C++/Tree Mapping User Manual

Revision 4.1.0 September 2017

Copyright © 2005-2017 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
2 C++/Tree Mapping 2
2.1 Preliminary Information 2
2.1.1 C++ Standard 2
2.1.2 Identifiers 2
2.1.3 Character Type and Encoding 3
2.1.4 XML Schema Namespace 3
2.1.5 Anonymous Types 4
2.2 Error Handling 4
2.2.1 `xml_schema::duplicate_id` 5
2.3 Mapping for `import` and `include` 5
2.3.1 Import 5
2.3.2 Inclusion with Target Namespace 6
2.3.3 Inclusion without Target Namespace 6
2.4 Mapping for Namespaces 7
2.5 Mapping for Built-in Data Types 7
2.5.1 Inheritance from Built-in Data Types 10
2.5.2 Mapping for `anyType` 11
2.5.3 Mapping for `anySimpleType` 12
2.5.4 Mapping for `QName` 13
2.5.5 Mapping for `IDREF` 14
2.5.6 Mapping for `base64Binary` and `hexBinary` 16
2.5.7 Time Zone Representation 19
2.5.8 Mapping for `date` 20
2.5.9 Mapping for `dateTime` 21
2.5.10 Mapping for `duration` 22
2.5.11 Mapping for `gDay` 24
2.5.12 Mapping for `gMonth` 24
2.5.13 Mapping for `gMonthDay` 25
2.5.14 Mapping for `gYear` 26
2.5.15 Mapping for `gYearMonth` 27
2.5.16 Mapping for `time` 28
2.6 Mapping for Simple Types 29
2.6.1 Mapping for Derivation by Restriction 29
2.6.2 Mapping for Enumerations 30
2.6.3 Mapping for Derivation by List 31
2.6.4 Mapping for Derivation by Union 32

2.7 Mapping for Complex Types 33
2.7.1 Mapping for Derivation by Extension 37
2.7.2 Mapping for Derivation by Restriction 37
2.8 Mapping for Local Elements and Attributes 37
2.8.1 Mapping for Members with the One Cardinality Class 39
2.8.2 Mapping for Members with the Optional Cardinality Class 41
2.8.3 Mapping for Members with the Sequence Cardinality Class 45
2.8.4 Element Order 48
2.9 Mapping for Global Elements 55
2.9.1 Element Types 55
2.9.2 Element Map 58
2.10 Mapping for Global Attributes 59
2.11 Mapping for `xsi:type` and Substitution Groups 60
2.12 Mapping for `any` and `anyAttribute` 62
2.12.1 Mapping for `any` with the One Cardinality Class 64
2.12.2 Mapping for `any` with the Optional Cardinality Class 65
2.12.3 Mapping for `any` with the Sequence Cardinality Class 68
2.12.4 Element Wildcard Order 74
2.12.5 Mapping for `anyAttribute` 75
2.13 Mapping for Mixed Content Models 80
3 Parsing 82
3.1 Initializing the Xerces-C++ Runtime 85
3.2 Flags and Properties 85
3.3 Error Handling 86
3.3.1 `xml_schema::parsing` 88
3.3.2 `xml_schema::expected_element` 89
3.3.3 `xml_schema::unexpected_element` 89
3.3.4 `xml_schema::expected_attribute` 90
3.3.5 `xml_schema::unexpected_enumerator` 91
3.3.6 `xml_schema::expected_text_content` 91
3.3.7 `xml_schema::no_type_info` 91
3.3.8 `xml_schema::not_derived` 92
3.3.9 `xml_schema::no_prefix_mapping` 92
3.4 Reading from a Local File or URI 93
3.5 Reading from `std::istream` 93
3.6 Reading from `xercesc::InputSource` 94
3.7 Reading from DOM 94
4 Serialization 94
4.1 Initializing the Xerces-C++ Runtime 96
4.2 Namespace Infomap and Character Encoding 97
4.3 Flags 99
4.4 Error Handling 99
4.4.1 `xml_schema::serialization` 100

4.4.2 xml_schema::unexpected_element 100
4.4.3 xml_schema::no_type_info 100
4.5 Serializing to std::ostream 101
4.6 Serializing to xercesc::XMLFormatTarget 101
4.7 Serializing to DOM 102
5 Additional Functionality 103
5.1 DOM Association 103
5.2 Binary Serialization 106
Appendix A — Default and Fixed Values 107

Preface

About This Document

This document describes the mapping of W3C XML Schema to the C++ programming language as implemented by CodeSynthesis XSD - an XML Schema to C++ data binding compiler. The mapping represents information stored in XML instance documents as a statically-typed, tree-like in-memory data structure and is called C++/Tree.

Revision 4.1.0
This revision of the manual describes the C++/Tree mapping as implemented by CodeSynthesis XSD version 4.1.0.

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

More Information

Beyond this manual, you may also find the following sources of information useful:

- C++/Tree Mapping Getting Started Guide
- C++/Tree Mapping Customization Guide
- C++/Tree Mapping Frequently Asked Questions (FAQ)
- XSD Compiler Command Line Manual
- The `examples/cxx/tree/` directory in the XSD distribution contains a collection of examples and a README file with an overview of each example.
- The `README` file in the XSD distribution explains how to compile the examples on various platforms.
- The xsd-users mailing list is a place to ask questions. Furthermore the archives may already have answers to some of your questions.

1 Introduction

C++/Tree is a W3C XML Schema to C++ mapping that represents the data stored in XML as a statically-typed, vocabulary-specific object model. Based on a formal description of an XML vocabulary (schema), the C++/Tree mapping produces a tree-like data structure suitable for in-memory processing as well as XML parsing and serialization code.

A typical application that processes XML documents usually performs the following three steps: it first reads (parses) an XML instance document to an object model, it then performs some useful computations on that model which may involve modification of the model, and finally it may write (serialize) the modified object model back to XML.

The C++/Tree mapping consists of C++ types that represent the given vocabulary (Chapter 2, "C++/Tree Mapping"), a set of parsing functions that convert XML documents to a tree-like in-memory data structure (Chapter 3, "Parsing"), and a set of serialization functions that convert the object model back to XML (Chapter 4, "Serialization"). Furthermore, the mapping provides a number of additional features, such as DOM association and binary serialization, that can be useful in some applications (Chapter 5, "Additional Functionality").

2 C++/Tree Mapping

2.1 Preliminary Information

2.1.1 C++ Standard

The C++/Tree mapping provides support for ISO/IEC C++ 1998/2003 (C++98) and ISO/IEC C++ 2011 (C++11). To select the C++ standard for the generated code we use the `--std` XSD compiler command line option. While the majority of the examples in this manual use C++98, support for the new functionality and library components introduced in C++11 are discussed throughout the document.

2.1.2 Identifiers

XML Schema names may happen to be reserved C++ keywords or contain characters that are illegal in C++ identifiers. To avoid C++ compilation problems, such names are changed (escaped) when mapped to C++. If an XML Schema name is a C++ keyword, the "_" suffix is added to it. All character of an XML Schema name that are not allowed in C++ identifiers are replaced with "_".

For example, XML Schema name `try` will be mapped to C++ identifier `try_`. Similarly, XML Schema name `strange.na-me` will be mapped to C++ identifier `strange_na_me`.

Furthermore, conflicts between type names and function names in the same scope are resolved using name escaping. Such conflicts include both a global element (which is mapped to a set of parsing and/or serialization functions or element types, see Section 2.9, "Mapping for Global Elements") and a global type sharing the same name as well as a local element or attribute inside a type having the same name as the type itself.

For example, if we had a global type `catalog` and a global element with the same name then the type would be mapped to a C++ class with name `catalog` while the parsing functions corresponding to the global element would have their names escaped as `catalog_`.

By default the mapping uses the so-called K&R (Kernighan and Ritchie) identifier naming convention which is also used throughout this manual. In this convention both type and function names are in lower case and words are separated by underscores. If your application code or schemas use a different notation, you may want to change the naming convention used by the mapping for consistency. The compiler supports a set of widely-used naming conventions that you can select with the `--type-naming` and `--function-naming` options. You can also further refine one of the predefined conventions or create a completely custom naming scheme by using the `--*-regex` options. For more detailed information on these options refer to the NAMING CONVENTION section in the XSD Compiler Command Line Manual.

2.1.3 Character Type and Encoding

The code that implements the mapping, depending on the `--char-type` option, is generated using either `char` or `wchar_t` as the character type. In this document code samples use symbol `C` to refer to the character type you have selected when translating your schemas, for example `std::basic_string<C>`.

Another aspect of the mapping that depends on the character type is character encoding. For the `char` character type the default encoding is UTF-8. Other supported encodings are ISO-8859-1, Xerces-C++ Local Code Page (LPC), as well as custom encodings and can be selected with the `--char-encoding` command line option.

For the `wchar_t` character type the encoding is automatically selected between UTF-16 and UTF-32/UCS-4 depending on the size of the `wchar_t` type. On some platforms (for example, Windows with Visual C++ and AIX with IBM XL C++) `wchar_t` is 2 bytes long. For these platforms the encoding is UTF-16. On other platforms `wchar_t` is 4 bytes long and UTF-32/UCS-4 is used.

2.1.4 XML Schema Namespace

The mapping relies on some predefined types, classes, and functions that are logically defined in the XML Schema namespace reserved for the XML Schema language (`http://www.w3.org/2001/XMLSchema`). By default, this namespace is mapped to C++ namespace `xml_schema`. It is automatically accessible from a C++ compilation unit that includes a header file generated from an XML Schema definition.

Note that, if desired, the default mapping of this namespace can be changed as described in Section 2.4, "Mapping for Namespaces".

2.1.5 Anonymous Types

For the purpose of code generation, anonymous types defined in XML Schema are automatically assigned names that are derived from enclosing attributes and elements. Otherwise, such types follows standard mapping rules for simple and complex type definitions (see Section 2.6, "Mapping for Simple Types" and Section 2.7, "Mapping for Complex Types"). For example, in the following schema fragment:

```
<element name="object">
  <complexType>
    ...
  </complexType>
</element>
```

The anonymous type defined inside element `object` will be given name `object`. The compiler has a number of options that control the process of anonymous type naming. For more information refer to the XSD Compiler Command Line Manual.

2.2 Error Handling

The mapping uses the C++ exception handling mechanism as a primary way of reporting error conditions. All exceptions that are specified in this mapping derive from `xml_schema::exception` which itself is derived from `std::exception`:

```
struct exception: virtual std::exception
{
  friend
  std::basic_ostream<C>&
  operator<< (std::basic_ostream<C>& os, const exception& e)
  {
    e.print (os);
    return os;
  }

protected:
  virtual void
  print (std::basic_ostream<C>&) const = 0;
};
```

The exception hierarchy supports "virtual" `operator<<` which allows you to obtain diagnostics corresponding to the thrown exception using the base exception interface. For example:

```
try
{
  ...
}
catch (const xml_schema::exception& e)
{
  cerr << e << endl;
}
```

The following sub-sections describe exceptions thrown by the types that constitute the object model. Section 3.3, "Error Handling" of Chapter 3, "Parsing" describes exceptions and error handling mechanisms specific to the parsing functions. Section 4.4, "Error Handling" of Chapter 4, "Serialization" describes exceptions and error handling mechanisms specific to the serialization functions.

2.2.1 `xml_schema::duplicate_id`

```
struct duplicate_id: virtual exception
{
  duplicate_id (const std::basic_string<C>& id);

  const std::basic_string<C>&
  id () const;

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

The `xml_schema::duplicate_id` is thrown when a conflicting instance of `xml_schema::id` (see Section 2.5, "Mapping for Built-in Data Types") is added to a tree. The offending ID value can be obtained using the `id` function.

2.3 Mapping for `import` and `include`

2.3.1 Import

The XML Schema `import` element is mapped to the C++ Preprocessor `#include` directive. The value of the `schemaLocation` attribute is used to derive the name of the header file that appears in the `#include` directive. For instance:

```
<import namespace="http://www.codesynthesis.com/test"
        schemaLocation="test.xsd"/>
```

is mapped to:

```
#include "test.hxx"
```

Note that you will need to compile imported schemas separately in order to produce corresponding header files.

2.3.2 Inclusion with Target Namespace

The XML Schema `include` element which refers to a schema with a target namespace or appears in a schema without a target namespace follows the same mapping rules as the `import` element, see Section 2.3.1, "Import".

2.3.3 Inclusion without Target Namespace

For the XML Schema `include` element which refers to a schema without a target namespace and appears in a schema with a target namespace (such inclusion sometimes called "chameleon inclusion"), declarations and definitions from the included schema are generated in-line in the namespace of the including schema as if they were declared and defined there verbatim. For example, consider the following two schemas:

```
<-- common.xsd -->
<schema>
  <complexType name="type">
  ...
  </complexType>
</schema>

<-- test.xsd -->
<schema targetNamespace="http://www.codesynthesis.com/test">
  <include schemaLocation="common.xsd"/>
</schema>
```

The fragment of interest from the generated header file for `text.xsd` would look like this:

```
// test.hxx
namespace test
{
  class type
  {
    ...
  };
}
```

2.4 Mapping for Namespaces

An XML Schema namespace is mapped to one or more nested C++ namespaces. XML Schema namespaces are identified by URIs. 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 instance:

```
<schema targetNamespace="http://www.codesynthesis.com/system/test">
  ...
</schema>
```

is mapped to:

```
namespace system
{
  namespace test
  {
    ...
  }
}
```

The default mapping of namespace URIs to C++ namespace names can be altered using the `--namespace-map` and `--namespace-regex` options. See the XSD Compiler Command Line Manual for more information.

2.5 Mapping for Built-in Data Types

The mapping of XML Schema built-in data types to C++ types is summarized in the table below.

XML Schema type	Alias in the `xml_schema` namespace	C++ type
anyType and anySimpleType types		
`anyType`	`type`	Section 2.5.2, "Mapping for `anyType`"
`anySimpleType`	`simple_type`	Section 2.5.3, "Mapping for `anySimpleType`"
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 long
unsignedLong	unsigned_long	unsigned long long
arbitrary-length integral types		
integer	integer	long long
nonPositiveInteger	non_positive_integer	long long
nonNegativeInteger	non_negative_integer	unsigned long long
positiveInteger	positive_integer	unsigned long long
negativeInteger	negative_integer	long 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	type derived from std::basic_string
normalizedString	normalized_string	type derived from string
token	token	type derived from normalized_string
Name	name	type derived from token
NMTOKEN	nmtoken	type derived from token
NMTOKENS	nmtokens	type derived from sequence<nmtoken>
NCName	ncname	type derived from name
language	language	type derived from token
qualified name		
QName	qname	Section 2.5.4, "Mapping for QName"
ID/IDREF types		

ID	id	type derived from ncname
IDREF	idref	Section 2.5.5, "Mapping for IDREF"
IDREFS	idrefs	type derived from sequence<idref>
URI types		
anyURI	uri	type derived from std::basic_string
binary types		
base64Binary	base64_binary	Section 2.5.6, "Mapping for base64Binary and hexBinary"
hexBinary	hex_binary	
date/time types		
date	date	Section 2.5.8, "Mapping for date"
dateTime	date_time	Section 2.5.9, "Mapping for dateTime"
duration	duration	Section 2.5.10, "Mapping for duration"
gDay	gday	Section 2.5.11, "Mapping for gDay"
gMonth	gmonth	Section 2.5.12, "Mapping for gMonth"
gMonthDay	gmonth_day	Section 2.5.13, "Mapping for gMonthDay"
gYear	gyear	Section 2.5.14, "Mapping for gYear"
gYearMonth	gyear_month	Section 2.5.15, "Mapping for gYearMonth"
time	time	Section 2.5.16, "Mapping for time"
entity types		
ENTITY	entity	type derived from name
ENTITIES	entities	type derived from sequence<entity>

All XML Schema built-in types are mapped to C++ classes that are derived from the xml_schema::simple_type class except where the mapping is to a fundamental C++ type.

The sequence class template is defined in an implementation-specific namespace. It conforms to the sequence interface as defined by the ISO/ANSI Standard for C++ (ISO/IEC 14882:1998, Section 23.1.1, "Sequences"). Practically, this means that you can treat such a sequence as if it was std::vector. One notable extension to the standard interface that is available only for sequences of non-fundamental C++ types is the addition of the overloaded push_back and insert member functions which instead of the constant reference to the element type accept

automatic pointer (`std::auto_ptr` or `std::unique_ptr`, depending on the C++ standard selected) to the element type. These functions assume ownership of the pointed to object and reset the passed automatic pointer.

2.5.1 Inheritance from Built-in Data Types

In cases where the mapping calls for an inheritance from a built-in type which is mapped to a fundamental C++ type, a proxy type is used instead of the fundamental C++ type (C++ does not allow inheritance from fundamental types). For instance:

```
<simpleType name="my_int">
  <restriction base="int"/>
</simpleType>
```

is mapped to:

```
class my_int: public fundamental_base<int>
{
  ...
};
```

The `fundamental_base` class template provides a close emulation (though not exact) of a fundamental C++ type. It is defined in an implementation-specific namespace and has the following interface:

```
template <typename X>
class fundamental_base: public simple_type
{
public:
  fundamental_base ();
  fundamental_base (X)
  fundamental_base (const fundamental_base&)

public:
  fundamental_base&
  operator= (const X&);

public:
  operator const X & () const;
  operator X& ();

  template <typename Y>
  operator Y () const;

  template <typename Y>
  operator Y ();
};
```

2.5.2 Mapping for `anyType`

The XML Schema `anyType` built-in data type is mapped to the `xml_schema::type` C++ class:

```
class type
{
public:
  virtual
  ~type ();

  type ();
  type (const type&);

  type&
  operator= (const type&);

  virtual type*
  _clone () const;

  // anyType DOM content.
  //
public:
  typedef element_optional dom_content_optional;

  const dom_content_optional&
  dom_content () const;

  dom_content_optional&
  dom_content ();

  void
  dom_content (const xercesc::DOMElement&);

  void
  dom_content (xercesc::DOMElement*);

  void
  dom_content (const dom_content_optional&);

  const xercesc::DOMDocument&
  dom_content_document () const;

  xercesc::DOMDocument&
  dom_content_document ();

  bool
  null_content () const;

  // DOM association.
```

```
  //
public:
  const xercesc::DOMNode*
  _node () const;

  xercesc::DOMNode*
  _node ();
};
```

When `xml_schema::type` is used to create an instance (as opposed to being a base of a derived type), it represents the XML Schema `anyType` type. `anyType` allows any attributes and any content in any order. In the C++/Tree mapping this content can be represented as a DOM fragment, similar to XML Schema wildcards (Section 2.12, "Mapping for `any` and `anyAttribute`").

To enable automatic extraction of `anyType` content during parsing, the `--generate-any-type` option must be specified. Because the DOM API is used to access such content, the Xerces-C++ runtime should be initialized by the application prior to parsing and should remain initialized for the lifetime of objects with the DOM content. For more information on the Xerces-C++ runtime initialization see Section 3.1, "Initializing the Xerces-C++ Runtime".

The DOM content is stored as the optional DOM element container and the DOM content accessors and modifiers presented above are identical to those generated for an optional element wildcard. Refer to Section 2.12.2, "Mapping for `any` with the Optional Cardinality Class" for details on their semantics.

The `dom_content_document()` function returns the DOM document used to store the raw XML content corresponding to the `anyType` instance. It is equivalent to the `dom_document()` function generated for types with wildcards.

The `null_content()` accessor is an optimization function that allows us to check for the lack of content without actually creating its empty representation, that is, empty DOM document for `anyType` or empty string for `anySimpleType` (see the following section for details on `anySimpleType`).

For more information on DOM association refer to Section 5.1, "DOM Association".

2.5.3 Mapping for `anySimpleType`

The XML Schema `anySimpleType` built-in data type is mapped to the `xml_schema::simple_type` C++ class:

```
class simple_type: public type
{
public:
  simple_type ();
```

```
  simple_type (const C*);
  simple_type (const std::basic_string<C>&);

  simple_type (const simple_type&);

  simple_type&
  operator= (const simple_type&);

  virtual simple_type*
  _clone () const;

  // anySimpleType text content.
  //
public:
  const std::basic_string<C>&
  text_content () const;

  std::basic_string<C>&
  text_content ();

  void
  text_content (const std::basic_string<C>&);
};
```

When `xml_schema::simple_type` is used to create an instance (as opposed to being a base of a derived type), it represents the XML Schema `anySimpleType` type. `anySimpleType` allows any simple content. In the C++/Tree mapping this content can be represented as a string and accessed or modified with the `text_content()` functions shown above.

2.5.4 Mapping for `QName`

The XML Schema `QName` built-in data type is mapped to the `xml_schema::qname` C++ class:

```
class qname: public simple_type
{
public:
  qname (const ncname&);
  qname (const uri&, const ncname&);
  qname (const qname&);

public:
  qname&
  operator= (const qname&);

public:
  virtual qname*
  _clone () const;

public:
```

```
  bool
  qualified () const;

  const uri&
  namespace_ () const;

  const ncname&
  name () const;
};
```

The `qualified` accessor function can be used to determine if the name is qualified.

2.5.5 Mapping for `IDREF`

The XML Schema `IDREF` built-in data type is mapped to the `xml_schema::idref` C++ class. This class implements the smart pointer C++ idiom:

```
class idref: public ncname
{
public:
  idref (const C* s);
  idref (const C* s, std::size_t n);
  idref (std::size_t n, C c);
  idref (const std::basic_string<C>&);
  idref (const std::basic_string<C>&,
         std::size_t pos,
         std::size_t n = npos);

public:
  idref (const idref&);

public:
  virtual idref*
  _clone () const;

public:
  idref&
  operator= (C c);

  idref&
  operator= (const C* s);

  idref&
  operator= (const std::basic_string<C>&)

  idref&
  operator= (const idref&);

public:
  const type*
```

```
  operator-> () const;

  type*
  operator-> ();

  const type&
  operator* () const;

  type&
  operator* ();

  const type*
  get () const;

  type*
  get ();

  // Conversion to bool.
  //
public:
  typedef void (idref::*bool_convertible)();
  operator bool_convertible () const;
};
```

The object, `idref` instance refers to, is the immediate container of the matching `id` instance. For example, with the following instance document and schema:

```
<!-- test.xml -->
<root>
  <object id="obj-1" text="hello"/>
  <reference>obj-1</reference>
</root>

<!-- test.xsd -->
<schema>
  <complexType name="object_type">
    <attribute name="id" type="ID"/>
    <attribute name="text" type="string"/>
  </complexType>

  <complexType name="root_type">
    <sequence>
      <element name="object" type="object_type"/>
      <element name="reference" type="IDREF"/>
    </sequence>
  </complexType>

  <element name="root" type="root_type"/>
</schema>
```

The `ref` instance in the code below will refer to an object of type `object_type`:

```
root_type& root = ...;
xml_schema::idref& ref (root.reference ());
object_type& obj (dynamic_cast<object_type&> (*ref));
cout << obj.text () << endl;
```

The smart pointer interface of the `idref` class always returns a pointer or reference to `xml_schema::type`. This means that you will need to manually cast such pointer or reference to its real (dynamic) type before you can use it (unless all you need is the base interface provided by `xml_schema::type`). As a special extension to the XML Schema language, the mapping supports static typing of `idref` references by employing the `refType` extension attribute. The following example illustrates this mechanism:

```
<!-- test.xsd -->
<schema
  xmlns:xse="http://www.codesynthesis.com/xmlns/xml-schema-extension">

  ...

      <element name="reference" type="IDREF" xse:refType="object_type"/>

  ...

</schema>
```

With this modification we do not need to do manual casting anymore:

```
root_type& root = ...;
root_type::reference_type& ref (root.reference ());
object_type& obj (*ref);
cout << ref->text () << endl;
```

2.5.6 Mapping for `base64Binary` and `hexBinary`

The XML Schema `base64Binary` and `hexBinary` built-in data types are mapped to the `xml_schema::base64_binary` and `xml_schema::hex_binary` C++ classes, respectively. The `base64_binary` and `hex_binary` classes support a simple buffer abstraction by inheriting from the `xml_schema::buffer` class:

```
class bounds: public virtual exception
{
public:
  virtual const char*
  what () const throw ();
};

class buffer
```

```
{
public:
  typedef std::size_t size_t;

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

public:
  buffer (const buffer&);

  buffer&
  operator= (const buffer&);

  void
  swap (buffer&);

public:
  size_t
  capacity () const;

  bool
  capacity (size_t);

public:
  size_t
  size () const;

  bool
  size (size_t);

public:
  const char*
  data () const;

  char*
  data ();

  const char*
  begin () const;

  char*
  begin ();

  const char*
```

```
  end () const;

  char*
  end ();
};
```

The last overloaded constructor reuses an existing data buffer instead of making a copy. If the `assume_ownership` argument is `true`, the instance assumes ownership of the memory block pointed to by the `data` argument and will eventually release it by calling `operator delete`. The `capacity` and `size` modifier functions return `true` if the underlying buffer has moved.

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

The `base64_binary` and `hex_binary` classes support the `buffer` interface and perform automatic decoding/encoding from/to the Base64 and Hex formats, respectively:

```
class base64_binary: public simple_type, public buffer
{
public:
  base64_binary (size_t size = 0);
  base64_binary (size_t size, size_t capacity);
  base64_binary (const void* data, size_t size);
  base64_binary (const void* data, size_t size, size_t capacity);
  base64_binary (void* data,
                 size_t size,
                 size_t capacity,
                 bool assume_ownership);

public:
  base64_binary (const base64_binary&);

  base64_binary&
  operator= (const base64_binary&);

  virtual base64_binary*
  _clone () const;

public:
  std::basic_string<C>
  encode () const;
};

class hex_binary: public simple_type, public buffer
{
public:
  hex_binary (size_t size = 0);
  hex_binary (size_t size, size_t capacity);
  hex_binary (const void* data, size_t size);
  hex_binary (const void* data, size_t size, size_t capacity);
```

```
  hex_binary (void* data,
              size_t size,
              size_t capacity,
              bool assume_ownership);

public:
  hex_binary (const hex_binary&);

  hex_binary&
  operator= (const hex_binary&);

  virtual hex_binary*
  _clone () const;

public:
  std::basic_string<C>
  encode () const;
};
```

2.5.7 Time Zone Representation

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

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

  bool
  zone_present () const;

  void
  zone_reset ();

  short
  zone_hours () const;

  void
  zone_hours (short);

  short
  zone_minutes () const;

  void
  zone_minutes (short);
};
```

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

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

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

2.5.8 Mapping for `date`

The XML Schema `date` built-in data type is mapped to the `xml_schema::date` C++ 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 2.5.7, "Time Zone Representation".

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

public:
  date (const date&);

  date&
  operator= (const date&);

  virtual date*
  _clone () const;

public:
  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&);
```

2.5.9 Mapping for `dateTime`

The XML Schema `dateTime` built-in data type is mapped to the `xml_schema::date_time` C++ 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 2.5.7, "Time Zone Representation".

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

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

  date_time&
  operator= (const date_time&);

  virtual date_time*
  _clone () const;

public:
  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&);
```

2.5.10 Mapping for `duration`

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

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

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

```
  virtual duration*
  _clone () const;

public:
  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&);
```

2.5.11 Mapping for `gDay`

The XML Schema `gDay` built-in data type is mapped to the `xml_schema::gday` C++ 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 2.5.7, "Time Zone Representation".

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

public:
  gday (const gday&);

  gday&
  operator= (const gday&);

  virtual gday*
  _clone () const;

public:
  unsigned short
  day () const;

  void
  day (unsigned short);
};

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

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

2.5.12 Mapping for `gMonth`

The XML Schema `gMonth` built-in data type is mapped to the `xml_schema::gmonth` C++ 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 2.5.7, "Time Zone Representation".

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

public:
  gmonth (const gmonth&);

  gmonth&
  operator= (const gmonth&);

  virtual gmonth*
  _clone () const;

public:
  unsigned short
  month () const;

  void
  month (unsigned short);
};

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

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

2.5.13 Mapping for `gMonthDay`

The XML Schema `gMonthDay` built-in data type is mapped to the `xml_schema::gmonth_day` C++ 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 2.5.7, "Time Zone Representation".

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

public:
  gmonth_day (const gmonth_day&);

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

```
  virtual gmonth_day*
  _clone () const;

public:
  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&);
```

2.5.14 Mapping for `gYear`

The XML Schema `gYear` built-in data type is mapped to the `xml_schema::gyear` C++ 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 2.5.7, "Time Zone Representation".

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

public:
  gyear (const gyear&);

  gyear&
  operator= (const gyear&);

  virtual gyear*
  _clone () const;

public:
  int
  year () const;
```

```
  void
  year (int);
};

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

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

2.5.15 Mapping for `gYearMonth`

The XML Schema `gYearMonth` built-in data type is mapped to the `xml_schema::gyear_month` C++ 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 2.5.7, "Time Zone Representation".

```
class gyear_month: public simple_type, public time_zone
{
public:
  gyear_month (int year, unsigned short month);
  gyear_month (int year, unsigned short month,
               short zone_hours, short zone_minutes);
public:
  gyear_month (const gyear_month&);

  gyear_month&
  operator= (const gyear_month&);

  virtual gyear_month*
  _clone () const;

public:
  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&);
```

2.5.16 Mapping for `time`

The XML Schema `time` built-in data type is mapped to the `xml_schema::time` C++ 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 2.5.7, "Time Zone Representation".

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

public:
  time (const time&);

  time&
  operator= (const time&);

  virtual time*
  _clone () const;

public:
  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&);
```

2.6 Mapping for Simple Types

An XML Schema simple type is mapped to a C++ class with the same name as the simple type. The class defines a public copy constructor, a public copy assignment operator, and a public virtual `_clone` function. The `_clone` function is declared `const`, does not take any arguments, and returns a pointer to a complete copy of the instance allocated in the free store. The `_clone` function shall be used to make copies when static type and dynamic type of the instance may differ (see Section 2.11, "Mapping for `xsi:type` and Substitution Groups"). For instance:

```
<simpleType name="object">
  ...
</simpleType>
```

is mapped to:

```
class object: ...
{
public:
  object (const object&);

public:
  object&
  operator= (const object&);

public:
  virtual object*
  _clone () const;

  ...

};
```

The base class specification and the rest of the class definition depend on the type of derivation used to define the simple type.

2.6.1 Mapping for Derivation by Restriction

XML Schema derivation by restriction is mapped to C++ public inheritance. The base type of the restriction becomes the base type for the resulting C++ class. In addition to the members described in Section 2.6, "Mapping for Simple Types", the resulting C++ class defines a public constructor with the base type as its single argument. For instance:

```
<simpleType name="object">
  <restriction base="base">
    ...
  </restriction>
</simpleType>
```

is mapped to:

```
class object: public base
{
public:
  object (const base&);
  object (const object&);

public:
  object&
  operator= (const object&);

public:
  virtual object*
  _clone () const;
};
```

2.6.2 Mapping for Enumerations

XML Schema restriction by enumeration is mapped to a C++ class with semantics similar to C++ `enum`. Each XML Schema enumeration element is mapped to a C++ enumerator with the name derived from the `value` attribute and defined in the class scope. In addition to the members described in Section 2.6, "Mapping for Simple Types", the resulting C++ class defines a public constructor that can be called with one of the enumerators as its single argument, a public constructor that can be called with enumeration's base value as its single argument, a public assignment operator that can be used to assign the value of one of the enumerators, and a public implicit conversion operator to the underlying C++ enum type.

Furthermore, for string-based enumeration types, the resulting C++ class defines a public constructor with a single argument of type `const C*` and a public constructor with a single argument of type `const std::basic_string<C>&`. For instance:

```
<simpleType name="color">
  <restriction base="string">
    <enumeration value="red"/>
    <enumeration value="green"/>
    <enumeration value="blue"/>
  </restriction>
</simpleType>
```

is mapped to:

```
class color: public xml_schema::string
{
public:
  enum value
  {
    red,
    green,
    blue
  };

public:
  color (value);
  color (const C*);
  color (const std::basic_string<C>&);
  color (const xml_schema::string&);
  color (const color&);

public:
  color&
  operator= (value);

  color&
  operator= (const color&);

public:
  virtual color*
  _clone () const;

public:
  operator value () const;
};
```

2.6.3 Mapping for Derivation by List

XML Schema derivation by list is mapped to C++ public inheritance from `xml_schema::simple_type` (Section 2.5.3, "Mapping for `anySimpleType`") and a suitable sequence type. The list item type becomes the element type of the sequence. In addition to the members described in Section 2.6, "Mapping for Simple Types", the resulting C++ class defines a public default constructor, a public constructor with the first argument of type `size_type` and the second argument of list item type that creates a list object with the specified number of copies of the specified element value, and a public constructor with the two arguments of an input iterator type that creates a list object from an iterator range. For instance:

```
<simpleType name="int_list">
  <list itemType="int"/>
</simpleType>
```

is mapped to:

```
class int_list: public simple_type,
                public sequence<int>
{
public:
  int_list ();
  int_list (size_type n, int x);

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

public:
  int_list&
  operator= (const int_list&);

public:
  virtual int_list*
  _clone () const;
};
```

The `sequence` class template is defined in an implementation-specific namespace. It conforms to the sequence interface as defined by the ISO/ANSI Standard for C++ (ISO/IEC 14882:1998, Section 23.1.1, "Sequences"). Practically, this means that you can treat such a sequence as if it was `std::vector`. One notable extension to the standard interface that is available only for sequences of non-fundamental C++ types is the addition of the overloaded `push_back` and `insert` member functions which instead of the constant reference to the element type accept automatic pointer (`std::auto_ptr` or `std::unique_ptr`, depending on the C++ standard selected) to the element type. These functions assume ownership of the pointed to object and reset the passed automatic pointer.

2.6.4 Mapping for Derivation by Union

XML Schema derivation by union is mapped to C++ public inheritance from `xml_schema::simple_type` (Section 2.5.3, "Mapping for `anySimpleType`") and `std::basic_string<C>`. In addition to the members described in Section 2.6, "Mapping for Simple Types", the resulting C++ class defines a public constructor with a single argument of type `const C*` and a public constructor with a single argument of type `const std::basic_string<C>&`. For instance:

```
<simpleType name="int_string_union">
  <xsd:union memberTypes="xsd:int xsd:string"/>
</simpleType>
```

is mapped to:

```
class int_string_union: public simple_type,
                        public std::basic_string<C>
{
public:
  int_string_union (const C*);
  int_string_union (const std::basic_string<C>&);
  int_string_union (const int_string_union&);

public:
  int_string_union&
  operator= (const int_string_union&);

public:
  virtual int_string_union*
  _clone () const;
};
```

2.7 Mapping for Complex Types

An XML Schema complex type is mapped to a C++ class with the same name as the complex type. The class defines a public copy constructor, a public copy assignment operator, and a public virtual `_clone` function. The `_clone` function is declared `const`, does not take any arguments, and returns a pointer to a complete copy of the instance allocated in the free store. The `_clone` function shall be used to make copies when static type and dynamic type of the instance may differ (see Section 2.11, "Mapping for `xsi:type` and Substitution Groups").

Additionally, the resulting C++ class defines two public constructors that take an initializer for each member of the complex type and all its base types that belongs to the One cardinality class (see Section 2.8, "Mapping for Local Elements and Attributes"). In the first constructor, the arguments are passed as constant references and the newly created instance is initialized with copies of the passed objects. In the second constructor, arguments that are complex types (that is, they themselves contain elements or attributes) are passed as either `std::auto_ptr` (C++98) or `std::unique_ptr` (C++11), depending on the C++ standard selected. In this case the newly created instance is directly initialized with and assumes ownership of the pointed to objects and the `std::[auto|unique]_ptr` arguments are reset to `0`. For instance:

```
<complexType name="complex">
  <sequence>
    <element name="a" type="int"/>
    <element name="b" type="string"/>
  </sequence>
</complexType>

<complexType name="object">
  <sequence>
```

```
    <element name="s-one" type="boolean"/>
    <element name="c-one" type="complex"/>
    <element name="optional" type="int" minOccurs="0"/>
    <element name="sequence" type="string" maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

is mapped to:

```
class complex: public xml_schema::type
{
public:
  object (const int& a, const xml_schema::string& b);
  object (const complex&);

public:
  object&
  operator= (const complex&);

public:
  virtual complex*
  _clone () const;

  ...

};

class object: public xml_schema::type
{
public:
  object (const bool& s_one, const complex& c_one);
  object (const bool& s_one, std::[auto|unique]_ptr<complex> c_one);
  object (const object&);

public:
  object&
  operator= (const object&);

public:
  virtual object*
  _clone () const;

  ...

};
```

Notice that the generated `complex` class does not have the second (`std::[auto|unique]_ptr`) version of the constructor since all its required members are of simple types.

If an XML Schema complex type has an ultimate base which is an XML Schema simple type then the resulting C++ class also defines a public constructor that takes an initializer for the base type as well as for each member of the complex type and all its base types that belongs to the One cardinality class. For instance:

```
<complexType name="object">
  <simpleContent>
    <extension base="date">
      <attribute name="lang" type="language" use="required"/>
    </extension>
  </simpleContent>
</complexType>
```

is mapped to:

```
class object: public xml_schema::string
{
public:
  object (const xml_schema::language& lang);

  object (const xml_schema::date& base,
          const xml_schema::language& lang);

  ...

};
```

Furthermore, for string-based XML Schema complex types, the resulting C++ class also defines two public constructors with the first arguments of type `const C*` and `std::basic_string<C>&`, respectively, followed by arguments for each member of the complex type and all its base types that belongs to the One cardinality class. For enumeration-based complex types the resulting C++ class also defines a public constructor with the first arguments of the underlying enum type followed by arguments for each member of the complex type and all its base types that belongs to the One cardinality class. For instance:

```
<simpleType name="color">
  <restriction base="string">
    <enumeration value="red"/>
    <enumeration value="green"/>
    <enumeration value="blue"/>
  </restriction>
</simpleType>

<complexType name="object">
  <simpleContent>
    <extension base="color">
```

```
      <attribute name="lang" type="language" use="required"/>
    </extension>
  </simpleContent>
</complexType>
```

is mapped to:

```
class color: public xml_schema::string
{
public:
  enum value
  {
    red,
    green,
    blue
  };

public:
  color (value);
  color (const C*);
  color (const std::basic_string<C>&);

  ...

};

class object: color
{
public:
  object (const color& base,
          const xml_schema::language& lang);

  object (const color::value& base,
          const xml_schema::language& lang);

  object (const C* base,
          const xml_schema::language& lang);

  object (const std::basic_string<C>& base,
          const xml_schema::language& lang);

  ...

};
```

Additional constructors can be requested with the `--generate-default-ctor` and `--generate-from-base-ctor` options. See the XSD Compiler Command Line Manual for details.

If an XML Schema complex type is not explicitly derived from any type, the resulting C++ class is derived from `xml_schema::type`. In cases where an XML Schema complex type is defined using derivation by extension or restriction, the resulting C++ base class specification depends on the type of derivation and is described in the subsequent sections.

The mapping for elements and attributes that are defined in a complex type is described in Section 2.8, "Mapping for Local Elements and Attributes".

2.7.1 Mapping for Derivation by Extension

XML Schema derivation by extension is mapped to C++ public inheritance. The base type of the extension becomes the base type for the resulting C++ class.

2.7.2 Mapping for Derivation by Restriction

XML Schema derivation by restriction is mapped to C++ public inheritance. The base type of the restriction becomes the base type for the resulting C++ class. XML Schema elements and attributes defined within restriction do not result in any definitions in the resulting C++ class. Instead, corresponding (unrestricted) definitions are inherited from the base class. In the future versions of this mapping, such elements and attributes may result in redefinitions of accessors and modifiers to reflect their restricted semantics.

2.8 Mapping for Local Elements and Attributes

XML Schema element and attribute definitions are called local if they appear within a complex type definition, an element group definition, or an attribute group definitions.

Local XML Schema element and attribute definitions have the same C++ mapping. Therefore, in this section, local elements and attributes are collectively called members.

While there are many different member cardinality combinations (determined by the `use` attribute for attributes and the `minOccurs` and `maxOccurs` attributes for elements), the mapping divides all possible cardinality combinations into three cardinality classes:

one
: attributes: `use == "required"`
: attributes: `use == "optional"` and has default or fixed value
: elements: `minOccurs == "1"` and `maxOccurs == "1"`

optional
: attributes: `use == "optional"` and doesn't have default or fixed value
: elements: `minOccurs == "0"` and `maxOccurs == "1"`

sequence
    elements: `maxOccurs > "1"`

An optional attribute with a default or fixed value acquires this value if the attribute hasn't been specified in an instance document (see Appendix A, "Default and Fixed Values"). This mapping places such optional attributes to the One cardinality class.

A member is mapped to a set of public type definitions (`typedefs`) and a set of public accessor and modifier functions. Type definitions have names derived from the member's name. The accessor and modifier functions have the same name as the member. For example:

```
<complexType name="object">
  <sequence>
    <element name="member" type="string"/>
  </sequence>
</complexType>
```

is mapped to:

```
class object: public xml_schema::type
{
public:
  typedef xml_schema::string member_type;

  const member_type&
  member () const;

  ...

};
```

In addition, if a member has a default or fixed value, a static accessor function is generated that returns this value. For example:

```
<complexType name="object">
  <attribute name="data" type="string" default="test"/>
</complexType>
```

is mapped to:

```
class object: public xml_schema::type
{
public:
  typedef xml_schema::string data_type;

  const data_type&
  data () const;

  static const data_type&
```

```
  data_default_value ();

  ...
};
```

Names and semantics of type definitions for the member as well as signatures of the accessor and modifier functions depend on the member's cardinality class and are described in the following sub-sections.

2.8.1 Mapping for Members with the One Cardinality Class

For the One cardinality class, the type definitions consist of an alias for the member's type with the name created by appending the `_type` suffix to the member's name.

The accessor functions come in constant and non-constant versions. The constant accessor function returns a constant reference to the member and can be used for read-only access. The non-constant version returns an unrestricted reference to the member and can be used for read-write access.

The first modifier function expects an argument of type reference to constant of the member's type. It makes a deep copy of its argument. Except for member's types that are mapped to fundamental C++ types, the second modifier function is provided that expects an argument of type automatic pointer (`std::auto_ptr` or `std::unique_ptr`, depending on the C++ standard selected) to the member's type. It assumes ownership of the pointed to object and resets the passed automatic pointer. For instance:

```
<complexType name="object">
  <sequence>
    <element name="member" type="string"/>
  </sequence>
</complexType>
```

is mapped to:

```
class object: public xml_schema::type
{
public:
  // Type definitions.
  //
  typedef xml_schema::string member_type;

  // Accessors.
  //
  const member_type&
  member () const;

  member_type&
```

```
  member ();

  // Modifiers.
  //
  void
  member (const member_type&);

  void
  member (std::[auto|unique]_ptr<member_type>);
  ...

};
```

In addition, if requested by specifying the `--generate-detach` option and only for members of non-fundamental C++ types, the mapping provides a detach function that returns an automatic pointer to the member's type, for example:

```
class object: public xml_schema::type
{
public:
  ...

  std::[auto|unique]_ptr<member_type>
  detach_member ();
  ...

};
```

This function detaches the value from the tree leaving the member value uninitialized. Accessing such an uninitialized value prior to re-initializing it results in undefined behavior.

The following code shows how one could use this mapping:

```
void
f (object& o)
{
  using xml_schema::string;

  string s (o.member ());                // get
  object::member_type& sr (o.member ()); // get

  o.member ("hello");           // set, deep copy
  o.member () = "hello";        // set, deep copy

  // C++98 version.
  //
  std::auto_ptr<string> p (new string ("hello"));
  o.member (p);                 // set, assumes ownership
  p = o.detach_member ();       // detach, member is uninitialized
  o.member (p);                 // re-attach
```

```
  // C++11 version.
  //
  std::unique_ptr<string> p (new string ("hello"));
  o.member (std::move (p));     // set, assumes ownership
  p = o.detach_member ();       // detach, member is uninitialized
  o.member (std::move (p));     // re-attach
}
```

2.8.2 Mapping for Members with the Optional Cardinality Class

For the Optional cardinality class, the type definitions consist of an alias for the member's type with the name created by appending the `_type` suffix to the member's name and an alias for the container type with the name created by appending the `_optional` suffix to the member's name.

Unlike accessor functions for the One cardinality class, accessor functions for the Optional cardinality class return references to corresponding containers rather than directly to members. The accessor functions come in constant and non-constant versions. The constant accessor function returns a constant reference to the container and can be used for read-only access. The non-constant version returns an unrestricted reference to the container and can be used for read-write access.

The modifier functions are overloaded for the member's type and the container type. The first modifier function expects an argument of type reference to constant of the member's type. It makes a deep copy of its argument. Except for member's types that are mapped to fundamental C++ types, the second modifier function is provided that expects an argument of type automatic pointer (`std::auto_ptr` or `std::unique_ptr`, depending on the C++ standard selected) to the member's type. It assumes ownership of the pointed to object and resets the passed automatic pointer. The last modifier function expects an argument of type reference to constant of the container type. It makes a deep copy of its argument. For instance:

```
<complexType name="object">
  <sequence>
    <element name="member" type="string" minOccurs="0"/>
  </sequence>
</complexType>
```

is mapped to:

```
class object: public xml_schema::type
{
public:
  // Type definitions.
  //
  typedef xml_schema::string member_type;
  typedef optional<member_type> member_optional;
```

```
  // Accessors.
  //
  const member_optional&
  member () const;

  member_optional&
  member ();

  // Modifiers.
  //
  void
  member (const member_type&);

  void
  member (std::[auto|unique]_ptr<member_type>);

  void
  member (const member_optional&);

  ...

};
```

The `optional` class template is defined in an implementation-specific namespace and has the following interface. The `[auto|unique]_ptr`-based constructor and modifier function are only available if the template argument is not a fundamental C++ type.

```
template <typename X>
class optional
{
public:
  optional ();

  // Makes a deep copy.
  //
  explicit
  optional (const X&);

  // Assumes ownership.
  //
  explicit
  optional (std::[auto|unique]_ptr<X>);

  optional (const optional&);

public:
  optional&
  operator= (const X&);

  optional&
```

```
  operator= (const optional&);

  // Pointer-like interface.
  //
public:
  const X*
  operator-> () const;

  X*
  operator-> ();

  const X&
  operator* () const;

  X&
  operator* ();

  typedef void (optional::*bool_convertible) ();
  operator bool_convertible () const;

  // Get/set interface.
  //
public:
  bool
  present () const;

  const X&
  get () const;

  X&
  get ();

  // Makes a deep copy.
  //
  void
  set (const X&);

  // Assumes ownership.
  //
  void
  set (std::[auto|unique]_ptr<X>);

  // Detach and return the contained value.
  //
  std::[auto|unique]_ptr<X>
  detach ();

  void
  reset ();
};
```

```
template <typename X>
bool
operator== (const optional<X>&, const optional<X>&);

template <typename X>
bool
operator!= (const optional<X>&, const optional<X>&);

template <typename X>
bool
operator< (const optional<X>&, const optional<X>&);

template <typename X>
bool
operator> (const optional<X>&, const optional<X>&);

template <typename X>
bool
operator<= (const optional<X>&, const optional<X>&);

template <typename X>
bool
operator>= (const optional<X>&, const optional<X>&);
```

The following code shows how one could use this mapping:

```
void
f (object& o)
{
  using xml_schema::string;

  if (o.member ().present ())       // test
  {
    string& s (o.member ().get ()); // get
    o.member ("hello");             // set, deep copy
    o.member ().set ("hello");      // set, deep copy
    o.member ().reset ();           // reset
  }

  // Same as above but using pointer notation:
  //
  if (o.member ())                  // test
  {
    string& s (*o.member ());       // get
    o.member ("hello");             // set, deep copy
    *o.member () = "hello";         // set, deep copy
    o.member ().reset ();           // reset
  }

  // C++98 version.
  //
```

```
  std::auto_ptr<string> p (new string ("hello"));
  o.member (p);                     // set, assumes ownership

  p = new string ("hello");
  o.member ().set (p);              // set, assumes ownership

  p = o.member ().detach ();        // detach, member is reset
  o.member ().set (p);              // re-attach

  // C++11 version.
  //
  std::unique_ptr<string> p (new string ("hello"));
  o.member (std::move (p));         // set, assumes ownership

  p.reset (new string ("hello"));
  o.member ().set (std::move (p));  // set, assumes ownership

  p = o.member ().detach ();        // detach, member is reset
  o.member ().set (std::move (p));  // re-attach
}
```

2.8.3 Mapping for Members with the Sequence Cardinality Class

For the Sequence cardinality class, the type definitions consist of an alias for the member's type with the name created by appending the `_type` suffix to the member's name, an alias of the container type with the name created by appending the `_sequence` suffix to the member's name, an alias of the iterator type with the name created by appending the `_iterator` suffix to the member's name, and an alias of the constant iterator type with the name created by appending the `_const_iterator` suffix to the member's name.

The accessor functions come in constant and non-constant versions. The constant accessor function returns a constant reference to the container and can be used for read-only access. The non-constant version returns an unrestricted reference to the container and can be used for read-write access.

The modifier function expects an argument of type reference to constant of the container type. The modifier function makes a deep copy of its argument. For instance:

```
<complexType name="object">
  <sequence>
    <element name="member" type="string" minOccurs="unbounded"/>
  </sequence>
</complexType>
```

is mapped to:

```
class object: public xml_schema::type
{
public:
  // Type definitions.
  //
  typedef xml_schema::string member_type;
  typedef sequence<member_type> member_sequence;
  typedef member_sequence::iterator member_iterator;
  typedef member_sequence::const_iterator member_const_iterator;

  // Accessors.
  //
  const member_sequence&
  member () const;

  member_sequence&
  member ();

  // Modifier.
  //
  void
  member (const member_sequence&);

  ...

};
```

The `sequence` class template is defined in an implementation-specific namespace. It conforms to the sequence interface as defined by the ISO/ANSI Standard for C++ (ISO/IEC 14882:1998, Section 23.1.1, "Sequences"). Practically, this means that you can treat such a sequence as if it was `std::vector`. Two notable extensions to the standard interface that are available only for sequences of non-fundamental C++ types are the addition of the overloaded `push_back` and `insert` as well as the `detach_back` and `detach` member functions. The additional `push_back` and `insert` functions accept an automatic pointer (`std::auto_ptr` or `std::unique_ptr`, depending on the C++ standard selected) to the element type instead of the constant reference. They assume ownership of the pointed to object and reset the passed automatic pointer. The `detach_back` and `detach` functions detach the element value from the sequence container and, by default, remove the element from the sequence. These additional functions have the following signatures:

```
template <typename X>
class sequence
{
public:
  ...

  void
  push_back (std::[auto|unique]_ptr<X>)
```

```
  iterator
  insert (iterator position, std::[auto|unique]_ptr<X>)

  std::[auto|unique]_ptr<X>
  detach_back (bool pop = true);

  iterator
  detach (iterator position,
          std::[auto|unique]_ptr<X>& result,
          bool erase = true)

  ...
}
```

The following code shows how one could use this mapping:

```
void
f (object& o)
{
  using xml_schema::string;

  object::member_sequence& s (o.member ());

  // Iteration.
  //
  for (object::member_iterator i (s.begin ()); i != s.end (); ++i)
  {
    string& value (*i);
  }

  // Modification.
  //
  s.push_back ("hello");  // deep copy

  // C++98 version.
  //
  std::auto_ptr<string> p (new string ("hello"));
  s.push_back (p);        // assumes ownership
  p = s.detach_back ();   // detach and pop
  s.push_back (p);        // re-append

  // C++11 version.
  //
  std::unique_ptr<string> p (new string ("hello"));
  s.push_back (std::move (p)); // assumes ownership
  p = s.detach_back ();        // detach and pop
  s.push_back (std::move (p)); // re-append

  // Setting a new container.
  //
  object::member_sequence n;
```

```
  n.push_back ("one");
  n.push_back ("two");
  o.member (n);           // deep copy
}
```

2.8.4 Element Order

C++/Tree is a "flattening" mapping in a sense that many levels of nested compositors (`choice` and `sequence`), all potentially with their own cardinalities, are in the end mapped to a flat set of elements with one of the three cardinality classes discussed in the previous sections. While this results in a simple and easy to use API for most types, in certain cases, the order of elements in the actual XML documents is not preserved once parsed into the object model. And sometimes such order has application-specific significance. As an example, consider a schema that defines a batch of bank transactions:

```
<complexType name="withdraw">
  <sequence>
    <element name="account" type="unsignedInt"/>
    <element name="amount" type="unsignedInt"/>
  </sequence>
</complexType>

<complexType name="deposit">
  <sequence>
    <element name="account" type="unsignedInt"/>
    <element name="amount" type="unsignedInt"/>
  </sequence>
</complexType>

<complexType name="batch">
  <choice minOccurs="0" maxOccurs="unbounded">
    <element name="withdraw" type="withdraw"/>
    <element name="deposit" type="deposit"/>
  </choice>
</complexType>
```

The batch can contain any number of transactions in any order but the order of transactions in each actual batch is significant. For instance, consider what could happen if we reorder the transactions and apply all the withdrawals before deposits.

For the `batch` schema type defined above the default C++/Tree mapping will produce a C++ class that contains a pair of sequence containers, one for each of the two elements. While this will capture the content (transactions), the order of this content as it appears in XML will be lost. Also, if we try to serialize the batch we just loaded back to XML, all the withdrawal transactions will appear before deposits.

To overcome this limitation of a flattening mapping, C++/Tree allows us to mark certain XML Schema types, for which content order is important, as ordered.

There are several command line options that control which schema types are treated as ordered. To make an individual type ordered, we use the `--ordered-type` option, for example:

```
--ordered-type batch
```

To automatically treat all the types that are derived from an ordered type also ordered, we use the `--ordered-type-derived` option. This is primarily useful if you would like to iterate over the complete hierarchy's content using the content order sequence (discussed below).

Ordered types are also useful for handling mixed content. To automatically mark all the types with mixed content as ordered we use the `--ordered-type-mixed` option. For more information on handling mixed content see Section 2.13, "Mapping for Mixed Content Models".

Finally, we can mark all the types in the schema we are compiling with the `--ordered-type-all` option. You should only resort to this option if all the types in your schema truly suffer from the loss of content order since, as we will discuss shortly, ordered types require extra effort to access and, especially, modify. See the XSD Compiler Command Line Manual for more information on these options.

Once a type is marked ordered, C++/Tree alters its mapping in several ways. Firstly, for each local element, element wildcard (Section 2.12.4, "Element Wildcard Order"), and mixed content text (Section 2.13, "Mapping for Mixed Content Models") in this type, a content id constant is generated. Secondly, an addition sequence is added to the class that captures the content order. Here is how the mapping of our `batch` class changes once we make it ordered:

```
class batch: public xml_schema::type
{
public:
  // withdraw
  //
  typedef withdraw withdraw_type;
  typedef sequence<withdraw_type> withdraw_sequence;
  typedef withdraw_sequence::iterator withdraw_iterator;
  typedef withdraw_sequence::const_iterator withdraw_const_iterator;

  static const std::size_t withdraw_id = 1;

  const withdraw_sequence&
  withdraw () const;

  withdraw_sequence&
  withdraw ();

  void
```

```
  withdraw (const withdraw_sequence&);

  // deposit
  //
  typedef deposit deposit_type;
  typedef sequence<deposit_type> deposit_sequence;
  typedef deposit_sequence::iterator deposit_iterator;
  typedef deposit_sequence::const_iterator deposit_const_iterator;

  static const std::size_t deposit_id = 2;

  const deposit_sequence&
  deposit () const;

  deposit_sequence&
  deposit ();

  void
  deposit (const deposit_sequence&);

  // content_order
  //
  typedef xml_schema::content_order content_order_type;
  typedef std::vector<content_order_type> content_order_sequence;
  typedef content_order_sequence::iterator content_order_iterator;
  typedef content_order_sequence::const_iterator content_order_const_iterator;

  const content_order_sequence&
  content_order () const;

  content_order_sequence&
  content_order ();

  void
  content_order (const content_order_sequence&);

  ...
};
```

Notice the `withdraw_id` and `deposit_id` content ids as well as the extra `content_order` sequence that does not correspond to any element in the schema definition. The other changes to the mapping for ordered types has to do with XML parsing and serialization code. During parsing the content order is captured in the `content_order` sequence while during serialization this sequence is used to determine the order in which content is serialized. The `content_order` sequence is also copied during copy construction and assigned during copy assignment. It is also taken into account during comparison.

The entry type of the `content_order` sequence is the `xml_schema::content_order` type that has the following interface:

```
namespace xml_schema
{
  struct content_order
  {
    content_order (std::size_t id, std::size_t index = 0);

    std::size_t id;
    std::size_t index;
  };

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

  bool
  operator!= (const content_order&, const content_order&);

  bool
  operator< (const content_order&, const content_order&);
}
```

The `content_order` sequence describes the order of content (elements, including wildcards, as well as mixed content text). Each entry in this sequence consists of the content id (for example, `withdraw_id` or `deposit_id` in our case) as well as, for elements of the sequence cardinality class, an index into the corresponding sequence container (the index is unused for the one and optional cardinality classes). For example, in our case, if the content id is `withdraw_id`, then the index will point into the `withdraw` element sequence.

With all this information we can now examine how to iterate over transaction in the batch in content order:

```
batch& b = ...

for (batch::content_order_const_iterator i (b.content_order ().begin ());
     i != b.content_order ().end ();
     ++i)
{
  switch (i->id)
  {
  case batch::withdraw_id:
    {
      const withdraw& t (b.withdraw ()[i->index]);
      cerr << t.account () << " withdraw " << t.amount () << endl;
      break;
    }
  case batch::deposit_id:
    {
```

```
      const deposit& t (b.deposit ()[i->index]);
      cerr << t.account () << " deposit " << t.amount () << endl;
      break;
    }
  default:
    {
      assert (false); // Unknown content id.
    }
  }
}
```

If we serialized our batch back to XML, we would also see that the order of transactions in the output is exactly the same as in the input rather than all the withdrawals first followed by all the deposits.

The most complex aspect of working with ordered types is modifications. Now we not only need to change the content, but also remember to update the order information corresponding to this change. As a first example, we add a deposit transaction to the batch:

```
using xml_schema::content_order;

batch::deposit_sequence& d (b.deposit ());
batch::withdraw_sequence& w (b.withdraw ());
batch::content_order_sequence& co (b.content_order ());

d.push_back (deposit (123456789, 100000));
co.push_back (content_order (batch::deposit_id, d.size () - 1));
```

In the above example we first added the content (deposit transaction) and then updated the content order information by adding an entry with `deposit_id` content id and the index of the just added deposit transaction.

Removing the last transaction can be easy if we know which transaction (deposit or withdrawal) is last:

```
d.pop_back ();
co.pop_back ();
```

If, however, we do not know which transaction is last, then things get a bit more complicated:

```
switch (co.back ().id)
{
case batch::withdraw_id:
  {
    d.pop_back ();
    break;
  }
case batch::deposit_id:
  {
```

```
    w.pop_back ();
    break;
  }
}

co.pop_back ();
```

The following example shows how to add a transaction at the beginning of the batch:

```
w.push_back (withdraw (123456789, 100000));
co.insert (co.begin (),
           content_order (batch::withdraw_id, w.size () - 1));
```

Note also that when we merely modify the content of one of the elements in place, we do not need to update its order since it doesn't change. For example, here is how we can change the amount in the first withdrawal:

```
w[0].amount (10000);
```

For the complete working code shown in this section refer to the `order/element` example in the `examples/cxx/tree/` directory in the XSD distribution.

If both the base and derived types are ordered, then the content order sequence is only added to the base and the content ids are unique within the whole hierarchy. In this case the content order sequence for the derived type contains ordering information for both base and derived content.

In some applications we may need to perform more complex content processing. For example, in our case, we may need to remove all the withdrawal transactions. The default container, `std::vector`, is not particularly suitable for such operations. What may be required by some applications is a multi-index container that not only allows us to iterate in content order similar to `std::vector` but also search by the content id as well as the content id and index pair.

While C++/Tree does not provide this functionality by default, it allows us to specify a custom container type for content order with the `--order-container` command line option. The only requirement from the generated code side for such a container is to provide the `vector`-like `push_back()`, `size()`, and const iteration interfaces.

As an example, here is how we can use the Boost Multi-Index container for content order. First we create the `content-order-container.hxx` header with the following definition (in C++11, use the alias template instead):

```
#ifndef CONTENT_ORDER_CONTAINER
#define CONTENT_ORDER_CONTAINER

#include <cstddef> // std::size_t

#include <boost/multi_index_container.hpp>
```

```
#include <boost/multi_index/member.hpp>
#include <boost/multi_index/identity.hpp>
#include <boost/multi_index/ordered_index.hpp>
#include <boost/multi_index/random_access_index.hpp>

struct by_id {};
struct by_id_index {};

template <typename T>
struct content_order_container:
  boost::multi_index::multi_index_container<
    T,
    boost::multi_index::indexed_by<
      boost::multi_index::random_access<>,
      boost::multi_index::ordered_unique<
        boost::multi_index::tag<by_id_index>,
        boost::multi_index::identity<T>
      >,
      boost::multi_index::ordered_non_unique<
        boost::multi_index::tag<by_id>,
        boost::multi_index::member<T, std::size_t, &T::id>
      >
    >
  >
{};

#endif
```

Next we add the following two XSD compiler options to include this header into every generated header file and to use the custom container type (see the XSD compiler command line manual for more information on shell quoting for the first option):

```
--hxx-prologue '#include "content-order-container.hxx"'
--order-container content_order_container
```

With these changes we can now use the multi-index functionality, for example, to search for a specific content id:

```
typedef batch::content_order_sequence::index<by_id>::type id_set;
typedef id_set::iterator id_iterator;

const id_set& ids (b.content_order ().get<by_id> ());

std::pair<id_iterator, id_iterator> r (
  ids.equal_range (std::size_t (batch::deposit_id));

for (id_iterator i (r.first); i != r.second; ++i)
```

```
{
  const deposit& t (b.deposit ()[i->index]);
  cerr << t.account () << " deposit " << t.amount () << endl;
}
```

2.9 Mapping for Global Elements

An XML Schema element definition is called global if it appears directly under the `schema` element. A global element is a valid root of an instance document. By default, a global element is mapped to a set of overloaded parsing and, optionally, serialization functions with the same name as the element. It is also possible to generate types for root elements instead of parsing and serialization functions. This is primarily useful to distinguish object models with the same root type but with different root elements. See Section 2.9.1, "Element Types" for details. It is also possible to request the generation of an element map which allows uniform parsing and serialization of multiple root elements. See Section 2.9.2, "Element Map" for details.

The parsing functions read XML instance documents and return corresponding object models as an automatic pointer (`std::auto_ptr` or `std::unique_ptr`, depending on the C++ standard selected). Their signatures have the following pattern (`type` denotes element's type and `name` denotes element's name):

```
std::[auto|unique]_ptr<type>
name (....);
```

The process of parsing, including the exact signatures of the parsing functions, is the subject of Chapter 3, "Parsing".

The serialization functions write object models back to XML instance documents. Their signatures have the following pattern:

```
void
name (<stream type>&, const type&, ....);
```

The process of serialization, including the exact signatures of the serialization functions, is the subject of Chapter 4, "Serialization".

2.9.1 Element Types

The generation of element types is requested with the `--generate-element-type` option. With this option each global element is mapped to a C++ class with the same name as the element. Such a class is derived from `xml_schema::element_type` and contains the same set of type definitions, constructors, and member function as would a type containing a single element with the One cardinality class named `"value"`. In addition, the element type also contains a set of member functions for accessing the element name and namespace as well as its value in a uniform manner. For example:

```
<complexType name="type">
  <sequence>
    ...
  </sequence>
</complexType>

<element name="root" type="type"/>
```

is mapped to:

```
class type
{
  ...
};

class root: public xml_schema::element_type
{
public:
  // Element value.
  //
  typedef type value_type;

  const value_type&
  value () const;

  value_type&
  value ();

  void
  value (const value_type&);

  void
  value (std::[auto|unique]_ptr<value_type>);

  // Constructors.
  //
  root (const value_type&);

  root (std::[auto|unique]_ptr<value_type>);

  root (const xercesc::DOMElement&, xml_schema::flags = 0);

  root (const root&, xml_schema::flags = 0);

  virtual root*
  _clone (xml_schema::flags = 0) const;

  // Element name and namespace.
  //
  static const std::string&
  name ();
```

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

  virtual const std::string&
  _name () const;

  virtual const std::string&
  _namespace () const;

  // Element value as xml_schema::type.
  //
  virtual const xml_schema::type*
  _value () const;

  virtual xml_schema::type*
  _value ();
};

void
operator<< (xercesc::DOMElement&, const root&);
```

The `xml_schema::element_type` class is a common base type for all element types and is defined as follows:

```
namespace xml_schema
{
  class element_type
  {
  public:
    virtual
    ~element_type ();

    virtual element_type*
    _clone (flags f = 0) const = 0;

    virtual const std::basic_string<C>&
    _name () const = 0;

    virtual const std::basic_string<C>&
    _namespace () const = 0;

    virtual xml_schema::type*
    _value () = 0;

    virtual const xml_schema::type*
    _value () const = 0;
  };
}
```

The `_value()` member function returns a pointer to the element value or 0 if the element is of a fundamental C++ type and therefore is not derived from `xml_schema::type`.

Unlike parsing and serialization functions, element types are only capable of parsing and serializing from/to a `DOMElement` object. This means that the application will need to perform its own XML-to-DOM parsing and DOM-to-XML serialization. The following section describes a mechanism provided by the mapping to uniformly parse and serialize multiple root elements.

2.9.2 Element Map

When element types are generated for root elements it is also possible to request the generation of an element map with the `--generate-element-map` option. The element map allows uniform parsing and serialization of multiple root elements via the common `xml_schema::element_type` base type. The `xml_schema::element_map` class is defined as follows:

```
namespace xml_schema
{
  class element_map
  {
  public:
    static std::[auto|unique]_ptr<xml_schema::element_type>
    parse (const xercesc::DOMElement&, flags = 0);

    static void
    serialize (xercesc::DOMElement&, const element_type&);
  };
}
```

The `parse()` function creates the corresponding element type object based on the element name and namespace and returns it as an automatic pointer (`std::auto_ptr` or `std::unique_ptr`, depending on the C++ standard selected) to `xml_schema::element_type`. The `serialize()` function serializes the passed element object to `DOMElement`. Note that in case of `serialize()`, the `DOMElement` object should have the correct name and namespace. If no element type is available for an element, both functions throw the `xml_schema::no_element_info` exception:

```
struct no_element_info: virtual exception
{
  no_element_info (const std::basic_string<C>& element_name,
                   const std::basic_string<C>& element_namespace);

  const std::basic_string<C>&
  element_name () const;

  const std::basic_string<C>&
  element_namespace () const;
```

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

The application can discover the actual type of the element object returned by `parse()` either using `dynamic_cast` or by comparing element names and namespaces. The following code fragments illustrate how the element map can be used:

```
// Parsing.
//
DOMElement& e = ... // Parse XML to DOM.

auto_ptr<xml_schema::element_type> r (
  xml_schema::element_map::parse (e));

if (root1 r1 = dynamic_cast<root1*> (r.get ()))
{
  ...
}
else if (r->_name == root2::name () &&
         r->_namespace () == root2::namespace_ ())
{
  root2& r2 (static_cast<root2&> (*r));

  ...
}

// Serialization.
//
xml_schema::element_type& r = ...

string name (r._name ());
string ns (r._namespace ());

DOMDocument& doc = ... // Create a new DOMDocument with name and ns.
DOMElement& e (*doc->getDocumentElement ());

xml_schema::element_map::serialize (e, r);

// Serialize DOMDocument to XML.
```

2.10 Mapping for Global Attributes

An XML Schema attribute definition is called global if it appears directly under the `schema` element. A global attribute does not have any mapping.

2.11 Mapping for `xsi:type` and Substitution Groups

The mapping provides optional support for the XML Schema polymorphism features (`xsi:type` and substitution groups) which can be requested with the `--generate-polymorphic` option. When used, the dynamic type of a member may be different from its static type. Consider the following schema definition and instance document:

```
<!-- test.xsd -->
<schema>
  <complexType name="base">
    <attribute name="text" type="string"/>
  </complexType>

  <complexType name="derived">
    <complexContent>
      <extension base="base">
        <attribute name="extra-text" type="string"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="root_type">
    <sequence>
      <element name="item" type="base" maxOccurs="unbounded"/>
    </sequence>
  </complexType>

  <element name="root" type="root_type"/>
</schema>

<!-- test.xml -->
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <item text="hello"/>
  <item text="hello" extra-text="world" xsi:type="derived"/>
</root>
```

In the resulting object model, the container for the `root::item` member will have two elements: the first element's type will be `base` while the second element's (dynamic) type will be `derived`. This can be discovered using the `dynamic_cast` operator as shown in the following example:

```
void
f (root& r)
{
  for (root::item_const_iterator i (r.item ().begin ());
       i != r.item ().end ()
       ++i)
  {
    if (derived* d = dynamic_cast<derived*> (&(*i)))
```

```
    {
      // derived
    }
    else
    {
      // base
    }
  }
}
```

The `_clone` virtual function should be used instead of copy constructors to make copies of members that might use polymorphism:

```
void
f (root& r)
{
  for (root::item_const_iterator i (r.item ().begin ());
       i != r.item ().end ()
       ++i)
  {
    std::auto_ptr<base> c (i->_clone ());
  }
}
```

The mapping 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 mapping 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`.

You can also indicate that all types should be treated as polymorphic with the `--polymorphic-type-all`. However, this may result in slower generated code with a greater footprint.

2.12 Mapping for `any` and `anyAttribute`

For the XML Schema `any` and `anyAttribute` wildcards an optional mapping can be requested with the `--generate-wildcard` option. The mapping represents the content matched by wildcards as DOM fragments. Because the DOM API is used to access such content, the Xerces-C++ runtime should be initialized by the application prior to parsing and should remain initialized for the lifetime of objects with the wildcard content. For more information on the Xerces-C++ runtime initialization see Section 3.1, "Initializing the Xerces-C++ Runtime".

The mapping for `any` is similar to the mapping for local elements (see Section 2.8, "Mapping for Local Elements and Attributes") except that the type used in the wildcard mapping is `xercesc::DOMElement`. As with local elements, the mapping divides all possible cardinality combinations into three cardinality classes: *one*, *optional*, and *sequence*.

The mapping for `anyAttribute` represents the attributes matched by this wildcard as a set of `xercesc::DOMAttr` objects with a key being the attribute's name and namespace.

Similar to local elements and attributes, the `any` and `anyAttribute` wildcards are mapped to a set of public type definitions (typedefs) and a set of public accessor and modifier functions. Type definitions have names derived from "`any`" for the `any` wildcard and

`"any_attribute"` for the `anyAttribute` wildcard. The accessor and modifier functions are named `"any"` for the `any` wildcard and `"any_attribute"` for the `anyAttribute` wildcard. Subsequent wildcards in the same type have escaped names such as `"any1"` or `"any_attribute1"`.

Because Xerces-C++ DOM nodes always belong to a `DOMDocument`, each type with a wildcard has an associated `DOMDocument` object. The reference to this object can be obtained using the accessor function called `dom_document`. The access to the document object from the application code may be necessary to create or modify the wildcard content. For example:

```
<complexType name="object">
  <sequence>
    <any namespace="##other"/>
  </sequence>
  <anyAttribute namespace="##other"/>
</complexType>
```

is mapped to:

```
class object: public xml_schema::type
{
public:
  // any
  //
  const xercesc::DOMElement&
  any () const;

  void
  any (const xercesc::DOMElement&);

  ...

  // any_attribute
  //
  typedef attribute_set any_attribute_set;
  typedef any_attribute_set::iterator any_attribute_iterator;
  typedef any_attribute_set::const_iterator any_attribute_const_iterator;

  const any_attribute_set&
  any_attribute () const;

  any_attribute_set&
  any_attribute ();

  ...

  // DOMDocument object for wildcard content.
  //
  const xercesc::DOMDocument&
```

```
  dom_document () const;

  xercesc::DOMDocument&
  dom_document ();

  ...
};
```

Names and semantics of type definitions for the wildcards as well as signatures of the accessor and modifier functions depend on the wildcard type as well as the cardinality class for the `any` wildcard. They are described in the following sub-sections.

2.12.1 Mapping for `any` with the One Cardinality Class

For `any` with the One cardinality class, there are no type definitions. The accessor functions come in constant and non-constant versions. The constant accessor function returns a constant reference to `xercesc::DOMElement` and can be used for read-only access. The non-constant version returns an unrestricted reference to `xercesc::DOMElement` and can be used for read-write access.

The first modifier function expects an argument of type reference to constant `xercesc::DOMElement` and makes a deep copy of its argument. The second modifier function expects an argument of type pointer to `xercesc::DOMElement`. This modifier function assumes ownership of its argument and expects the element object to be created using the DOM document associated with this instance. For example:

```
<complexType name="object">
  <sequence>
    <any namespace="##other"/>
  </sequence>
</complexType>
```

is mapped to:

```
class object: public xml_schema::type
{
public:
  // Accessors.
  //
  const xercesc::DOMElement&
  any () const;

  xercesc::DOMElement&
  any ();

  // Modifiers.
  //
  void
```

```
  any (const xercesc::DOMElement&);

  void
  any (xercesc::DOMElement*);

  ...

};
```

The following code shows how one could use this mapping:

```
void
f (object& o, const xercesc::DOMElement& e)
{
  using namespace xercesc;

  DOMElement& e1 (o.any ());             // get
  o.any (e)                              // set, deep copy
  DOMDocument& doc (o.dom_document ());
  o.any (doc.createElement (...));       // set, assumes ownership
}
```

2.12.2 Mapping for `any` with the Optional Cardinality Class

For `any` with the Optional cardinality class, the type definitions consist of an alias for the container type with name `any_optional` (or `any1_optional`, etc., for subsequent wildcards in the type definition).

Unlike accessor functions for the One cardinality class, accessor functions for the Optional cardinality class return references to corresponding containers rather than directly to `DOMElement`. The accessor functions come in constant and non-constant versions. The constant accessor function returns a constant reference to the container and can be used for read-only access. The non-constant version returns an unrestricted reference to the container and can be used for read-write access.

The modifier functions are overloaded for `xercesc::DOMElement` and the container type. The first modifier function expects an argument of type reference to constant `xercesc::DOMElement` and makes a deep copy of its argument. The second modifier function expects an argument of type pointer to `xercesc::DOMElement`. This modifier function assumes ownership of its argument and expects the element object to be created using the DOM document associated with this instance. The third modifier function expects an argument of type reference to constant of the container type and makes a deep copy of its argument. For instance:

```
<complexType name="object">
  <sequence>
    <any namespace="##other" minOccurs="0"/>
  </sequence>
</complexType>
```

is mapped to:

```
class object: public xml_schema::type
{
public:
  // Type definitions.
  //
  typedef element_optional any_optional;

  // Accessors.
  //
  const any_optional&
  any () const;

  any_optional&
  any ();

  // Modifiers.
  //
  void
  any (const xercesc::DOMElement&);

  void
  any (xercesc::DOMElement*);

  void
  any (const any_optional&);

  ...

};
```

The `element_optional` container is a specialization of the `optional` class template described in Section 2.8.2, "Mapping for Members with the Optional Cardinality Class". Its interface is presented below:

```
class element_optional
{
public:
  explicit
  element_optional (xercesc::DOMDocument&);

  // Makes a deep copy.
  //
```

```
  element_optional (const xercesc::DOMElement&, xercesc::DOMDocument&);

  // Assumes ownership.
  //
  element_optional (xercesc::DOMElement*, xercesc::DOMDocument&);

  element_optional (const element_optional&, xercesc::DOMDocument&);

public:
  element_optional&
  operator= (const xercesc::DOMElement&);

  element_optional&
  operator= (const element_optional&);

  // Pointer-like interface.
  //
public:
  const xercesc::DOMElement*
  operator-> () const;

  xercesc::DOMElement*
  operator-> ();

  const xercesc::DOMElement&
  operator* () const;

  xercesc::DOMElement&
  operator* ();

  typedef void (element_optional::*bool_convertible) ();
  operator bool_convertible () const;

  // Get/set interface.
  //
public:
  bool
  present () const;

  const xercesc::DOMElement&
  get () const;

  xercesc::DOMElement&
  get ();

  // Makes a deep copy.
  //
  void
  set (const xercesc::DOMElement&);

  // Assumes ownership.
```

```
  //
  void
  set (xercesc::DOMElement*);

  void
  reset ();
};

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

bool
operator!= (const element_optional&, const element_optional&);
```

The following code shows how one could use this mapping:

```
void
f (object& o, const xercesc::DOMElement& e)
{
  using namespace xercesc;

  DOMDocument& doc (o.dom_document ());

  if (o.any ().present ())                  // test
  {
    DOMElement& e1 (o.any ().get ());       // get
    o.any ().set (e);                       // set, deep copy
    o.any ().set (doc.createElement (...)); // set, assumes ownership
    o.any ().reset ();                      // reset
  }

  // Same as above but using pointer notation:
  //
  if (o.member ())                          // test
  {
    DOMElement& e1 (*o.any ());             // get
    o.any (e);                              // set, deep copy
    o.any (doc.createElement (...));        // set, assumes ownership
    o.any ().reset ();                      // reset
  }
}
```

2.12.3 Mapping for `any` with the Sequence Cardinality Class

For `any` with the Sequence cardinality class, the type definitions consist of an alias of the container type with name `any_sequence` (or `any1_sequence`, etc., for subsequent wildcards in the type definition), an alias of the iterator type with name `any_iterator` (or `any1_iterator`, etc., for subsequent wildcards in the type definition), and an alias of the constant iterator type with name `any_const_iterator` (or `any1_const_iterator`, etc.,

for subsequent wildcards in the type definition).

The accessor functions come in constant and non-constant versions. The constant accessor function returns a constant reference to the container and can be used for read-only access. The non-constant version returns an unrestricted reference to the container and can be used for read-write access.

The modifier function expects an argument of type reference to constant of the container type. The modifier function makes a deep copy of its argument. For instance:

```
<complexType name="object">
  <sequence>
    <any namespace="##other" minOccurs="unbounded"/>
  </sequence>
</complexType>
```

is mapped to:

```
class object: public xml_schema::type
{
public:
  // Type definitions.
  //
  typedef element_sequence any_sequence;
  typedef any_sequence::iterator any_iterator;
  typedef any_sequence::const_iterator any_const_iterator;

  // Accessors.
  //
  const any_sequence&
  any () const;

  any_sequence&
  any ();

  // Modifier.
  //
  void
  any (const any_sequence&);

  ...

};
```

The `element_sequence` container is a specialization of the `sequence` class template described in Section 2.8.3, "Mapping for Members with the Sequence Cardinality Class". Its interface is similar to the sequence interface as defined by the ISO/ANSI Standard for C++ (ISO/IEC 14882:1998, Section 23.1.1, "Sequences") and is presented below:

```
class element_sequence
{
public:
  typedef xercesc::DOMElement        value_type;
  typedef xercesc::DOMElement*       pointer;
  typedef const xercesc::DOMElement* const_pointer;
  typedef xercesc::DOMElement&       reference;
  typedef const xercesc::DOMElement& const_reference;

  typedef <implementation-defined>   iterator;
  typedef <implementation-defined>   const_iterator;
  typedef <implementation-defined>   reverse_iterator;
  typedef <implementation-defined>   const_reverse_iterator;

  typedef <implementation-defined>   size_type;
  typedef <implementation-defined>   difference_type;
  typedef <implementation-defined>   allocator_type;

public:
  explicit
  element_sequence (xercesc::DOMDocument&);

  // DOMElement cannot be default-constructed.
  //
  // explicit
  // element_sequence (size_type n);

  element_sequence (size_type n,
                    const xercesc::DOMElement&,
                    xercesc::DOMDocument&);

  template <typename I>
  element_sequence (const I& begin,
                    const I& end,
                    xercesc::DOMDocument&);

  element_sequence (const element_sequence&, xercesc::DOMDocument&);

  element_sequence&
  operator= (const element_sequence&);

public:
  void
  assign (size_type n, const xercesc::DOMElement&);

  template <typename I>
  void
  assign (const I& begin, const I& end);

public:
  // This version of resize can only be used to shrink the
```

```
  // sequence because DOMElement cannot be default-constructed.
  //
  void
  resize (size_type);

  void
  resize (size_type, const xercesc::DOMElement&);

public:
  size_type
  size () const;

  size_type
  max_size () const;

  size_type
  capacity () const;

  bool
  empty () const;

  void
  reserve (size_type);

  void
  clear ();

public:
  const_iterator
  begin () const;

  const_iterator
  end () const;

  iterator
  begin ();

  iterator
  end ();

  const_reverse_iterator
  rbegin () const;

  const_reverse_iterator
  rend () const

    reverse_iterator
  rbegin ();

  reverse_iterator
  rend ();
```

```
public:
  xercesc::DOMElement&
  operator[] (size_type);

  const xercesc::DOMElement&
  operator[] (size_type) const;

  xercesc::DOMElement&
  at (size_type);

  const xercesc::DOMElement&
  at (size_type) const;

  xercesc::DOMElement&
  front ();

  const xercesc::DOMElement&
  front () const;

  xercesc::DOMElement&
  back ();

  const xercesc::DOMElement&
  back () const;

public:
  // Makes a deep copy.
  //
  void
  push_back (const xercesc::DOMElement&);

  // Assumes ownership.
  //
  void
  push_back (xercesc::DOMElement*);

  void
  pop_back ();

  // Makes a deep copy.
  //
  iterator
  insert (iterator position, const xercesc::DOMElement&);

  // Assumes ownership.
  //
  iterator
  insert (iterator position, xercesc::DOMElement*);

  void
```

```
  insert (iterator position, size_type n, const xercesc::DOMElement&);

  template <typename I>
  void
  insert (iterator position, const I& begin, const I& end);

  iterator
  erase (iterator position);

  iterator
  erase (iterator begin, iterator end);

public:
  // Note that the DOMDocument object of the two sequences being
  // swapped should be the same.
  //
  void
  swap (sequence& x);
};

inline bool
operator== (const element_sequence&, const element_sequence&);

inline bool
operator!= (const element_sequence&, const element_sequence&);
```

The following code shows how one could use this mapping:

```
void
f (object& o, const xercesc::DOMElement& e)
{
  using namespace xercesc;

  object::any_sequence& s (o.any ());

  // Iteration.
  //
  for (object::any_iterator i (s.begin ()); i != s.end (); ++i)
  {
    DOMElement& e (*i);
  }

  // Modification.
  //
  s.push_back (e);                        // deep copy
  DOMDocument& doc (o.dom_document ());
  s.push_back (doc.createElement (...)); // assumes ownership
}
```

2.12.4 Element Wildcard Order

Similar to elements, element wildcards in ordered types (Section 2.8.4, "Element Order") are assigned content ids and are included in the content order sequence. Continuing with the bank transactions example started in Section 2.8.4, we can extend the batch by allowing custom transactions:

```
<complexType name="batch">
  <choice minOccurs="0" maxOccurs="unbounded">
    <element name="withdraw" type="withdraw"/>
    <element name="deposit" type="deposit"/>
    <any namespace="##other" processContents="lax"/>
  </choice>
</complexType>
```

This will lead to the following changes in the generated `batch` C++ class:

```
class batch: public xml_schema::type
{
public:
  ...

  // any
  //
  typedef element_sequence any_sequence;
  typedef any_sequence::iterator any_iterator;
  typedef any_sequence::const_iterator any_const_iterator;

  static const std::size_t any_id = 3UL;

  const any_sequence&
  any () const;

  any_sequence&
  any ();

  void
  any (const any_sequence&);

  ...
};
```

With this change we also need to update the iteration code to handle the new content id:

```
for (batch::content_order_const_iterator i (b.content_order ().begin ());
     i != b.content_order ().end ();
     ++i)
{
  switch (i->id)
  {
```

```
    ...

  case batch::any_id:
    {
      const DOMElement& e (b.any ()[i->index]);
      ...
      break;
    }

    ...
  }
}
```

For the complete working code that shows the use of wildcards in ordered types refer to the `order/element` example in the `examples/cxx/tree/` directory in the XSD distribution.

2.12.5 Mapping for `anyAttribute`

For `anyAttribute` the type definitions consist of an alias of the container type with name `any_attribute_set` (or `any1_attribute_set`, etc., for subsequent wildcards in the type definition), an alias of the iterator type with name `any_attribute_iterator` (or `any1_attribute_iterator`, etc., for subsequent wildcards in the type definition), and an alias of the constant iterator type with name `any_attribute_const_iterator` (or `any1_attribute_const_iterator`, etc., for subsequent wildcards in the type definition).

The accessor functions come in constant and non-constant versions. The constant accessor function returns a constant reference to the container and can be used for read-only access. The non-constant version returns an unrestricted reference to the container and can be used for read-write access.

The modifier function expects an argument of type reference to constant of the container type. The modifier function makes a deep copy of its argument. For instance:

```
<complexType name="object">
  <sequence>
    ...
  </sequence>
  <anyAttribute namespace="##other"/>
</complexType>
```

is mapped to:

```
class object: public xml_schema::type
{
public:
  // Type definitions.
  //
  typedef attribute_set any_attribute_set;
```

```
  typedef any_attribute_set::iterator any_attribute_iterator;
  typedef any_attribute_set::const_iterator any_attribute_const_iterator;

  // Accessors.
  //
  const any_attribute_set&
  any_attribute () const;

  any_attribute_set&
  any_attribute ();

  // Modifier.
  //
  void
  any_attribute (const any_attribute_set&);

  ...

};
```

The `attribute_set` class is an associative container similar to the `std::set` class template as defined by the ISO/ANSI Standard for C++ (ISO/IEC 14882:1998, Section 23.3.3, "Class template set") with the key being the attribute's name and namespace. Unlike `std::set`, `attribute_set` allows searching using names and namespaces instead of `xercesc::DOMAttr` objects. It is defined in an implementation-specific namespace and its interface is presented below:

```
class attribute_set
{
public:
  typedef xercesc::DOMAttr         key_type;
  typedef xercesc::DOMAttr         value_type;
  typedef xercesc::DOMAttr*        pointer;
  typedef const xercesc::DOMAttr*  const_pointer;
  typedef xercesc::DOMAttr&        reference;
  typedef const xercesc::DOMAttr&  const_reference;

  typedef <implementation-defined> iterator;
  typedef <implementation-defined> const_iterator;
  typedef <implementation-defined> reverse_iterator;
  typedef <implementation-defined> const_reverse_iterator;

  typedef <implementation-defined> size_type;
  typedef <implementation-defined> difference_type;
  typedef <implementation-defined> allocator_type;

public:
  attribute_set (xercesc::DOMDocument&);

  template <typename I>
```

```
  attribute_set (const I& begin, const I& end, xercesc::DOMDocument&);

  attribute_set (const attribute_set&, xercesc::DOMDocument&);

  attribute_set&
  operator= (const attribute_set&);

public:
  const_iterator
  begin () const;

  const_iterator
  end () const;

  iterator
  begin ();

  iterator
  end ();

  const_reverse_iterator
  rbegin () const;

  const_reverse_iterator
  rend () const;

  reverse_iterator
  rbegin ();

  reverse_iterator
  rend ();

public:
  size_type
  size () const;

  size_type
  max_size () const;

  bool
  empty () const;

  void
  clear ();

public:
  // Makes a deep copy.
  //
  std::pair<iterator, bool>
  insert (const xercesc::DOMAttr&);
```

```
  // Assumes ownership.
  //
  std::pair<iterator, bool>
  insert (xercesc::DOMAttr*);

  // Makes a deep copy.
  //
  iterator
  insert (iterator position, const xercesc::DOMAttr&);

  // Assumes ownership.
  //
  iterator
  insert (iterator position, xercesc::DOMAttr*);

  template <typename I>
  void
  insert (const I& begin, const I& end);

public:
  void
  erase (iterator position);

  size_type
  erase (const std::basic_string<C>& name);

  size_type
  erase (const std::basic_string<C>& namespace_,
         const std::basic_string<C>& name);

  size_type
  erase (const XMLCh* name);

  size_type
  erase (const XMLCh* namespace_, const XMLCh* name);

  void
  erase (iterator begin, iterator end);

public:
  size_type
  count (const std::basic_string<C>& name) const;

  size_type
  count (const std::basic_string<C>& namespace_,
         const std::basic_string<C>& name) const;

  size_type
  count (const XMLCh* name) const;

  size_type
```

```
  count (const XMLCh* namespace_, const XMLCh* name) const;

  iterator
  find (const std::basic_string<C>& name);

  iterator
  find (const std::basic_string<C>& namespace_,
        const std::basic_string<C>& name);

  iterator
  find (const XMLCh* name);

  iterator
  find (const XMLCh* namespace_, const XMLCh* name);

  const_iterator
  find (const std::basic_string<C>& name) const;

  const_iterator
  find (const std::basic_string<C>& namespace_,
        const std::basic_string<C>& name) const;

  const_iterator
  find (const XMLCh* name) const;

  const_iterator
  find (const XMLCh* namespace_, const XMLCh* name) const;

public:
  // Note that the DOMDocument object of the two sets being
  // swapped should be the same.
  //
  void
  swap (attribute_set&);
};

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

bool
operator!= (const attribute_set&, const attribute_set&);
```

The following code shows how one could use this mapping:

```
void
f (object& o, const xercesc::DOMAttr& a)
{
  using namespace xercesc;

  object::any_attribute_set& s (o.any_attribute ());
```

```
  // Iteration.
  //
  for (object::any_attribute_iterator i (s.begin ()); i != s.end (); ++i)
  {
    DOMAttr& a (*i);
  }

  // Modification.
  //
  s.insert (a);                           // deep copy
  DOMDocument& doc (o.dom_document ());
  s.insert (doc.createAttribute (...)); // assumes ownership

  // Searching.
  //
  object::any_attribute_iterator i (s.find ("name"));
  i = s.find ("http://www.w3.org/XML/1998/namespace", "lang");
}
```

2.13 Mapping for Mixed Content Models

For XML Schema types with mixed content models C++/Tree provides mapping support only if the type is marked as ordered (Section 2.8.4, "Element Order"). Use the `--ordered-type-mixed` XSD compiler option to automatically mark all types with mixed content as ordered.

For an ordered type with mixed content, C++/Tree adds an extra text content sequence that is used to store the text fragments. This text content sequence is also assigned the content id and its entries are included in the content order sequence, just like elements. As a result, it is possible to capture the order between elements and text fragments.

As an example, consider the following schema that describes text with embedded links:

```
<complexType name="anchor">
  <simpleContent>
    <extension base="string">
      <attribute name="href" type="anyURI" use="required"/>
    </extension>
  </simpleContent>
</complexType>

<complexType name="text" mixed="true">
  <sequence>
    <element name="a" type="anchor" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

The generated `text` C++ class will provide the following API (assuming it is marked as ordered):

```
class text: public xml_schema::type
{
public:
  // a
  //
  typedef anchor a_type;
  typedef sequence<a_type> a_sequence;
  typedef a_sequence::iterator a_iterator;
  typedef a_sequence::const_iterator a_const_iterator;

  static const std::size_t a_id = 1UL;

  const a_sequence&
  a () const;

  a_sequence&
  a ();

  void
  a (const a_sequence&);

  // text_content
  //
  typedef xml_schema::string text_content_type;
  typedef sequence<text_content_type> text_content_sequence;
  typedef text_content_sequence::iterator text_content_iterator;
  typedef text_content_sequence::const_iterator text_content_const_iterator;

  static const std::size_t text_content_id = 2UL;

  const text_content_sequence&
  text_content () const;

  text_content_sequence&
  text_content ();

  void
  text_content (const text_content_sequence&);

  // content_order
  //
  typedef xml_schema::content_order content_order_type;
  typedef std::vector<content_order_type> content_order_sequence;
  typedef content_order_sequence::iterator content_order_iterator;
  typedef content_order_sequence::const_iterator content_order_const_iterator;

  const content_order_sequence&
  content_order () const;
```

```
  content_order_sequence&
  content_order ();

  void
  content_order (const content_order_sequence&);

  ...
};
```

Given this interface we can iterate over both link elements and text in content order. The following code fragment converts our format to plain text with references.

```
const text& t = ...

for (text::content_order_const_iterator i (t.content_order ().begin ());
     i != t.content_order ().end ();
     ++i)
{
  switch (i->id)
  {
  case text::a_id:
    {
      const anchor& a (t.a ()[i->index]);
      cerr << a << "[" << a.href () << "]";
      break;
    }
  case text::text_content_id:
    {
      const xml_schema::string& s (t.text_content ()[i->index]);
      cerr << s;
      break;
    }
  default:
    {
      assert (false); // Unknown content id.
    }
  }
}
```

For the complete working code that shows the use of mixed content in ordered types refer to the `order/mixed` example in the `examples/cxx/tree/` directory in the XSD distribution.

3 Parsing

This chapter covers various aspects of parsing XML instance documents in order to obtain corresponding tree-like object model.

Each global XML Schema element in the form:

```
<element name="name" type="type"/>
```

is mapped to 14 overloaded C++ functions in the form:

```
// Read from a URI or a local file.
//

std::[auto|unique]_ptr<type>
name (const std::basic_string<C>& uri,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());

std::[auto|unique]_ptr<type>
name (const std::basic_string<C>& uri,
      xml_schema::error_handler&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());

std::[auto|unique]_ptr<type>
name (const std::basic_string<C>& uri,
      xercesc::DOMErrorHandler&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());


// Read from std::istream.
//

std::[auto|unique]_ptr<type>
name (std::istream&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());

std::[auto|unique]_ptr<type>
name (std::istream&,
      xml_schema::error_handler&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());

std::[auto|unique]_ptr<type>
name (std::istream&,
      xercesc::DOMErrorHandler&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());


std::[auto|unique]_ptr<type>
name (std::istream&,
      const std::basic_string<C>& id,
```

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

std::[auto|unique]_ptr<type>
name (std::istream&,
      const std::basic_string<C>& id,
      xml_schema::error_handler&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());

std::[auto|unique]_ptr<type>
name (std::istream&,
      const std::basic_string<C>& id,
      xercesc::DOMErrorHandler&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());


// Read from InputSource.
//

std::[auto|unique]_ptr<type>
name (xercesc::InputSource&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());

std::[auto|unique]_ptr<type>
name (xercesc::InputSource&,
      xml_schema::error_handler&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());

std::[auto|unique]_ptr<type>
name (xercesc::InputSource&,
      xercesc::DOMErrorHandler&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());


// Read from DOM.
//

std::[auto|unique]_ptr<type>
name (const xercesc::DOMDocument&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());

std::[auto|unique]_ptr<type>
name (xml_schema::dom::[auto|unique]_ptr<xercesc::DOMDocument>,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());
```

You can choose between reading an XML instance from a local file, URI, `std::istream`, `xercesc::InputSource`, or a pre-parsed DOM instance in the form of `xercesc::DOMDocument`. All the parsing functions return a dynamically allocated object model as either `std::auto_ptr` or `std::unique_ptr`, depending on the C++ standard selected. Each of these parsing functions is discussed in more detail in the following sections.

3.1 Initializing the Xerces-C++ Runtime

Some parsing functions expect you to initialize the Xerces-C++ runtime while others initialize and terminate it as part of their work. The general rule is as follows: if a function has any arguments or return a value that is an instance of a Xerces-C++ type, then this function expects you to initialize the Xerces-C++ runtime. Otherwise, the function initializes and terminates the runtime for you. Note that it is legal to have nested calls to the Xerces-C++ initialize and terminate functions as long as the calls are balanced.

You can instruct parsing functions that initialize and terminate the runtime not to do so by passing the `xml_schema::flags::dont_initialize` flag (see Section 3.2, "Flags and Properties").

3.2 Flags and Properties

Parsing flags and properties are the last two arguments of every parsing function. They allow you to fine-tune the process of instance validation and parsing. Both arguments are optional.

The following flags are recognized by the parsing functions:

`xml_schema::flags::keep_dom`
: Keep association between DOM nodes and the resulting object model nodes. For more information about DOM association refer to Section 5.1, "DOM Association".

`xml_schema::flags::own_dom`
: Assume ownership of the DOM document passed. This flag only makes sense together with the `keep_dom` flag in the call to the parsing function with the `xml_schema::dom::[auto|unique]_ptr<DOMDocument>` argument.

`xml_schema::flags::dont_validate`
: Do not validate instance documents against schemas.

`xml_schema::flags::dont_initialize`
: Do not initialize the Xerces-C++ runtime.

You can pass several flags by combining them using the bit-wise OR operator. For example:

```
using xml_schema::flags;

std::auto_ptr<type> r (
  name ("test.xml", flags::keep_dom | flags::dont_validate));
```

By default, validation of instance documents is turned on even though parsers generated by XSD do not assume instance documents are valid. They include a number of checks that prevent construction of inconsistent object models. This, however, does not mean that an instance document that was successfully parsed by the XSD-generated parsers is valid per the corresponding schema. If an instance document is not "valid enough" for the generated parsers to construct consistent object model, one of the exceptions defined in `xml_schema` namespace is thrown (see Section 3.3, "Error Handling").

For more information on the Xerces-C++ runtime initialization refer to Section 3.1, "Initializing the Xerces-C++ Runtime".

The `xml_schema::properties` class allows you to programmatically specify schema locations to be used instead of those specified with the `xsi::schemaLocation` and `xsi::noNamespaceSchemaLocation` attributes in instance documents. The interface of the `properties` class is presented below:

```
class properties
{
public:
  void
  schema_location (const std::basic_string<C>& namespace_,
                   const std::basic_string<C>& location);
  void
  no_namespace_schema_location (const std::basic_string<C>& location);
};
```

Note that all locations are relative to an instance document unless they are URIs. For example, if you want to use a local file as your schema, then you will need to pass `file:///absolute/path/to/your/schema` as the location argument.

3.3 Error Handling

As discussed in Section 2.2, "Error Handling", the mapping uses the C++ exception handling mechanism as its primary way of reporting error conditions. However, to handle recoverable parsing and validation errors and warnings, a callback interface maybe preferred by the application.

To better understand error handling and reporting strategies employed by the parsing functions, it is useful to know that the transformation of an XML instance document to a statically-typed tree happens in two stages. The first stage, performed by Xerces-C++, consists of parsing an XML document into a DOM instance. For short, we will call this stage the XML-DOM stage. Validation, if not disabled, happens during this stage. The second stage, performed by the generated parsers, consist of parsing the DOM instance into the statically-typed tree. We will call this stage the DOM-Tree stage. Additional checks are performed during this stage in order to prevent construction of inconsistent tree which could otherwise happen when validation is disabled, for

example.

All parsing functions except the one that operates on a DOM instance come in overloaded triples. The first function in such a triple reports error conditions exclusively by throwing exceptions. It accumulates all the parsing and validation errors of the XML-DOM stage and throws them in a single instance of the `xml_schema::parsing` exception (described below). The second and the third functions in the triple use callback interfaces to report parsing and validation errors and warnings. The two callback interfaces are `xml_schema::error_handler` and `xercesc::DOMErrorHandler`. For more information on the `xercesc::DOMErrorHandler` interface refer to the Xerces-C++ documentation. The `xml_schema::error_handler` interface is presented below:

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

  virtual bool
  handle (const std::basic_string<C>& id,
          unsigned long line,
          unsigned long column,
          severity,
          const std::basic_string<C>& message) = 0;

  virtual
  ~error_handler ();
};
```

The `id` argument of the `error_handler::handle` function identifies the resource being parsed (e.g., a file name or URI).

By returning `true` from the `handle` function you instruct the parser to recover and continue parsing. Returning `false` results in termination of the parsing process. An error with the `fatal` severity level results in termination of the parsing process no matter what is returned from the `handle` function. It is safe to throw an exception from the `handle` function.

The DOM-Tree stage reports error conditions exclusively by throwing exceptions. Individual exceptions thrown by the parsing functions are described in the following sub-sections.

3.3.1 `xml_schema::parsing`

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

  severity (value);
  operator value () const;
};

struct error
{
  error (severity,
         const std::basic_string<C>& id,
         unsigned long line,
         unsigned long column,
         const std::basic_string<C>& message);

  severity
  severity () const;

  const std::basic_string<C>&
  id () const;

  unsigned long
  line () const;

  unsigned long
  column () const;

  const std::basic_string<C>&
  message () const;
};

std::basic_ostream<C>&
operator<< (std::basic_ostream<C>&, const error&);

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

std::basic_ostream<C>&
operator<< (std::basic_ostream<C>&, const diagnostics&);

struct parsing: virtual exception
{
  parsing ();
```

```
  parsing (const diagnostics&);

  const diagnostics&
  diagnostics () const;

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

The `xml_schema::parsing` exception is thrown if there were parsing or validation errors reported during the XML-DOM stage. If no callback interface was provided to the parsing function, the exception contains a list of errors and warnings accessible using the `diagnostics` function. The usual conditions when this exception is thrown include malformed XML instances and, if validation is turned on, invalid instance documents.

3.3.2 `xml_schema::expected_element`

```
struct expected_element: virtual exception
{
  expected_element (const std::basic_string<C>& name,
                    const std::basic_string<C>& namespace_);


  const std::basic_string<C>&
  name () const;

  const std::basic_string<C>&
  namespace_ () const;


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

The `xml_schema::expected_element` exception is thrown when an expected element is not encountered by the DOM-Tree stage. The name and namespace of the expected element can be obtained using the `name` and `namespace_` functions respectively.

3.3.3 `xml_schema::unexpected_element`

```
struct unexpected_element: virtual exception
{
  unexpected_element (const std::basic_string<C>& encountered_name,
                      const std::basic_string<C>& encountered_namespace,
                      const std::basic_string<C>& expected_name,
                      const std::basic_string<C>& expected_namespace)


  const std::basic_string<C>&
```

```
  encountered_name () const;

  const std::basic_string<C>&
  encountered_namespace () const;


  const std::basic_string<C>&
  expected_name () const;

  const std::basic_string<C>&
  expected_namespace () const;


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

The `xml_schema::unexpected_element` exception is thrown when an unexpected element is encountered by the DOM-Tree stage. The name and namespace of the encountered element can be obtained using the `encountered_name` and `encountered_namespace` functions respectively. If an element was expected instead of the encountered one, its name and namespace can be obtained using the `expected_name` and `expected_namespace` functions respectively. Otherwise these functions return empty strings.

3.3.4 `xml_schema::expected_attribute`

```
struct expected_attribute: virtual exception
{
  expected_attribute (const std::basic_string<C>& name,
                      const std::basic_string<C>& namespace_);


  const std::basic_string<C>&
  name () const;

  const std::basic_string<C>&
  namespace_ () const;


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

The `xml_schema::expected_attribute` exception is thrown when an expected attribute is not encountered by the DOM-Tree stage. The name and namespace of the expected attribute can be obtained using the `name` and `namespace_` functions respectively.

3.3.5 `xml_schema::unexpected_enumerator`

```
struct unexpected_enumerator: virtual exception
{
  unexpected_enumerator (const std::basic_string<C>& enumerator);

  const std::basic_string<C>&
  enumerator () const;

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

The `xml_schema::unexpected_enumerator` exception is thrown when an unexpected enumerator is encountered by the DOM-Tree stage. The enumerator can be obtained using the `enumerator` functions.

3.3.6 `xml_schema::expected_text_content`

```
struct expected_text_content: virtual exception
{
  virtual const char*
  what () const throw ();
};
```

The `xml_schema::expected_text_content` exception is thrown when a content other than text is encountered and the text content was expected by the DOM-Tree stage.

3.3.7 `xml_schema::no_type_info`

```
struct no_type_info: virtual exception
{
  no_type_info (const std::basic_string<C>& type_name,
                const std::basic_string<C>& type_namespace);

  const std::basic_string<C>&
  type_name () const;

  const std::basic_string<C>&
  type_namespace () const;

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

The `xml_schema::no_type_info` exception is thrown when there is no type information associated with a type specified by the `xsi:type` attribute. This exception is thrown by the DOM-Tree stage. The name and namespace of the type in question can be obtained using the

`type_name` and `type_namespace` functions respectively. Usually, catching this exception means that you haven't linked the code generated from the schema defining the type in question with your application or this schema has been compiled without the `--generate-polymorphic` option.

3.3.8 `xml_schema::not_derived`

```
struct not_derived: virtual exception
{
  not_derived (const std::basic_string<C>& base_type_name,
               const std::basic_string<C>& base_type_namespace,
               const std::basic_string<C>& derived_type_name,
               const std::basic_string<C>& derived_type_namespace);

  const std::basic_string<C>&
  base_type_name () const;

  const std::basic_string<C>&
  base_type_namespace () const;


  const std::basic_string<C>&
  derived_type_name () const;

  const std::basic_string<C>&
  derived_type_namespace () const;

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

The `xml_schema::not_derived` exception is thrown when a type specified by the `xsi:type` attribute is not derived from the expected base type. This exception is thrown by the DOM-Tree stage. The name and namespace of the expected base type can be obtained using the `base_type_name` and `base_type_namespace` functions respectively. The name and namespace of the offending type can be obtained using the `derived_type_name` and `derived_type_namespace` functions respectively.

3.3.9 `xml_schema::no_prefix_mapping`

```
struct no_prefix_mapping: virtual exception
{
  no_prefix_mapping (const std::basic_string<C>& prefix);

  const std::basic_string<C>&
  prefix () const;
```

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

The `xml_schema::no_prefix_mapping` exception is thrown during the DOM-Tree stage if a namespace prefix is encountered for which a prefix-namespace mapping hasn't been provided. The namespace prefix in question can be obtained using the `prefix` function.

3.4 Reading from a Local File or URI

Using a local file or URI is the simplest way to parse an XML instance. For example:

```
using std::auto_ptr;

auto_ptr<type> r1 (name ("test.xml"));
auto_ptr<type> r2 (name ("http://www.codesynthesis.com/test.xml"));
```

Or, in the C++11 mode:

```
using std::unique_ptr;

unique_ptr<type> r1 (name ("test.xml"));
unique_ptr<type> r2 (name ("http://www.codesynthesis.com/test.xml"));
```

3.5 Reading from `std::istream`

When using an `std::istream` instance, you may also pass an optional resource id. This id is used to identify the resource (for example in error messages) as well as to resolve relative paths. For instance:

```
using std::auto_ptr;

{
  std::ifstream ifs ("test.xml");
  auto_ptr<type> r (name (ifs, "test.xml"));
}

{
  std::string str ("..."); // Some XML fragment.
  std::istringstream iss (str);
  auto_ptr<type> r (name (iss));
}
```

3.6 Reading from `xercesc::InputSource`

Reading from a `xercesc::InputSource` instance is similar to the `std::istream` case except the resource id is maintained by the `InputSource` object. For instance:

```
xercesc::StdInInputSource is;
std::auto_ptr<type> r (name (is));
```

3.7 Reading from DOM

Reading from a `xercesc::DOMDocument` instance allows you to setup a custom XML-DOM stage. Things like DOM parser reuse, schema pre-parsing, and schema caching can be achieved with this approach. For more information on how to obtain DOM representation from an XML instance refer to the Xerces-C++ documentation. In addition, the C++/Tree Mapping FAQ shows how to parse an XML instance to a Xerces-C++ DOM document using the XSD runtime utilities.

The last parsing function is useful when you would like to perform your own XML-to-DOM parsing and associate the resulting DOM document with the object model nodes. The automatic `DOMDocument` pointer is reset and the resulting object model assumes ownership of the DOM document passed. For example:

```
// C++98 version.
//
xml_schema::dom::auto_ptr<xercesc::DOMDocument> doc = ...

std::auto_ptr<type> r (
  name (doc, xml_schema::flags::keep_dom | xml_schema::flags::own_dom));

// At this point doc is reset to 0.

// C++11 version.
//
xml_schema::dom::unique_ptr<xercesc::DOMDocument> doc = ...

std::unique_ptr<type> r (
  name (std::move (doc),
        xml_schema::flags::keep_dom | xml_schema::flags::own_dom));

// At this point doc is reset to 0.
```

4 Serialization

This chapter covers various aspects of serializing a tree-like object model to DOM or XML. In this regard, serialization is complimentary to the reverse process of parsing a DOM or XML instance into an object model which is discussed in Chapter 3, "Parsing". Note that the generation of the serialization code is optional and should be explicitly requested with the `--gener-`

`ate-serialization` option. See the XSD Compiler Command Line Manual for more information.

Each global XML Schema element in the form:

```
<xsd:element name="name" type="type"/>
```

is mapped to 8 overloaded C++ functions in the form:

```
// Serialize to std::ostream.
//
void
name (std::ostream&,
      const type&,
      const xml_schema::namespace_fomap& =
        xml_schema::namespace_infomap (),
      const std::basic_string<C>& encoding = "UTF-8",
      xml_schema::flags = 0);

void
name (std::ostream&,
      const type&,
      xml_schema::error_handler&,
      const xml_schema::namespace_infomap& =
        xml_schema::namespace_infomap (),
      const std::basic_string<C>& encoding = "UTF-8",
      xml_schema::flags = 0);

void
name (std::ostream&,
      const type&,
      xercesc::DOMErrorHandler&,
      const xml_schema::namespace_infomap& =
        xml_schema::namespace_infomap (),
      const std::basic_string<C>& encoding = "UTF-8",
      xml_schema::flags = 0);


// Serialize to XMLFormatTarget.
//
void
name (xercesc::XMLFormatTarget&,
      const type&,
      const xml_schema::namespace_infomap& =
        xml_schema::namespace_infomap (),
      const std::basic_string<C>& encoding = "UTF-8",
      xml_schema::flags = 0);

void
name (xercesc::XMLFormatTarget&,
```

```
      const type&,
      xml_schema::error_handler&,
      const xml_schema::namespace_infomap& =
        xml_schema::namespace_infomap (),
      const std::basic_string<C>& encoding = "UTF-8",
      xml_schema::flags = 0);

void
name (xercesc::XMLFormatTarget&,
      const type&,
      xercesc::DOMErrorHandler&,
      const xml_schema::namespace_infomap& =
        xml_schema::namespace_infomap (),
      const std::basic_string<C>& encoding = "UTF-8",
      xml_schema::flags = 0);


// Serialize to DOM.
//
xml_schema::dom::[auto|unique]_ptr<xercesc::DOMDocument>
name (const type&,
      const xml_schema::namespace_infomap&
        xml_schema::namespace_infomap (),
      xml_schema::flags = 0);

void
name (xercesc::DOMDocument&,
      const type&,
      xml_schema::flags = 0);
```

You can choose between writing XML to `std::ostream` or `xercesc::XMLFormatTarget` and creating a DOM instance in the form of `xercesc::DOMDocument`. Serialization to `ostream` or `XMLFormatTarget` requires a considerably less work while serialization to DOM provides for greater flexibility. Each of these serialization functions is discussed in more detail in the following sections.

4.1 Initializing the Xerces-C++ Runtime

Some serialization functions expect you to initialize the Xerces-C++ runtime while others initialize and terminate it as part of their work. The general rule is as follows: if a function has any arguments or return a value that is an instance of a Xerces-C++ type, then this function expects you to initialize the Xerces-C++ runtime. Otherwise, the function initializes and terminates the runtime for you. Note that it is legal to have nested calls to the Xerces-C++ initialize and terminate functions as long as the calls are balanced.

You can instruct serialization functions that initialize and terminate the runtime not to do so by passing the `xml_schema::flags::dont_initialize` flag (see Section 4.3, "Flags").

4.2 Namespace Infomap and Character Encoding

When a document being serialized uses XML namespaces, custom prefix-namespace associations can to be established. If custom prefix-namespace mapping is not provided then generic prefixes (`p1`, `p2`, etc) are automatically assigned to namespaces as needed. Also, if you would like the resulting instance document to contain the `schemaLocation` or `noNamespaceSchemaLocation` attributes, you will need to provide namespace-schema associations. The `xml_schema::namespace_infomap` class is used to capture this information:

```
struct namespace_info
{
  namespace_info ();
  namespace_info (const std::basic_string<C>& name,
                  const std::basic_string<C>& schema);

  std::basic_string<C> name;
  std::basic_string<C> schema;
};

// Map of namespace prefix to namespace_info.
//
struct namespace_infomap: public std::map<std::basic_string<C>,
                                          namespace_info>
{
};
```

Consider the following associations as an example:

```
xml_schema::namespace_infomap map;

map["t"].name = "http://www.codesynthesis.com/test";
map["t"].schema = "test.xsd";
```

This map, if passed to one of the serialization functions, could result in the following XML fragment:

```
<?xml version="1.0" ?>
<t:name xmlns:t="http://www.codesynthesis.com/test"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.codesynthesis.com/test test.xsd">
```

As you can see, the serialization function automatically added namespace mapping for the `xsi` prefix. You can change this by providing your own prefix:

```
xml_schema::namespace_infomap map;

map["xsn"].name = "http://www.w3.org/2001/XMLSchema-instance";

map["t"].name = "http://www.codesynthesis.com/test";
map["t"].schema = "test.xsd";
```

This could result in the following XML fragment:

```
<?xml version="1.0" ?>
<t:name xmlns:t="http://www.codesynthesis.com/test"
        xmlns:xsn="http://www.w3.org/2001/XMLSchema-instance"
        xsn:schemaLocation="http://www.codesynthesis.com/test test.xsd">
```

To specify the location of a schema without a namespace you can use an empty prefix as in the example below:

```
xml_schema::namespace_infomap map;

map[""].schema = "test.xsd";
```

This would result in the following XML fragment:

```
<?xml version="1.0" ?>
<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="test.xsd">
```

To make a particular namespace default you can use an empty prefix, for example:

```
xml_schema::namespace_infomap map;

map[""].name = "http://www.codesynthesis.com/test";
map[""].schema = "test.xsd";
```

This could result in the following XML fragment:

```
<?xml version="1.0" ?>
<name xmlns="http://www.codesynthesis.com/test"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.codesynthesis.com/test test.xsd">
```

Another bit of information that you can pass to the serialization functions is the character encoding method that you would like to use. Common values for this argument are `"US-ASCII"`, `"ISO8859-1"`, `"UTF-8"`, `"UTF-16BE"`, `"UTF-16LE"`, `"UCS-4BE"`, and `"UCS-4LE"`. The default encoding is `"UTF-8"`. For more information on encoding methods see the "Character Encoding" article from Wikipedia.

4.3 Flags

Serialization flags are the last argument of every serialization function. They allow you to fine-tune the process of serialization. The flags argument is optional.

The following flags are recognized by the serialization functions:

`xml_schema::flags::dont_initialize`
: Do not initialize the Xerces-C++ runtime.

`xml_schema::flags::dont_pretty_print`
: Do not add extra spaces or new lines that make the resulting XML slightly bigger but easier to read.

`xml_schema::flags::no_xml_declaration`
: Do not write XML declaration (<?xml ... ?>).

You can pass several flags by combining them using the bit-wise OR operator. For example:

```
std::auto_ptr<type> r = ...
std::ofstream ofs ("test.xml");
xml_schema::namespace_infomap map;
name (ofs,
      *r,
      map,
      "UTF-8",
      xml_schema::flags::no_xml_declaration |
      xml_schema::flags::dont_pretty_print);
```

For more information on the Xerces-C++ runtime initialization refer to Section 4.1, "Initializing the Xerces-C++ Runtime".

4.4 Error Handling

As with the parsing functions (see Section 3.3, "Error Handling"), to better understand error handling and reporting strategies employed by the serialization functions, it is useful to know that the transformation of a statically-typed tree to an XML instance document happens in two stages. The first stage, performed by the generated code, consist of building a DOM instance from the statically-typed tree . For short, we will call this stage the Tree-DOM stage. The second stage, performed by Xerces-C++, consists of serializing the DOM instance into the XML document. We will call this stage the DOM-XML stage.

All serialization functions except the two that serialize into a DOM instance come in overloaded triples. The first function in such a triple reports error conditions exclusively by throwing exceptions. It accumulates all the serialization errors of the DOM-XML stage and throws them in a single instance of the `xml_schema::serialization` exception (described below). The second and the third functions in the triple use callback interfaces to report serialization errors

and warnings. The two callback interfaces are `xml_schema::error_handler` and `xercesc::DOMErrorHandler`. The `xml_schema::error_handler` interface is described in Section 3.3, "Error Handling". For more information on the `xercesc::DOMErrorHandler` interface refer to the Xerces-C++ documentation.

The Tree-DOM stage reports error conditions exclusively by throwing exceptions. Individual exceptions thrown by the serialization functions are described in the following sub-sections.

4.4.1 `xml_schema::serialization`

```
struct serialization: virtual exception
{
  serialization ();
  serialization (const diagnostics&);

  const diagnostics&
  diagnostics () const;

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

The `xml_schema::diagnostics` class is described in Section 3.3.1, "`xml_schema::parsing`". The `xml_schema::serialization` exception is thrown if there were serialization errors reported during the DOM-XML stage. If no callback interface was provided to the serialization function, the exception contains a list of errors and warnings accessible using the `diagnostics` function.

4.4.2 `xml_schema::unexpected_element`

The `xml_schema::unexpected_element` exception is described in Section 3.3.3, "`xml_schema::unexpected_element`". It is thrown by the serialization functions during the Tree-DOM stage if the root element name of the provided DOM instance does not match with the name of the element this serialization function is for.

4.4.3 `xml_schema::no_type_info`

The `xml_schema::no_type_info` exception is described in Section 3.3.7, "`xml_schema::no_type_info`". It is thrown by the serialization functions during the Tree-DOM stage when there is no type information associated with a dynamic type of an element. Usually, catching this exception means that you haven't linked the code generated from the schema defining the type in question with your application or this schema has been compiled without the `--generate-polymorphic` option.

4.5 Serializing to `std::ostream`

In order to serialize to `std::ostream` you will need an object model, an output stream and, optionally, a namespace infomap. For instance:

```
// Obtain the object model.
//
std::auto_ptr<type> r = ...

// Prepare namespace mapping and schema location information.
//
xml_schema::namespace_infomap map;

map["t"].name = "http://www.codesynthesis.com/test";
map["t"].schema = "test.xsd";

// Write it out.
//
name (std::cout, *r, map);
```

Note that the output stream is treated as a binary stream. This becomes important when you use a character encoding that is wider than 8-bit `char`, for instance UTF-16 or UCS-4. For example, things will most likely break if you try to serialize to `std::ostringstream` with UTF-16 or UCS-4 as an encoding. This is due to the special value, `'\0'`, that will most likely occur as part of such serialization and it won't have the special meaning assumed by `std::ostringstream`.

4.6 Serializing to `xercesc::XMLFormatTarget`

Serializing to an `xercesc::XMLFormatTarget` instance is similar the `std::ostream` case. For instance:

```
using std::auto_ptr;

// Obtain the object model.
//
auto_ptr<type> r = ...

// Prepare namespace mapping and schema location information.
//
xml_schema::namespace_infomap map;

map["t"].name = "http://www.codesynthesis.com/test";
map["t"].schema = "test.xsd";

using namespace xercesc;

XMLPlatformUtils::Initialize ();
```

```
{
  // Choose a target.
  //
  auto_ptr<XMLFormatTarget> ft;

  if (argc != 2)
  {
    ft = auto_ptr<XMLFormatTarget> (new StdOutFormatTarget ());
  }
  else
  {
    ft = auto_ptr<XMLFormatTarget> (
      new LocalFileFormatTarget (argv[1]));
  }

  // Write it out.
  //
  name (*ft, *r, map);
}

XMLPlatformUtils::Terminate ();
```

Note that we had to initialize the Xerces-C++ runtime before we could call this serialization function.

4.7 Serializing to DOM

The mapping provides two overloaded functions that implement serialization to a DOM instance. The first creates a DOM instance for you and the second serializes to an existing DOM instance. While serializing to a new DOM instance is similar to serializing to `std::ostream` or `xercesc::XMLFormatTarget`, serializing to an existing DOM instance requires quite a bit of work from your side. You will need to set all the custom namespace mapping attributes as well as the `schemaLocation` and/or `noNamespaceSchemaLocation` attributes. The following listing should give you an idea about what needs to be done:

```
// Obtain the object model.
//
std::auto_ptr<type> r = ...

using namespace xercesc;

XMLPlatformUtils::Initialize ();

{
  // Create a DOM instance. Set custom namespace mapping and schema
  // location attributes.
  //
  DOMDocument& doc = ...
```

```
  // Serialize to DOM.
  //
  name (doc, *r);

  // Serialize the DOM document to XML.
  //
  ...
}

XMLPlatformUtils::Terminate ();
```

For more information on how to create and serialize a DOM instance refer to the Xerces-C++ documentation. In addition, the C++/Tree Mapping FAQ shows how to implement these operations using the XSD runtime utilities.

5 Additional Functionality

The C++/Tree mapping provides a number of optional features that can be useful in certain situations. They are described in the following sections.

5.1 DOM Association

Normally, after parsing is complete, the DOM document which was used to extract the data is discarded. However, the parsing functions can be instructed to preserve the DOM document and create an association between the DOM nodes and object model nodes. When there is an association between the DOM and object model nodes, you can obtain the corresponding DOM element or attribute node from an object model node as well as perform the reverse transition: obtain the corresponding object model from a DOM element or attribute node.

Maintaining DOM association is normally useful when the application needs access to XML constructs that are not preserved in the object model, for example, XML comments. Another useful aspect of DOM association is the ability of the application to navigate the document tree using the generic DOM interface (for example, with the help of an XPath processor) and then move back to the statically-typed object model. Note also that while you can change the underlying DOM document, these changes are not reflected in the object model and will be ignored during serialization. If you need to not only access but also modify some aspects of XML that are not preserved in the object model, then type customization with custom parsing constructors and serialization operators should be used instead.

To request DOM association you will need to pass the `xml_schema::flags::keep_dom` flag to one of the parsing functions (see Section 3.2, "Flags and Properties" for more information). In this case the DOM document is retained and will be released when the object model is deleted. Note that since DOM nodes "out-live" the parsing function call, you need to initialize the

Xerces-C++ runtime before calling one of the parsing functions with the `keep_dom` flag and terminate it after the object model is destroyed (see Section 3.1, "Initializing the Xerces-C++ Runtime").

If the `keep_dom` flag is passed as the second argument to the copy constructor and the copy being made is of a complete tree, then the DOM association is also maintained in the copy by cloning the underlying DOM document and reestablishing the associations. For example:

```
using namespace xercesc;

XMLPlatformUtils::Initialize ();

{
  // Parse XML to object model.
  //
  std::auto_ptr<type> r (root (
    "root.xml",
     xml_schema::flags::keep_dom |
     xml_schema::flags::dont_initialize));

   // Copy without DOM association.
   //
   type copy1 (*r);

   // Copy with DOM association.
   //
   type copy2 (*r, xml_schema::flags::keep_dom);
}

XMLPlatformUtils::Terminate ();
```

To obtain the corresponding DOM node from an object model node you will need to call the `_node` accessor function which returns a pointer to `DOMNode`. You can then query this DOM node's type and cast it to either `DOMAttr*` or `DOMElement*`. To obtain the corresponding object model node from a DOM node, the DOM user data API is used. The `xml_schema::dom::tree_node_key` variable contains the key for object model nodes. The following schema and code fragment show how to navigate from DOM to object model nodes and in the opposite direction:

```
<complexType name="object">
  <sequence>
    <element name="a" type="string"/>
  </sequence>
</complexType>

<element name="root" type="object"/>
```

```
using namespace xercesc;

XMLPlatformUtils::Initialize ();

{
  // Parse XML to object model.
  //
  std::auto_ptr<type> r (root (
    "root.xml",
     xml_schema::flags::keep_dom |
     xml_schema::flags::dont_initialize));

  DOMNode* n = r->_node ();
  assert (n->getNodeType () == DOMNode::ELEMENT_NODE);
  DOMElement* re = static_cast<DOMElement*> (n);

  // Get the 'a' element. Note that it is not necessarily the
  // first child node of 'root' since there could be whitespace
  // nodes before it.
  //
  DOMElement* ae;

  for (n = re->getFirstChild (); n != 0; n = n->getNextSibling ())
  {
    if (n->getNodeType () == DOMNode::ELEMENT_NODE)
    {
      ae = static_cast<DOMElement*> (n);
      break;
    }
  }

  // Get from the 'a' DOM element to xml_schema::string object model
  // node.
  //
  xml_schema::type& t (
    *reinterpret_cast<xml_schema::type*> (
       ae->getUserData (xml_schema::dom::tree_node_key)));

  xml_schema::string& a (dynamic_cast<xml_schema::string&> (t));
}

XMLPlatformUtils::Terminate ();
```

The 'mixed' example which can be found in the XSD distribution shows how to handle the mixed content using DOM association.

5.2 Binary Serialization

Besides reading from and writing to XML, the C++/Tree 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 serialization saves only the data without any meta information or markup. As a result, saving to and loading from a binary representation can be an order of magnitude faster than 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 binary serialization 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 operators and extraction constructors 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 Compiler Command Line Manual for more information.

Once the insertion operators and extraction constructors are generated, you can use the `xml_schema::istream` and `xml_schema::ostream` wrapper stream templates to save the object model to and load it from a specific format. The following code fragment shows how to do this using ACE (Adaptive Communication Environment) CDR streams as an example:

```
<complexType name="object">
  <sequence>
    <element name="a" type="string"/>
    <element name="b" type="int"/>
  </sequence>
</complexType>

<element name="root" type="object"/>

// Parse XML to object model.
//
std::auto_ptr<type> r (root ("root.xml"));

// Save to a CDR stream.
//
ACE_OutputCDR ace_ocdr;
xml_schema::ostream<ACE_OutputCDR> ocdr (ace_ocdr);

ocdr << *r;

// Load from a CDR stream.
```

```
//
ACE_InputCDR ace_icdr (buf, size);
xml_schema::istream<ACE_InputCDR> icdr (ace_icdr);

std::auto_ptr<object> copy (new object (icdr));

// Serialize to XML.
//
root (std::cout, *copy);
```

The XSD distribution contains a number of examples that show how to save the object model to and load it from CDR, XDR, and a custom format.

Appendix A — Default and Fixed Values

The following table summarizes the effect of default and fixed values (specified with the `default` and `fixed` attributes, respectively) on attribute and element values. The `default` and `fixed` attributes are mutually exclusive. It is also worthwhile to note that the fixed value semantics is a superset of the default value semantics.

<table>
<tr><th></th><th></th><th colspan="2">default</th><th colspan="2">fixed</th></tr>
<tr><td rowspan="4">element</td><td rowspan="2">not present</td><td>optional</td><td>required</td><td>optional</td><td>required</td></tr>
<tr><td>not present</td><td>invalid instance</td><td>not present</td><td>invalid instance</td></tr>
<tr><td>empty</td><td colspan="2">default value is used</td><td colspan="2">fixed value is used</td></tr>
<tr><td>value</td><td colspan="2">value is used</td><td colspan="2">value is used provided it’s the same as fixed</td></tr>
<tr><td rowspan="4">attribute</td><td rowspan="2">not present</td><td>optional</td><td>required</td><td>optional</td><td>required</td></tr>
<tr><td>default value is used</td><td>invalid schema</td><td>fixed value is used</td><td>invalid instance</td></tr>
<tr><td>empty</td><td colspan="2">empty value is used</td><td colspan="2">empty value is used provided it’s the same as fixed</td></tr>
<tr><td>value</td><td colspan="2">value is used</td><td colspan="2">value is used provided it’s the same as fixed</td></tr>
</table>