the object model without relying on STL, iostream, or C++ exceptions.

The first step is to instruct the XSD/e compiler not to use any of the above features in the generated code. You may also need to re-configure and rebuild the XSD/e runtime library (`libxsde.a`) to disable STL, iostream, and exceptions.

```
$ xsde cxx-hybrid --no-stl --no-iostream --no-exceptions \
  --generate-parser --generate-serializer --generate-aggregate \
  hello.xsd
```

If you now study the generated `hello.hxx` file, you will notice that the use of `std::string` type is replaced with `char*`. When STL is disabled, built-in XML Schema type `string` is mapped to a C string. The following listing presents the content of `driver.cxx` in full:

```
#include <stdio.h>

#include "people.hxx"

#include "people-pimpl.hxx"
#include "people-simpl.hxx"

using namespace std;

struct writer: xml_schema::writer
{
  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 (int argc, char* argv[])
{
  // Open the file or use STDIN.
  //
  FILE* f = fopen (argv[1], "rb");

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

  // Parse.
  //
  using xml_schema::parser_error;

  parser_error pe;
  bool io_error = false;
  hello* h = 0;

  do
  {
```

```
    hello_paggr hello_p;
    xml_schema::document_pimpl doc_p (hello_p.root_parser (),
                                      hello_p.root_name ());
    if (pe = doc_p._error ())
      break;

    hello_p.pre ();

    if (pe = hello_p._error ())
      break;

    char buf[4096];

    do
    {
      size_t s = fread (buf, 1, sizeof (buf), f);

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

      doc_p.parse (buf, s, feof (f) != 0);
      pe = doc_p._error ();

    } while (!pe && !feof (f));

    if (io_error || pe)
      break;

    h = hello_p.post ();
    pe = hello_p._error ();

  } while (false);

  fclose (f);

  // Handle parsing errors.
  //
  if (io_error)
  {
    fprintf (stderr, "%s: read failure\n", argc);
    return 1;
  }

  if (pe)
  {
    switch (pe.type ())
    {
    case parser_error::sys:
```

```
    {
      fprintf (stderr, "%s: %s\n", argc, pe.sys_text ());
      break;
    }
  case parser_error::xml:
    {
      fprintf (stderr, "%s:%lu:%lu: %s\n",
               argc, pe.line (), pe.column (), pe.xml_text ());
      break;
    }
  case parser_error::schema:
    {
      fprintf (stderr, "%s:%lu:%lu: %s\n",
               argc, pe.line (), pe.column (), pe.schema_text ());
      break;
    }
  default:
    break;
  }

  return 1;
}

// Print what we've got.
//
for (hello::name_const_iterator i = h->name ().begin ();
     i != h->name ().end ();
     ++i)
{
  printf ("%s, %s!\n", h->greeting (), *i);
}

using xml_schema::strdupx;

// Change the greeting phrase.
//
char* str = strdupx ("Hi");

if (str == 0)
{
  fprintf (stderr, "error: no memory\n");
  delete h;
  return 1;
}

h->greeting (str);

// Add another entry to the name sequence.
//
str = strdupx ("mars");
```

```
  if (str == 0)
  {
    fprintf (stderr, "error: no memory\n");
    delete h;
    return 1;
  }

  if (h->name ().push_back (str) != 0)
  {
    // The sequence has already freed str.
    //
    fprintf (stderr, "error: no memory\n");
    delete h;
    return 1;
  }

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

  serializer_error se;
  writer w;

  do
  {
    hello_saggr hello_s;
    xml_schema::document_simpl doc_s (hello_s.root_serializer (),
                                      hello_s.root_name ());
    if (se = doc_s._error ())
      break;

    hello_s.pre (*h);

    if (se = hello_s._error ())
      break;

    doc_s.serialize (w, xml_schema::document_simpl::pretty_print);

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

    hello_s.post ();

    se = hello_s._error ();

  } while (false);

  delete h;

  // Handle serializer errors.
  //
```

```
  if (se)
  {
    switch (se.type ())
    {
    case serializer_error::sys:
      {
        fprintf (stderr, "error: %s\n", se.sys_text ());
        break;
      }
    case serializer_error::xml:
      {
        fprintf (stderr, "error: %s\n", se.xml_text ());
        break;
      }
    case serializer_error::schema:
      {
        fprintf (stderr, "error: %s\n", se.schema_text ());
        break;
      }
    default:
      break;
    }

    return 1;
  }
}
```

The parsing and serialization parts of the above example got quite a bit more complex due to the lack of exceptions and iostream support. For more information on what's going on there, refer to Chapter 6, "Parsing and Serialization". On the other hand, the access and modification of the object model stayed relatively unchanged. The only noticeable change is the use of the `xml_schema::strdupx` function to create C strings from string literals. We have to use this function because the object model assumes ownership of the strings passed. We also cannot use the standard C `strdup` because the object model expects the strings to be allocated with C++ operator `new[]` while C `strdup` uses `malloc` (on most implementations operator `new` is implemented in terms of `malloc` so you can probably use `strdup` if you really want to).

# 3 Mapping Configuration

The Embedded C++/Hybrid mapping has a number of configuration parameters that determine the overall properties and behavior of the generated code, such as the use of Standard Template Library (STL), Input/Output Stream Library (iostream), C++ exceptions, XML Schema validation, 64-bit integer types, as well as parser and serializer implementation reuse styles. In the previous chapter we have already got an overview of the changes to the generated code that happen when we disable STL, iostream, and C++ exceptions. In this chapter we will discuss these and other configuration parameters in more detail.

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

While the XML documents can use various encodings, the C++/Hybrid object model always stores character data in the same encoding, called application encoding. The application encoding can either be UTF-8 (default) or ISO-8859-1. To select a particular encoding, configure the XSD/e runtime library accordingly and pass the `--char-encoding` option to the XSD/e compiler when translating your schemas.

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

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

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

The underlying XML parser used by the mapping includes built-in support for XML documents encoded in UTF-8, UTF-16, ISO-8859-1, and US-ASCII. Other encodings can be supported by providing application-specific decoder functions. The underlying XML serializer used by C++/Hybrid produces the resulting XML documents in the UTF-8 encoding.

# 3.1 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 (see Chapter 5, "Mapping for Built-In XML Schema Types") are mapped to C-style `char*` instead of `std::string`. In this configuration when you set an element or attribute value of a string-based type, the object model assumes ownership of the string and expects that it was allocated with operator `new[]`. To simplify creation of such strings from string literals, the generated code provides the `strdupx` and `strndupx` functions in the `xml_schema` namespace. These functions are similar to C `strdup` and `strndup` except that they use operator `new[]` instead of `malloc` to allocate the string:

```
namespace xml_schema
{
  char*
  strdupx (const char*);

  char*
  strndupx (const char*, size_t);
}
```

## 3.2 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, a number of overloaded `parse()` and `serialize()` functions in the document parser (`xml_schema::document_pimpl`) and document serializer (`xml_schema::document_simpl`) become unavailable. See Chapter 7, "Document Parser and Error Handling" in the Embedded C++/Parser Mapping Getting Started Guide and Chapter 8, "Document Serializer and Error Handling" in the Embedded C++/Serializer Mapping Getting Started Guide for details.

## 3.3 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 that may arise while parsing, serializing, and modifying the object model are indicated with error codes instead of exceptions. For more information on error handling during parsing, see Chapter 7, "Document Parser and Error Handling" in the Embedded C++/Parser Mapping Getting Started Guide. For more information on error handling during serialization, see Chapter 8, "Document Serializer and Error Handling" in the Embedded C++/Serializer Mapping Getting Started Guide. For more information on error handling in the object model, see Chapter 4, "Working with Object Models" below.

## 3.4 XML Schema Validation

By default, XML Schema validation is enabled during both parsing and serialization. To disable validation during parsing, you will need to configure the XSD/e runtime to disable support for validation in the C++/Parser mapping as well as pass the `--suppress-parser-val` option to the XSD/e compiler when translating your schemas. To disable validation during serialization, you will need to configure the XSD/e runtime to disable support for validation in the C++/Serializer mapping as well as pass the `--suppress-serializer-val` option to the XSD/e compiler when translating your schemas. If you are disabling validation during both parsing and serialization, you can use the `--suppress-validation` option instead of the two options

mentioned above.

Disabling XML Schema validation allows to further increase the parsing and serialization performance as well as reduce footprint in cases where the data being parsed and/or serialized is known to be valid.

## 3.5 64-bit Integer Type

By default the 64-bit `long` and `unsignedLong` built-in XML Schema 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.

## 3.6 Parser and Serializer Reuse

When one type in XML Schema inherits from another, it is often desirable to be able to reuse the parser and serializer implementations corresponding to the base type in the parser and serializer implementations corresponding to the derived type. XSD/e provides support for two reuse styles: the so-called *mixin* (generated when the `--reuse-style-mixin` option is specified) and *tiein* (generated by default) styles. The XSD/e runtime should be configured in accordance with the reuse style used in the generated code. See Section 5.6, "Parser Reuse" in the Embedded C++/Parser Mapping Getting Started Guide and Section 6.6, "Serializer Reuse" in the Embedded C++/Serializer Mapping Getting Started Guide for details.

## 3.7 Support for Polymorphism

By default the XSD/e compiler generates non-polymorphic code. If your vocabulary uses XML Schema polymorphism in the form of `xsi:type` and/or substitution groups, then you will need to configure the XSD/e runtime with support for polymorphism, compile your schemas with the `--generate-polymorphic` option to produce polymorphism-aware code, as well as pass `true` as the last argument to the `xml_schema::document_pimpl` and `xml_schema::document_simpl` constructors (see Chapter 6, "Parsing and Serialization" for details). If some of your schemas do not require support for polymorphism then you can compile them with the `--runtime-polymorphic` option and still use the XSD/e runtime configured with polymorphism support.

The XSD/e compiler can often automatically determine which types are polymorphic based on the substitution group declarations. However, if your XML vocabulary is not using substitution groups or if substitution groups are defined in a separate schema, then you will need to use the `--polymorphic-type` option to specify which types are polymorphic. When using this

option you only need to specify the root of a polymorphic type hierarchy and the XSD/e compiler will assume that all the derived types are also polymorphic. Also note that you need to specify this option when compiling every schema file that references the polymorphic type. Consider the following two schemas as an example:

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

  <xs:complexType name="base">
    <xs:sequence>
      <xs:element name="b" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>

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

</xs:schema>

<!-- derived.xsd -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <include schemaLocation="base.xsd"/>

  <xs:complexType name="derived">
    <xs:complexContent>
      <xs:extension base="base">
        <xs:sequence>
          <xs:element name="d" type="xs:string"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:element name="derived" type="derived" substitutionGroup="base"/>

</xs:schema>
```

In this example we need to specify "`--polymorphic-type base`" when compiling both schemas because the substitution group is declared in a schema other than the one defining type `base`.

Another issue that may arise when compiling polymorphic schemas is the situation where the XSD/e compiler is unaware of all the derivations of a polymorphic type while generating parser and serializer aggregates. As a result, the generated code may not be able to parse and serialize these "invisible" to the compiler types. The following example will help illustrate this case. Consider a modified version of `base.xsd` from the above example:

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

  <xs:complexType name="base">
    <xs:sequence>
      <xs:element name="b" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>

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

  <xs:complexType name="root">
    <xs:sequence>
      <xs:element ref="base" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <!-- document root -->
  <xs:element name="root" type="root"/>

</xs:schema>
```

Suppose we compile this schema as follows:

```
$ xsde cxx-hybrid --generate-parser --generate-serializer \
--generate-polymorphic --polymorphic-type base \
--generate-aggregate --root-element root base.xsd
```

The resulting parser and serializer aggregates for the `root` element will not include the parser and serializer for the `derived` type that can be used instead of the `base` type. This is because the XSD/e compiler has no knowledge of the `derived`'s existence when compiling `base.xsd`.

There are two ways to overcome this problem. The easier but potentially slower approach is to compile all your schemas at once, for example:

```
$ xsde cxx-hybrid --generate-parser --generate-serializer \
--generate-polymorphic --polymorphic-type base \
--generate-aggregate --root-element root base.xsd derived.xsd
```

This will make sure the XSD/e compiler "sees" all the derivations of the polymorphic types. The other approach allows you to explicitly specify, with the `--polymorphic-schema` option, additional schemas that may contain derivations of the polymorphic types. Using this approach we would compile `base.xsd` and `derived.xsd` like this:

```
$ xsde cxx-hybrid --generate-parser --generate-serializer \
--generate-polymorphic --polymorphic-type base \
--generate-aggregate --root-element root \
--polymorphic-schema derived.xsd base.xsd

$ xsde cxx-hybrid --generate-parser --generate-serializer \
--generate-polymorphic --polymorphic-type base derived.xsd
```

For information on how to use object models with polymorphic types, refer to Section 4.10, "Polymorphic Object Models".

# 3.8 Custom Allocators

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

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

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

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

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

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

  void
  free (void*);

  struct alloc_guard
```

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

    void*
    get () const;

    void
    release ();

  private:
    ...
  };
}
```

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

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

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

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

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

...

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

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

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

...

if (x)
```

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

For a complete example that shows how to use custom allocators, see the `allocator` example which can be found in the `examples/cxx/hybrid/` directory of the XSD/e distribution.

# 4 Working with Object Models

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

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

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

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

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

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

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

</xs:schema>
```

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

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

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

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

</people>
```

Compiling `people.xsd` with the XSD/e compiler results in three generated object model classes: `gender`, `person` and `people`. Here is how they look with STL enabled:

```
// gender (fixed-length)
//
class gender
{
public:
  enum value_type
  {
    male,
    female
  };

  gender ();
  gender (value_type);
  gender (const gender&);
  gender& operator= (const gender&);

  void
  value (value_type);

  operator value_type () const;

  const char*
  string () const;

private:
  ...
```

```
};

// person (fixed-length)
//
class person
{
public:
  person ();
  person (const person&);
  person& operator= (const person&);

  // id
  //
  unsigned int
  id () const;

  unsigned int&
  id ();

  void
  id (unsigned int);

  // first-name
  //
  const std::string&
  first_name () const;

  std::string&
  first_name ();

  void
  first_name (const std::string&);

  // middle-name
  //
  bool
  middle_name_present () const;

  void
  middle_name_present (bool);

  const std::string&
  middle_name () const;

  std::string&
  middle_name ();

  void
  middle_name (const std::string&);

  // last-name
```

```
  //
  const std::string&
  last_name () const;

  std::string&
  last_name ();

  void
  last_name (const std::string&);

  // gender
  //
  const ::gender&
  gender () const;

  ::gender&
  gender ();

  void
  gender (const ::gender&);

  // age
  //
  unsigned short
  age () const;

  unsigned short&
  age ();

  void
  age (unsigned short);

private:
  ...
};

// people (variable-length)
//
class people
{
public:
  people ();

private:
  people (const people&);
  people& operator= (const people&);

public:
  // person
  //
  typedef xml_schema::fix_sequence<person> person_sequence;
```

```
  typedef person_sequence::iterator person_iterator;
  typedef person_sequence::const_iterator person_const_iterator;

  const person_sequence&
  person () const;

  person_sequence&
  person ();

private:
  ...
};
```

We will examine these classes in detail in the subsequent sections.

# 4.1 Namespaces

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

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

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

A vocabulary without a namespace is mapped to the global scope. This also can be altered with the above options by using an empty name for the XML namespace. For example, we could place the generated object model classes for the `people.xsd` schema into the `records` C++ namespace by adding the following option:

```
--namespace-map =records
```

# 4.2 Memory Management

To ensure that objects are allocated and passed efficiently, the C++/Hybrid mapping divides all object model types into fixed-length and variable-length. A type is variable-length if any of the following is true:

1. it is an XML Schema `list` type
2. it is an XML Schema `union` type and STL is disabled
3. it derives from a variable-length type
4. it contains an element or attribute of a variable-length type
5. it contains an element or compositor (`sequence` or `choice`) with `maxOccurs` greater than one
6. it is recursive (that is, one of its elements contains a reference, directly or indirectly, to the type itself)
7. it is polymorphic (see Section 4.10, "Polymorphic Object Models" for details)

The following build-in XML Schema types are variable-length: `base64Binary`, `hexBinary`, `NMTOKENS`, and `IDREFS`. Furthermore, if STL is disabled, all string-based build-in XML Schema types are variable-length, namely: `string`, `normalizedString`, `token`, `Name`, `NMTOKEN`, `NCName`, `language`, `QName`, `ID`, `IDFER`, and `anyURI`.

Otherwise, a type is fixed-length. As you might have noticed from the previous code listings, the XSD/e compiler adds a comment before each generated object model class that states whether it is fixed or variable-length. For example, the `people` type is variable-length because it contains a sequence of `person` elements (`maxOccurs="unbounded"`). If we recompile the `people.xsd` schema with the `--no-stl` option, the `person` type will also become variable-length since it contains elements of the `string` built-in type. And when STL is disabled, `string` is variable-length.

The object model uses different methods for storing and passing around fixed-length and variable-length types. Instances of fixed-length types are stored and passed by value since it is cheaper to copy than to allocate them dynamically (in the STL case, the `std::string` is expected to support the referenced-counted copy-on-write optimization, which makes copying cheap).

Variable-length types are always allocated dynamically and are stored and passed as pointers. Because copying an instance of a variable-length type can be expensive, such types make their copy constructor and copy assignment operators unavailable.

When you set a value of an element or attribute of a variable-length type, the object model assumes ownership of the pointed to object. Unless you are using custom allocators (see Section 3.8, "Custom Allocators"), the object model expects you to allocate such an object with operator `new` and will eventually delete it with operator `delete`.

If you wish to make copies of variable-length objects, then you can request the generation of the object cloning functions with the `--generate-clone` compiler option. When this option is specified, each variable-length type implements the `_clone()` function which returns a dynamically-allocated copy of the object or `NULL` if the allocation failed and C++ exceptions are disabled (see Section 3.3, "C++ Exceptions").

You can also request generation of detach functions with the `--generate-detach` compiler option. These functions allow you to detach a variable-length object from the object model. As an example, let us extend our `people.xsd` schema with the following type:

```
<xs:complexType name="staff">
  <xs:sequence>
    <xs:element name="permanent" type="people"/>
    <xs:element name="contract" type="people"/>
  </xs:sequence>
</xs:complexType>
```

If we compile it with XSD/e and specify the `--generate-clone` and `--generate-detach` options, we will get the following C++ class:

```
// staff (variable-length)
//
class staff
{
public:
  staff ();

  staff*
  _clone () const;

private:
  staff (const staff&);
  staff& operator= (const staff&);

public:
  // permanent
  //
  const people&
  permanent () const;

  people&
  permanent ();

  void
  permanent (people*);

  people*
  permanent_detach ();

  // contract
  //
  const people&
  contract () const;

  people&
  contract ();
```

```
  void
  contract (people*);

  people*
  contract_detach ();

private:
  ...
};
```

Notice that unlike, say, the `first_name()` modifier function in the `person` class, the `permanent()` and `contract()` modifiers expect a pointer to the `people` object. The following listing shows how we can create and populate an instance of the `staff` class. The use of smart pointers to hold the results of dynamic allocations is omitted for brevity:

```
people* per = new people;
people* con = new people;

// Populate per and con.

staff s;
s->permanent (per) // Assumes ownership of per.
s->contract (con)  // Assumes ownership of con.
```

# 4.3 Enumerations

By default, string-based types that use XML Schema restriction by enumeration are mapped to C++ classes with semantics similar to C++ enum (you can suppress this mapping and instead get the plain inheritance by specifying the `--suppress-enum` compiler option). The following code fragment again shows the C++ class that was generated for the `gender` XML Schema type presented at the beginning of this chapter:

```
// gender (fixed-length)
//
class gender
{
public:
  enum value_type
  {
    male,
    female
  };

  gender ();
  gender (value_type);
  gender (const gender&);
  gender& operator= (const gender&);
```

```
  void
  value (value_type);

  operator value_type () const;

  const char*
  string () const;

private:
  value_type v_;
};
```

The `gender` class defines the underlying C++ enum type (`value_type`) with enumerators corresponding to the `enumeration` elements in XML Schema. The class also defines the default constructor, copy constructor, constructor with the underlying enum type as its argument, and the assignment operator. The `gender` class also supports the implicit conversion to the underlying enum type and the explicit conversion to string via the `string()` function. Finally, it provides the `value()` modifier function which allows you to set the underlying enum value explicitly. Note also that such an enumeration class is always fixed-length since it only contains the C++ enum value. The following example shows how we can use the `gender` class:

```
gender g = gender::male;
g = gender::female;
g.value (gender::female); // Same as above.

cerr << g.string () << endl;

if (g != gender::male)
  ...

switch (g)
{
case gender::male:
  ...
case gender::female:
  ...
}
```

# 4.4 Attributes and Elements

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

The *one* cardinality class covers all elements that should occur exactly once as well as the required attributes. In our example, the `first-name`, `last-name`, `gender`, and `age` elements as well as the `id` attribute belong to this cardinality class. The following code fragment

again shows the accessor and modifier functions that are generated for the `first-name` element in the `person` class:

```
class person
{
  // first-name
  //
  const std::string&
  first_name () const;

  std::string&
  first_name ();

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

The first two accessor functions return read-only (constant) and read-write references to the element's value, respectively. The modifier function sets the new value for the element. Note that the signature of the modifier function varies depending on whether the element or attribute is of a fixed or variable-length type, as was discussed in the previous section.

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

```
class person
{
  // middle-name
  //
  bool
  middle_name_present () const;

  void
  middle_name_present (bool);

  const std::string&
  middle_name () const;

  std::string&
  middle_name ();

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

Compared to the *one* cardinality class, *optional* adds functions for querying and modifying the member's presence status. The following example shows how we can use these functions:

```
person& p = ...

if (p.middle_name_present ())
{
  cout << p.middle_name () << endl;
  p.middle_name_present (false); // Reset to the "not present" state.
}
```

If an optional member is of a variable-length type, then the second `_present()` function is omitted. This is done to help detect programming errors that result from a type becoming variable-length due to schema changes. In this situation, before the type becomes variable-length, calling the presence function with `true` as its argument and then accessing the member is valid. Once the type becomes variable-length, the same sequence of calls would lead to a runtime error. By omitting the second `_present()` function for variable-length types, this kind of errors can be detected at compile time. To reset an optional member of a variable-length type you can call the member modifier function with `NULL` as its argument. For example, if the `middle_name` member was of a variable-length type, then the above code fragment would look like this:

```
person& p = ...

if (p.middle_name_present ())
{
  cout << *p.middle_name () << endl;
  p.middle_name (0); // Reset to the "not present" state.
}
```

There are two cases in the *optional* cardinality class that are handled differently. These are optional attributes with default and fixed values. When an optional attribute declaration in XML Schema specifies a default or fixed value and such an attribute is not present in the XML document, the attribute is assumed to have the default or fixed value, respectively. Furthermore, if an attribute with the fixed value is set in the XML document, then the attribute value should be the same as its fixed value.

For an optional attribute with a default value, the functions for querying and modifying the attribute's presence status are replaced with functions that allow you to determine whether the attribute has the default value. The accessor functions can be called at any time since an optional attribute with a default value always has some value. Also an extra static function is provided to allow you to obtain the default value. Consider the following modification to the `person` type which adds the `verified` attribute with the default value:

```
<xs:complexType name="person">
  <xs:sequence>
    <xs:element name="first-name" type="xs:string"/>
    ...
  </xs:sequence>
  <xs:attribute name="id" type="xs:unsignedInt" use="required"/>
  <xs:attribute name="verified" type="xs:boolean" default="false"/>
</xs:complexType>
```

The code fragment below shows the accessor and modifier functions that are generated for this new attribute in the `person` class:

```
class person
{
  // verified
  //
  bool
  verified_default () const;

  void
  verified_default (bool);

  bool
  verified () const;

  bool&
  verified ();

  void
  verified (bool);

  static bool
  verified_default_value ();
};
```

When we create an object of the `person` class, the `verified` member is automatically initialized to the default value. The following example shows how we can manipulate the `verified` attribute value:

```
person p; // verified is set to the default value (false).

if (p.verified_default ())
  p.verified (true);
else
  p.verified_default (true); // Revert to the default value.

bool v = p.verified (); // Ok, can always be called.
bool vd = person::verified_default_value ();
```

Note that modifying an attribute of a variable-length type via the reference when the attribute is set to the default value is illegal since this will modify the default value shared by all instances. For example:

```
type& x = ...

if (x.foo_default ())
{
  foo& f = x.foo (); // foo is variable-length, for example NMTOKENS
  f.push_back ("aaa"); // Illegal.
}

if (x.foo_default ())
{
  foo* f = new foo;
  f->push_back ("aaa");
  x.foo (f); // Ok.
}
```

Because an attribute with a fixed value can only be set to that value, only the read-only (constant) accessor and the static function for obtaining the fixed value are provided for such attributes. Similar to the default values, members with fixed values of a newly created object are automatically initialized to their respective fixed values. Consider the following modification to the `verified` attribute from the schema above:

```
<xs:complexType name="person">
  ...
  <xs:attribute name="verified" type="xs:boolean" fixed="true"/>
</xs:complexType>
```

The code fragment below shows the accessor functions that are generated for this attribute in the `person` class:

```
class person
{
  // verified
  //
  bool
  verified () const;

  static bool
  verified_fixed_value ();
};
```

During serialization, attributes that are set to default and fixed values are explicitly specified in the resulting XML document. You can use the `--omit-default-attributes` XSD/e compiler option to omit such attributes from the serialized XML.

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

```
class people
{
  // person
  //
  typedef xml_schema::fix_sequence<person> person_sequence;
  typedef person_sequence::iterator person_iterator;
  typedef person_sequence::const_iterator person_const_iterator;

  const person_sequence&
  person () const;

  person_sequence&
  person ();
};
```

The `person_sequence` type is a sequence container for the element's values. It has an interface similar to `std::vector` and we will discuss it in more detail shortly. The `person_iterator` and `person_const_iterator` types are read-write and read-only (constant) iterators for the `person_sequence` container.

Unlike other two cardinality classes, the *sequence* class only provides accessor functions that return read-only (constant) and read-write references to the sequence container. The modification of the element values is performed my manipulating the returned sequence container and elements that it contains.

In the remainder of this section we will examine the interfaces of the sequence containers which differ slightly depending on whether the element type is fixed or variable-length and whether C++ exceptions are enabled. Also, when STL is disabled, string sequences have a special interface which is also discussed below.

When exceptions are enabled, the fixed-length type sequences are implemented in terms of the following class template:

```
namespace xml_schema
{
  template <typename T>
  class fix_sequence
  {
  public:
    typedef T         value_type;
    typedef T*        pointer;
    typedef const T*  const_pointer;
    typedef T&        reference;
```

```
  typedef const T&  const_reference;

  typedef size_t    size_type;
  typedef ptrdiff_t difference_type;

  typedef T*        iterator;
  typedef const T*  const_iterator;

public:
  fix_sequence ();

  void
  swap (fix_sequence&);

private:
  fix_sequence (const fix_sequence&);

  fix_sequence&
  operator= (fix_sequence&);

public:
  iterator
  begin ();

  const_iterator
  begin () const;

  iterator
  end ();

  const_iterator
  end () const;

  T&
  front ();

  const T&
  front () const;

  T&
  back ();

  const T&
  back () const;

  T&
  operator[] (size_t);

  const T&
  operator[] (size_t) const;
```

```
  public:
    bool
    empty () const;

    size_t
    size () const;

    size_t
    capacity () const;

    size_t
    max_size () const;

  public:
    void
    clear ();

    void
    pop_back ();

    iterator
    erase (iterator);

    void
    push_back (const T&);

    iterator
    insert (iterator, const T&);

    void
    reserve (size_t);

    void
    assign (const T* src, size_t n);
  };
}
```

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

```
namespace xml_schema
{
  template <typename T>
  class fix_sequence
  {
  public:
    enum error
    {
      error_none,
      error_no_memory
    };
```

```
    ...

  public:
    error
    push_back (const T&);

    error
    insert (iterator, const T&);

    error
    insert (iterator, const T&, iterator& result);

    error
    reserve (size_t);

    error
    assign (const T* src, size_t n);
  };
}
```

That is, the functions that may require memory allocation now return an error code that you will need to check in order to detect the out of memory condition.

When exceptions are enabled, the variable-length type sequences are implemented in terms of the following class template:

```
namespace xml_schema
{
  template <typename T>
  class var_sequence
  {
  public:
    typedef T         value_type;
    typedef T*        pointer;
    typedef const T*  const_pointer;
    typedef T&        reference;
    typedef const T&  const_reference;

    typedef size_t    size_type;
    typedef ptrdiff_t difference_type;

    typedef <implementation details> iterator;
    typedef <implementation details> const_iterator;

  public:
    var_sequence ();

    void
    swap (var_sequence&);
```

```
  private:
    var_sequence (const var_sequence&);

    var_sequence&
    operator= (var_sequence&);

  public:
    iterator
    begin ();

    const_iterator
    begin () const;

    iterator
    end ();

    const_iterator
    end () const;

    T&
    front ();

    const T&
    front () const;

    T&
    back ();

    const T&
    back () const;

    T&
    operator[] (size_t);

    const T&
    operator[] (size_t) const;

  public:
    bool
    empty () const;

    size_t
    size () const;

    size_t
    capacity () const;

    size_t
    max_size () const;
```

```
  public:
    void
    clear ();

    void
    push_back (T*);

    iterator
    insert (iterator, T*);

    void
    pop_back ();

    iterator
    erase (iterator);

    void
    reserve (size_t);

    T*
    detach (iterator);

    void
    attach (iterator, T*);
  };
}
```

Most of this interface is identical to the fixed-length type version except for the `push_back()`, and `insert()` functions. Similar to the modifier functions for elements and attributes of variable-length types, these two functions expect a pointer to the dynamically-allocated instance of the type and assume ownership of the passed object. To simplify error handling, these two functions delete the passed object if the reallocation of the underlying sequence buffer fails. The `var_sequence` class template also provides the `detach()` and `attach()` functions. The `detach()` function allows you to detach the contained object at the specified position. A detached object should eventually be deallocated with operator `delete`. Similarly, the `attach()` function allows you to attach a new object at the specified position.

When C++ exceptions are disabled, the `push_back()`, `insert()`, and `reserve()` functions return an error code to signal the out of memory condition:

```
namespace xml_schema
{
  template <typename T>
  class var_sequence
  {
  public:
    enum error
    {
      error_none,
```

```
      error_no_memory
    };

    ...

  public:
    error
    push_back (T*);

    error
    insert (iterator, T*);

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

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

When STL is enabled, the `string_sequence` class has the same interface as `fix_sequence<std::string>`. When STL is disabled and strings are mapped to `char*`, `string_sequence` has a special interface. When C++ exceptions are enabled, it has the following definition:

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

    typedef size_t        size_type;
    typedef ptrdiff_t     difference_type;

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

    string_sequence ();

    void
    swap (string_sequence&);

  private:
    string_sequence (string_sequence&);
```

```
  string_sequence&
  operator= (string_sequence&);

public:
  iterator
  begin ();

  const_iterator
  begin () const;

  iterator
  end ();

  const_iterator
  end () const;

  char*
  front ();

  const char*
  front () const;

  char*
  back ();

  const char*
  back () const;

  char*
  operator[] (size_t);

  const char*
  operator[] (size_t) const;

public:
  bool
  empty () const;

  size_t
  size () const;

  size_t
  capacity () const;

  size_t
  max_size () const;

public:
  void
  clear ();
```

```
    void
    pop_back ();

    iterator
    erase (iterator);

    void
    push_back (char*);

    void
    push_back_copy (const char*);

    iterator
    insert (iterator, char*);

    void
    reserve (size_t);

    char*
    detach (iterator);

    void
    attach (iterator, char*);
  };
}
```

The `push_back()` and `insert()` functions assume ownership of the passed string which should be allocated with operator `new[]` and will be deallocated with operator `delete[]` by the `string_sequence` object. Similar to `var_sequence`, these two functions free the passed string if the reallocation of the underlying sequence buffer fails. The `push_back_copy()` function makes a copy of the passed string. The `string_sequence` class also provides the `detach()` and `attach()` functions. The `detach()` function allows you to detach the contained string at the specified position. A detached string should eventually be deallocated with operator `delete[]`. Similarly, the `attach()` function allows you to attach a new string at the specified position.

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

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

```
    };

    ...

  public:
    error
    push_back (char*);

    error
    push_back_copy (const char*);

    error
    insert (iterator, char*);

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

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

# 4.5 Compositors

The XML Schema language provides three compositor constructs that are used to group elements: `all`, `sequence`, and `choice`. If a compositor has an *optional* or *sequence* cardinality class (see Section 4.4, "Attributes and Elements") or if a compositor is inside `choice`, then the C++/Hybrid mapping generates a nested class for such a compositor as well as a set of accessor and modifier functions similar to the ones defined for elements and attributes. Otherwise, the member functions, corresponding to elements defined in a compositor, are generated directly in the containing class.

Compositor classes are either fixed or variable-length and obey the same storage and passing rules as object model classes corresponding to XML Schema types (see Section 4.2, "Memory Management"). Consider the following schema fragment as an example:

```
<complexType name="type">
  <sequence>
    <sequence minOccurs="0">
      <element name="a" type="int"/>
      <element name="b" type="string" maxOccurs="unbounded"/>
    </sequence>
    <sequence maxOccurs="unbounded">
      <element name="c" type="int"/>
      <element name="d" type="string"/>
    </sequence>
  </sequence>
</complexType>
```

The corresponding object model class is shown below:

```
// type (variable-length)
//
class type
{
public:
  type ();

private:
  type (const type&);
  type& operator= (const type&);

public:
  // sequence (variable-length)
  //
  class sequence_type
  {
  public:
    sequence_type ();

  private:
    sequence_type (const sequence_type&);
    sequence_type& operator= (const sequence_type&);

  public:
    // a
    //
    int
    a () const;

    int&
    a ();

    void
    a (int);

    // b
    //
    typedef xml_schema::string_sequence b_sequence;
    typedef b_sequence::iterator b_iterator;
    typedef b_sequence::const_iterator b_const_iterator;

    const b_sequence&
    b () const;

    b_sequence&
    b ();

  private:
    ...
```

```
};

bool
sequence_present () const;

const sequence_type&
sequence () const;

sequence_type&
sequence ();

void
sequence (sequence_type*);

// sequence1 (fixed-length)
//
class sequence1_type
{
public:
  sequence1_type ();
  sequence1_type (const sequence1_type&);
  sequence1_type& operator= (const sequence1_type&);

  // c
  //
  int
  c () const;

  int&
  c ();

  void
  c (int);

  // d
  //
  const std::string&
  d () const;

  std::string&
  d ();

  void
  d (const std::string&);

private:
  ...
};

typedef xml_schema::fix_sequence<sequence1_type> sequence1_sequence;
typedef sequence1_sequence::iterator sequence1_iterator;
```

```
  typedef sequence1_sequence::const_iterator sequence1_const_iterator;

  const sequence1_sequence&
  sequence1 () const;

  sequence1_sequence&
  sequence1 ();

private:
  ...
};
```

The content of the outer `sequence` compositor is generated in-line since this compositor belongs to the *one* cardinality class. The first nested `sequence` compositor is optional (`minOccurs="0"`), which results in a corresponding nested class. Notice that the `sequence_type` is variable-length and the accessor and modifier functions corresponding to this `sequence` compositor are the same as for an optional element or attribute. Similarly, the second nested compositor is of the *sequence* cardinality class (`maxOccurs="unbounded"`), which also results in a nested class and a set of accessor functions.

Generated code corresponding to an `all` and `sequence` compositor, whether in-line or as a nested class, simply define accessor and modifier functions for the elements that this compositor contains. For the `choice` compositor, on the other hand, additional types and functions are generated to support querying and selecting the choice arm that is in effect. Consider the following simple example:

```
<complexType name="type">
  <choice>
    <element name="a" type="int"/>
    <element name="b" type="string"/>
    <element name="c" type="boolean"/>
  </choice>
</complexType>
```

The corresponding object model class is shown next:

```
// type (fixed-length)
//
class type
{
public:
  type ();
  type (const type&);
  type& operator= (const type&);

  // choice
  //
  enum choice_arm_tag
  {
```

```
    a_tag,
    b_tag,
    c_tag
  };

  choice_arm_tag
  choice_arm () const;

  void
  choice_arm (choice_arm_tag);

  // a
  //
  int
  a () const;

  int&
  a ();

  void
  a (int);

  // b
  //
  const std::string&
  b () const;

  std::string&
  b ();

  void
  b (const std::string&);

  // c
  //
  bool
  c () const;

  bool&
  c ();

  void
  c (bool);

private:
  ...
};
```

The extra type is the `choice_arm_tag` enumeration which defines a set of tags corresponding to each choice arm. There are also the `choice_arm()` accessor and modifier functions that can be used to query and set the current choice arm. The following code fragment shows how we can use this class:

```
type& x = ...

switch (x.choice_arm ())
{
case type::a_tag:
  {
    cout << "a: " << x.a () << endl;
    break;
  }
case type::b_tag:
  {
    cout << "b: " << x.b () << endl;
    break;
  }
case type::c_tag:
  {
    cout << "c: " << x.c () << endl;
    break;
  }
}

// Modifiers automatically set the corresponding arm.
//
x.a (10);

// For accessors we need to select the arm explicitly.
//
x.choice_arm (type::b_tag);
x.b () = "b";
```

The following slightly more complex example triggers the generation of nested classes for the `choice` compositor as well as for the `sequence` compositor inside `choice`. Notice that the nested class for `sequence` is generated because it is in `choice` even though its cardinality class is *one*.

```
<complexType name="type">
  <choice maxOccurs="unbounded">
    <sequence>
      <element name="a" type="int"/>
      <element name="b" type="string"/>
    </sequence>
    <element name="c" type="boolean"/>
  </choice>
</complexType>
```

The corresponding object model class is shown next:

```
// type (variable-length)
//
class type
{
public:
  type ();

private:
  type (const type&);
  type& operator= (const type&);

public:
  // choice (fixed-length)
  //
  class choice_type
  {
  public:
    choice_type ();
    choice_type (const choice_type&);
    choice_type& operator= (const choice_type&);

    enum choice_arm_tag
    {
      sequence_tag,
      c_tag
    };

    choice_arm_tag
    choice_arm () const;

    void
    choice_arm (choice_arm_tag);

    // sequence (fixed-length)
    //
    class sequence_type
    {
    public:
      sequence_type ();
      sequence_type (const sequence_type&);
      sequence_type& operator= (const sequence_type&);

      // a
      //
      int
      a () const;

      int&
      a ();
```

```
      void
      a (int);

      // b
      //
      const std::string&
      b () const;

      std::string&
      b ();

      void
      b (const std::string&);

    private:
      ...
    };

    const sequence_type&
    sequence () const;

    sequence_type&
    sequence ();

    void
    sequence (const sequence_type&);

    // c
    //
    bool
    c () const;

    bool&
    c ();

    void
    c (bool);

  private:
    ...
  };

  typedef xml_schema::fix_sequence<choice_type> choice_sequence;
  typedef choice_sequence::iterator choice_iterator;
  typedef choice_sequence::const_iterator choice_const_iterator;

  const choice_sequence&
  choice () const;

  choice_sequence&
```

```
  choice ();

private:
  ...
};
```

## 4.6 Accessing the Object Model

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

```
#include <memory>
#include <iostream>

#include "people.hxx"
#include "people-pimpl.hxx"

using namespace std;

int
main ()
{
  // Parse.
  //
  people_paggr people_p;
  xml_schema::document_pimpl doc_p (people_p.root_parser (),
                                    people_p.root_name ());
  people_p.pre ();
  doc_p.parse ("people.xml");
  auto_ptr<people> ppl (people_p.post ());

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

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

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

    if (p.middle_name_present ())
      cout << p.middle_name () << " ";

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

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

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

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

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

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

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

# 4.7 Modifying the Object Model

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

```
#include <memory>
#include <iostream>

#include "people.hxx"
#include "people-pimpl.hxx"
#include "people-simpl.hxx"
```

```
using namespace std;

int
main ()
{
  // Parse.
  //
  people_paggr people_p;
  xml_schema::document_pimpl doc_p (people_p.root_parser (),
                                    people_p.root_name ());
  people_p.pre ();
  doc_p.parse ("people.xml");
  auto_ptr<people> ppl (people_p.post ());

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

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

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

  john.middle_name ("Mary");
  jane.middle_name_present (false);

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

  // Serialize the modified object model to XML.
  //
  people_saggr people_s;
  xml_schema::document_simpl doc_s (people_s.root_serializer (),
                                    people_s.root_name ());
  people_s.pre (*ppl);
  doc_s.serialize (cout, xml_schema::document_simpl::pretty_print);
  people_s.post ();
}
```

The first modification the above application performs is iterating over person records and incrementing the age value. This code fragment shows how to modify the value of a required attribute or element. The next modification shows how to set a new value for the optional `middle-name`

element as well as clear its value. Finally, the example adds a copy of the John Doe record to the `person` element sequence.

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

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

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

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

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

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

</people>
```

# 4.8 Creating the Object Model from Scratch

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

```
#include <iostream>

#include "people.hxx"
#include "people-simpl.hxx"

using namespace std;
```

```
int
main ()
{
  people ppl;
  people::person_sequence& ps = ppl.person ();

  // John
  //
  {
    person p;
    p.first_name ("John");
    p.last_name ("Doe");
    p.gender (gender::male);
    p.age (32);
    p.id (1);

    ps.push_back (p);
  }

  // Jane
  //
  {
    person p;
    p.first_name ("Jane");
    p.middle_name ("Mary");
    p.last_name ("Doe");
    p.gender (gender::female);
    p.age (28);
    p.id (2);

    ps.push_back (p);
  }

  // Serialize the object model to XML.
  //
  people_saggr people_s;
  xml_schema::document_simpl doc_s (people_s.root_serializer (),
                                    people_s.root_name ());
  people_s.pre (ppl);
  doc_s.serialize (cout, xml_schema::document_simpl::pretty_print);
  people_s.post ();
}
```

The only new part in the above application is the calls to the `people` and `person` constructors. As a general rule, a newly created instance does not assign any values to its elements and attributes. That is, members with the *one* cardinality class are left uninitialized, members with the *optional* cardinality class are set to the "not present" state, and members with the *sequence* cardinality class have empty containers. After the instance has been created, we can set its element and attribute values using the modifier functions.

The above application produces the following output:

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

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

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

</people>
```

# 4.9 Customizing the Object Model

Sometimes it is desirable to add extra, application-specific data or functionality to some object model classes or nested compositor classes. Cases where this may be required include handling of typeless content matched by XML Schema wildcards as well as a need for an application to pass extra data or provide custom functions as part of the object model. The C++/Hybrid mapping provides two mechanisms for accomplishing this: custom data and custom types. Custom data is a light-weight mechanism for storing application-specific data by allowing you to add a sequence of opaque objects, stored as `void*`, to select generated classes. Type customization is a more powerful mechanism that allows you to provide custom implementations for select object model classes. You have the option of either extending the generated version of the class (for example, by adding extra data members and/or functions) or providing your own implementation from scratch. The latter approach essentially allows you to change the mapping of XML Schema to C++ on a case by case basis.

It is also possible to customize the parsing and serialization code, for example, to populate the custom data sequence or custom data members during parsing and later serialize them to XML. See Section 6.1, "Customizing Parsers and Serializers" for details. The remainder of this section discusses the custom data and custom types mechanisms in more detail.

To instruct the XSD/e compiler to include custom data in a specific object model class, we need to use the `--custom-data` option with the corresponding XML Schema type name as its argument. To include custom data into a nested compositor class, use its qualified name starting with the XML Schema type, for example `type::sequence1`. If we would like to add the ability to

store custom data in the generated `person` class from our person records vocabulary, we can compile `people.xsd` like this:

```
$ xsde cxx-hybrid --custom-data person people.xsd
```

The resulting `person` class will have the following extra set of type definitions and functions:

```
// person (variable-length)
//
class person
{
public:

  ...

  // Custom data.
  //
  typedef xml_schema::data_sequence custom_data_sequence;
  typedef custom_data_sequence::iterator custom_data_iterator;
  typedef custom_data_sequence::const_iterator custom_data_const_iterator;

  const custom_data_sequence&
  custom_data () const;

  custom_data_sequence&
  custom_data ();
};
```

Notice also that the `person` class is now variable-length since it contains a sequence. When C++ exceptions are enabled, the custom data sequence has the following interface:

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

    typedef size_t        size_type;
    typedef ptrdiff_t     difference_type;

    typedef void** iterator;
    typedef const void* const* const_iterator;

    typedef void (*destroy_func) (void* data, size_t pos);
    typedef void* (*clone_func) (void* data, size_t pos);
```

```
  public:
    data_sequence ();

    void
    destructor (destroy_func);

    void
    clone (clone_func);

    void
    swap (data_sequence&);

  private:
    data_sequence (const data_sequence&);

    data_sequence&
    operator= (data_sequence&);

  public:
    iterator
    begin ();

    const_iterator
    begin () const;

    iterator
    end ();

    const_iterator
    end () const;

    void*
    front ();

    const void*
    front () const;

    void*
    back ();

    const void*
    back () const;

    void*
    operator[] (size_t);

    const void*
    operator[] (size_t) const;

  public:
```

```
    bool
    empty () const;

    size_t
    size () const;

    size_t
    capacity () const;

    size_t
    max_size () const;

  public:
    void
    clear ();

    void
    pop_back ();

    iterator
    erase (iterator);

    void
    push_back (void*);

    iterator
    insert (iterator, void*);

    void
    reserve (size_t);
  };
}
```

The `destructor()` modifier allows you to specify the clean up function used to free the sequence elements. Similarly, the `clone()` modifier allows you to specify the cloning function used to copy the sequence elements. The second argument in these functions is the position of the element in the sequence. This allows you to store objects of different types in the same custom data sequence.

The `push_back()` and `insert()` functions free the passed object if the reallocation of the underlying sequence buffer fails. When exceptions are disabled, the `push_back()`, `insert()`, and `reserve()` functions return an error code to signal the out of memory condition:

```
namespace xml_schema
{
  class data_sequence
  {
  public:
```

```
    enum error
    {
      error_none,
      error_no_memory
    };

    ...

  public:
    error
    push_back (void*);

    error
    insert (iterator, void*);

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

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

The following code fragment shows how we can store and retrieve custom data in the `person` class:

```
class data
{
  ...
};

void
destroy_data (void* p, size_t)
{
  delete static_cast<data*> (p);
}

person& = ...;
person::custom_data_sequence& cd = p.custom_data ();

cd.destructor (&destroy_data);

// Store.
//
data* d = new data;
cd.push_back (d);

// Retrieve.
//
```

```
for (person::custom_data_iterator i = cd.begin (); i != cd.end (); ++i)
{
  data* d = static_cast<data*> (*i);
}
```

To instruct the XSD/e compiler to use a custom implementation for a specific object model class, we need to use the `--custom-type` option. The argument format for this option is `name[=[flags][/[type][/[base][/include]]]]`. The *`name`* component is the XML Schema type name being customized. Optional *`flags`* allow you to specify whether the custom class is fixed or variable-length since customization can alter this property, normally from fixed-length to variable-length. The `f` flag indicates the type is fixed-length and the `v` flag indicates the type is variable-length. If omitted, the default rules are used to determine the type length (see Section 4.2, "Memory Management"). Optional *`type`* is a C++ type name, potentially qualified, that should be used as a custom implementation. If specified, the object model type is defined as a `typedef` alias for this C++ type. Optional *`base`* is a C++ name that should be given to the generated version. It is normally used as a base for the custom implementation. Optional *`include`* is the header file that defines the custom implementation. It is `#include`'ed into the generated code immediately after (if *`base`* is specified) or instead of the generated version. The following examples show how we can use this option:

```
--custom-type foo
--custom-type foo=///foo.hxx
--custom-type foo=v///foo.hxx
--custom-type foo=f/int
--custom-type foo=//foo_base/my/foo.hxx
--custom-type foo=v/wrapper<foo_base>/foo_base
```

The first version instructs the XSD/e compiler not to generate the object model class for the `foo` XML Schema type. The generated code simply forward-declares `foo` as a class and leaves it to you to provide the implementation. The second version is similar to the first, except now we specify the header file which defines the custom implementation. This file is automatically included into the generated header file instead of the standard implementation. The third version is similar to the second, except now we specify that the `foo` type is variable-length. In the previous two cases the type length was determined automatically based on the type definition in the schema. In the fourth version we specify that schema type `foo` is fixed-length and should be mapped to `int`. The fifth version instructs the XSD/e compiler to generate the object model class for type `foo` but call it `foo_base`. It also tells the compiler to generate the `#include` directive with the `my/foo.hxx` file (which presumably defines `foo`) right after the `foo_base` class. Finally, the last version specifies that schema type `foo` is variable-length and should be mapped to `wrapper<foo_base>`. The compiler is also instructed to generate the standard object model class for type `foo` but call it `foo_base`. If you omit the last component (*`include`*), as in the final version, then you can provide the custom type definitions using one of the prologue or epilogue XSD/e compiler options. See the XSD/e Compiler Command Line Manual for details.

Note also that if the type length you specified with the `--custom-type` option differs from the default type length that would have been determined by the XSD/e compiler, then you need to specify this `--custom-type` option when compiling every schema file that includes or imports the schema that defines the type being customized.

As an example, let us add a flag to the `person` class from our person records vocabulary. This flag can be used by the application to keep track of whether a particular person record has been verified. To customize the `person` type we can compile `people.xsd` like this:

```
$ xsde cxx-hybrid --custom-type person=//person_base/person.hxx \
people.xsd
```

The relevant code fragment from the generated header file looks like this:

```
// person_base (fixed-length)
//
class person_base
{
  ...
};

#include "person.hxx"

// people (variable-length)
//
class people
{
  ...

  // person
  //
  typedef xml_schema::fix_sequence<person> person_sequence;
  typedef person_sequence::iterator person_iterator;
  typedef person_sequence::const_iterator person_const_iterator;

  const person_sequence&
  person () const;

  person_sequence&
  person ();

private:
  ...
};
```

We base our custom implementation of the `person` class on generated `person_base` and save it to `person.hxx`:

```
class person: public person_base
{
public:
  person ()
    : verified_ (false)
  {
  }

  bool
  verified () const
  {
    return verified_;
  }

  void
  verified (bool v)
  {
    verified_ = v;
  }

private:
  bool verified_;
};
```

The client code can use our custom implementation as if the flag was part of the vocabulary:

```
people::person_sequence& ps = ...;

for (people::person_iterator i = ps.begin (); i != ps.end (); ++i)
{
  if (!i->verified ())
  {
    // Verify the record.

    ...

    i->verified (true);
  }
}
```

# 4.10 Polymorphic Object Models

When generating polymorphism-aware code (see Section 3.7, "Support for Polymorphism"), some objects in the resulting object model will be polymorphic. By polymorphic we mean that the object's (static) type as specified in the object model's interface may differ from the object's actual (dynamic) type. Because of this, it may be necessary to discover the object's actual type at runtime and cast it to this type to gain access to the object's extended interface. Consider the following schema as an example:

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

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

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

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

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

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

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

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

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

</xs:schema>
```

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

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

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

  <superman can-fly="false">
    <name>James "007" Bond</name>
  </superman>

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

</supermen>
```

When compiling the schema above with the `--generate-polymorphic` option, the XSD/e compiler automatically detects that the type hierarchy starting with the `person` type is polymorphic. A polymorphic type is always variable-length which means objects of polymorphic types are allocated dynamically and are stored and passed around as pointers or references. A polymorphic type also defines a virtual `_clone()` function (see Section 4.2, "Memory Management") and a virtual destructor which allow you to copy and delete an instance of a polymorphic type via a pointer to its base. The following code fragment shows how we can parse, access, modify, and serialize the above XML document:

```
// Parse.
//
supermen_paggr supermen_p;

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

supermen_p.pre ();
doc_p.parse ("supermen.xml");
auto_ptr<supermen> sm (supermen_p.post ());

// Print what we've got.
//
for (supermen::person_iterator i = sm->person ().begin ();
     i != sm->person ().end ();
     ++i)
{
  person& p = *i;

  if (batman* b = dynamic_cast<batman*> (&p))
```

```
  {
    cerr << b->name () << ", batman, wing span " <<
      b->wing_span () << endl;
  }
  else if (superman* s = dynamic_cast<superman*> (&p))
  {
    cerr << s->name () << ", ";

    if (s->can_fly ())
      cerr << "flying ";

    cerr << "superman" << endl;
  }
  else
  {
    cerr << p.name () << ", ordinary person" << endl;
  }
}

// Add another superman entry.
//
auto_ptr<superman> s (new superman);
s->name ("Clark Kent");
s->can_fly (true);
sm->person ().push_back (s.release ());

// Serialize.
//
supermen_saggr supermen_s;

// The last argument to the document's constructor indicates that we
// are serializing polymorphic XML documents.
//
xml_schema::document_simpl doc_s (
  supermen_s.root_serializer (),
  supermen_s.root_name (),
  true);

doc_s.add_no_namespace_schema ("supermen.xsd");

supermen_s.pre (*sm);
doc_s.serialize (cout, xml_schema::document_simpl::pretty_print);
supermen_s.post ();
```

In the example above we used the standard C++ RTTI mechanism to detect the object's actual (dynamic) type. If RTTI is not available on your platform, then you can request the generation of custom runtime type information for polymorphic types with the `--generate-typeinfo` XSD/e compiler option. When this option is specified, each polymorphic type provides the following two public functions:

```
virtual const std::string&
_dynamic_type () const;

static const std::string&
_static_type ();
```

Or, if STL is disabled (Section 3.1, "Standard Template Library"), the following two functions:

```
virtual const char*
_dynamic_type () const;

static const char*
_static_type ();
```

The `_dynamic_type()` function returns the object's dynamic type id. The `_static_type()` function returns the type's static id that can be compared to the dynamic id. The following code fragment shows how we can change the previous example to use custom type information instead of C++ RTTI:

```
for (supermen::person_iterator i = sm->person ().begin ();
     i != sm->person ().end ();
     ++i)
{
  person& p = *i;
  const string& dt = p._dynamic_type ();

  if (dt == batman::_static_type ())
  {
    batman& b = static_cast<batman&> (p)
    cerr << b.name () << ", batman, wing span " <<
      b.wing_span () << endl;
  }
  else if (dt == superman::_static_type ())
  {
    superman& s = static_cast<superman&> (p)
    cerr << s.name () << ", ";

    if (s.can_fly ())
      cerr << "flying ";

    cerr << "superman" << endl;
  }
  else
  {
    cerr << p.name () << ", ordinary person" << endl;
  }
}
```

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

# 5 Mapping for Built-In XML Schema Types

In XML Schema, built-in types, such as `int`, `string`, etc., are defined in the XML Schema namespace. By default this namespace is mapped to C++ namespace `xml_schema` (this mapping can be altered with the `--namespace-map` option). The following table summarizes the mapping of XML Schema built-in types to C++ types in the C++/Hybrid mapping. Declarations for these types are automatically included into each generated header file.

| XML Schema type | Alias in the `xml_schema` namespace | C++ type |
|---|---|---|
| **fixed-length integral types** | | |
| `byte` | `byte` | `signed char` |
| `unsignedByte` | `unsigned_byte` | `unsigned char` |
| `short` | `short_` | `short` |
| `unsignedShort` | `unsigned_short` | `unsigned short` |
| `int` | `int_` | `int` |
| `unsignedInt` | `unsigned_int` | `unsigned int` |
| `long` | `long_` | `long` or<br>`long long`<br>Section 3.5, "64-bit Integer Type" |
| `unsignedLong` | `unsigned_long` | `unsigned long` or<br>`unsigned long long`<br>Section 3.5, "64-bit Integer Type" |
| **arbitrary-length integral types** | | |
| `integer` | `integer` | `long` |
| `nonPositiveInteger` | `non_positive_integer` | `long` |
| `nonNegativeInteger` | `non_negative_integer` | `unsigned long` |
| `positiveInteger` | `positive_integer` | `unsigned long` |

| | | |
|---|---|---|
| negativeInteger | negative_integer | long |
| **boolean types** | | |
| boolean | boolean | bool |
| **fixed-precision floating-point types** | | |
| float | float_ | float |
| double | double_ | double |
| **arbitrary-precision floating-point types** | | |
| decimal | decimal | double |
| **string types** | | |
| string | string | std::string or char* Section 3.1, "Standard Template Library" |
| normalizedString | normalized_string | std::string or char* Section 3.1, "Standard Template Library" |
| token | token | std::string or char* Section 3.1, "Standard Template Library" |
| Name | name | std::string or char* Section 3.1, "Standard Template Library" |
| NMTOKEN | nmtoken | std::string or char* Section 3.1, "Standard Template Library" |
| NMTOKENS | nmtokens | Section 5.2, "Mapping for NMTOKENS and IDREFS" |
| NCName | ncname | std::string or char* Section 3.1, "Standard Template Library" |
| language | language | std::string or char* Section 3.1, "Standard Template Library" |

| qualified name | | |
|---|---|---|
| QName | qname | Section 5.1, "Mapping for QName" |
| **ID/IDREF types** | | |
| ID | id | std::string or char* Section 3.1, "Standard Template Library" |
| IDREF | idref | std::string or char* Section 3.1, "Standard Template Library" |
| IDREFS | idrefs | Section 5.2, "Mapping for NMTOKENS and IDREFS" |
| **URI types** | | |
| anyURI | uri | std::string or char* Section 3.1, "Standard Template Library" |
| **binary types** | | |
| base64Binary | base64_binary | Section 5.3, "Mapping for base64Binary and hexBinary" |
| hexBinary | hex_binary | Section 5.3, "Mapping for base64Binary and hexBinary" |
| **date/time types** | | |
| date | date | Section 5.5, "Mapping for date" |
| dateTime | date_time | Section 5.6, "Mapping for dateTime" |
| duration | duration | Section 5.7, "Mapping for duration" |
| gDay | gday | Section 5.8, "Mapping for gDay" |
| gMonth | gmonth | Section 5.9, "Mapping for gMonth" |

| | | |
|---|---|---|
| gMonthDay | gmonth_day | Section 5.10, "Mapping for gMonthDay" |
| gYear | gyear | Section 5.11, "Mapping for gYear" |
| gYearMonth | gyear_month | Section 5.12, "Mapping for gYearMonth" |
| time | time | Section 5.13, "Mapping for time" |
| **anyType and anySimpleType** | | |
| anyType | any_type | Section 5.14, "Mapping for anyType" |
| anySimpleType | any_simple_type | std::string or char* Section 3.1, "Standard Template Library" |

As you can see from the table above a number of built-in XML Schema types are mapped to fundamental C++ types such as int or bool. All string-based XML Schema types are mapped to either std::string or char*, depending on whether the use of STL is enabled or not. A number of built-in types, such as QName, the binary types, and the date/time types, do not have suitable fundamental or standard C++ types to map to. These types are implemented from scratch in the XSD/e runtime and are discussed in more detail in the subsequent sections.

In cases where the schema calls for an inheritance from a built-in type which is mapped to a fundamental C++ type, a special base type corresponding to the fundamental type and defined in the xml_schema namespace is used (C++ does not allow inheritance from fundamental types). For example:

```
<complexType name="measure">
  <simpleContent>
    <extension base="int">
      <attribute name="unit" type="string" use="required"/>
    </extension>
  </simpleContent>
</complexType>
```

The corresponding object model class is shown below:

```
// measure (fixed-length)
//
class measure: public xml_schema::int_base
{
```

```
public:
  measure ();
  measure (const measure&);
  measure& operator= (const measure&);

  // unit
  //
  const std::string&
  unit () const;

  std::string&
  unit ();

  void
  unit (const std::string&);

private:
  ...
};
```

The `xml_schema::int_base` class has the following interface:

```
namespace xml_schema
{
  class int_base
  {
  public:
    int_base ();

    int_base&
    operator= (int);

  public:
    int
    base_value () const;

    int&
    base_value ();

    void
    base_value (int);

    operator const int& () const;
    operator int& ();
  };
}
```

All other base types for fundamental C++ types have similar interfaces. The only exception is the base type for string types when STL is disabled:

```
namespace xml_schema
{
  class string_base
  {
  public:
    string_base ();

    string_base&
    operator= (char* x)

  public:
    const char*
    base_value () const;

    char*
    base_value ();

    void
    base_value (char* x);

    char*
    base_value_detach ();

    operator const char* () const;
    operator char* ();
  };
}
```

Note that the `string_base` object assumes ownership of the strings passed to the assignment operator and the `base_value()` modifier. If you detach the string value then it should eventually be deallocated with operator `delete[]`.

# 5.1 Mapping for `QName`

The `QName` built-in XML Schema type is mapped to the `qname` class which represents an XML qualified name. With STL enabled (Section 3.1, "Standard Template Library"), it has the following interface:

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

    explicit
```

```
    qname (const std::string& name);
    qname (const std::string& prefix, const std::string& name);

    void
    swap (qname&);

    const std::string&
    prefix () const;

    std::string&
    prefix ();

    void
    prefix (const std::string&);

    const std::string&
    name () const;

    std::string&
    name ();

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

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

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

When STL is disabled and C++ exceptions are enabled (Section 3.3, "C++ Exceptions"), the `qname` type has the following interface:

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

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

    void
    swap (qname&);
```

```
  private:
    qname (const qname&);

    qname&
    operator= (const qname&);

  public:
    char*
    prefix ();

    const char*
    prefix () const;

    void
    prefix (char*);

    void
    prefix_copy (const char*);

    char*
    prefix_detach ();

  public:
    char*
    name ();

    const char*
    name () const;

    void
    name (char*);

    void
    name_copy (const char*);

    char*
    name_detach ();
  };

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

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

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

strings, then they should eventually be deallocated with operator `delete[]`.

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

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

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

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

    void
    swap (qname&);

  private:
    qname (const qname&);

    qname&
    operator= (const qname&);

  public:
    char*
    prefix ();

    const char*
    prefix () const;

    void
    prefix (char*);

    error
    prefix_copy (const char*);

    char*
    prefix_detach ();

  public:
```

```
    char*
    name ();

    const char*
    name () const;

    void
    name (char*);

    error
    name_copy (const char*);

    char*
    name_detach ();
  };

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

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

## 5.2 Mapping for `NMTOKENS` and `IDREFS`

The `NMTOKENS` and `IDREFS` built-in XML Schema types are mapped to the string sequence type which is discussed in Section 4.4, "Attributes and Elements".

## 5.3 Mapping for `base64Binary` and `hexBinary`

The `base64Binary` and `hexBinary` built-in XML Schema types are mapped to the `buffer` class. With C++ exceptions enabled (Section 3.3, "C++ Exceptions"), it 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
    assign (void* data, size_t size);

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

    void*
    detach ();


    void
    swap (buffer&);

  public:
    size_t
    capacity () const;

    bool
    capacity (size_t);

  public:
    size_t
    size () const;

    bool
    size (size_t);

  public:
    const char*
    data () const;

    char*
    data ();

    const char*
    begin () const;
```

```
    char*
    begin ();

    const char*
    end () const;

    char*
    end ();
  };

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

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

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

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

If C++ exceptions are disabled, the `buffer` class 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
    assign (void* data, size_t size);

    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&);
}
```

# 5.4 Time Zone Representation

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

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

    bool
    zone_present () const;

    void
    zone_reset ();

    short
    zone_hours () const;

    void
    zone_hours (short);

    short
    zone_minutes () const;

    void
    zone_minutes (short);
  };

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

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

The `zone_present()` accessor function returns `true` if the time zone is specified. The `zone_reset()` modifier function resets the time zone object to the "not specified" state. If the time zone offset is negative then both hours and minutes components should be negative.

## 5.5 Mapping for `date`

The `date` built-in XML Schema type is mapped to the `date` class which represents a year, a day, and a month with an optional time zone. Its interface is presented below. For more information on the base `xml_schema::time_zone` class refer to Section 5.4, "Time Zone Representation".

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

    date (int year, unsigned short month, unsigned short day);

    date (int year, unsigned short month, unsigned short day,
          short zone_hours, short zone_minutes);

    int
    year () const;

    void
    year (int);

    unsigned short
    month () const;

    void
    month (unsigned short);

    unsigned short
    day () const;

    void
    day (unsigned short);
  };

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

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

## 5.6 Mapping for `dateTime`

The `dateTime` built-in XML Schema type is mapped to the `date_time` class which represents a year, a month, a day, hours, minutes, and seconds with an optional time zone. Its interface is presented below. For more information on the base `xml_schema::time_zone` class refer to Section 5.4, "Time Zone Representation".

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

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

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

    int
    year () const;

    void
    year (int);

    unsigned short
    month () const;

    void
    month (unsigned short);

    unsigned short
    day () const;

    void
    day (unsigned short);

    unsigned short
    hours () const;
```

```
    void
    hours (unsigned short);

    unsigned short
    minutes () const;

    void
    minutes (unsigned short);

    double
    seconds () const;

    void
    seconds (double);
  };

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

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

## 5.7 Mapping for `duration`

The `duration` built-in XML Schema type is mapped to the `duration` class which represents a potentially negative duration in the form of years, months, days, hours, minutes, and seconds. Its interface is presented below.

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

    duration (bool negative,
              unsigned int years, unsigned int months, unsigned int days,
              unsigned int hours, unsigned int minutes, double seconds);

    bool
    negative () const;

    void
    negative (bool);
```

```
    unsigned int
    years () const;

    void
    years (unsigned int);

    unsigned int
    months () const;

    void
    months (unsigned int);

    unsigned int
    days () const;

    void
    days (unsigned int);

    unsigned int
    hours () const;

    void
    hours (unsigned int);

    unsigned int
    minutes () const;

    void
    minutes (unsigned int);

    double
    seconds () const;

    void
    seconds (double);
  };

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

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

# 5.8 Mapping for `gDay`

The `gDay` built-in XML Schema type is mapped to the `gday` class which represents a day of the month with an optional time zone. Its interface is presented below. For more information on the base `xml_schema::time_zone` class refer to Section 5.4, "Time Zone Representation".

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

    explicit
    gday (unsigned short day);

    gday (unsigned short day, short zone_hours, short zone_minutes);

    unsigned short
    day () const;

    void
    day (unsigned short);
  };

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

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

# 5.9 Mapping for `gMonth`

The `gMonth` built-in XML Schema type is mapped to the `gmonth` class which represents a month of the year with an optional time zone. Its interface is presented below. For more information on the base `xml_schema::time_zone` class refer to Section 5.4, "Time Zone Representation".

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

    explicit
    gmonth (unsigned short month);

    gmonth (unsigned short month,
```

```
            short zone_hours, short zone_minutes);

    unsigned short
    month () const;

    void
    month (unsigned short);
  };

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

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

## 5.10 Mapping for `gMonthDay`

The `gMonthDay` built-in XML Schema type is mapped to the `gmonth_day` class which represents a day and a month of the year with an optional time zone. Its interface is presented below. For more information on the base `xml_schema::time_zone` class refer to Section 5.4, "Time Zone Representation".

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

    gmonth_day (unsigned short month, unsigned short day);

    gmonth_day (unsigned short month, unsigned short day,
                short zone_hours, short zone_minutes);

    unsigned short
    month () const;

    void
    month (unsigned short);

    unsigned short
    day () const;

    void
    day (unsigned short);
  };
```

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

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

## 5.11 Mapping for `gYear`

The `gYear` built-in XML Schema type is mapped to the `gyear` class which represents a year with an optional time zone. Its interface is presented below. For more information on the base `xml_schema::time_zone` class refer to Section 5.4, "Time Zone Representation".

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

    explicit
    gyear (int year);

    gyear (int year, short zone_hours, short zone_minutes);

    int
    year () const;

    void
    year (int);
  };

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

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

## 5.12 Mapping for `gYearMonth`

The `gYearMonth` built-in XML Schema type is mapped to the `gyear_month` class which represents a year and a month with an optional time zone. Its interface is presented below. For more information on the base `xml_schema::time_zone` class refer to Section 5.4, "Time

Zone Representation".

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

    gyear_month (int year, unsigned short month);

    gyear_month (int year, unsigned short month,
                 short zone_hours, short zone_minutes);

    int
    year () const;

    void
    year (int);

    unsigned short
    month () const;

    void
    month (unsigned short);
  };

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

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

# 5.13 Mapping for `time`

The `time` built-in XML Schema type is mapped to the `time` class which represents hours, minutes, and seconds with an optional time zone. Its interface is presented below. For more information on the base `xml_schema::time_zone` class refer to Section 5.4, "Time Zone Representation".

```
namespace xml_schema
{
  class time: public time_zone
  {
  public:
```

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

    time (unsigned short hours, unsigned short minutes, double seconds);

    time (unsigned short hours, unsigned short minutes, double seconds,
          short zone_hours, short zone_minutes);

    unsigned short
    hours () const;

    void
    hours (unsigned short);

    unsigned short
    minutes () const;

    void
    minutes (unsigned short);

    double
    seconds () const;

    void
    seconds (double);
  };

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

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

## 5.14 Mapping for `anyType`

The `anyType` built-in XML Schema type is mapped to the `any_type` class in the `xml_schema` namespace. With C++ exceptions enabled (Section 3.3, "C++ Exceptions"), it has the following interface:

```
namespace xml_schema
{
  class any_type
  {
  public:
    // Custom data.
    //
    typedef xml_schema::data_sequence custom_data_sequence;
```

```
    typedef custom_data_sequence::iterator custom_data_iterator;
    typedef custom_data_sequence::const_iterator custom_data_const_iterator;

    void
    allocate_custom_data ();

    const custom_data_sequence&
    custom_data () const;

    custom_data_sequence&
    custom_data ();
  };
}
```

If C++ exceptions are disabled, the `any_type` class has the following interface:

```
namespace xml_schema
{
  class any_type
  {
  public:
    // Custom data.
    //
    typedef xml_schema::data_sequence custom_data_sequence;
    typedef custom_data_sequence::iterator custom_data_iterator;
    typedef custom_data_sequence::const_iterator custom_data_const_iterator;

    bool
    allocate_custom_data ();

    const custom_data_sequence&
    custom_data () const;

    custom_data_sequence&
    custom_data ();
  };
}
```

The `allocate_custom_data()` function allocates the custom data sequence. With C++ exceptions disabled, it returns `false` if memory allocation has failed and `true` otherwise. For more information on custom data, refer to Section 4.9, "Customizing the Object Model".

The default parser and serializer implementations for the `anyType` built-in type ignore all its content and return an empty `any_type` instance. If your application needs to access this content, then you will need to provide your own implementations of these parser and serializer and use the custom data sequence to store the extracted data.

# 6 Parsing and Serialization

As was mentioned in the introduction, the C++/Hybrid mapping uses the C++/Parser and C++/Serializer mappings for XML parsing and serialization. If your parsing and serialization requirements are fairly basic, for example, parsing from and serializing to a file or a memory buffer, then you don't need to concern yourself with these two underlying mappings. On the other hand, the C++/Parser and C++/Serializer mappings provide well-defined APIs which allow a great amount of flexibility that may be useful in certain situations. In such cases, you may need to get an understanding of how the C++/Parser and C++/Serializer mappings work. See the Embedded C++/Parser Mapping Getting Started Guide and the Embedded C++/Serializer Mapping Getting Started Guide for more detailed information on these mappings.

For each type defined in XML Schema, the C++/Parser and C++/Serializer mappings generate a parser skeleton class and serializer skeleton class, respectively. These classes manage parsing/serialization state, convert data between text and C++ types, and perform XML Schema validation, if enabled. Parser skeletons deliver the parsed data and serializer skeletons request the data to be serialized with callbacks. These callbacks are implemented by parser and serializer implementation classes that are derived from the skeletons. If the application uses the C++/Parser and C++/Serializer mappings directly, these implementation classes are normally written by the application developer to perform some application-specific actions. In case of the C++/Hybrid mapping, these implementations are automatically generated by the XSD/e compiler to parse XML to object models and to serialize object models to XML. To request the generation of parser skeletons and implementations, you need to specify the `--generate-parser` XSD/e command line option. Similarly, to generate serializer skeletons and implementations, you will need to use the `--generate-serializer` option.

Before an XML document can be parsed or serialized, the individual parser and serializer implementations need to be instantiated and connected to each other. Again, if the application uses the C++/Parser and C++/Serializer mappings directly, this is done by the application developer. While you can also do this with the generated C++/Hybrid parser and serializer implementations, it is easier to request the generation of parser and serializer aggregate classes with the `--generate-aggregate` options. Aggregate classes instantiate and connect all the necessary individual parser and serializer implementations for a particular root element or type. Consider again the `hello.xsd` schema from Chapter 2, "Hello World Example":

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

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

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

</xs:schema>
```

If we compile this schema with the `--generate-parser`, `--generate-serializer`, and `--generate-aggregate` options, we will have two aggregate classes, `hello_paggr` and `hello_saggr`, generated for the root `hello` element. The interface of the `hello_paggr` class is presented below:

```
class hello_paggr
{
public:
  hello_paggr ();

  void
  pre ();

  hello*
  post ();

  hello_pimpl&
  root_parser ();

  static const char*
  root_name ();

  static const char*
  root_namespace ();
};
```

The `pre()` and `post()` functions call the corresponding callbacks on the root parser implementation. The `root_parser()` function returns the root parser implementation. The `root_name()` and `root_namespace()` functions return the root element name and namespace, respectively.

As was shown in Chapter 2, "Hello World Example", we can use this parser aggregate to create the document parser (supplied by the C++/Parser mapping) and perform the parsing:

```
hello_paggr hello_p;
xml_schema::document_pimpl doc_p (hello_p.root_parser (),
                                  hello_p.root_name ());
hello_p.pre ();
doc_p.parse ("hello.xml");
hello* h = hello_p.post ();
```

For more information on the `document_pimpl` class, including the other variants of the `parse()` function as well as error handling during parsing, see Chapter 7, "Document Parser and Error Handling" in the Embedded C++/Parser Mapping Getting Started Guide.

The interface of the `hello_saggr` serializer aggregate mirrors that of `hello_paggr` and is presented below:

```
class hello_saggr
{
public:
  hello_saggr ();

  void
  pre (const hello&);

  void
  post ();

  hello_simpl&
  root_serializer ();

  static const char*
  root_name ();

  static const char*
  root_namespace ();
};
```

The `pre()` and `post()` functions call the corresponding callbacks on the root serializer implementation. The `root_serializer()` function returns the root serializer implementation. The `root_name()` and `root_namespace()` functions return the root element name and namespace, respectively.

As was shown in Chapter 2, "Hello World Example", we can use this serializer aggregate to create the document serializer (supplied by the C++/Serializer mapping) and perform the serialization:

```
hello_saggr hello_s;
xml_schema::document_simpl doc_s (hello_s.root_serializer (),
                                  hello_s.root_name ());
hello_s.pre (*h);
doc_s.serialize (std::cout, xml_schema::document_simpl::pretty_print);
hello_s.post ();
```

For more information on the `document_simpl` class, including the other variants of the `serialize()` function as well as error handling during serialization, see Chapter 8, "Document Serializer and Error Handling" in the Embedded C++/Serializer Mapping Getting Started Guide.

# 6.1 Customizing Parsers and Serializers

The C++/Hybrid mapping allows you to customize the generated parser and serializer implementations. This mechanism can be used, for example, to implement filtering, partially event-driven XML processing, as well as parsing of content matched by XML Schema wildcards. Filtering allows only parts of the XML document to be parsed into the object model or only parts of the object model to be serialized to XML. With partially event-driven parsing and serialization, we can process parts of the document as they become available as well as handle documents that are too large to fit into memory. This section expects you to have an understanding of the C++/Parser and C++/Serializer programming models.

To request customization of a parser or serializer implementation, you will need to specify the `--custom-parser` or `--custom-serializer` option, respectively. The argument format for these two options is `name[=[base][/include]]`. The *`name`* component is the XML Schema type name being customized. Optional *`base`* is a C++ name that should be given to the generated version. It is normally used as a base for the custom implementation. Optional *`include`* is the header file that defines the custom implementation. It is `#include`'ed into the generated code immediately after (if *`base`* is specified) or instead of the generated version. The following examples show how we can use these options:

```
--custom-parser foo
--custom-parser foo=foo_base_pimpl
--custom-parser foo=foo_base_pimpl/foo/foo-custom.hxx
--custom-parser foo=/foo/foo-custom.hxx
```

The first version instructs the XSD/e compiler not to generate the parser implementation for the `foo` XML Schema type. The second version instructs the compiler to generate the parser implementation for type `foo` but call it `foo_base_pimpl`. The third version is similar to the second except that the compiler generates the `#include` directive with the `foo/foo-custom.hxx` file (which presumably defines `foo_pimpl`) right after the `foo_base_pimpl` class. The last version instructs the XSD/e compiler to include `foo/foo-custom.hxx` instead of generating the parser implementation for `foo`. If you omit the last component (*`include`*), then you can include the custom parser/serializer definitions using one of the prologue or epilogue XSD/e compiler options. See the XSD/e Compiler Command Line Manual for details.

Once you specify the `--custom-parser` or `--custom-serializer` option, you will need to provide the custom implementation. You have a choice of either basing it on the generated version and overriding some callbacks or implementing it from scratch.

In the remainder of this section we will examine how to customize the `people` parser and serializer implementations from the example presented in Chapter 4, "Working with Object Models". Our custom parser implementation will filter the records being parsed based on a person's age. Similarly, the serializer will only serialize records of a specific gender. The code presented below is taken from the `filter` example in the XSD/e distribution. Other examples related to

parser/serializer customization are `wildcard` and `streaming`.

First, we compile the `people.xsd` schema and instruct the XSD/e compiler to customize the parser and serializer implementations for the `people` XML Schema type:

```
$ xsde cxx-hybrid --generate-parser --generate-serializer \
--custom-parser people=people_base_pimpl/people-custom-pimpl.hxx \
--custom-serializer people=people_base_simpl/people-custom-simpl.hxx \
--generate-aggregate people.xsd
```

The custom `people_pimpl` parser implementation is based on the generated version and is saved to `people-custom-pimpl.hxx`:

```
class people_pimpl: public people_base_pimpl
{
public:
  void
  age_filter (unsigned short min, unsigned short max)
  {
    min_age_ = min;
    max_age_ = max;
  }

  virtual void
  person (const ::person& p)
  {
    // Check if the age constraints are met.
    //
    unsigned short age = p.age ();

    if (age >= min_age_ && age <= max_age_)
      people_base_pimpl::person (p);
  }

private:
  unsigned short min_age_;
  unsigned short max_age_;
};
```

Here we override the `person()` callback and, if the filter conditions are satisfied, call the original version which adds the person record to the object model. Note that if the `person` object model class were variable-length, then the instance would be dynamically allocated and passed as a pointer. In this situation, if we don't use the object, we need to delete it, for example:

```
virtual void
person (const ::person* p)
{
  unsigned short age = p->age ();

  if (age >= min_age_ && age <= max_age_)
    people_base_pimpl::person (p);
  else
    delete p;
}
```

The custom `people_simpl` parser implementation is also based on the generated version and is saved to `people-custom-simpl.hxx`:

```
class people_simpl: public people_base_simpl
{
public:
  void
  gender_filter (gender g)
  {
    gender_ = g;
  }

  virtual bool
  person_next ()
  {
    // See if we have any more person records with the gender we
    // are interested in.
    //
    people::person_const_iterator& i = people_base_simpl_state_.person_;
    people::person_const_iterator& e = people_base_simpl_state_.person_end_;

    for (; i != e; ++i)
    {
      if (i->gender () == gender_)
        break;
    }

    return i != e;
  }

private:
  gender gender_;
};
```

Here we override the `person_next()` callback where we locate the next record that satisfies the filter conditions. Note that we use the serialization state provided by the generated `people_base_simpl` implementation.

The following code fragment shows a test driver that uses the above implementations to filter the data during parsing and serialization:

```
#include <memory>
#include <iostream>

#include "people.hxx"

#include "people-pimpl.hxx"
#include "people-simpl.hxx"

using namespace std;

int
main (int argc, char* argv[])
{
  // Parse.
  //
  people_paggr people_p;
  people_pimpl& root_p = people_p.root_parser ();

  // Initialize the filter.
  //
  root_p.age_filter (1, 30);

  xml_schema::document_pimpl doc_p (root_p, people_p.root_name ());

  people_p.pre ();
  doc_p.parse (argv[1]);
  auto_ptr<people> ppl (people_p.post ());

  // Print what we've got.
  //
  people::person_sequence& ps = ppl->person ();

  for (people::person_iterator i = ps.begin (); i != ps.end (); ++i)
  {
    cerr << "first:  " << i->first_name () << endl
         << "last:   " << i->last_name () << endl
         << "gender: " << i->gender ().string () << endl
         << "age:    " << i->age () << endl
         << endl;
  }

  // Serialize.
  //
  people_saggr people_s;
  people_simpl& root_s = people_s.root_serializer ();

  // Initialize the filter.
  //
```

```
  root_s.gender_filter (gender::female);

  xml_schema::document_simpl doc_s (root_s, people_s.root_name ());

  people_s.pre (*ppl);
  doc_s.serialize (cout, xml_schema::document_simpl::pretty_print);
  people_s.post ();
}
```

If we run this test driver on the following XML document:

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

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

  <person>
    <first-name>Jane</first-name>
    <last-name>Doe</last-name>
    <gender>female</gender>
    <age>28</age>
  </person>

  <person>
    <first-name>Joe</first-name>
    <last-name>Dirt</last-name>
    <gender>male</gender>
    <age>25</age>
  </person>

</people>
```

We will get the following output:

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

first:  Joe
last:   Dirt
gender: male
age:    25

<people>
  <person>
```

```
    <first-name>Jane</first-name>
    <last-name>Doe</last-name>
    <gender>female</gender>
    <age>28</age>
  </person>
</people>
```

# 7 Binary Representation

Besides reading from and writing to XML, the C++/Hybrid mapping also allows you to save the object model to and load it from a number of predefined as well as custom data representation formats. The predefined binary formats are CDR (Common Data Representation) and XDR (eXternal Data Representation). A custom format can easily be supported by providing insertion and extraction operators for basic types.

Binary representations contain only the data without any meta information or markup. Consequently, saving to and loading from a binary representation can be an order of magnitude faster as well as result in a much smaller footprint compared to parsing and serializing the same data in XML. Furthermore, the resulting representation is normally several times smaller than the equivalent XML representation. These properties make a binary representation ideal for internal data exchange and storage. A typical application that uses this facility stores the data and communicates within the system using a binary format and reads/writes the data in XML when communicating with the outside world.

In order to request the generation of insertion and extraction operators for a specific predefined or custom data representation stream, you will need to use the `--generate-insertion` and `--generate-extraction` compiler options. See the XSD/e Compiler Command Line Manual for more information.

The XSD/e runtime provides implementations of the base insertion and extraction operators for the ACE (Adaptive Communication Environment) CDR streams and the XDR API. The XDR API is available out of the box on most POSIX systems as part of Sun RPC. On other platforms you may need to install a third-party library which provides the XDR API. The XSD/e compiler recognizes two special argument values to the `--generate-insertion` and `--generate-extraction` options: `CDR` and `XDR`. When one of these arguments is specified, the corresponding implementation from the XSD/e runtime is automatically used. The following two sections describe each of these two formats in more detail. It is also possible to add support for saving the object model to and loading it from custom data representation formats as discussed in the last section of this chapter.

The saving of the object model types to a representation stream is implemented with stream insertion operators (`operator<<`). Similarly, loading of the object model from a representation stream is implemented with stream extraction operators (`operator>>`). The insertion and extraction operators for the built-in XML Schema types as well as the sequence templates are

provided by the stream implementation (that is, by the XSD/e runtime in case of CDR and XDR and by you for custom formats). The XSD/e compiler automatically generates insertion and extraction operators for the generated object model types.

When C++ exceptions are enabled (Section 3.3, "C++ Exceptions"), the signatures of the insertion and extraction operators are as follows:

```
void
operator<< (ostream&, const type&);

void
operator>> (istream&, type&);
```

The insertion and extraction errors are indicated by throwing stream-specific exceptions. When C++ exceptions are disabled, the signatures of the insertion and extraction operators are as follows:

```
bool
operator<< (ostream&, const type&);

bool
operator>> (istream&, type&);
```

In this case the insertion and extraction operators return `true` if the operation was successful and `false` otherwise. The stream object may provide additional error information.

# 7.1 CDR (Common Data Representation)

When you request the generation of CDR stream insertion and extraction operators, the `ocdrstream` and `icdrstream` types are defined in the `xml_schema` namespace. Additionally, if C++ exceptions are enabled, the `cdr_exception` exception is also defined in `xml_schema`. The `icdrstream` and `ocdrstream` types are simple wrappers for the ACE_InputCDR and ACE_OutputCDR streams. The following code fragment shows how we can use these types when C++ exceptions are enabled:

```
try
{
  const type& x = ... // Object model.

  // Save to a CDR stream.
  //
  ACE_OutputCDR ace_ocdr;
  xml_schema::ocdrstream ocdr (ace_ocdr);

  ocdr << x;

  // Load from a CDR stream.
```

```
  //
  ACE_InputCDR ace_icdr (buf, size);
  xml_schema::icdrstream icdr (ace_icdr);

  type copy;
  icdr >> copy;
}
catch (const xml_schema::cdr_exception&)
{
  cerr << "CDR operation failed" << endl;
}
```

The same code fragment but when C++ exceptions are disabled:

```
const type& x = ... // Object model.

// Save to a CDR stream.
//
ACE_OutputCDR ace_ocdr;
xml_schema::ocdrstream ocdr (ace_ocdr);

if (!(ocdr << x))
{
  cerr << "CDR operation failed" << endl;
}

// Load from a CDR stream.
//
ACE_InputCDR ace_icdr (buf, size);
xml_schema::icdrstream icdr (ace_icdr);

type copy;

if (!(icdr >> copy))
{
  cerr << "CDR operation failed" << endl;
}
```

The `cdr` example which can be found in the `examples/cxx/hybrid/binary/` directory of the XSD/e distribution includes complete source code that shows how to save the object model to and load it from the CDR format.

# 7.2 XDR (eXternal Data Representation)

When you request the generation of XDR stream insertion and extraction operators, the `oxdrstream` and `xcdrstream` types are defined in the `xml_schema` namespace. Additionally, if C++ exceptions are enabled, the `xdr_exception` exception is also defined in `xml_schema`. The `ixdrstream` and `oxdrstream` types are simple wrappers for the XDR API. The following code fragment shows how we can use these types when C++ exceptions are

enabled:

```
try
{
  const type& x = ... // Object model.

  // Save to a XDR stream.
  //
  XDR xdr;
  xdrrec_create (&xdr, ...);
  xml_schema::oxdrstream oxdr (xdr);

  oxdr << x;

  // Load from a XDR stream.
  //
  xdrrec_create (&xdr, ...);
  xml_schema::ixdrstream ixdr (xdr);

  type copy;
  ixdr >> copy;
}
catch (const xml_schema::xdr_exception&)
{
  cerr << "XDR operation failed" << endl;
}
```

The same code fragment but when C++ exceptions are disabled:

```
const type& x = ... // Object model.

// Save to a XDR stream.
//
XDR xdr;
xdrrec_create (&xdr, ...);
xml_schema::oxdrstream oxdr (xdr);

if (!(oxdr << x))
{
  cerr << "XDR operation failed" << endl;
}

// Load from a XDR stream.
//
xdrrec_create (&xdr, ...);
xml_schema::ixdrstream ixdr (xdr);

type copy;
```

```
if (!(ixdr >> copy))
{
  cerr << "XDR operation failed" << endl;
}
```

The `xdr` example which can be found in the `examples/cxx/hybrid/binary/` directory of the XSD/e distribution includes complete source code that shows how to save the object model to and load it from the XDR format.

## 7.3 Custom Representations

To add support for saving the object model to and loading it from a custom format, you will need to perform the following general steps:

1. Generate a header file corresponding to the XML Schema namespace using the `--generate-xml-schema` compiler option.
2. Implement custom stream insertion and extraction operators for the built-in XML Schema types and sequence templates. Include the header file obtained in the previous step to get definitions for these types.
3. Compile your schemas with the `--generate-insertion` and `--generate-extraction` options. The arguments to these options will be your custom output and input stream types, respectively. Use the `--hxx-prologue` option to include the definitions for these stream types into the generated code. Also use the `--extern-xml-schema` option to include the header file obtained in the first step instead of generating the same code directly.

The `custom` example which can be found in the `examples/cxx/hybrid/binary/` directory of the XSD/e distribution includes complete source code that shows how to save the object model to and load it from a custom format using the raw binary representation as an example. You can use the source code from this example as a base to implement support for your own format.