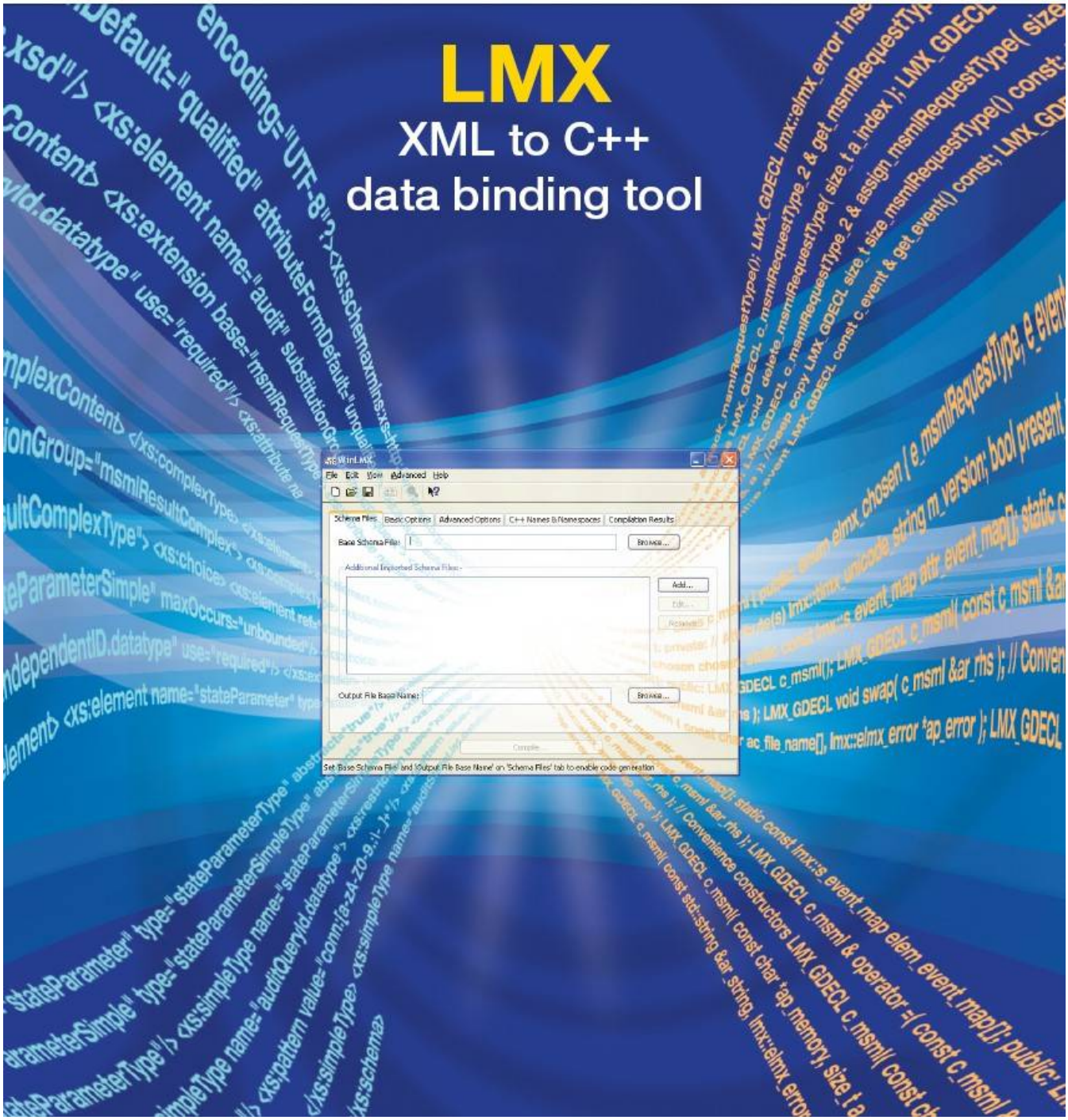
Codalogic

LMX

XML to C++ data binding tool



Version 7.7
Document Revision 1

Table of Contents

Table of Contents

Table of Contents

Introduction

This is the main documentation for the LMX™ XML Schema to C++ code generator. It is available as a single HTML file for easy on-line viewing and viewing on Linux, as a Windows® HTML Help file (.chm) for viewing on Windows, and as a PDF file for printing. A printed version of this manual can also be purchased.

If the LMX code generator seems like the tool you need (see What is LMX? to help you with that), the first thing to do is download and install a copy. You can evaluate a limited functionality version of LMX without a license. Having decided that LMX is what you need, you need to acquire a license file and store it on your PC. You will then be able to generate code by following the Quick Start instructions. Later you may wish to adapt LMX to better meet your requirements. Information on how to do this is covered in section 3 - In More Depth.

Quick Index

Quick links to popular topics:

1 - Orientation

This section will give you an overview of what LMX does, and describes how to get ready to use LMX, including how to install and license LMX.

1.1 - What is LMX?

LMX is a W3C XML Schema to C++ code generator. It takes a W3C XML Schema and generates C++ code that allows you to read and write XML conforming to the Schema by interacting with the C++ classes/objects generated by the tool.

It is primarily intended to be used in data oriented applications, but can also be used in document oriented applications.

The code generator runs on a Windows or Linux PC, but the output is generated in C++ source code that is portable to most platforms that support C++ templates and basic STL containers. (Contact us if the generated code does not run on your platform.)

The code generator comes with its own lightweight XML pull parser and uses C++ ostreams for output. Thus it is a complete XML solution that does not require additional components in order to interact with XML data.

1.2 - Benefits of Using LMX

- Quick and easy to manipulate XML data.
- Knowledge of XML and XML Schema not required - You can simply use the documentation in the generated header file.
- The names of variables are checked at compile time - Simple typos in names are detected automatically and early.
- No programming of tedious state code as required with SAX (and StAX) - LMX does all that in a couple of seconds (or less!)
- Beneficial for both small and large schema.
- Easy to implement web services.
- Can read and generate XML fragments.
- Simple to use - A minimal number of source files so that you don't need to spend time managing code.
- Data types can be customized using C++ language syntax - No need to learn complex code generator configuration options, and less chance of introducing accidental differences between builds by using the wrong configuration. (See table below.)
- Support for SOAP 'mustUnderstand' attribute detection.
- Easy to switch between wide character strings and 'narrow' character strings.
- Exception safe design.
- Portable code - Run on different platforms. Even build and test on one platform and run on another.

1.3 - Download & Installation

You can download the LMX code generator via http://www.codalogic.com/lmx/download.php#licensed.

To install on Windows simply run the installation program in the usual way.

On Linux, after extracting the file from the tarball you can either manually copy the files to appropriate locations or use the `install.pl` script to install LMX. The latter can be used in interactive and non-interactive mode. For an interactive install, simply execute the script in a terminal window, for example:

```
./install.pl
```

To enable a non-interactive install you can save the settings you want to use in a configuration file, an example of which is:

```
# This is a comment
bin=/usr/local/bin
include=/usr/local/include/lmx-$version
lib32_static=/usr/local/lib/lmx-$version
lib32_shared=/usr/lib
lib64_static=/usr/local/lib64/lmx-$version
lib64_shared=/usr/lib64
doc=~/lmx/lmx-$version/doc
src=~/lmx/lmx-$version/src
examples=
license_ok=yes
```

Each specification has the form:

<category>=<install path>

There should be no leading white-space on a line. Comments may be preceded by a # character. Blank lines and lines that contain all white-space are ignored.

Any occurrence of the string `$version` will be replaced with the value of the LMX version you are installing. If an installation variable is set to the empty string (such as `examples` shown above) then that category of items will not be installed. The script will ask you for any values that are not specified.

If the configuration settings are stored in a file such as `my-install-info.txt`, then installation using those settings can be invoked using the command:

```
./install.pl -f my-install-info.txt
```

A simple way to create a configuration file is to do an interactive install and tell the script to save the results. This can be done by doing:

```
./install.pl -o my-install-info.txt
```

You can operate the code generator with limited functionality for evaluation purposes without obtaining a license. For more information on purchasing and licensing see 1.6 - Licensing & Purchase.

1.4 - Evaluating LMX

To help with evaluation, LMX comes with a Windows based Evaluation Support Suite. This is a separate installation packaged as part of the main installation. The Evaluation Support Suite can be installed during the main LMX installation, or it can be installed at a later time by executing the Evaluation Support Suite installation from the LMX folder on the Windows Start menu.

Additional information about the Evaluation Support Suite can be found in the LMX folder on the Windows Start menu.

If you have your own schema and example XML instances, LMX can generate a simple test framework. This consists of an additionally generated C++ file that exercises the main generated code. The test framework code unmarshals an XML instance from a file, does a deep copy to a new set of objects, and then marshals the result out as XML to a new file. You can then visually compare the original and generated files. To use this feature, check the "Generate Test Framework File" box on the "Options" tab of WinLMX (or specify the `-tframework` flag on the command-line versions).

If you would like to see examples of how to use the generated code, then the generated copy constructors (of the form `c_myClass::c_myClass( const c_myClass & rhs )`) provide convenient examples.

A worked example is also available at http://www.codalogic.com/lmx/examples.php to further help with the evaluation process.

1.5 - A Simple Example

We present here a simple example of the sort of code you'll write to use the LMX generated C++ code. While short, the example does illustrate the majority of the aspects of using LMX generated code. We first present the schema used by LMX to generate the code. In this case the generated code is stored in the files `po.h` and `po.cpp`. This is followed by an example XML instance for the schema. Finally, the code used to interface to the LMX generated classes is shown.

We have not included a description of this example, other than the comments that appear in the code. (Full documentation on how to interface with LMX generated code appears in 2.5 - Accesing the Data in the Generated Classes.) You should find however that this example is straightforward and intuitive to understand, demonstrating that it is easy to learn how to use LMX. This makes your code faster to develop and easier to understand.

Example code similar to this can be found in the `examples` sub-directory of the installation or at http://www.codalogic.com/lmx/lmx-example.zip.

The schema:

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

<xsd:annotation>
<xsd:documentation xml:lang="en">
Purchase order schema for Example.com.
Copyright 2000 Example.com. All rights reserved.
</xsd:documentation>
</xsd:annotation>
```

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

<xsd:element name="comment" type="xsd:string"/>

<xsd:complexType name="PurchaseOrderType">
<xsd:sequence>
<xsd:element name="shipTo" type="USAddress"/>
<xsd:element name="billTo" type="USAddress"/>
<xsd:element ref="comment" minOccurs="0"/>
<xsd:element name="items"  type="Items"/>
</xsd:sequence>
<xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name="USAddress">
<xsd:sequence>
<xsd:element name="name"   type="xsd:string"/>
<xsd:element name="street" type="xsd:string"/>
<xsd:element name="city"   type="xsd:string"/>
<xsd:element name="state"  type="xsd:string"/>
<xsd:element name="zip"    type="xsd:decimal"/>
</xsd:sequence>
<xsd:attribute name="country" type="xsd:NMTOKEN"
    fixed="US"/>
</xsd:complexType>

<xsd:complexType name="Items">
<xsd:sequence>
<xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
    <xsd:complexType>
    <xsd:sequence>
    <xsd:element name="productName" type="xsd:string"/>
    <xsd:element name="quantity">
    <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
        <xsd:maxExclusive value="100"/>
        </xsd:restriction>
    </xsd:simpleType>
    </xsd:element>
    <xsd:element name="USPrice"  type="xsd:decimal"/>
    <xsd:element ref="comment"   minOccurs="0"/>
    <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="partNum" type="SKU" use="required"/>
    </xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>

<!-- Stock Keeping Unit, a code for identifying products -->
<xsd:simpleType name="SKU">
<xsd:restriction base="xsd:string">
<xsd:pattern value="\d{3}-[A-Z]{2}"/>
</xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```

An XML Instance:

```
<?xml version='1.0'?>
<purchaseOrder orderDate='1999-10-20'>
```

```
    <shipTo country='US'>
        <name>Alice Smith</name>
        <street>123 Maple Street</street>
        <city>Mill Valley</city>
        <state>CA</state>
        <zip>90952</zip>
    </shipTo>
    <billTo country='US'>
        <name>Robert Smith</name>
        <street>8 Oak Avenue</street>
        <city>Old Town</city>
        <state>PA</state>
        <zip>95819</zip>
    </billTo>
    <comment>Hurry, my lawn is going wild!</comment>
    <items>
        <item partNum='872-AA'>
            <productName>Lawnmower</productName>
            <quantity>1</quantity>
            <USPrice>148.95</USPrice>
            <comment>Confirm this is electric</comment>
        </item>
        <item partNum='926-AA'>
            <productName>Baby Monitor</productName>
            <quantity>1</quantity>
            <USPrice>39.98</USPrice>
            <shipDate>1999-05-21</shipDate>
        </item>
    </items>
</purchaseOrder>
```

And the example code to read the instance:

```
#include <iostream>      // For std::cout etc.
#include "po.h"          // Generated by LMX
#include "lmxparse.h"    // For the LMX parser

int main()
{
    // Allocate a place to store the an error code returned by
    // the unmarshaling operation.
    lmx::elmx_error l_error;

    // c_root is the class generated by LMX that will store the
    // unmarshalled XML.  This constructor unmarshals the data
    // stored in the file "po.xml" and places any error code in
    // l_error.
    c_root l_po( "po.xml", &l_error );

    if( l_error != lmx::ELMX_OK )
    {
        std::cout << "An error occurred while reading XML\n";
        return 0;
    }

    // Or we could have done:
    // c_root l_po;
    // lmx::elmx_error l_error = l_po.unmarshal( "po.xml" );
    // if( l_error ...

    // Interrogate the parsed XML
    //---------------------------
```

```
if( l_po.getchosen() == c_root::e_purchaseOrder )
{
    // Make some intermediate variables so that components are
    // easier to reference
    const c_PurchaseOrderType *lp_pot = &l_po.get_purchaseOrder();

    if( lp_pot->isset_orderDate() )    // If the orderDate is set...
    {
        std::cout << "Ordered on: " <<
                lp_pot->get_orderDate() << "\n";

        if( lp_pot->get_orderDate().get_year() <= 2000 )
            std::cout <<
                "This was ordered last century - It is now overdue!\n";
    }

    // Reference another part of the data structure
    const c_Items *lp_items = &lp_pot->get_items();

    float l_total_cost = 0;

    // Print the total number of items
    std::cout << "Number of items = " <<
            lp_items->size_item() << "\n";

    // Print out some information about each order item
    for( size_t l_i=0; l_i<lp_items->size_item(); ++l_i )
    {
        std::cout << "Item " <<
            (l_i + 1) << ": " <<
            lp_items->get_item( l_i ).get_quantity() <<
            " off - " <<
            lmx::as_ascii( lp_items->get_item( l_i ).get_productName() ) <<
            " (" <<
            lmx::as_ascii( lp_items->get_item( l_i ).get_partNum() ) << ")";

        // The comment is optional, so only print it out if it is present
        if( lp_items->get_item( l_i ).isset_comment() )
            std::cout << " [" <<
            lmx::as_ascii( lp_items->get_item( l_i ).get_comment() ) << "]";

        std::cout << " - $" <<
            lp_items->get_item( l_i ).get_USPrice() << "\n";

        l_total_cost +=
            lp_items->get_item( l_i ).get_USPrice().get_scaled( 2 );
    }

    std::cout << "Total Cost: $" << l_total_cost/100 << "\n";

    std::cout << "Ship to: " <<
            lmx::as_ascii( lp_pot->get_shipTo().get_name() ) << "\n";
    std::cout << "Bill to: " <<
            lmx::as_ascii( lp_pot->get_billTo().get_name() ) << "\n";
    std::cout << "\n";


    // Modify the XML
    //---------------
    c_root l_alt_po( l_po );    // We don't actually need to create
                                // a new instance before modifying

    l_alt_po.get_purchaseOrder().get_items().append_item();
```

```
        c_Items::c_item *lp_item =
                &l_alt_po.get_purchaseOrder().get_items().back_item();

        lp_item->set_productName( L"Fence" );
        lp_item->set_quantity( 2 );
        lp_item->set_partNum( L"100-AB" );
        lp_item->set_USPrice( lmx::c_decimal( 12.95, 2U ) );

        // We can also do:
        // lp_item->set_USPrice( 12.95 );  // Although the schema needs to
        //                                 // specify the fraction digits
        //                                 // facet in order to get the number
        //                                 // of decimal places correct!
        // lp_item->set_USPrice( lmx::c_decimal( 1295, 2 ) );
        //                                 // e.g. 1295 / (10^2)
        // lp_item->set_USPrice( "12.95" );

        lp_item->set_comment( L"Will this stop the baby getting on the lawn?" );

        assert( lp_item->is_occurs_ok() );  // Check sufficient elements and
                                            // attributes added


        // Write the modified version of the XML to the file po-out.xml
        //-------------------------------------------------------------
        if( l_alt_po.marshal( "po-out.xml" ) == lmx::ELMX_OK )
            std::cout << "Modified XML written successfully\n";
        else
            std::cout << "Error writing Modified XML\n";
    }

    return 0;
}
```

1.6 - Licensing & Purchase

For the purposes of evaluation, the LMX code generator can be used in a restricted mode without a license. In the restricted mode, the code generator limits the number of XML objects that it will compile.

To fully use the product a license file must be acquired. This will be e-mailed to you when you purchase a license.

To purchase a license, follow the links and instructions for purchasing at: http://www.codalogic.com/lmx/

When using a Windows version of LMX the license file can be placed in the same folder as the `lmx.exe`/`winlmx.exe` executable (by default, `C:\Program Files\LMX` on 32-bit systems and `C:\Program Files (x86)\LMX` on 64-bit systems), or in the folder specified by the `HKEY_CURRENT_USER\Software\Codalogic\LMX\dir` registry key.

On Linux (and other Unix versions) you can either place the license file in one of the following directories; `$HOME/.linmx`, `$HOME/.settings/codalogic/linmx`, `$HOME/.config/codalogic/linmx`, or `$HOME/.codalogic/linmx`; or in an alternate directory specified by setting and exporting the `LMXDIR` shell environment variable with the name of the directory containing the license file (possibly in your login script such as .bash_profile). For example:-

```
LMXDIR=$HOME/lmx
export LMXDIR
```

Note that to avoid problems with email firewalls, the namespace URI in license files has been changed from `http://codalogic.com/lmx/license` to `com.codalogic.lmx.license`. If you have bought a new license to work with old versions of LMX you will need to change the namespace URI in the license file back to `http://codalogic.com/lmx/license` for the license file to be recognised.

1.7 - Version History

7.7

- Support nillable complexTypes with empty content.
- Added `-ignore-mixed-attr` to ignore `mixed="true"` attributes in schemas.
- `minLength=0` ignored to avoid generating warning on GCC
- Various `throw()` specifications replaced with `LMX_NOEXCEPT`.
- Specialisations of `lmx::uniq_ptr<char>` added to avoid GCC address sanity warnings.

7.6

- Added mitigation for the XML Billion Laughs attack. See 3.31 - XML Billion Laughs Mitigation for more information.
- Added `-enum-to-string` flag along with `-method-chosen_to_string` and `-method-all_to_string` flags to allow optional conversion of `xs:choice` and `xs:all` enum values to string values.
- Added the `-alt-lmxuser YYY` flag to allow inclusion of an alternative version of the `lmxuser.h` file. See 3.1 - Configuration using lmxuser.h.
- Added `c_xml_reader_c_string` class to support easy unmarshalling from C-Style `char []` style arrays.
- Added `-enum-options-above` flag to enable control of where documentation of enum options is generated in relation to the methods that use the enums.
- Changed the namespace uri of license files from `http://codalogic.com/lmx/license` to `com.codalogic.lmx.license` to address problems encountered with email firewalls.
- Updated regular expression code to derive from libxml2-2.9.13.
- `linmx` and `glinmx` are now compiled as 64-bit applications. (Windows versions remain as 32-bit application.)

7.5

- Added LMX_USE_64BIT_INTEGER_FALLBACK conditional define to lmxuser.h to permit selecting fallback handling of 64-bit integers.
- Added `-no-choice-enum-vals` flag to disable assigning numerical values to each enum when defining choice enums.
- Added support for `on_lmx_marshal_start()` and `on_lmx_marshal_end()` snippet event handlers to global elements that use a type attribute.
- Added LMX_WRITER_DEFAULT_CONVENIENCE_NO_XML_DECL #define as an easier way to disable XML Decl output when using convenience methods.
- Added `no_xml_decl_with_convenience_methods()` to disable output of XML Decl when marshalling using generated convenience methods. See 3.21 - Adding an XMLDecl to the XML

output.
- Added generation of C++11 move constructor and move assignment operator methods.
- Added XSD1.1 style support for `xs:any` in `xs:all` constructs. Other XSD1.0 limitations of `xs:all` currently remain.
- Documented the `-autover2` flag.
- Modified parser code to remove Level 4 Warning generated by Visual Studio.
- Modified default settings of Visual Studio declspecs. Some users may need to adjust their build files.

7.4

- Tested on VS2019 and g++ 9.1.0.
- Support g++ -Wextra and -pedantic flags.
- Added `on_lmx_copy_construct()`, `on_lmx_swap()` and `on_lmx_eq()` snippet event handlers. See 3.14.1 - Augmenting Generated Classes with Snippet Event Handlers.
- Added `c_xml_reader_string` and `c_xml_reader_any_info` classes.
- Added `c_xml_reader::add_namespace_mappings( const c_any_info & )` and `c_xml_reader::add_namespace_mappings( const c_namespace_context & )`.
- Added `c_xml_reader::is_xml_end()` method. See 3.2.3 - Checking for trailing non-XML material.
- Updated regular expression code.

7.3

- Added support for C++11 range-based for loops, allowing code of the form: `for( auto & i : top.in_NAME() )`. See 2.5.5 - Multiple Simple Type and List Simple Type C++ Interface and similar sections for more details. Also added `-prefix-in` flag.
- Added `on_lmx_alt_unmarshal_xs_any()` snippet event handler. See 3.14.1 - Augmenting Generated Classes with Snippet Event Handlers.
- Various enums are given explicit values to make debugging easier.
- Introduced LMX_FLOAT_PRECISION, LMX_DOUBLE_PRECISION and LMX_FLOAT_HIGH_PRECISION defines to allow user to select precision when converting floating point values to text representation.
- Added c_xml_reader::extract_namespace_context() method.
- Modified relationship of operator<) and output_convert_to_xml(), so that providing specialisations of operator<) can act as a customisation point.
- Added `-#define` flag.
- Added `-lmxuser-defs` and `-lmxuser-defs-end` flags.

7.2

- Added testing on VS 2015, VS 2017, g++ 5.4.0, g++ 6.3.0 and g++ 7.1.0
- Added `c_xml_writer::set_convenience_options()` and `c_xml_writer::get_convenience_options()` to enable ability to configure whether XMLDecl and 'standalone' are output when marshalling using the convenience methods. See 3.21 - Adding an XMLDecl to the XML output.
- Corrected default whitespace handling of xs:token and xs:normalizedString and their derived types.
- On a list type, calling clear_NAME() now marks a list as present.
- Added `-fname-sep` flag to specify separator character sequence for constructed file names.
- Added better support for using command-line wildcards to specify input XSD files.
- Added `-method-getid` and `-method-has_id` flags.

- Implemented `on_lmx_unmarshal_outer_start()` and `on_lmx_unmarshal_outer_end()` snippet event hanlders.
- Added operator << (std::ostream &, const c_binary &) to facilitate output of raw binary data.
- Enhanced `s_debug_error` to allow customized descriptions for LMX error codes allocated for user defined purposes. See 3.16.4 - Collecting Debug Error Information when using Convenience Methods.

7.1

- Virtual methods added to `c_xml_reader` to allow conditional acceptance / rejection of unknown attributes and elements in auto-versioning modes. See 3.27 - Allowing Unknown Items in an XML Instance for more details.
- Modifications to allow for C++11 deprecation of `std::auto_ptr`.
- Added methods `c_binary::get_as_hex_string()` and `c_binary::get_as_base64_string()`.
- Enhanced support for non-C locales for converting strings to `xs:float` values.
- Added `DECIMAL_INIT_MODE` section to `lmxuser.h` to allow choice between initialising `xs:decimal` values using C++ strings or doubles.
- Added `lmx_assert_is_occurs_ok()` macro to allow user control of what happens if insufficient items are present for marshalling.
- Fixed application of pattern facets to a whole `xs:list` construct as opposed to its individual members.
- Use of `ptrdiff_t` fixed to `std::ptrdiff_t`.

7

- To accommodate the increasing number of versions of Visual Studio and GCC we have decided to only ship the source code for the Runtime Supporting Software and allow users to create their own binaries for this code. This allows greater user flexibility and complete cross-platform solutions.
- Added ability to install external character transcoders. See 3.17 - Adding External Character Set Transcoders.
- Added the `-expose-storage` option. This allows writing of more generic code.
- Improved handling of serialisation of floating point numbers on non-Windows platforms.

6.4

- Improved formatting of fractional seconds values in times and durations.
- is_occurs_ok() method can now collect better debugging information.
- Added generation of the check() method, effectively a deep version of the is_occurs_ok() method.
- Added `-no-check-code` flag.

6.3

- The Windows library naming convention has been changed from including the compiler version in the name (e.g. vc11) to the IDE version (e.g. vs2012).
- Fixed HTML documentation of attribute cardinality.
- Reduced rounding errors when round-tripping float values through c_decimal.
- c_binary::operator [] const and at() const now return const unsigned char & to allow the return value to be used in functions like fwrite() etc.
- Changed the internal variable naming convention.

- Classes that contain binary members now have added a non-const method for accessing the binary member.

6.2

- Added the `-enums-per-type` flag option to generate separate C++ enums for each schema type that has xs:enumeration facets defined.
- Added `-narrow-strings` flag to generate `#defines` suitable for instructing C++ compiler to use narrow strings for Unicode strings.
- Added `-prefix-enumeration` and `-prefix-enumeration-name` to improve control of naming of c++ enumerations associated with `xs:enumeration` facets.
- Added generation of const version of getany_attributes() method.
- Added facility to change whether seconds in types containing time are represented using C++ float or double.
- Includes binaries for Microsoft VS 2012.

6.1

- The generated operator == () methods now take into account polymorphism of types.
- The generated code has been modified so that try/catch blocks are not required. Exception safety is maintained.
- Added the `-prefix-declash` flag. See 3.22.1 - Handling Multiple Independent Schemas for more details.
- Fixed Micro Format handling (see 3.26 - Specifying Micro Formats) when schema specifies a type to be an `xs:list` type.
- If a project build environment defines `LMX_XML_ALWAYS_ESCAPE_GT` to be 1 when the `lmxtypes.cpp` source code is compiled then > will always be escaped to > (Professional Edition required).

6

- Refactored the interface between the generated objects and the XML parser/writer.
- Restored the ability to allow globally stored objects that have string-based default values.
- Improved conditional automatic inclusion of xml.xsd namespace attributes.

5.7

- Improved support for choice of CamelCase or underscore_separated C++ names. See `-naming` flag.

5.6

- Convenience methods now take an optional pointer to an error object to allow capture of enhanced error information. See 2.3 - Unmarshaling (simple form) for more information.
- Enhancements have been made to enable better control of attribute layout when formatting. See 3.25 - XML Output Format Customization for more information.
- Some concrete implementations of key template methods have been placed in the binary libraries to reduce the size of binaries on some platforms.

5.5

- Enhancements to the Annotated XML Example (AXE) parser.

- Added `-release-switch` flag.
- Added `on_lmx_is_occurs_ok()` snippet event handler. (See 3.14.1 - Augmenting Generated Classes with Snippet Event Handlers.)
- Added generation of `assign_XYZ( size_t index, T * p )` methods for complexTypes.
- Added `c_xml_reader::ignore_element(...)` method mainly to be used with the `on_lmx_alt_unmarshal_element_XYZ()` snippet event handler to enable ignoring the contents of elements the user is not interested in.

5.4

- Added support for specifying the XML data format using Annotated XML Example (AXE™) format.
- Better support when using empty prefixes for C++ identifiers.

5.3

- Global setting of the XML output formatting is supported. (See 3.25 - XML Output Format Customization.)
- Added `on_lmx_alt_unmarshal_attribute_XYZ()`, `on_lmx_alt_unmarshal_element_XYZ()`, `on_lmx_alt_marshal_attribute_XYZ()` and `on_lmx_alt_marshal_element_XYZ()` snippet event handlers. (See 3.14.1 - Augmenting Generated Classes with Snippet Event Handlers.)
- Added the `c_xml_writer::disable_ns_map_output()` method to disable output of the `xmlns` namespace attributes. See 3.20 - Disabling Output of XML Namespaces for more.
- Added additional variants of the `unmarshal_partial(...)` template method. See 3.2.4 - Unmarshalling Schema Fragments.
- Documented the `unmarshal_find()` template functions. See 3.3 - Unmarshalling a Sub-element within an XML Instance.
- Refactored internal marshalling and unmarshaling interface to `c_xml_writer` and `c_xml_reader`.

5.2

- Added the `-micro-format` flag to allow support of micro formats. See 3.26 - Specifying Micro Formats.
- Added generation of `insert_XYZ( size_t index, T * p )` methods for complexTypes.
- Added generation of non-const `get_XYZ(???)` methods for compound simpleTypes such as xs:date and micro formats to enable easier access.
- Fixed code generation bug for complexTypes containing fixed simpleContent.

5.1

- Added the `-pattern-out` flag to allow specification of custom output converters for data types requiring xs:pattern specific output formatting. See 3.24 - Custom Pattern Facet Output Formatting.
- Changed name of .isset() methods on classes to .is_value_set() to avoid problems when other code defines an isset() macro (e.g. HP aCC).
- Corrected handling of decimal number comparisons that contained 0s as final fractional digits.
- Add binaries for g++ 4.2.3.
- Includes support for Microsoft VS 2010.
- Enhanced Linux install.pl script to allow unattended install.

5

- Re-factored the inner workings of the generated code. The generated API to use the generated code remains the same.
- Some short accessor methods can now be generated inline as part of the class definition rather than in the .cpp file.
- Added `-max-inline` flag.
- Added `-error-fast` flag.
- Added `-no-local-classes` flag.
- Updated regex code derived libxml2 2.7.7.

4.2

- Added improved support for handling and subsequently unmarshalling contents of `xs:any`.
- Added `on_lmx_unmarshal_attribute_XYZ()` and `on_lmx_unmarshal_element_XYZ()` snippet event handlers.
- Added option to specify the base class of a named class. See `-class-base` flag.
- Add option to be able to specify C++ include guard string. See `-include-guard` flag.
- Default C++ include guard text now includes the C++ namespace.
- Modified generated code to accommodate GCC version 4.3.3 and 4.4.1. warnings.
- Added a Perl based install script (install.pl) for Linux platforms.

4.1

- Added option to generate equality (and inequality) operators. See `-eq` flag.
- Added `exec-pre-all`, `exec-post-all` and `exec-post-all-error` flags.
- Added option to create a makefile fragment. See `-makefile` flag.
- Significantly sped up the compile time of some large schemas.

4.0

- GUI interface for Linux added, called glinmx.
- `-expose-containers` flag added, allowing more direct access to the containers (e.g. std::vector) in the objects.
- Changed defaulting of `LMX_NARROW_URI_STRINGS` so that it is defaulted to the value of `LMX_NARROW_UNICODE_STRINGS` rather than 0.

3.10

- Enhancements made so that string classes that have different method names to std::basic_string can be used.
- Supporting Software and generated code changed so that GCC can be run with the -Wunused-parameter flag without causing warnings.
- Wide-string based path names supported in generated convenience methods.

3.9

- Added the `-naming YYY` flag which specifies the naming convention to be used for names in the generated code. If `camel` is specified, LMX will attempt to convert names generated in the code to CamelCase. If `underscore` is specified, LMX will attempt to make names underscore separated.
- Added `-file-ext-snippets YYY` flag.

- Reworked the output converters code so that alternative output converters can be more readily used with the Standard Edition of LMX.
- WinLMX now automatically checks to see if there are any new versions of LMX available.

3.8

- Added `c_soap` and `c_winhttp` classes to aid implementing web services. See 3.11 - Use with Web Services for more information.
- Added the ability for users to customize the behavior of generated classes using snippet event handlers. These call user defined methods during the marshal and unmarshal processes. See 3.14.1 - Augmenting Generated Classes with Snippet Event Handlers for more information.
- Added the `-alt-xml-reader` and `-alt-xml-writer` flags. See the flags section for more information.
- To provide custom error feedback, primarily for snippet event handlers, the `lmx::elmx_error` enumeration has had added to it the enumeration values `ELMX_USER_DEFINED_1` to `ELMX_USER_DEFINED_9`.
- Added `-lmx-include-path` flag to allow specifying the directory where the LMX header files are stored.
- Added the `-check-is-occurs-ok-on-marshal` flag.
- Added `tlmx_uri_string` typedef so that URI string can have an independent type to Unicode strings.
- Added #ifndef LMX_NO_WSTRING sections to the supporting software source code to allow conditional removal of std::wstring from a build.

3.7.1

- Fixed handling of hierarchies of substitution groups.
- Auto-versioning of substitution groups now has an explicit option. Previously auto-versioning of substitution groups was the default (but non-standard) behavior. See `-autover-subst-groups` flag.

3.7

- Added options to specify the file extensions of the generated files. See `-file-ext-cpp` and `-file-ext-h` flags.
- Added option to specify that the C++ enums associated with schema enumeration facets should be local to a class. See `-local-enums` flag.
- Added options to specify that documentation from the schema (included in `xs:documentation` elements) should be output in the .h and/or HTML file. See `-doc-in-h` and `-doc-in-html` flags.
- A flag has been added to cause any warnings generated during code generation to return the same program error code returned by errors. See the `-werror` flag.
- The Windows DLLs now have version information in them.
- A bug associated with determining the used XML namespaces has been fixed.
- Added option to suppress generation of root class. See `-no-root-class` flag.
- Added option to not generate code for insert/delete/clear operations on items that can occur more than once. See `-no-container-ops` flag.

3.6

- Added options to specify suffixes to be added to the code names of attributes, elements, types, and groups. See `-suffix-attribute`, `-suffix-element`, `-suffix-type` and

`-suffix-group` flags.
- Corrected handling of `xmlns=""` idiom according to XML Namespaces 1.1.
- Added static `debug_error` object to aid the debugging process.
- Added one .cpp per schema file generation mode. See `-cpp-per-schema` flag.
- Added ability for developers to augment the generated classes with their own code. See `-snippets` flag.
- More accurate HTML documentation generation.

3.5

- Added options to control whether only marshaling or only unmarshaling code is generated. See flags `-no-marshal` and `-no-unmarshal`.

3.4

- Added polymorphic behavior for XSD extension and restriction types. Use -no-polymorphic to switch off polymorphic behavior.

3.3

- Adopted the Windows XP look-and-feel for WinLMX.
- Corrected some issues with simpleContent types.
- Changed company name.

3.2.5

- Fixed a dependency issue in the generated C++ code between C++ declarations and definitions for attributes in restricted simple content.

3.2.4

- Improved support for QNames.

3.2.3

- Improved support for multiple instances of simple type lists.
- Fixed handling of fractional seconds less than 10.0 in time, datetime and duration types.

3.2

- Further enhancements to WinLMX's method prototype viewing feature made.
- Improvements in the way xs:include files are handled.
- Corrected handling of multiple occurrences of information items with fixed or default value constraints.

3.1

- Further enhancements to WinLMX's method prototype viewing feature made.
- Fixed an XML namespace handling issue.
- Modified c_read_file so that reading XML from files is faster in applications generated with Visual Studio 2005.

- Added the ability to set schema location attributes in generated XML.
- Formerly the validation of the length facets for xs:string types was only correct if they were mapped to std::wstring. This has now been corrected so that the validation is correct irrespective of whether the xs:string types are mapped to std::wstring or std::string.

3.0

- WinLMX enhanced. The base names of variables and methods can now be edited using the interface.
- Editing of a schema can be invoked from WinLMX.
- The generated HTML documentation can be displayed from WinLMX.

3.0 Beta

- XML Parser refactored.
- Additional marshal and unmarshal methods generated to make it more convenient to marshal and unmarshal to/from files and memory.
- Test framework output (-tframework) simplified.
- WinLMX enhanced.
- If multiple schemas are treated as global, each such schema has its own XML namespace URI to namespace prefix map.

2.11

- Nested C++ namespaces can be specified.
- Removed use of global_event_map in 'global' unmarshal functions.
- Added provision for extension from xs:anyType.
- Improved parser efficiency.
- User has control over XML namespace prefixes used via -ns-prefix-map command-line flag.

2.10

- Primarily an upgrade to support the Basic Edition license.
- Includes code for VS2005 libraries.
- Enforce the "C" locale for string to float conversions in locale aware implementations.
- Unicode handling extended beyond the Basic Multi-lingual Plane to full Unicode range (0x-0x10ffff).
- Moved position of generated braces!

2.9

- Improved 'One .h/.cpp file per class' handling.
- insert_NAME() methods generated for interfacing with collections.
- Mapping of namespace prefix flags supported (see -ns-prefix-map).
- Improved mapping of punctuation characters in enumeration identifiers to C++ names.
- Unions are no longer anonymous, leading to improved portability on compilers such as HP-UX aCC.

2.8

- Can specify wide-char file names when opening XML files for reading.
- Added string converters for std::string to std::string and std::wstring to std::wstring. These non-converters make it easier to operate in a environment where _TCHAR type mechanisms are being used.

- Added features to help MS IntelliSense.
- Added work-arounds for IBM Visual Age Compiler.

2.7

- Code can be generated from an XML external DTD definition - see 3.6 - Selecting the Input File Type (XSD, WSDL, DTD).
- A W3C Schema embedded in a WSDL file can be automatically parsed - see 3.6 - Selecting the Input File Type (XSD, WSDL, DTD).
- Modified header file layout so that it is possible to switch between wide and narrow strings without requiring the supporting software source code.
- Fixed a bug in the generated code that did a double memory allocation when calling c_class &get_class(size_t index), causing a memory leak.
- Fixed problem in the generated code when parsing XML instances from schemas that have multiple global elements and have multiple namespaces associated with them.
- Added lmx_assert() macro to allow customisation of how assertions are handled. The macro is currently mapped directly to assert(), but could be modified to allow throwing exceptions in release code if desired.
- Character entities in entity definitions are now expanded immediately when first parsing, rather than when the entity text is used.

2.6

- Modified constructor initialisation order to remove GCC -Wall warnings.
- Added at() and get(unsigned char [], size_t) methods to c_binary.
- Added functions to convert between wide and narrow strings (and vice versa).
- Added some defaults to switch statements to address compiling with GCC -Wall.
- Fixed problem with single quote appearing in attribute values when parsing xs:any elements.
- Fixed problem with comments that contained no space after the <!-- sequence and contained a / character (e.g. <!--My comment with / char -->).

2.5

- Customized naming of methods now supported - see 3.12 - Naming of Methods and Variables.
- Union simpleType as base of simpleContent now supported.
- Improve recording of code generator options in generated code when WinLMX used.
- c_xml_reader destructor made virtual and unused variable removed from c_big_int::operator >.

2.4

- Added -multi-file flag to allow one class per file operation.
- WinLMX allows saving of default project settings for easier, more consistent project start up.
- Fixed issue with -output-defaults.

2.3

- Added -no-nested-classes flag to allow specifying that nested classes should not be used.

1.8 - Supported XML Schema Features

LMX XML C++ Databinder offers industry leading schema feature support for C++ to XML data binding as shown in the following table:

XML Schema Feature	Supported?
Schema Handling	
Import	Yes
Include	Yes
Redefine	Yes
Chameleon Design	Yes
Qualified / Unqualified Elements	Yes
Qualified / Unqualified Attributes	Yes
Types	
Empty Types	Yes
simpleType	Yes
complexType / simpleContent	Yes
complexType / complexContent	Yes
Nillable	Yes
Simple Types & Facets	
All Simple Types Supported	Yes
Lists	Yes
Unions	Partial[1]
enumeration	Yes
Allow reading/writing using enumerated values	Yes
fractionDigits	Yes
length	Yes
maxExclusive	Yes
maxInclusive	Yes
minExclusive	Yes
minInclusive	Yes
maxLength	Yes
minLength	Yes
pattern	Yes
totalDigits	Yes
whiteSpace	Yes
List length facets	Yes
default values	Yes
fixed values	Yes
xs:anyAttribute	Yes
xs:anyAttribute namespace validation	Yes
xs:anyType	Yes

Automatic entity substitution (& etc)	Yes
Expansion of DTD defined entities	Yes
complexType / complexContent	
Sequence	Yes
Choice	Yes
All	Yes
Capture / control order of xs:all	Yes
Anonymous Compositors	Yes
xs:any	Yes
xs:any namespace validation	Yes
Mixed	Partial[2]
Extension	Yes
Restriction	Mostly[3]
Polymorphic Extension using xsi:type	Yes
Polymorphic Restriction using xsi:type	Yes
Groups	Yes
Recursive Definitions	Yes
Cardinality	
Optional	Yes
Mandatory	Yes
Multiple (0-n, 1-n, m-n, m-unbounded etc.)	Yes
Miscellaneous	
Attribute Groups	Yes
Substitution Groups	Yes
Name clash prevention	Yes
UTF-8, UTF-16 (BE & LE), UCS2 (BE & LE), ISO-8859-1	Yes
Generate code from XML external DTDs	Yes
Generate code from Schemas embedded in WSDL files	Yes

Notes:
[1]All XML schema Union simpleTypes are stored as strings.
[2]The contents of an XML schema Mixed type is stored as a string, including any additional markup.
[3]Some corner cases that don't map cleanly to C++, such as an element that has a child element that is restricted by using a type that is a restriction of the original type, are not supported.

2 - Quick Start

Having installed and licensed your LMX product, you will want to generate code. This section gives a brief introduction to this and will probably be sufficient for the majority of your use of the tool. For more detailed information, see section 3 - In More Depth.

2.1 - Code Generation

The LMX code generator can be run on Windows® and Linux platforms. On the Windows platform LMX is available as both a GUI and a command-line version.

The generated code is cross-platform.

2.1.1 - Code Generation Using the Windows Interface

The Windows® Interface version of LMX makes it easy to use the LMX tool. The Windows version is called WinLMX.

When you start WinLMX, you will see a window displayed that contains a tabbed dialog. You can specify what you want WinLMX to do by selecting the various tabs, and completing the dialog items that are displayed.

The main aspect of configuration is selecting the files to be compiled, and the names of the output files. This is configured using the left-most tab labelled 'Schema Files'. Enter the base schema file in the top-most edit box, and the root of the output files (e.g. 'File' to generate 'File.h' and 'File.cpp') in the bottom-most edit box. The files that the schema imports should be entered into the middle list box by pressing the associated 'Add...' button.

LMX can parse both W3C Schema (XSD) and XML (external) DTD files. LMX can also locate and parse W3C Schema definitions embedded in WSDL files. The parsing that LMX performs depends on the file extension of the base schema file. See 3.6 - Selecting the Input File Type (XSD, WSDL, DTD) for more information.

When you have completed configuring WinLMX you can save the configuration using the 'File' menu.

To generate code for a configuration, press the 'Compile...' button at the bottom of the window. This will automatically select the 'Compilation Results' tab and display the results of the compilation.

Note that when a configuration file is loaded into WinLMX, the current working directory is set to the path of the configuration file. This allows you to specify relative file names for the schema and output files.

In addition to using the 'File' menu to open configuration files, WinLMX also supports drag and drop.

The configuration files created by WinLMX can be used by the command-line version of LMX. This allows you to develop configuration files using the windows version, and use that configuration in your build strategy with the command-line version. (If the command-line version of LMX is called with a single argument that ends in '.lmxprj' the tool assumes that it is a configuration file. Alternatively, the '-f' command-line option can

be used to specify a configuration file.)

If WinLMX's default project settings are not suitable for your development environment, configure the settings as you would like them and then select the 'File->Save Default New Project' menu item. These settings will then be used when a new project is started, or a '.xsd' file is dragged over WinLMX.

When using a licensed version LMX needs to know the location of the license file. See 1.6 - Licensing & Purchase for further information.

2.1.2 - Code Generation Using the Windows (DOS) Command Line Version

The LMX code generator can be run from a (DOS) command line prompt. This is useful when LMX is used as part of a batch build process such as a nightly build. The basic command line syntax is:

```
lmx [flags] primary-xsd-file *[++ additional-global-xsd-file] *[+ additional-library-xsd-file] outpu
```

The flags are optional and are described in 3.8 - Command-line Flags.

The `primary-xsd-file` is the name of the file that contains the main schema definition. See 3.6 - Selecting the Input File Type (XSD, WSDL, DTD) for more information on selecting between W3C Schema files, WSDL files and XML external DTD files.

If the primary schema definition references other schemas, these additional schema files are specified on the command line by including the + character, a space, and then the name of the file. Any number of additional schema files can be specified in this way.

If the additional schema files contain global elements that you would like to treat as possible XML instance document elements, then instead of using the + character in front of the schema name as above, use the ++ character sequence. This can be useful when you are generating code for two or more schemas that share common imported schemas.

The last argument specifies the names of the C++ files into which the generated code is to be placed. The LMX code generator will append .cpp to the name specified for the C++ source file, and append .h to the name for the header file.

For example, the command line:

```
lmx my_xsd.xsd my_xsd_code
```

will compile the schema specified in my_xsd.xsd and generate the files my_xsd_code.h and my_xsd_code.cpp.

If my_xsd.xsd refers to other schemas, an example command line would be:

```
lmx my_xsd.xsd + my_xsd_lib_1.xsd + my_xsd_lib_2.xsd my_xsd_code
```

This will combine the schemas specified in `my_xsd.xsd`, `my_xsd_lib_1.xsd` and `my_xsd_lib_2.xsd`, and generate the files `my_xsd_code.h` and `my_xsd_code.cpp`.

If two schemas refer to a common schema, one way to generate code is to use a command line similar to:

```
lmx my_xsd_1.xsd ++ my_xsd_2.xsd + my_xsd_lib.xsd my_xsd_code
```

(N.B. notice the use of ++ rather than just +.) In this case, in addition to allowing global elements in `my_xsd_1.xsd` to be document level elements in an XML instance, global elements in `my_xsd_2.xsd` will also be treated as candidate document level elements.

LMX has support for command-line shells that provide wildcard expansion (LMX itself does not perform wildcard expansion). A command-line of the form below will treat all files matching `my_xsd_*.xsd` as `additional-library-xsd-file`s. Only files not already mentioned on the command-line are treated as `additional-library-xsd-file`s. Therefore, in the example below, `my_xsd_1.xsd` will not be treated as an `additional-library-xsd-file` even though it matches the wildcard, because it is explicitly named earlier on the command-line.

```
lmx my_xsd_1.xsd + my_xsd_*.xsd my_xsd_code
```

When using a licensed version the LMX executable needs to know the location of the license file. See 1.6 - Licensing & Purchase for further information.

2.1.3 - Code Generation Using the Linux Command Line Version

The operation of the Linux command-line version is the same as the Windows (DOS) command-line version as described in 2.1.2 - Code Generation Using the Windows (DOS) Command Line Version, except that the executable is called '`linmx`'. Hence the basic command-line syntax becomes:

```
linmx [flags] primary-xsd-file *[++ additional-global-xsd-file] *[+ additional-library-xsd-fi
```

For example:

```
linmx my_xsd.xsd my_xsd_code
```

When using a licensed version the LMX executable needs to know the location of the license file. See 1.6 - Licensing & Purchase for further information.

2.2 - C++ Compiling and Linking

An LMX project consists of three different types of source code.

- Code generated by the LMX code generator.
- The LMX Runtime Supporting Software. This is code common to all LMX projects and consists of the low level XML parser and type classes. The `.cpp` source files for this code are located in the `supporting-software/src` sub-directory of the installation, and the `.h` header files are located in the `supporting-software/include` sub-directory.
- The code written by you that interfaces to the LMX generated code.

The simplest solution is to include the source code of all three types of code directly into your project. This is the quickest and easiest solution for the evaluation phase. However, you may choose to compile the different types of code separately into libraries or Windows DLLs. The following sections give guidance on how to do

this, although note that there are many different ways in which you may choose to partition the code.

2.2.1 - Compiling the Runtime Supporting Software Source Code

As mentioned above, the Runtime Supporting Software is common to all LMX based projects. The `.cpp` source files for this code are located in the `supporting-software/src` sub-directory of the installation, and the `.h` header files are located in the `supporting-software/include` sub-directory.

The set of source files you need for your build will depend on your requirements:

- `lmxparse.cpp` contains the low-level XML parser and related code. Almost all LMX projects will need this file.
- `lmxtypes.cpp` contains the LMX type classes. Most projects will use this file, although there is the option to provide your own alternative.
- `lmxregex.cpp` and `lmxunicode.cpp` are needed if you have not disabled regular expression handling in your build and you don't want to use your own variant of these (see 3.23 - Pattern Facet Handling Customization).
- `lmxsoap.cpp` may be included if you wish to use LMX's SOAP functionality.
- `lmxwinhttp.cpp` can optionally be included if you wish to use LMX's Windows HTTP functionality. (Note that `lmxwinhttp.cpp` is Windows specific and will not compile on other platforms. We recommend using libraries such as Linux's `libcurl` on other platforms.)

During the build process, these files will require access to the Supporting Software include files, typically located in the `supporting-software/include` sub-directory of the installation. (Note that you might want to move the various files out of their default location when building a library so that you can more easily place them under version control.)

The exact process of building the source code will depend on your platform. The simplest approach is to directly include the various source files in with your main project or makefile. (Note that if you use the Visual Studio DLL C Runtime, the LMX code assumes by default that ultimately you will put the supporting software code in its own DLL. As a result, if you include it directly in your project you will get `LNK4217: locally defined symbol` warnings (or similar). In initial evaluation you can either ignore these warnings, or include `LMX_WANT_DLL=0` in the `Preprocessor Definitions` of your project settings.)

The Windows version of LMX includes a batch file called `build-libs.bat` in the `supporting-software/src` sub-directory of the installation that can be used to build the source code into static `.lib` files and dynamic `.dll` files. This batch file should be run inside a Visual Studio Command Prompt window. If you choose to build the files within the default `c:\Program Files\LMX` directory location then you will need to run the Visual Studio Command Prompt as an Administrator. Read the comments in `build-libs.bat` for further details on how to customize the build to your needs.

Note that to build the supporting software using `build-libs.bat` on Visual Studio 2015, 2017 & later, you need to ensure that the Windows SDK component of the Visual Studio installation process has been included. (If not you can modify your Visual Studio configuration by re-running the Visual Studio installer.)

As you may notice from the `build-libs.bat` batch file, if you wish to build the Supporting Software Source Code into your own DLL, you need to configure your project to define `LMX_MAKE_DLL` (i.e. "Project->Properties->Configuration Properties->C/C++->Preprocessor->Preprocessor Definitions" includes `LMX_MAKE_DLL;`.) and also define `LMX_RT_SOURCE` (the value is not important in this latter case). (For versions prior to version 6.1 of LMX or if you only want part of the LMX Supporting Software in a DLL, instead of defining `LMX_RT_SOURCE`, define `LMX_PARSE_SOURCE`, `LMX_TYPES_SOURCE`, and `LMX_REGEX_SOURCE`.)

If the Supporting Software Source Code has been built into a DLL and you which to combine the DLL with other code then you need to configure the project that references the DLL to define `LMX_WANT_DLL` to `1`.

The Linux version of LMX includes a makefile called `MakefileLMXlib` that can be used as a starting point for writing a makefile for building the Supporting Software into a static or shared library.

2.2.2 - Compiling the Generated Code

The code generated by LMX will need access to the Supporting Software include files, typically located in the `supporting-software/include` sub-directory of the installation, in order to compile.

If you wish to build the generated code into a DLL, you need to configure your project to define `LMX_MAKE_DLL` (i.e. "Project->Properties->Configuration Properties->C/C++->Preprocessor->Preprocessor Definitions" includes `LMX_MAKE_DLL;`).

If the generated code has been built into a DLL and you which to combine the DLL with other code then you need to configure the project that references the DLL to define `LMX_WANT_GEN_IMPORT_DLL` to `1`.

2.2.3 - Compiling Your Code to Use LMX Code

To compile your code so that it can use the LMX generated code and Runtime Support Software your code will need to have access to the generated header file. For example, if the LMX generated header file is `generated.h`, then you will need to include the following lines in your .cpp file:

```
#include "generated.h"
```

If your code performs 'advanced' forms of marshaling and unmarshaling by calling the generated `marshal()` and `unmarshal()` methods directly (see 3.5 - Marshaling (advanced forms) and 3.2 - Unmarshaling (advanced forms)), then you will need to include both the LMX generated `.h` file and the LMX parser header file; `lmxparse.h`. For example:

```
#include "generated.h"
#include "lmxparse.h"
```

(If you choose to configure LMX to generate multiple header files (e.g. one header file per class), you should substitute the line `#include "generated.h"` mentioned above as appropriate.)

The generated header file will need access to the Runtime Supporting Software header files, typically located in the `supporting-software/include` sub-directory of the installation.

If the Runtime Supporting Software Source Code has been built into a DLL and you wish to combine it with your code then you need to configure your code sub-project to define `LMX_WANT_DLL` to `1`.

If the generated code has been built into a DLL and you wish to combine it with your code then you need to configure your code sub-project to define `LMX_WANT_GEN_IMPORT_DLL` to `1`.

2.3 - Unmarshaling (simple form)

Unmarshaling is the process of reading in XML and converting it to C++ objects. LMX can parse UTF-8, UTF-16 (big and little endian), ISO-10646-UCS-2 (big and little endian), ISO-8859-1 (aka Latin-1), and US-ASCII. The source XML can either be in an in-memory data buffer, or a file.

For the class generated at the top of the class hierarchy (e.g. c_root) LMX generates convenience functions to help the marshaling and unmarshaling process. These functions are not generated for other classes in order to reduce generated code size. This section first presents the method for unmarshaling using these convenience methods. Later in the section an alternative method of unmarshaling is presented, which can be used with other classes. Additionally, advanced unmarshaling techniques are presented in 3.2 - Unmarshaling (advanced forms).

To read XML from a `std::string`, code similar to the following can be used:

```
// Allocate somewhere to store any error code that the unmarshaling may
// produce
lmx::elmx_error result;
lmx::s_debug_error error_detail;

// Construct an object from the XML contained in the std::string
// xml_message_data_string and store the result code of the
// unmarshaling in the 'result' variable.
const c_generated_xsd_root_class top_object(
                                    xml_message_data_string,
                                    &result,
                                    &error_detail );

// If the 'result' variable indicates that all is OK, interact with the
// object.
if( result == lmx::ELMX_OK )
{
    // Work with unmarshalled data...
```

The `error_detail` argument is optional, so the following code is also valid if detailed error information is not required:

```
lmx::elmx_error result;

const c_generated_xsd_root_class top_object(
                                    xml_message_data_string,
                                    &result );

if( result == lmx::ELMX_OK )
{
    // Work with unmarshalled data...
```

To read XML from an in-memory buffer, where `xml_message_data_buffer` is of type `const char`

* or similar, code similar to the following can be used:

```
// Allocate somewhere to store any error code that the unmarshaling may
// produce
lmx::elmx_error result;
lmx::s_debug_error error_detail;    // Using error_detail is optional

// Construct an object from the XML contained in the memory located at
// xml_message_data_buffer and store the result code of the unmarshaling in
// the 'result' variable.
const c_generated_xsd_root_class top_object(
                                    xml_message_data_buffer,
                                    number_of_bytes_in_buffer,
                                    &result,
                                    &error_detail );    // error_detail is optional

// If the 'result' variable indicates that all is OK, interact with the
// object.
if( result == lmx::ELMX_OK )
{
    // Work with unmarshalled data...
```

To read XML from a file, code similar to the following can be used (which doesn't use the option to retrieve detailed error information):

```
// Allocate somewhere to store any error code that the unmarshaling may
// produce
lmx::elmx_error result;

// Construct an object from the XML contained in the file 'c:/myxml.xml'
// and store the result code of the unmarshaling in the 'result' variable.
const c_generated_xsd_root_class top_object( "c:/myxml.xml", &result );

// If the 'result' variable indicates that all is OK, interact with the
// object.
if( result == lmx::ELMX_OK )
{
    // Work with unmarshalled data...
```

Convenience methods are also generated to unmarshal instances corresponding to `xs:any` constructs that have previously been unmarshalled from an XML instance. This can be done using code similar to the following:

```
// Allocate somewhere to store any error code that the unmarshaling may
// produce
lmx::elmx_error result;

// Construct an object from the XML corresponding to an xs:any instance
// unmarshalled from a previous unmarshaling run.
const c_generated_xsd_root_class top_object(
                          my_previously_unmarshaled_object.get_any(),
                          &result );

// If the 'result' variable indicates that all is OK, interact with the
// object.
if( result == lmx::ELMX_OK )
{
    // Work with unmarshalled data...
```

where the `get_any()` method is described in 2.5.10 - Singular xs:any C++ Interface and 2.5.12 - Multiple xs:any C++ Interface.

If it is not convenient to unmarshal at the time the object is constructed (for example, if the object is a static global store of the application's user preferences) then the following methods can be used.

To read XML from a `std::string`, code similar to the following can be used:

```
// Create an instance of the class
c_generated_xsd_root_class top_object;

// ... additional code ...

// Use the unmarshal method, giving the std::string to read from
lmx::elmx_error result = top_object.unmarshal( xml_message_data_string );

// If unmarshaling was successful, interact with the object
if( result == lmx::ELMX_OK )
{
    // Work with unmarshalled data...
```

To read XML from an in-memory buffer (of type `const char *` or similar), code similar to the following can be used:

```
// Create an instance of the class
c_generated_xsd_root_class top_object;

// ... additional code ...

// Use the unmarshal method, giving the details of the memory to read from
lmx::elmx_error result = top_object.unmarshal( xml_message_data_buffer,
                                    number_of_bytes_in_buffer );

// If unmarshaling was successful, interact with the object
if( result == lmx::ELMX_OK )
{
    // Work with unmarshalled data...
```

To read XML from a file:

```
// Create an instance of the class
c_generated_xsd_root_class top_object;

// ... additional code ...

// Use the unmarshal method, giving the details of the file to read from
lmx::elmx_error result = top_object.unmarshal( "c:\\myxml.xml" );

// If unmarshaling was successful, interact with the object
if( result == lmx::ELMX_OK )
{
    // Work with unmarshalled data...
```

If you wish to unmarshal from an object for which the convenience functions were not generated, then you can use the `unmarshal` template functions that are defined in `lmxparse.h`. In this case the code form is:

- for XML in a `std::string`:

```
// Create an instance of the class
c_generated_class an_object;

// Use the template unmarshal method, giving the details of the memory to
// read from
lmx::elmx_error result = lmx::unmarshal( &an_object,
                                         xml_message_data_string );

// If unmarshaling was successful, interrogate the object
if( result == lmx::ELMX_OK )
{
    // Work with unmarshalled data...
```

- for XML in memory:

```
// Create an instance of the class
c_generated_class an_object;

// Use the template unmarshal method, giving the details of the memory to
// read from
lmx::elmx_error result = lmx::unmarshal( &an_object, xml_message_data_buffer,
                                    number_of_bytes_in_buffer );

// If unmarshaling was successful, interrogate the object
if( result == lmx::ELMX_OK )
{
    // Work with unmarshalled data...
```

- for XML in a file:

```
// Create an instance of the class
c_generated_class an_object;

// Use the unmarshal method, giving the details of the file to read from
lmx::elmx_error result = lmx::unmarshal( &an_object, "c:\\myxml.xml" );

// If unmarshaling was successful, interrogate the object
if( result == lmx::ELMX_OK )
{
    // Work with unmarshalled data...
```

For information on how to diagnose errors, particularly during debugging, when using these convenience methods, see 3.16.4 - Collecting Debug Error Information when using Convenience Methods.

As mentioned previously, additional unmarshaling techniques are described in 3.2 - Unmarshaling (advanced forms).

2.4 - Marshaling (simple form)

Marshaling is the process converting the C++ object content into XML data.

For the class generated at the top of the class hierarchy (e.g. c_root) LMX generates convenience functions to help the marshaling and unmarshaling process. These functions are not generated for other classes in order to reduce generated code size. This section first presents the method for marshaling using these convenience methods. Later in the section an alternative method of marshaling is presented, which can be used with other classes. Additionally, advanced marshaling techniques are presented in 3.5 - Marshaling (advanced forms).

A typical section of marshaling code for writing the XML to a string looks as follows:

```
// Create an instance of the class
c_generated_xsd_root_class top_object;

// ... Populate and manipulate top_object ...

// Allocate a string in which to store the marshalled XML
std::string my_string;

// Marshal the object to the string
top_object.marshal( &my_string );
```

The equivalent code for writing to a file is as follows:

```
// Create an instance of the class
c_generated_xsd_root_class top_object;

// ... Populate and manipulate top_object ...

// Marshal the object to the file "c:\myxml.xml"
if( top_object.marshal( "c:\\myfile.xml" ) != lmx::ELMX_OK )
{
    // Something went wrong!
}
```

If you wish to marshal an object for which the convenience functions are not generated, then you can use the `marshal` template functions defined in `lmxparse.h`. In this case, the code for marshaling to string is:

```
// Create an instance of the class
c_generated_class an_object;

// ... Populate and manipulate an_object ...

// Allocate a string in which to store the marshalled XML
std::string my_string;

// Marshal the object to the string
lmx::marshal( an_object, &my_string );
```

and the code for marshaling to a file is:

```
// Create an instance of the class
c_generated_class an_object;

// ... Populate and manipulate an_object ...

// Marshal the object to the file "c:\myxml.xml"
if( lmx::marshal( an_object, "c:\\myfile.xml" ) != lmx::ELMX_OK )
{
    // Something went wrong!
}
```

As in the unmarshalling case, for information on how to diagnose errors, particularly during debugging, when using these convenience methods, see 3.16.4 - Collecting Debug Error Information when using Convenience Methods.

As mentioned previously, additional marshaling techniques are described in 3.5 - Marshaling (advanced forms).

2.5 - Accesing the Data in the Generated Classes

The C++ class interface to a generated item is independent of whether the item is an element or an attribute. Hence the difference between these two forms is abstracted away. The term 'item' is used here to refer to either an element or an attribute.

An interface to an item depends in one respect on whether an item is:

- empty.
- a simpleType.
- a complexType.
- xs:anyAttribute.
- xs:any.

The other aspect that affects the interface is whether the item is:

- Singular - minOccurs = maxOccurs = 1, or a required attribute.
- Optional - minOccurs = 0, maxOccurs = 1, or a normal attribute.
- Multiple - maxOccurs > 1 (or unbounded), or a list.

A further consideration for optional items is whether the item is in an xs:choice construct.

There are also special functions to interface with enumerated values, and additional functions to interface to complex types that are polymorphic.

The following sections describe the interface for each of these combinations. Throughout the discussion, the string `NAME` is used to represent the name given to the item.

An element of type `xs:any` will by default have the `NAME` part of the method name set to `any`. However, if there is more than one xs:any element in a complex type, or another element is named `any`, numbers will be appended to the name to provide differentiation.

2.5.1 - Singular Empty Type C++ Interface

No interface is generated.

2.5.2 - Optional Empty Type C++ Interface

If the item is not in an xs:choice, the following method is generated:

```
void set_NAME();
```

Marks the item `NAME` as present.

```
bool isset_NAME() const;
```

Returns `true` if the item `NAME` is present, and `false` otherwise.

`void unset_NAME();`
Marks the item `NAME` as not present.

If the item is allowed in an xs:choice, but not present, the `getchosen()` method returns `<class name>::e_choice_not_set`.

2.5.3 - Singular Simple Type C++ Interface

Note that for the interface to list types, see 2.5.5 - Multiple Simple Type and List Simple Type C++ Interface.

`<return type> get_NAME() const;`
Return either the value of item `NAME` or a reference to `NAME` depending on the particular type.

`<non-const ref to type> get_NAME();`
Generated if the type of NAME is a compound class such as for xs:date. Returns a non-const reference to `NAME`.

`void set_NAME( simple_type );`
Set item `NAME` to the specified value.

If the type has enumerations, the following additional methods are generated:

`elmx_enums getenum_NAME() const;`
Returns the enumerated constant corresponding to the value of item `NAME`.

`bool setenum_NAME( elmx_enums );`
Sets the value of item `NAME` to the value corresponding to the specified enumerated constant. `true` is returned if the value is correctly set, and `false` otherwise.

If a simple type is nillable, then it is treated as complex type / simple content. Thus the interaction with nillable types is discussed below.

2.5.4 - Optional Simple Type C++ Interface

(Note: See 2.5.5 - Multiple Simple Type and List Simple Type C++ Interface for the interface to list types.) The Optional Simple Type has the same methods as defined in 2.5.3 - Singular Simple Type C++ Interface, plus the following methods:

If the item is not in an xs:choice, the following methods are generated:

`bool isset_NAME() const;`
Returns `true` if the item `NAME` is present, and `false` otherwise.

`void unset_NAME();`
Marks the item `NAME` as not present.

If the item is allowed in an xs:choice, but not present, the `getchosen()` method returns `<class name>::e_choice_not_set`.

2.5.5 - Multiple Simple Type and List Simple Type C++ Interface

This interface is used for elements that have a cardinality greater than 1, and elements or attributes with lists.

`size_t size_NAME() const;`
: Returns the number of instances to type `NAME`.

`<return type> get_NAME( size_t n ) const;`
: Returns the n-th value of item `NAME`. The first instance is n=0.

`<non-const ref to type> get_NAME( size_t n );`
: Generated if the type of NAME is a compound class such as for xs:date. Returns a non-const reference to the n-th value of item `NAME`. The first instance is n=0.

`lmx::elmx_error set_NAME( size_t n, type );`
: Sets the instance at the n-th position of the collection of `NAME` items. The item previously at the n-th position is over-written. If a value of n is specified that is larger than the size of the collection, the `append_NAME( type )` method is called. The first instance is n=0.

`lmx::elmx_error append_NAME( type );`
: Appends an instance to the collection of `NAME` items.

`lmx::elmx_error insert_NAME( size_t n, type );`
: Inserts an instance into the n-th position of the collection of `NAME` items. The item previously at the n-th position is moved to the (n+1)-th position and so on. If a value of n is specified that is larger than the size of the collection, the `append_NAME( type )` method is called. The first instance is n=0.

`void delete_NAME( size_t n );`
: Deletes the n-th value of item `NAME`. The first instance is n=0.

`void clear_NAME();`
: Removes all items from the collection. In the case of a list item, calling this method can be used to create an empty list if a list with no members is wanted.

`auto in_NAME();`
: Returns iterator details that can be used in a C++11 range-based for statement. See example in 2.5.5.1 - Multiple Simple Type Usage Examples.

If the type has enumerations, the following additional methods are generated:

`elmx_enums getenum_NAME( size_t n ) const;`
: Returns the enumeration constant corresponding to the n-th instance of item `NAME`. The first instance is n=0.

`bool setenum_NAME( size_t n, elmx_enums set );`
: Sets the value of the item at the n-th position of the `NAME` collection to the value corresponding to the specified enumeration constant. If a value of n is specified that is larger than the size of the collection, the `appendenum_NAME( type )` method is called. The first instance is n=0. `true` is returned if the value is correctly set, and `false` otherwise.

`bool appendenum_NAME( elmx_enums set );`
: Appends a value to the item `NAME` corresponding to the specified enumeration constant. `true` is returned if the value is correctly set, and `false` otherwise.

`bool insertenum_NAME( size_t n, elmx_enums set );`
: Inserts a value at the n-th position of the collection `NAME` corresponding to the specified enumeration constant. If a value of n is specified that is larger than the size of the collection, the `appendenum_NAME( type )` method is called. The first instance is n=0. `true` is returned if the value is correctly set, and `false` otherwise.

If the type is an optional list, the following additional methods are generated (this allows an empty list to be present):

`bool isset_NAME() const;`
: Returns `true` if the item `NAME` is present, and `false` otherwise.

`void unset_NAME();`
: Marks the item `NAME` as not present.

2.5.5.1 - Multiple Simple Type Usage Examples

For reading:

```
    for( size_t i=0; i<top.size_NAME(); ++i )
        do_something( top.get_NAME( i ) );
```

Or:

```
    for( auto & i : top.in_NAME() )
        do_something( i );
```

2.5.6 - Singular Complex Type C++ Interface

`<const ref to type> get_NAME() const;`
: This method provides read-only access to `NAME`. It is recommended that this method be used for reading the item rather than the read/write variant.

`<non-const ref to type> get_NAME();`
: This method allows read/write access to `NAME` by returning a reference.

If the complex type is an xs:choice, the following method is generated:

`<class name>::elmx_chosen getchosen() const;`
: Returns the class specific enumerated value corresponding to the present choice.

For the marshaling operation, the appropriate choice is selected when the relevant `set_NAME` or non-const `get_NAME` method among the choice's children is called. For example, if `CHOICE` is a choice, represented by the class `c_CHOICE`, and `OPTION` is one of the elements within the choice, then if we have:

```
    c_CHOICE &my_choice = ...;
```

we can select the desired option within the choice by doing:

```
    my_choice.set_OPTION(...);
```

Similarly, if we do not immediately have a pointer or reference to `c_CHOICE`, we can set the desired option by doing:

```
my_item.get_CHOICE().set_OPTION(...);
```

If `OPTION` is a complex type which has associated with it a non-const get method, then the act of calling that non-const get method will cause the appropriate choice option to be selected. For example:

```
c_OPTION &my_option = my_item.get_CHOICE().get_OPTION(...);
```

or:

```
my_item.get_CHOICE().get_OPTION().set_an_option_child(12);
```

If the complex type is an xs:all, the following methods are generated:

`<class name>::elmx_all getorder( size_t n ) const;`
: Returns the enumeration constant corresponding to the n-th present element in the xs:all construct. The first element corresponds to n == 0.

`size_t getorder( elmx_all enumeration_const ) const;`
: Returns the position within the `xs:all` construct of the element corresponding to the enumeration constant `enumeration_const`. The function returns `lmx::k_all_order_not_present` if the element is not present.

`size_t getorder_NAME() const;`
: Returns the position of element NAME within the `xs:all` construct. The function returns `lmx::k_all_order_not_present` if the element is not present.

`size_t sizeorder() const;`
: Returns the number of elements present in the xs:all construct.

During marshaling, the order that the elements are output corresponds to the order in which they are written to the object.

A nillable type will always have a class generated for it, even if it is a simple type. (In effect a nillable simple type is treated as complex type / simple content.) If a type is nillable, the following functions will be generated:-

`void setnil_NAME();`
: Used to set the element to a nil value.

`bool isnil_NAME() const;`
: Returns `true` is a value is nil, and false otherwise.

If a value is nillable, you should test `isnil_NAME()` prior to attempting to read the element's body value(s) using `get_NAME()` etc.

If a type is polymorphic, the following method is generated:

`void assign_NAME( c_NAME_TYPE *p_derived_type );`
: Sets the polymorphic type associated with item NAME to the object pointed to by p_derived_type, which must be a derived type of c_NAME_TYPE. Once the pointer is assigned to the generated

object, the generated object takes responsibility for deleting the pointer on destruction. (Note: the `get_NAME()` method can be used to access a polymorphic type, and no additional methods are generated for this purpose.)

2.5.6.1 - Singular Complex Type Usage Examples

To read a child of `NAME`:

```
int an_int = top.get_NAME().get_CHILD();
```

Or, if you intend to read a lot of items from `NAME`, another possibility is:

```
const c_NAME & name_ref = top.get_NAME();
int an_int = name_ref.get_CHILD();
float a_float = name_ref.get_CHILD2();
```

To write to `NAME`:

```
top.get_NAME().set_CHILD( 12 );
```

Or:

```
c_NAME & name_ref = top.get_NAME();
name_ref.set_CHILD( 12 );
name_ref.set_CHILD2( 1.2 );
```

If the complex type is an `xs:choice`:

```
switch( top.getchosen() )
    {
case c_top::e_my_int:
    ...
break;
case c_top::e_my_float:
    ...
break;
default:
    assert(0);
    }
```

An alternative to the above with better compile time checking is:

```
c_top::elmx_chosen chosen = top.getchosen();
if( chosen == c_top::e_my_int ) {
    ...
}
else if( chosen == c_top::e_my_float ) {
    ...
}
```

To set the choice option when marshaling, use one of the children's `set` or non-const `get` methods. For example:

```
top.get_NAME().set_CHILD( 12 );
```

Or:

```
c_NAME & name_ref = top.get_NAME();
name_ref.set_CHILD( 12 );
```

If the complex type is an `xs:all`:

```
for( size_t i=0; i<top.sizeorder(); ++i )
    {
    switch( top.getorder(i) )
        {
    case c_top::e_my_int:
        ...
    break;
    case c_top::e_my_float:
        ...
    break;
    default:
        assert(0);
        }
    }
```

2.5.7 - Optional Complex Type C++ Interface

An Optional Complex Type has the same methods as a Singular Complex type, plus the additional methods specified 2.5.4 - Optional Simple Type C++ Interface.

If the complex type is an xs:choice, a `getchosen()` method is generated as described in 2.5.6 - Singular Complex Type C++ Interface. If the optional xs:choice is not present, the `getchosen()` method returns `<class name>::e_choice_not_set`.

2.5.7.1 - Optional Complex Type Usage Examples

To read an optional complex type:

```
if( top.isset_NAME() ) {
    const c_NAME & name_ref = top.get_NAME();
    int an_int = name_ref.get_CHILD();
    float a_float = name_ref.get_CHILD2();
}
```

To write to an optional complex type:

```
c_NAME & name_ref = top.get_NAME();
name_ref.set_CHILD( 12 );
name_ref.set_CHILD2( 1.2 );
```

2.5.8 - Multiple Complex Type C++ Interface

Common to reading and writing:

```
size_t size_NAME() const;
```

Returns the number of instances of item NAME.

For reading:

```
<const ref to type> get_NAME( size_t n ) const;
```

Returns a const reference to the n-th instance of item NAME. This version should always be used when reading from item NAME. The first instance is n=0.

For writing:

```
void append_NAME();
```

Appends a new instance of item NAME. This method is used in conjunction with the back_NAME() method. For more information see the example below.

```
void append_NAME( <type> * p );
```

Appends the value pointed to by p to the collection of NAME items. The LMX generated code takes ownership of the object pointed to by p when the method is called, and will delete it when the LMX object is deleted. Use get_NAME(n) to access and modify the inserted value.

```
<non-const ref to type> back_NAME();
```

Returns a reference to the instance at the back of the list of item NAME. This method is used in conjunction with the append_NAME() method. For more information, see the example below.

```
void insert_NAME( size_t n );
```

Inserts an instance into the n-th position of the collection of NAME items. The item previously at the n-th position is moved to the (n+1)-th position and so on. The first instance is n=0. If a value of n is specified that is larger than the size of the collection, the append_NAME() method is called. Use get_NAME(n) to access and modify the inserted value.

```
void insert_NAME( size_t n, <type> * p );
```

Inserts the value pointed to by p into the n-th position of the collection of NAME items. The item previously at the n-th position is moved to the (n+1)-th position and so on. The first instance is n=0. If a value of n is specified that is larger than the size of the collection, the append_NAME(p) method is called. The LMX generated code takes ownership of the object pointed to by p when the method is called, and will delete it when the LMX object is deleted. Use get_NAME(n) to access and modify the inserted value.

```
<non-const ref to type> get_NAME( size_t n );
```

Returns a reference to the n-th instance of item NAME. The first instance is n=0. If such an instance does not currently exist, an instance (and all intervening instances) will be created. For example, if m instances already exist, and instance n is requested (where m < n) n - m instances will be created.

```
<non-const ref to type> assign_NAME( size_t n, <const ref to type> );
```

Does a deep copy of the input parameter to the n-th position of the collection. The first instance is n=0. If such an instance does not currently exist, an instance (and all intervening instances) will be created. For example, if m instances already exist, and instance n is requested (where m < n) n - m instances will be created.

```
void delete_NAME( size_t n );
```

Deletes the n-th instance of item NAME. The first instance is n=0.

```
void clear_NAME();
```

Removes all items from the collection.

`auto in_NAME();`
: Returns iterator details that can be used in a C++11 range-based for statement. See example in 2.5.8.1 - Multiple Complex Type Usage Examples.

If `NAME` is an optional xs:list, it also has the methods defined for an Optional Complex Type.

If the complex type is an xs:choice, the following method is also generated:

`<class name>::elmx_chosen getchosen( size_t n ) const;`
: Returns the class specific enumerated value corresponding to the chosen element in the n-th instance of the xs:choice. n=0 specifies the first instance. Other than the `n` parameter to select the index, the `getchosen()` method operates the same way as described in 2.5.6 - Singular Complex Type C++ Interface.

If a type is polymorphic, the following method is generated:

`void append_NAME( c_NAME_TYPE *p_derived_type );`
: Appends the polymorphic type pointed to by p_derived_type to the item NAME, which must be a derived type of c_NAME_TYPE. Once the pointer is assigned to the generated object, the generated object takes responsibility for deleting the pointer on destruction. (Note: the `get_NAME(size_t n)` method can be used to access a polymorphic type, and no additional methods are generated for this purpose.)

2.5.8.1 - Multiple Complex Type Usage Examples

For reading:

```
for( size_t i=0; i<top.size_NAME(); ++i )
    do_something( top.get_NAME( i ) );
```

Or:

```
for( auto & i : top.in_NAME() )
    do_something( i );
```

For writing, a new instance of the complex type is appended using the `append_NAME` method. The new instance is then populated by repeated use of the `back_NAME` method. For example:

```
while( not_finished( &an_int, &a_float ) ) {
    top.append_NAME();
    top.back_NAME().set_int( an_int );
    top.back_NAME().set_float( a_float );
}
```

2.5.9 - Multiple xs:anyAttribute C++ Interface

If xs:anyAttribute is specified, only the 'multiple' case is allowed.

When a document has instances xs:anyAttribute it may be necessary to add extra namespace information to the `c_xml_writer` class. See 3.18 - Adding Extra Namespace Information for more information.

```
size_t sizeany_attribute() const;
```
Returns the number of xs:anyAttributes present.

```
void getany_attribute( size_t n,
                       std::string *p_namespace,
                       std::string *p_name,
                       std::string *p_value ) const;
```
Retrieves the n-th xs:anyAttribute. The namespace of the attribute (e.g. 'http://...') is stored in the string pointed to by p_namespace, the name of the attribute is stored in the string pointed to by p_name, and the value is stored in the string pointed to by p_value. n=0 refers to the first instance.

```
void appendany_attribute( const std::string &name,
                          const std::string &value );
```
Appends an xs:anyAttribute to the list of xs:anyAttributes. The name of the attribute is stored in name, and the value in value. When written to the XML document, the value part will be surrounded by quote marks, and standard XML entities will be escaped (e.g. & ' " etc).

```
void insertany_attribute( size_t n,
                          const std::string &name,
                          const std::string &value );
```
Inserts an xs:anyAttribute at the n-th position of the list of xs:anyAttributes. The item previously at the n-th position is moved to the (n+1)-th position and so on. The first instance is n=0. If a value of n is specified that is larger than the size of the collection, the `appendany_attribute()` method is called. The name of the attribute is stored in name, and the value in value. When written to the XML document, the value part will be surrounded by quote marks, and standard XML entities will be escaped (e.g. & ' " etc).

```
void deleteany_attribute( size_t n );
```
Deletes the n-th instance of xs:anyAttribute. n=0 refers to the first instance.

```
void clearany_attribute();
```
Removes all items from the collection.

2.5.10 - Singular xs:any C++ Interface

When a document has instances xs:any it may be necessary to add extra namespace information to the `c_xml_reader` and `c_xml_writer` class. See 3.18 - Adding Extra Namespace Information for more information.

To access the xs:any information, use the following methods:

```
const lmx::c_any_info & get_any() const;
lmx::c_any_info       & get_any();
```
Retrieves the `lmx::c_any_info` object corresponding to the data in the `xs:any` instance, where `lmx::c_any_info` has the following public interface:

```
class c_any_info
{
public:
    c_any_info();
    // Compiler generated c_any_info( const c_any_info & ar_rhs ) is OK
    // For writing during unmarshaling
    c_any_info( const std::string &ar_namespace,
                          const std::string &ar_name,
```

```
                                const std::string &ar_value );
        // For user writing xs:anyAttribute
        c_any_info( const std::string &ar_name,
                                const std::string &ar_value )
        // For user writing xs:any
        c_any_info( const std::string &ar_value );
        // Compiler generated operator = ( const c_any_info & ar_rhs ) is OK
        const std::string & get_namespace() const;
        const std::string & get_name() const;
        const std::string & get_value() const;
        std::string get_local_name() const;
        void set_namespace( const std::string &ar_namespace );
        void set_name( const std::string &ar_name );
        void set_value( const std::string &ar_value );

        const c_namespace_context & get_namespace_context() const;
        c_namespace_context & get_namespace_context();
    };
```

If the `xs:any` element corresponds to:

```
<nsp:myElement xmlns:nsp="http://codalogic.com/example"
etc....>...</nsp:myElement>
```

then:

- `get_namespace()` returns: `http://codalogic.com/example`
- `get_name()` returns: `nsp:myElement`
- `get_local_name()` returns: `myElement`
- `get_value()` returns: `<nsp:myElement xmlns:nsp="http://codalogic.com/example" etc....>...</nsp:myElement>`.

```
void set_any( const lmx::tlmx_string &any );
```

Set the value of an `xs:any` element. The specified value must include the start tag and end tag, and be UTF-8 encoded. The value is inserted into the output using a simple copy operation.

(See 2.5 - Accesing the Data in the Generated Classes for the naming of `xs:any` access methods.)

Additionally, the generated code supports a legacy mode interface to `xs:any` using the following interface:

```
void get_any( std::string *p_namespace,
              std::string *p_name,
              std::string *p_value ) const;
```

Retrieves the details of an `xs:any` element. If the `xs:any` element corresponds to:

```
<nsp:myElement xmlns:nsp="http://codalogic.com/example"
etc....>...</nsp:myElement>
```

then:

- `*p_namespace` is set to: `http://codalogic.com/example`
- `*p_name` is set to: `nsp:myElement`
- `*p_value` is set to: `<nsp:myElement xmlns:nsp="http://codalogic.com/example"`

`etc....>...</nsp:myElement>`.

`*p_namespace` and/or `*p_name` may be set to 0 (NULL) if it is not desired to retrieve their corresponding values.

2.5.11 - Optional xs:any C++ Interface

An Optional `xs:any` has the same interface methods as a Singular xs:any instance as described in 2.5.10 - Singular xs:any C++ Interface, plus the methods of any described in 2.5.4 - Optional Simple Type C++ Interface.

2.5.12 - Multiple xs:any C++ Interface

When a document has instances xs:any it may be necessary to add extra namespace information to the `c_xml_reader` and `c_xml_writer` class. See 3.18 - Adding Extra Namespace Information for more information.

```
size_t size_any() const;
```
Returns the number of instances of a particular `xs:any`.

```
const lmx::c_any_info & get_any( size_t n ) const;
lmx::c_any_info       & get_any( size_t n );
```
Retrieves a reference to the `lmx::c_any_info` object representing the n-th instance of a particular `xs:any`. The first instance is n=0. `lmx::c_any_info` is described in 2.5.10 - Singular xs:any C++ Interface.

(See 2.5 - Accesing the Data in the Generated Classes for the naming of `xs:any` access methods.)

```
void append_any( const lmx::tlmx_string &any );
```
Appends an instance to a particular `xs:any`. The specified value must include the start tag and end tag, and be UTF-8 encoded. The value is inserted into the output using a simple copy operation.

```
void insert_any( size_t n, const lmx::tlmx_string &any );
```
Inserts an instance to the n-th position of a particular `xs:any`. The first instance is n=0. The item previously at the n-th position is moved to the (n+1)-th position and so on. If a value of n is specified that is larger than the size of the collection, the `append_any()` method is called. The specified value must include the start tag and end tag, and be UTF-8 encoded. The value is inserted into the output using a simple copy operation.

```
void delete_any( size_t n );
```
Deletes the n-th instance of a particular `xs:any`. The first instance is n=0.
```
void clear_any();
```
Removes all items from the collection.

```
auto in_any();
```
Returns iterator details that can be used in a C++11 range-based for statement. See a related example in 2.5.8.1 - Multiple Complex Type Usage Examples.

Additionally, the generated code supports a legacy mode interface to the `xs:any` using the following interface:

```
void get_any( size_t n,
              std::string *p_namespace,
              std::string *p_name,
              std::string *p_value ) const;
```

Retrieves the details of n-th element of a particular `xs:any`. On completion of the function, the string pointed to by p_value will include the start tag and end tag. The first instance is n=0.

If the `xs:any` element corresponds to:

```
<nsp:myElement xmlns:nsp="http://codalogic.com/example"
etc....>...</nsp:myElement>
```

then:

- `*p_namespace` is set to: `http://codalogic.com/example`
- `*p_name` is set to: `nsp:myElement`
- `*p_value` is set to: `<nsp:myElement xmlns:nsp="http://codalogic.com/example" etc....>...</nsp:myElement>`.

`*p_namespace` and/or `*p_name` may be set to 0 (NULL) if it is not desired to retrieve their corresponding values.

2.5.13 - Additional Polymorphic Methods

2.5.13.1 - Finding the Identity of a Polymorphic Class

One way to determine the type of a polymorphic class is to use C++'s RTTI. Additionally, LMX generates methods and members that allow the identity of a class to be determined.

Each polymorphic class has the following public data member:

```
t_class_identity id
```

Returns a constant that is unique for the particular class.

Each polymorphic class has the following public methods:

```
t_class_identity getid() const;
```

A virtual function that returns the id constant that identifies the class type.

```
bool has_id( t_class_identity sought_id ) const;
```

A virtual function that returns true if the class hierarchy contains the type corresponding to sought_id.

For example:

```
c_base *p_base = &item.get_base();
if( p_base->getid() == c_derived::id )
{
    c_derived *p_derived = dynamic_cast<c_derived *>( p_base );
}
```

Or:

```
c_base *p_base = &item.get_base();
if( p_base->has_id( c_derived::id ) )
{
    c_derived *p_derived = dynamic_cast<c_derived *>( p_base );
    // Do the common things specific to c_derived
}
if( p_base->has_id( c_more_derived::id ) )
{
    c_more_derived *p_more_derived = dynamic_cast<c_more_derived *>( p_base );
    // Do the common things specific to c_more_derived
}
```

2.5.13.2 - Polymorphic Cloning

To support polymorphic object duplication, LMX generates a virtual `clone()` method. This returns a pointer to a deep copy of the object on which the `clone()` method is called. For example:

```
c_base *p_base = item.get_base().clone();
```

2.5.14 - Resetting the Class

Each class has a function that resets the object to the state it would be after construction. The prototype for this function is:

`void reset();`
: Resets the state of the object to that as if it had just been constructed.

2.5.15 - Run-time Checking

LMX generates two functions for each class that can be used to check whether sufficient attributes and elements have been set for a class to be minimally valid.

`bool is_occurs_ok( lmx::c_check_tracker * p_tracker_in = LMXNULL );`
: Tests whether sufficient elements and attributes have been set for the class to be valid. `is_occurs_ok()` only checks members of its class and not members of any child or base classes. Returns `true` if sufficient items have been set, and `false` otherwise.

`bool check( lmx::c_check_tracker * p_tracker_in = LMXNULL );`
: Recursively calls the current classes `is_occurs_ok()` method and the `check()` methods of the child classes offering a 'deep' version of `is_occurs_ok()`. Returns `true` if sufficient items have been set, and `false` otherwise.

One way to use these functions is in a debug version `assert` statement, e.g.:

```
c_NAME & name_ref = top.get_NAME();
name_ref.set_CHILD( 12 );
name_ref.set_CHILD2( 1.2 );

assert( name_ref.is_occurs_ok() );
```

If the above call to `is_occurs_ok()` finds missing components it will populate the global instance of `lmx::c_check_exception` called `lmx::global_check_exception`. If the `#define LMX_USE_CHECK_EXCEPTIONS` is set to `1` or `LMX_USE_CHECK_EXCEPTIONS` is not set and `LMX_USE_EXCEPTIONS` is set to `1` then `is_occurs_ok()` will throw a `lmx::c_check_exception` exception if it finds missing components.

`is_occurs_ok()` can also be configured to populate a local instance of `c_check_exception` using the following code:

```
lmx::c_check_exception check_exception;
lmx::c_check_tracker check_tracker( &check_exception );
if( ! name_ref.is_occurs_ok( &check_tracker ) )
{
    // Look at check_exception for details of error
}
```

The `check()` method has similar behavior to the `is_occurs_ok()` method, but recursively calls the child members versions of `check()` to give a deep version of `is_occurs_ok()`.

To test the facets of simple type components LMX allows optional testing of facets when the various `set_NAME` functions are called.

See section 3.16.5 - Debugging Support for more information on these features.

2.6 - The Abbreviated Type Notation

In both the generated C++ header file and the HTML documentation file an abbreviated notation is used to describe the various types. The intention is to allow you to work mainly from the generated C++ header file or HTML file without having to refer to the actual schema definition, thus speeding up the development process. This section describes the format.

The complete abbreviated type notation has the form:

```
name_in_schema --> type { facets } [ cardinality ]
```

The `name_in_schema` field is the name of the item from the schema (possibly made ASCII safe). If the item is an xs:any element, then the field is set to `{any}`. If the item is the body of a Simple Content element, the field is set to `{body}`.

The `type` is the type of the item. This may be one of the built-in schema types (e.g. xs:string) or the name of a type defined elsewhere in the schema. If the type is a built-in schema type, its name is prefixed by `xs:` irrespective of whether the schema namespace prefix has been set to `xs`.

The `facets` field specifies (some of) the facets applied to a type by a schema. The `facets` are wrapped in curly braces and each facet description is separated by a comma. If there are no relevant facets, the `facets` field and the enclosing curly braces are omitted. Currently this field captures pattern, min/max and length facets. Enumeration facets are documented along with the methods for reading and writing items via enumeration (e.g. `getenum_NAME` and `setenum_NAME`).

A pattern facet follows the Perl regular expression format without the beginning and end anchors. That is:

```
/pattern/
```

The min and max range facets have the form:

```
min_value <= x < max_value
```

If either the min or the max limits are not specified, then the corresponding leg of the facet description is omitted. For example, if no min_value were specified, the following format would be used:

```
x < max_value
```

The comparison operator shown in the description (e.g. < or <=) depends on whether the facet is exclusive or inclusive.

Length facets are documented for `xs:string`, `xs:hexBinary` and `xs:base64Binary` types where specified. The facet is documented as:

```
min_length .. max_length
```

If the maximum length is unbounded, the following form is used:

```
min_length .. *
```

If the minimum length and the maximum length are the same, this is documented as:

```
length .. length
```

The cardinality field specifies how many instances of the item are allowed. If the field is absent (including absence of the square brackets), one, and only one, occurrence of the item is allowed. If the field is present, the general format is:

```
[ min .. max ]
```

where `min` specifies the minimum number of times the item can occur and `max` specifies the maximum number of times an item can occur. If `min` and `max` are the same values, then the following form is used:

```
[ min ]
```

If the maximum number of occurrences is unbounded, then the following form is used:

```
[ min .. * ]
```

2.6.1 - Abbreviated Type Notation Examples

To illustrate the Abbreviated Type Notation, the following examples are presented:

```
e1 --> xs:unsignedShort
```

`e1` is an unsigned short that can occur once, and only once.

```
e2 --> xs:unsignedShort [0..1]
```

`e2` is an optional unsigned short. It can occur zero or one time.

```
e3 --> xs:unsignedShort { 0<=x<1000 }
```

`e3` is an unsigned short that is constrained to be greater than or equal to 0 and less than 1000. It can occur once, and only once.

```
e4 --> xs:unsignedShort { 0<=x<1000 }[0..1]
```

`e4` is an unsigned short that is constrained to be greater than or equal to 0 and less than 1000. It can occur zero or one time.

```
e5 --> xs:unsignedShort { 0<x<=1000 }[1..*]
```

`e5` is an unsigned short that is constrained to be greater than 0 and less than or equal to 1000. It can occur one or more times.

```
e6 --> xs:string { /a\d/,/b\d\d/ }[1..*]
```

`e6` is a string that must match the pattern a\d or b\d\d (e.g. 'a1' or 'b12'). It can occur one or more times.

```
e7 --> xs:string { 1..5 }[1..*]
```

`e7` is a string that can be between 1 and 5 characters long. It can occur one or more times.

3 - In More Depth

3.1 - Configuration using lmxuser.h

As already alluded to, in addition to configuring data binding behavior via the LMX executable, you can also control the behavior by setting various C++ #defines and editing USER: sections in lmxuser.h.

For a number of features, the effect of lmxuser.h can be changed by setting C++ #defines. This can be done using -#define flag or in your C++ project or makefiles.

If a large number of defines need to be used, it may be easier to use the -lmxuser-defs flag, or define LMX_WANT_USER_DEFS in your project/makefile. This will cause lmxuser.h to #include "lmxuser-defs.h" and you can set your other #defines via the included file.

You can also specify the -lmxuser-defs-end flag or define LMX_WANT_USER_DEFS_END in your project/makefile, (perhaps in the lmxuser-defs.h file), and this will cause the file lmxuser-defs-end.h to be included at the end of lmxuser.h. This allows further customization of LMX behavior using features that have been defined by lmxuser.h.

If your setup requires both -lmxuser-defs and -lmxuser-defs-end to be used, another option is to use the -alt-lmxuser YYY flag. This will replace of line #include "lmxuser.h" generated in the .h files with the line #include "YYY". If you want to use the #include <YYY> form, then include the <> brackets as part of the parameter, e.g. do -alt-lmxuser <YYY>. In the alternative file you would set any #defines required, then insert #include "lmxuser.h" and then follow that with any other #defines you require.

3.2 - Unmarshaling (advanced forms)

2.3 - Unmarshaling (simple form) describes the simple method for unmarshaling object content direct from files and memory. In certain situations more flexibility is required, and the following more advanced techniques may be required.

To read XML from an in-memory buffer, code similar to the following can be used:

```
lmx::c_xml_reader_memory reader( xml_message_data_buffer,
                                 number_of_bytes_in_buffer );

c_generated_xsd_root_class top_object;

lmx::elmx_error error = top_object.unmarshal( reader );

if( error == lmx::ELMX_OK )
{
    const c_generated_xsd_root_class & const_top_object = top_object;
    ...
```

To read XML from a C++ std::string, the lmx::c_xml_reader_string class can be used in place of lmx::c_xml_reader_memory in the code above.

Similary, to read XML from a C-style `char []` style arrays, the `lmx::c_xml_reader_c_string` class can be used in place of `lmx::c_xml_reader_memory` in the code above.

To read XML from an instance of `lmx::c_any_info`, the `lmx::c_xml_reader_any_info` class can be used.

To read XML from a file, code similar to the following can be used:

```
lmx::c_xml_reader_file reader( "c:\\myxml.xml" );

if( reader.is_open() )
{
    c_generated_xsd_root_class top_object;

    lmx::elmx_error error = top_object.unmarshal( reader );

    if( error == lmx::ELMX_OK )
        {
        const c_generated_xsd_root_class & const_top_object = top_object;
        ...
```

Looking more closely at the in-memory case, we first create an object that will do low-level read operations from memory. This is an instance of `c_xml_reader_memory` from the `lmx` namespace. The constructor for this object takes a pointer to the buffer as the first argument and the number of valid bytes in the buffer as the second argument.

```
lmx::c_xml_reader_memory reader( xml_message_data_buffer,
                                    number_of_bytes_in_buffer );
```

We then create an instance of the top-level class generated by the LMX code generator. The name of this class will differ. If there is only one global element, the relevant class will be that of the global element. Alternatively, the name is either derived from the namespace prefix assigned to the schema's target namespace, or is c_root. It is normally the last class defined in the .h file.

```
c_generated_xsd_root_class top_object;
```

To do the unmarshaling, the unmarshal method of the top level object is called, giving it a reference to the low-level reader class. Note that you must always unmarshal into a freshly created object otherwise the results may be unpredictable. The method returns an error code.

```
lmx::elmx_error error = top_object.unmarshal( reader );
```

If the returned error code is `lmx::ELMX_OK` then unmarshaling has been successful, and the data structure can be interrogated. Section 3.9 - Error Codes lists other possible error codes.

```
if( error == lmx::ELMX_OK )
{
```

Having determined that the unmarshalled object is suitable for processing, it is advisable to create a const version of the class reference to avoid accidentally modifying the various objects.

```
    const c_generated_xsd_root_class & const_top_object = top_object;
    ...
```

Unmarshaling from a file is a similar process, however in this case the `c_xml_reader_file` class from the `lmx` namespace is used as the type for the low-level reader. The constructor takes a single argument, which is the name of the file from which to read.

```
lmx::c_xml_reader_file reader( "c:\\myxml.xml" );
```

Having created the `c_xml_reader_file` instance, it is necessary to test whether the file has been successfully opened by calling the `is_open` method. (This method can also be called on an object of type `c_xml_reader_memory`, `c_xml_reader_string`, `c_xml_reader_c_string` or `c_xml_reader_any_info`, but this always returns true.)

```
if( reader.is_open() )
{
```

If the file is successfully opened, the unmarshaling operation continues as for the from memory case, e.g.:

```
    c_generated_xsd_root_class top_object;

    lmx::elmx_error error = top_object.unmarshal( reader );

    if( error == lmx::ELMX_OK )
    {
        const c_generated_xsd_root_class & const_top_object = top_object;
        ...
```

Note that during unmarshaling memory may be dynamically allocated, which may cause exceptions to be thrown. Therefore it may be appropriate to include `try/catch` blocks at some level in your code.

3.2.1 - Unmarshaling with Additional Control when Reading XML Input

If additional control is required over the way XML is read, a further method for unmarshaling is available. The interface class for reading XML input is `c_read`, which is defined in `lmxparse.h`. To implement reading from different sources, or perform special processing while reading (such a reading from a socket or a compressed file), implement a concrete class that derives from `c_read` and implements the relevant virtual methods. Assuming such a concrete class is called `c_my_reader`, the code to unmarshal using this class would be:

```
c_my_reader low_level_reader( ...relevant parameters... );

lmx::c_xml_reader reader( low_level_reader );

c_generated_xsd_root_class top_object;

lmx::elmx_error error = top_object.unmarshal( reader );

if( error == lmx::ELMX_OK )
{
    const c_generated_xsd_root_class & const_top_object = top_object;
    ...
```

The key part is creating an instance of the concrete class and then passing it to an instance of `lmx::c_xml_reader`. The rest of the unmarshaling operation is the same as described in 3.2 - Unmarshaling (advanced forms).

3.2.2 - Unmarshalling Multiple XML Instances from the Same Source

If you wish to read multiple XML instances from the same input source you can create a single low-level reader (such as `lmx::c_read_file`, `c_read_memory` and `c_read_string`) and use that to construct multiple instances of `c_xml_reader` from which to read each new XML instance. For example:

```
// Setup the input source
lmx::c_read_file file_reader( file_in );

for(;;)
{
    // Create new c_xml_reader to read from the input source
    lmx::c_xml_reader xml_reader( file_reader );

    // Read the XML
    lmx::elmx_error lmx_error;
    c_MyElement item( xml_reader, &lmx_error );

    // An error will be reported if there is no input XML
    if( lmx_error != lmx::ELMX_OK )
        break;

    // TODO: Process data in 'item'
}
```

See also 3.5.1 - Marshalling Multiple XML Instances to the Same Destination.

3.2.3 - Checking for trailing non-XML material

If you wish to check that your unmarshalled XML does not have non-valid XML end matter following it, you can use the `c_xml_reader::is_xml_end()` method.

```
lmx::c_xml_reader_string reader( xml_message );

lmx::elmx_error error;
c_generated_xsd_root_class top_object( reader, &error );

if( error == lmx::ELMX_OK && reader.is_xml_end() )
{
    ...
```

3.2.4 - Unmarshalling Schema Fragments

Typically an entry point for initiating an unmarshal process corresponds to a global element definition. If you wish to unmarshal an item that corresponds to a global type you need to use the `unmarshal_partial` template functions located in `lmxparse.h`. These have the prototypes:

```
template< class T >
elmx_error unmarshal_partial(
        c_xml_reader & ar_reader,
        T *ap_item,
        const s_ns_map ac_ns_map[],
        const std::string &ar_namespace,
        const std::string &ar_local_name );
```

```
template< class T >
elmx_error unmarshal_partial(
        T *ap_item,
        const char ac_file_name[],
        const s_ns_map ac_ns_map[],
        const std::string &ar_namespace,
        const std::string &ar_local_name );

template< class T >
elmx_error unmarshal_partial(
        T *ap_item,
        const wchar_t ac_file_name[],
        const s_ns_map ac_ns_map[],
        const std::string &ar_namespace,
        const std::string &ar_local_name );

template< class T >
elmx_error unmarshal_partial(
        T *ap_item,
        const char *ap_memory,
        size_t a_memory_size,
        const s_ns_map ac_ns_map[],
        const std::string &ar_namespace,
        const std::string &ar_local_name );

template< class T >
elmx_error unmarshal_partial(
        T *ap_item,
        const std::string &ar_string,
        const s_ns_map ac_ns_map[],
        const std::string &ar_namespace,
        const std::string &ar_local_name )
```

The first two read XML from a file, the third from an array of bytes, the fourth from a std::string and the last one from an `lmx::c_xml_reader` object.

The parameters have the following meaning:

ar_reader
: A reference to an LMX reader object.

ap_item
: A pointer to the object that is will receive the unmarshalled data.

ac_file_name
: The name of a file to be parsed.

ap_memory
: A pointer to memory containing XML to be parsed.

a_memory_size
: The number of bytes pointer to be ap_memory.

ar_string
: A std::string containing XML to be parsed.

ac_ns_map
: The generated namespace map for the schema to be unmarshalled. It can be found by looking in the generated `.h` file and has the name `ns_map_reader`. You may need to include the C++ namespace with the variable name, e.g. `my_cpp_namespace::ns_map_reader`.

ar_namespace
: The XML namespace of the root element that is to be unmarshalled. If there is no namespace, set this to `""`.

ar_local_name
: The local part of the name of the root element to be unmarshalled, without any namespace or namespace prefix information.

An example usage is:

```
c_my_object   my_object;
lmx::elmx_error error = lmx::unmarshal_partial(
        "myFile.xml",
        &my_object,
        my_cpp_namespace::ns_map_reader,
        "http://myDomain.com",    // or "" if no namespace
        "localName" );
```

3.3 - Unmarshalling a Sub-element within an XML Instance

If you wish to only unmarshal a particular element within an XML instance you can use the `unmarshal_find` template function located in `lmxparse.h`. This is available in the following forms:

```
template< class T >
bool /*is_found*/ unmarshal_find(
        c_xml_reader & ar_reader,
        elmx_error * ap_error,
        T * ap_item,
        const s_ns_map ac_ns_map[],
        const std::string & ar_sought_namespace,
        const std::string & ar_sought_local_name );

template< class T >
bool /*is_found*/ unmarshal_find(
        elmx_error * ap_error,
        T * ap_item,
        const char ac_file_name[],
        const s_ns_map ac_ns_map[],
        const std::string & ar_namespace,
        const std::string & ar_local_name );

template< class T >
bool /*is_found*/ unmarshal_find(
        elmx_error * ap_error,
        T * ap_item,
        const wchar_t ac_file_name[],
        const s_ns_map ac_ns_map[],
        const std::string & ar_namespace,
        const std::string & ar_local_name );

template< class T >
bool /*is_found*/ unmarshal_find(
        elmx_error * ap_error,
        T * ap_item,
        const char * ap_memory,
        size_t a_memory_size,
        const s_ns_map ac_ns_map[],
        const std::string & ar_namespace,
        const std::string & ar_local_name );

template< class T >
bool /*is_found*/ unmarshal_find(
```

```
        elmx_error * ap_error,
        T * ap_item,
        const std::string & ar_string,
        const s_ns_map ac_ns_map[],
        const std::string & ar_namespace,
        const std::string & ar_local_name );
```

The parameters have the following meaning:

ar_reader
: A reference to an LMX reader object.

ap_error
: A pointer to where any error code is to be placed when unmarshalling is completed.

ap_item
: A pointer to the object that is will receive the unmarshalled data.

ac_file_name
: The name of a file to be parsed.

ap_memory
: A pointer to memory containing XML to be parsed.

a_memory_size
: The number of bytes pointer to be ap_memory.

ar_string
: A std::string containing XML to be parsed.

ac_ns_map
: The generated namespace map for the schema to be unmarshalled. It can be found by looking in the generated `.h` file and has the name `ns_map_reader`. You may need to include the C++ namespace with the variable name, e.g. `my_cpp_namespace::ns_map_reader`.

ar_namespace
: The XML namespace of the root element that is to be unmarshalled. If there is no namespace, set this to `""`.

ar_local_name
: The local name of the root element to be unmarshalled, without any namespace or namespace prefix information.

The function returns `true` if the specified element was found and `false` if not.

An example usage is:

```
c_my_object  my_object;
lmx::elmx_error  error;
bool is_found = lmx::unmarshal_find(
        &error,
        &my_object,
        "myfile.xml",
        my_cpp_namespace::ns_map_reader,
        "http://myDomain.com",    // or "" if no namespace
        "localName" );
if( is_found && error != lmx::ELMX_OK )
{...}
```

3.4 - Unmarshalling Multiple Sub-elements within an XML Instance

If you wish to only unmarshal a subset of elements within an XML instance you can use the `unmarshal_select` functions located in `lmxselected.h`. These have the prototypes:

```
bool /*is_complete*/ unmarshal_select(
        c_xml_reader & ar_reader,
        lmx::elmx_error * ap_error,
        const lmx::s_ns_map ac_ns_map[],
        lmx::tc_elements_to_select ac_elements_to_select,
        lmx::c_selected_element * ap_handler );

bool /*is_complete*/ unmarshal_select(
        lmx::elmx_error * ap_error,
        const char ac_file_name[],
        const lmx::s_ns_map ac_ns_map[],
        lmx::tc_elements_to_select ac_elements_to_select,
        lmx::c_selected_element * ap_handler );

#if LMX_WANT_WIDE_FILE_NAMES == 1
bool /*is_complete*/ unmarshal_select(
        lmx::elmx_error * ap_error,
        const wchar_t ac_file_name[],
        const lmx::s_ns_map ac_ns_map[],
        lmx::tc_elements_to_select ac_elements_to_select,
        lmx::c_selected_element * ap_handler );
#endif

bool /*is_complete*/ unmarshal_select(
        lmx::elmx_error * ap_error,
        const char * ap_memory,
        size_t a_memory_size,
        const lmx::s_ns_map ac_ns_map[],
        lmx::tc_elements_to_select ac_elements_to_select,
        lmx::c_selected_element * ap_handler );

bool /*is_complete*/ unmarshal_select(
        lmx::elmx_error * ap_error,
        const std::string & ar_string,
        const lmx::s_ns_map ac_ns_map[],
        lmx::tc_elements_to_select ac_elements_to_select,
        lmx::c_selected_element * ap_handler );
```

The parameters have the following meaning:

ar_reader
: A reference to an LMX reader object.

ap_error
: A pointer to where any error code is to be placed when unmarshalling is completed.

ac_file_name
: The name of a file to be parsed.

ap_memory
: A pointer to memory containing XML to be parsed.

a_memory_size
: The number of bytes pointer to be ap_memory.

ar_string

A std::string containing XML to be parsed.

ac_ns_map
: The generated namespace map for the schema to be unmarshalled. It can be found by looking in the generated `.h` file and has the name `ns_map_reader`. You may need to include the C++ namespace with the variable name, e.g. `my_cpp_namespace::ns_map_reader`.

ac_elements_to_select
: An array containing the namespaces and local names of the elements being selected. The final entry in the array is marked by both fields being set to NULL.

ap_handler
: A pointer to a base class of type `lmx::c_selected_element` that is called when a selected element is found. This is described further below.

The function returns `true` if the operation was complete and `false` if not.

`lmx::tc_elements_to_select` is defined as:

```
typedef struct s_elements_to_select
{
    const char *p_namespace;
    const char *p_local_name;
} tc_elements_to_select[];    // Last member marked by NULL values
```

`lmx::c_selected_element` is defined as:

```
class c_selected_element
{
public:
    enum elmx_unmarshal_select_result { E_STOP, E_PARSED, E_SKIP };

    virtual elmx_unmarshal_select_result element_found(
        c_xml_reader & ar_reader,
        elmx_error *ap_error,
        const std::string &ar_sought_namespace,
        const std::string &ar_sought_local_name,
        const std::string &ar_full_element_name ) = 0;
};
```

To use `unmarshal_select()` you define a class that derives from `lmx::c_selected_element` and implements an `element_found()` virtual method.

You then give the `unmarshal_select()` function a list of element names that you want to process in an instance of the `lmx::tc_elements_to_select` array.

Each time `unmarshal_select()` finds one of the elements you asked for it calls the `lmx::c_selected_element::element_found()` virtual method and you can decide how to process it.

`lmx::c_selected_element::element_found()` returns either `lmx::c_selected_element::E_PARSED` to indicate that it parsed the element, `lmx::c_selected_element::E_SKIP` to indicate that it did not parse the element and the `unmarshal_select()` function should skip over it, or `lmx::c_selected_element::E_STOP` to indicate that parsing should be stopped.

An example usage of `unmarshal_select()` is as follows:

```
#include "selected.h"
#include "lmxparse.h"
#include <fstream>
#include <sstream>
#include <iostream>

#include "lmxselected.h"

// 1. Setup an array with a list of elements that we are interested in.
// Need namespace and local name.  The end of the list is marked by all
// NULLs.
lmx::tc_elements_to_select mc_elements_to_select = {
    { "http://xml2cpp.com/selected.xsd", "stat" },
    { 0, 0 } };

// 2. Define a class that contains a callback that will be called when the
// sought element is found.  This is derived from lmx::c_selected_element.
// This example is intended to just find the max and total values of the
// input.
class c_selected : public lmx::c_selected_element
{
private:
    int count;
    int max_height;
    int total_height;
    int max_width;
    int total_width;

public:
    c_selected() : count( 0 ), max_height( 0 ),
                total_height( 0 ), max_width( 0 ), total_width( 0 )
    {}

    // 3. This is the virtual function that is called when the element is found.
    virtual elmx_unmarshal_select_result
                    element_found(
                            lmx::c_xml_reader & ar_reader,
                            lmx::elmx_error *ap_error,
                            const std::string &ar_sought_namespace,
                            const std::string &ar_sought_local_name,
                            const std::string &ar_full_element_name );

    void print( std::wostream &ar_os ) const
    {
        ar_os << L"Max Height: " << max_height<< L"\n" <<
                L"Total Height: " << total_height<< L"\n" <<
                L"Max Width: " << max_width<< L"\n" <<
                L"Total Width: " << total_width << L"\n";
    }
};

std::wostream & operator << ( std::wostream &ar_os, const c_selected &ar_selected )
{
    ar_selected.print( ar_os );
    return ar_os;
}

// 4. The implementation of the virtual function that is called when the
// element is found.
c_selected::elmx_unmarshal_select_result
        c_selected::element_found(
                lmx::c_xml_reader & ar_reader,
                lmx::elmx_error *ap_error,
```

```
                const std::string &ar_sought_namespace,
                const std::string &ar_sought_local_name,
                const std::string &ar_full_element_name )
{
    // 5. Let's say we skip the second occurence of the element
    if( ++count == 2 )
        return c_selected_element::E_SKIP;

    // 6. ...and stop at the 5th occurence
    else if( count >= 5 )
        return c_selected_element::E_STOP;

    // 7. Unmarshal what we are interested in
    c_statType l_stat;
    *ap_error = l_stat.unmarshal( ar_reader, ar_full_element_name );

    // 8. And if everything is all right, do something with it
    if( *ap_error == lmx::ELMX_OK )
    {
        std::wcout << L"An element: Name=" << l_stat.get_name() <<
                    L", Height=" << l_stat.get_height() <<
                    L", Width=" << l_stat.get_width() << L"\n";

        if( l_stat.get_height() > max_height )
            max_height = l_stat.get_height();
        total_height += l_stat.get_height();
        if( l_stat.get_width() > max_width )
            max_width = l_stat.get_width();
        total_width += l_stat.get_width();
    }

    // 9. Say that we parsed the XML
    return c_selected_element::E_PARSED;
}

// 10. Now to actually use what we have written...
int test_it( const char ac_file_in[] )
{
    // 11. Create an instance of our callback class.
    c_selected l_selected;

    // 12. Create an LMX reader to read XML from a file
    lmx::c_xml_reader_file l_reader( ac_file_in );
    if( ! l_reader.is_open() )
        return 1;

    // 13. We need to be able to return an LMX error code
    lmx::elmx_error l_error;

    // 14. Actually call the LMX selective unmarsharshalling functionality
    if( lmx::unmarshal_select(
                &l_error,               // Place to store any LMX error code
                ac_file_in,             // XML file name
                ns_map_reader,          // List of namespaces relevant to
                                        // this XML schema (created by LMX)
                mc_elements_to_select,  // List of sought element names
                &l_selected ) )         // Object to be called when elements
                                        // found
        std::wcout << L"All the XML was parsed\n";
    else
        std::wcout << L"XML parsed was terminated before the end of the input\n";

    // 15. Check if everything went OK
```

```
    if( l_error != lmx::ELMX_OK )
    {
        std::wcout << L"Unable to unmarshal " << lmx::convert( ac_file_in ) << L"\n";

        return 2;
    }

    // 16. Print out the callback classes collected results
    std::wcout << l_selected;

    return 0;
}
```

3.5 - Marshaling (advanced forms)

2.4 - Marshaling (simple form) describes the simple method for marshaling object content direct to files and memory. In certain situations more flexibility is required, and the following more advanced techniques may be required.

Marshaling is the process converting the C++ object content into XML data. An advanced section of marshaling code for writing the XML to a string looks as follows:

```
c_generated_xsd_root_class top_object;

// ... Populate and manipulate top_object ...

std::ostringstream sos;

lmx::c_xml_writer writer( sos );

top_object.marshal( writer );

std::string string_out( sos.str() );
```

Breaking this down into its component parts, we first create an instance of the top-level class generated by the LMX code generator. (See the description in section 2.3 - Unmarshaling (simple form) on the name given to the top level class.) This is populated either by interacting with the object's methods (see 2.5 - Accesing the Data in the Generated Classes), performing an unmarshaling operation as described in 2.3 - Unmarshaling (simple form) or a combination of the two.

The first part of the marshaling operation is to create an instance of a class that is derived from `std::ostream`. This is where the output will be written. The line below uses the class derived from `std::ostream` by the C++ library for outputting to a string, but the line `std::ofstream sos;` could equally be used for outputting to a file.

```
std::ostringstream sos;
```

Now create a low-level LMX XML writer object of type `lmx::c_xml_writer`, telling it to write using the `sos` object.

```
lmx::c_xml_writer writer( sos );
```

Tell the generated top-level object to marshal itself using the `writer` object by calling the top-level object's `marshal` method and giving it a reference to the `writer` low-level XML writer object:

```
top_object.marshal( writer );
```

Finally, for when writing to a string, get the output into a form where it is more readily usable:

```
std::string string_out( sos.str() );
```

`string_out` is then a `std::string` object containing the marshaled XML.

The equivalent code for writing to a file is as follows:

```
c_generated_xsd_root_class top_object;

// Open a file using the C++ class derived from ostream
// for writing to a file
std::ofstream fos( "c:\\myfile.xml" );

// If the file is opened successfully
if( fos.is_open() )
{
    // Create an instance of an LMX writer class,
    // giving it a reference to the opened file's
    // stream
    lmx::c_xml_writer writer( fos );

    // Tell the top_object to marshal its contents
    // using the LMX writer class
    top_object.marshal( writer );
}
```

At the end of the above operations, the file `c:\myfile.xml` will contain the marshaled XML.

3.5.1 - Marshalling Multiple XML Instances to the Same Destination

If you wish to marshal multiple XML instances to the same output destination you can create a single C++ output stream and use that to construct multiple instances of `c_xml_writer` with which to write each new XML instance. For example:

```
// Setup output destination
std::ofstream fos( file_out );

if( fos.is_open() )
{
    c_MyElement * p_item;

    while( (p_item = get_item_data()) != 0 )
    {
        // Create a new c_xml_writer to the output destination
        // (The 0 for the t_writer_options prevents the XMLDecl being output)
        lmx::c_xml_writer xml_writer( fos, 0 );

        // Write out the XML
        p_item->marshal( xml_writer );
    }
}
```

See also 3.2.2 - Unmarshalling Multiple XML Instances from the Same Source.

3.6 - Selecting the Input File Type (XSD, WSDL, DTD)

LMX can parse both W3C Schema (XSD) and XML (external) DTD files. LMX can also locate and parse W3C Schema definitions embedded in WSDL files. The parsing that LMX performs depends on the file extension of the base schema file.

If the file extension is '`.wsdl`', LMX with attempt to locate a schema within a WSDL file and parse that. Note that LMX will only extract the first schema within a WSDL file. If multiple schema are present in a WSDL file then it is necessary to manually extract the second and subsequent schemas into separate schema files prior to parsing.

If the file extension is '`.dtd`', LMX will assume the file is an XML external DTD. When parsing DTD definitions, the Imported files window has no effect. Files names referenced as external entities within a DTD are opened relative to file location of the DTD file.

All other file extensions, including '`.xsd`' are assumed to be W3C Schema definitions.

3.7 - The Command-line and Configuration File Format

LMX and WinLMX can be configured using command-line flags and configuration files. The contents of a configuration file is simply a number of command-line flags (See 3.8 - Command-line Flags). The layout of the flags in the file (whether on the same line or different lines etc) is not significant, although one command option per line is suggested. Comments can be included in a configuration file using a '#' character. Any characters after the '#' character until the end of the line are ignored. For example:

```
# This is my LMX config file
# Created: today
# By: Me
-cpplines 2000
MyProject.xsd     # The schema for our project
MyProjectXML      # The root of the output files
```

When a set of command-line arguments are first received by either tool, it is subject to some special processing:

- If a single command-line argument is present, and it ends with '.lmxprj' it is assumed to be the name of a configuration file, and the tool reads and processes the file.
- In the case of the command-line tool, if a single argument is present, but it does not end in '.lmxprj', the tool tries appending '.lmxprj' to the end of the name and tests whether a file of the resultant name exists. If it does, the tool treats the resultant file as a configuration file and executes its contents. This allows configuration files to be associated with particular schema files.

3.8 - Command-line Flags

The LMX code generator understands the following flags:-

-#define YYY
: Outputs the line `#define YYY` in the generated .h file before the inclusion of `lmxuser.h`. This

allows configuration of the `lmxuser.h` file. If the `YYY` field includes spaces, place it in `"` marks for correct command line argument parsing. For example, `-#define "LMX_WANT_DLL 1"`

-alt-lmxuser YYY

Specifies that in place of line `#include "lmxuser.h"` in the generated .h files, the line `#include "YYY"` should be used. If you would like to use the `#include <YYY>` form, then include the <> brackets as part of the parameter, e.g. do `-alt-lmxuser <YYY>`. This provides an alternative way of configuring LMX. It is assumed that the `lmxuser.h` file is included via a `#include` in the specified alternative lmxuser file.

-alt-xml-reader YYY

Changes the XML reader class that the unmarshal code uses to read XML. The default XML reader class is `lmx::c_xml_reader`. Typically this flag would be used to specify a class that derives from `lmx::c_xml_reader` and the derived class would contain extra data and methods to be used in conjunction with custom event handlers during the unmarshal process.

-alt-xml-writer YYY

Changes the XML writer class that the marshal code uses to write XML. The default XML writer class is `lmx::c_xml_writer`. Typically this flag would be used to specify a class that derives from `lmx::c_xml_writer` and the derived class would contain extra data and methods to be used in conjunction with custom event handlers during the marshal process.

-no-assign

Switches off the generation of operator = and copy constructor code. The code generated uses the construct and swap pattern for exception safety. This can generate a lot of code and if the assignment operator is not needed in your code you can have a smaller executable by specifying this flag.

-autover

This option allows the generated code to support limited automatic versioning of your schema. If this option is selected, the parser will allow unknown elements to appear before an end tag. This allows new elements to be added to a schema in subsequent versions, but still allow old installations of the code to operate successfully. Note that selecting this option results in non-standard schema behavior.

-autover-subst-groups

This option allows the generated code to support automatic versioning of substitution groups. With this option, if an unknown element is encountered in the place of a mandatory substitution group member, then the element is skipped, the element is recorded as unknown, and no error is generated. Without this option being specified an error is reported under these circumstances.

-autover2

This option allows the generated code to ignore unknown XML elements encountered at any point during unmarshalling. This non-standard feature offers an easy way to make your schemas extensible. See also the `-autover` flag that enables a less permissive form of extensibility.

-braces-indent

Causes the braces ({}) to be indented.

-build YYY

Tells the code generator to generate code for the project described in configuration file YYY and return. A configuration file may have multiple -build arguments allowing multiple projects to be built from a single configuration file.

-cd YYY
: Changes the current working directory to YYY. This allows the configuration file to specify relative file names rather than absolute file names.

-no-check-code
: Disables generation of the check() methods. See 2.5.15 - Run-time Checking.

-check-is-occurs-ok-on-marshal
: Adds code to the marshal methods to check that `is_occurs_ok()` returns `true`. The `lmx::ELMX_OCCURRENCE_ERROR` error is returned if `is_occurs_ok()` returns `false`. By default this check is not done in order to maximize performance.

-no-choice-enum-vals
: Disables explicitly assigning numerical values to each enum when defining choice enums.

-class-base CLASS BASE
: Specifies that the generated class named CLASS should be generated such that it derives from BASE. Note the CLASS field should not include the class name prefix. Thus by default the option `-class-base MyClass MyBase` will generate code of the form `class c_MyClass : public MyBase`.

-cns YYY
: Sets the C++ namespace to YYY. In other words, the generated code is wrapped in a `namespace YYY { ... }` statement. By default the C++ namespace is derived from the XML namespace prefix assigned to the primary schema's target namespace. Nested C++ namespaces can also be specified. To do this use the form `-cns YYY::ZZZ`.

-no-cns
: Indicates that no C++ namespace statements should be generated, even if a default namespace name can be derived from the XML schema.

-no-container-ops
: By default items that can have multiple instances will have container operations generated which allow you to insert, delete or clear members of the container. Specifying this option disables the generation of these methods.

-no-convenience
: Switches off the generation of convenience functions that allow direct marshaling and unmarshaling from files and strings.

-no-convenience-xs-any
: Switches off the generation of convenience functions that allow direct unmarshaling from content retrieved `xs:any` instances.

-cpp-compiler YYY
: Indicates that code should be generated that is specific to the C++ compiler identified by YYY. The only compiler that requires special support is the IBM Visual Age compiler. This is identified by setting YYY to 'VisualAge'.

-cpp-per-class

Tells LMX to store the code for each class in a separate .h/.cpp file pair. -cpplines has no effect when this option is specified. The -multi-file flag has the same effect. If neither -cpp-per-class or -cpp-per-schema is specified, then a single .h and a single .cpp file are generated.

-cpp-per-schema
: Tells LMX to generate one .cpp file for each schema. Note that the generated code for included schemas is included in the same .cpp as the schema doing the including. This mode generates a single .h file. -cpplines has no effect when this option is specified. If neither -cpp-per-class or -cpp-per-schema is specified, then a single .h and a single .cpp file are generated.

-cpplines YYY
: Some schemas can generate a large number of lines of code. This may cause a compiler problems if all the lines are stored in a single file. This flag allows you to control the number of lines that are stored in each `.cpp` file. During code generation, before the methods and data for a class is output, LMX tests whether the number of lines in the `.cpp` file exceeds the value specified by `YYY`. If it does, LMX closes the current file and starts outputting code to a new file. Files are differentiated by appending a number to the end of the file name root; e.g. `my_xsd_code2.cpp`.

-doc-in-h
: Causes the `xs:documentation` information from the schemas to be included in the generated `.h` files.

-doc-in-html
: Causes the `xs:documentation` information from the schemas to be included in the generated HTML documentation.

-no-dynamic-cast
: Forces use of a static_cast rather than a dynamic_cast when a down cast is required. This is less safe, but may be more efficient and permits use of compilers that don't support C++'s RTTI.

-enum-options-above
: Controls where comments describing enum options are generated in relation to the methods that use the enums. By default, the comments are output below the methods. This option outputs the comments above the methods.

-enum-to-string
: Enables generation of `chosen_to_string()` methods in `xs:choice` classes and `all_to_string()` methods in `xs:all` classes that convert the enumerations used to indicate the applicable elements present to string values.

-enums-global
: Makes the C++ enums associated with schema xs:enumerations be all coalesced into a single global C++ enum. This is the default behavior.

-enums-local
: Generate separate C++ enums for each element or attribute having xs:enumeration facets within the relevant class. This mode is deprecated and it is recommended to use -enums-per-type instead.

-enums-per-type
: Generate separate C++ enums for each type that has xs:enumeration facets defined in the schema. C++ enums associated with global schema types will be in C++ global space, and C++ enums

associated with local schema elements and attributes will be in the relevant C++ class.

-eq
: Enables generation of operator ==() and operator !=() methods.

-error-fast
: Normally when an error is detected LMX writes code that calls a common error handler method giving you the opportunity to intercept the error and implement custom error handling. This also provides a convenient place to put a breakpoint when debugging. When the `-error-fast` is specified the error code is returned immediately without calling the handler. This results in faster, more compact code, but does not have the other benefits.

-no-ev
: If a type such as a string is enumerated, by default LMX will generate code checks that a read XML document matches one of the enumerations. Setting this flag turns off generation of such code.

-no-exceptions
: Indicates that try/catch blocks should not be generated in C++ functions.

-exec-pre YYY
: Specifies the name of a program or script that should be executed prior to starting to generate code into a file. The script is given the name of the file that is about to be generated. For example, if LMX is about to starting generating code into myproject.cpp, then, with this option, LMX will execute the shell command `YYY myproject.cpp`.

-exec-pre-all YYY
: Specifies the name of a program or script that should be executed prior to starting the code generation process. The script is given the Output File Base Name for the project. For example, if LMX is about to starting generating code for a project whose Out File Base name is myproject, then, with this option, LMX will execute the shell command `YYY myproject`.

-exec-post YYY
: Specifies the name of a program or script that should be executed when LMX has finished generating code into a file. The script is given the name of the file that has been generated. For example, if LMX finished generating code into myproject.h, then, with this option, LMX will execute the shell command `YYY myproject.h`.

-exec-post-all YYY
: Specifies the name of a program or script that should be executed when the code generation process is completed. The script is given the Output File Base Name for the project. For example, if LMX has completed code generation for a project whose Out File Base name is myproject, then, with this option, LMX will execute the shell command `YYY myproject`. Note that the program/script will not be executed if errors have occurred during the code generation process.

-exec-post-all-error YYY
: Similar to `-exec-post-all`, except that the program/script is run when an error occurs during code generation.

-expose-containers
: Causes LMX to generate methods that return references to the containers (e.g. instances of std::vector) used in the objects. This allows more fine-grained control of the containers. Specifying

`-no-container-ops` in conjunction with `-expose-containers` allows smaller code to be generated while still maintaining the ability to manipulate the contents of the containers when required.

-expose-storage
: Generates methods that expose the underlying objects that store the data. Direct use of these objects allows more generic handling of similar data in different parent objects.

-f YYY
: Tells the code generator to add the configuration information contained in the file YYY to the current configuration. The contents of the configuration file is the same as the command-line flags described here.

-file-ext-cpp YYY
: Set the extension for generated CPP files to YYY. The default is `.cpp`.

-file-ext-h YYY
: Set the extension for generated header files to YYY. The default is `.h`.

-fname-sep YYY
: Sets the character sequence used to separate parts of file names that are constructed from multiple parts during multiple file generation modes to YYY. The default is `-`.

-file-ext-snippets YYY
: By default a `.h` is checked to locate user specified snippets and snippet events. Using this option specifies the file extension of files that are used to store snippets. This allows snippets to be stored in a file other than the `.h` file, and means that the source file storing snippets will not be overwritten by LMX. The separate snippets file should include zero or more sections of snippets consisting of both the snippet markers and the additional user code.

-forcegen
: Normally LMX will not generate code if errors are found when parsing the schema. Setting this flag forces code generation even if errors are found. The results of setting this flag when errors are encountered are unpredictable.

-no-gen
: Tells LMX not to generate any code (even if there are no errors). This can be used when developing a schema and code generation is not required. LMX can be used in evaluation mode with this flag set without displaying any warning messages about licensing!

-no-gen-date
: Normally, when writing a file, LMX will include a comment specifying the date when the file was generated. This flag prevents this. This means that each time LMX is run it will generate the same output, minimizing diffs in version control systems.

-no-html
: Tells the tool NOT to create HTML documentation for the project.

-ignore-mixed-attr
: Ignores `mixed="true"` attributes in schemas.

-include-cpp YYY

Specifies that when generating a .cpp file, a line of the form `#include "YYY"` should be included in the file. If you would like to use the `#include <YYY>` form, then include the <> brackets as part of the parameter, e.g. do `-include-cpp <YYY>`. Multiple instances of this flag are allowed.

-include-h YYY

Specifies that when generating a .h file, a line of the form `#include "YYY"` should be included in the file. If you would like to use the `#include <YYY>` form, then include the <> brackets as part of the parameter, e.g. do `-include-h <YYY>`. Multiple instances of this flag are allowed.

-include-guard YYY

Sets the .h file include guard text to YYY rather that the default derived from the C++ namespace and name of the file.

-lmx-cns YYY

Tells the code generator to use the C++ namespace YYY:: when referring to the LMX parser and other similar items. This allows the user to define an alternative parser to that supplied with LMX.

-no-lmx-cns

Tells the code generator not to use a C++ namespace when referring to the LMX parser components and other similar items.

-lmx-include-path YYY

Allows the path to be specified when code is generated to `#include` the LMX header files. To avoid making assumptions about directory separator characters, LMX does not append a `/` or `\` character if no such character is present. Thus the flag sequence `-lmx-include-path YYY` will generate a line similar to `#include "YYYlmxuser.h"`, and in general it will be necessary to specify the directory separator as part of the parameter; for example `-lmx-include-path YYY/`. If the LMX files are to be considered system files (i.e. so that code similar to `#include <YYY/lmxuser.h>` is generated), then include the <> brackets as part of the parameter, e.g. `-lmx-include-path <YYY/>`. If you want the LMX headers to be considered as system header files, but don't want them in a sub-directory, do `-lmx-include-path <>`. This will generate lines similar to `#include <lmxuser.h>`. Note that if the directory name contains spaces, then, as for all such parameters, the parameter will need to be included in double quote marks, for example `-lmx-include-path "Y Y Y/"`. This is also the case if a system path contains spaces, requiring a flag sequence of the form `-lmx-include-path "<Y Y Y/>"`. In this case the `"` characters will be stripped by the command-line processor, and won't appear in the generated code.

-lmx-types-cns YYY

Tells the code generator to use the C++ namespace YYY:: when referring to the LMX defined data types. By default `lmx::` is used. Changing this allows different C++ classes and types to be used to represent the data objects in the XML schema.

-no-lmx-types-cns

Tells LMX not to generate a C++ namespace qualifier (such as lmx::) when referring to LMX defined types. This allows you to define and control the types and classes used by the generated code.

-lmxuser-defs

Outputs the line `#define LMX_WANT_USER_DEFS` in the generated .h file before the inclusion of `lmxuser.h`. This causes the file `lmxuser-defs.h` to be included by `lmxuser.h`, which allows configuration of `lmxuser.h`.

-lmxuser-defs-end
: Outputs the line `#define LMX_WANT_USER_DEFS_END` in the generated .h file before the inclusion of `lmxuser.h`. This causes the file `lmxuser-defs-end.h` to be included by `lmxuser.h`, which allows configuration of `lmxuser.h`.

-no-local-classes
: By default LMX generates the unmarshal helper class as a local class in the unmarshal method. If your compiler does not correctly support the access rights of local classes then use this flag to generate the unmarshal helper class outside the unmarshal method.

-local-enums
: Obsolete version of -enums-local flag.

-no-local-enums
: Obsolete version of -enums-global flag.

-makefile
: Generates a makefile fragment that contains makefile variables that specify the names of C++ files that have been generated. See 3.10 - Generating makefiles for more information.

-makefile-lib YYY
: When creating a makefile, specifies the name of the library containing the LMX Supporting Software Runtime Source Code. For example, `-makefile-lib lmx410` will include a `-llmx410` in the makefile. See 3.10 - Generating makefiles for more information.

-markers
: Indicates that comment based markers should be output in the code. This can facilitate post processing of generated code, for example by using the `-exec-post` flag.

-no-marshal
: Disables the generation of the methods associated with marshaling the C++ objects to XML. See also -no-unmarshal.

-max-else-ifs YYY
: Some sections of code have the form if() ... else if() ... else if() .... Some compilers have limitations on how much nesting of this form can be used. This option allows specification of the maximum number of else if() constructs. For constructs larger than YYY, the form if() ...; if() ...; is used. This is less efficient in some cases.

-max-inline YYY
: Some short methods can now be generated in-line with the class definition. This flag allows specification of the maximum number of lines that are allowed to be generated in-line. The default is 3.

-method-all_to_string YYY
: Set the name of the `all_to_string()` methods to YYY, which are generated when the `-enum-to-string` flag has been enabled.

-method-chosen_to_string YYY
: Set the name of the `chosen_to_string()` methods to YYY, which are generated when the `-enum-to-string` flag has been enabled.

-method-getchosen YYY
: Set the name of the `getchosen()` method to YYY.

-method-getid YYY
: Set the name of the `getid()` method to YYY.

-method-getorder YYY
: Set the name of the indexed `getorder( index )` method to YYY.

-method-has_id YYY
: Set the name of the `has_id()` method to YYY.

-method-is_occurs_ok YYY
: Set the name of the `is_occurs_ok()` method to YYY.

-method-marshal YYY
: Set the name of the `marshal(...)` method to YYY.

-method-reset YYY
: Set the name of the `reset()` method to YYY.

-method-sizeorder YYY
: Set the name of the `sizeorder()` method to YYY.

-method-unmarshal YYY
: Set the name of the `unmarshal(...)` method to YYY.

-micro-format PATH CLASS
: Specifies that the simpleType item identified by PATH should be represented by a micro format using C++ class CLASS. See 3.26 - Specifying Micro Formats for more information.

-multi-file
: Same as the -cpp-per-class flag.

-names-dir YYY
: Optionally specifies the directory that the names files are to be read from or saved to. This flag does not enable names file usage by itself. Such behavior is enabled by using the -names-in and -names-out flags. See 3.12 - Naming of Methods and Variables for more information.

-names-in
: Use a names file to set the names of C++ variables in the generated code. The name of the names file is derived from the schema file name by appending '.lmxnames-*in*.xml' to the file name. If the schema is named mySchema.xsd, the names file looked for will be mySchema.xsd.lmxnames-*in*.xml. See the section 3.12 - Naming of Methods and Variables for more information.

-names-out
: Output the C++ names used in the generation to a names file. The name of the names file is derived from the schema file name by appending '.lmxnames-*out*.xml' to the file name. If the schema is named mySchema.xsd, the names file looked for will be mySchema.xsd.lmxnames-*out*.xml. See the section 3.12 - Naming of Methods and Variables for more information.

-naming YYY

Specifies the naming convention to be used for names in the generated code. If `YYY` corresponds to `camel` LMX will attempt to convert base names generated in the code to CamelCase. If `YYY` corresponds to `underscore` LMX will attempt to make base names underscore separated. If `YYY` corresponds to `camel-no-xform` a set of name prefixes will be selected that are compatible with the CamelCase naming convention, but base names derived from the schemas will not be transformed to CamelCase. The absence of this option is equivalent to `YYY` being set to `underscore-no-xform`.

-narrow-strings

Generates suitable `#defines` to instruct C++ compiler to use narrow strings for storing Unicode strings.

-no-nested-classes

By default, C++ classes generated for types defined locally in a schema are nested in their parent class to echo the structure of the original schema. For deeply nested schema (such as Russian Doll design) this approach may not be desirable. This flag therefore allows you to specify that C++ classes should not be nested.

-ns-map-base YYY

Specifies an alternate prefix name for the `s_ns_map` instances containing namespace mapping information. This can be used to prevent name clashes when multiple instances of LMX generated code are in the same project. While this option is one solution to this problem, the preferred solution is to generate the code into separate C++ namespaces using the `-cns` flag.

-ns-prefix-map FROM TO

Specifies that if a schema defines a namespace prefix of `FROM`, the marshaling code should use a namespace prefix of `TO`. For example, if the schema uses the namespace prefix `tns`, LMX can be instructed to use the default namespace for the marshalled XML using the flag sequence:

```
-ns-prefix-map tns ""
```

It is the responsibility of the user to ensure that mapped namespace prefixes do not overlap. Also, FROM may specify the relevant namespace URI, such as `http://www.example.com/myschema.xsd`. For example, the flag sequence:

```
-ns-prefix-map http://www.example.com/myschema.xsd myns
```

would cause elements and attributes in the `http://www.example.com/myschema.xsd` namespace to use the namespace prefix `myns`.

-old-enums

Specifies that the old way of generating enum names should be used.

-output-defaults

Selecting this option results in the default values of attributes and elements being marshalled into the output if the value is not set by the user access functions. This can be useful when debugging. This is non-standard schema behavior.

-pattern-out PATTERN FUNCTION

Allows the specification of a custom output converter so that output can be formatted to conform to a specified xs:pattern definition. See 3.24 - Custom Pattern Facet Output Formatting for more information.

-no-polymorphic

Disable generation of polymorphic code for schema extensions and restrictions.

-prefix-anonymous YYY
: Specifies that all anonymous class names are prefixed by YYY.

-prefix-append YYY
: Set the prefix of the methods named append_{name}(...) to YYY{name}(...).

-prefix-appendenum YYY
: Set the prefix of the methods named appendenum_{name}(...) to YYY{name}(...).

-prefix-assign YYY
: Set the prefix of the methods named assign_{name}(...) to YYY{name}(...).

-prefix-back YYY
: Set the prefix of the methods named back_{name}(...) to YYY{name}(...).

-prefix-class YYY
: Set the prefix of class names to YYY.

-prefix-clear YYY
: Set the prefix of the methods named clear_{name}(...) to YYY{name}(...).

-prefix-container YYY
: Set the prefix of the methods named container_{name}(...) to YYY{name}(...).

-prefix-declash YYY
: Specifies a prefix to use to avoid name clashes when including multiple sets of generated code into a C++ project. We do not recommend using this flag. See 3.22.1 - Handling Multiple Independent Schemas for more information.

-prefix-default YYY
: Set the prefix of the default variables to YYY.

-prefix-delete YYY
: Set the prefix of the methods named delete_{name}(...) to YYY{name}(...).

-prefix-enum YYY
: Set the prefix of enum names to YYY.

-prefix-enumeration YYY
: Sets the prefix used for the C++ enumerated values associated with `xs:enumeration` facets to YYY. By default it is the same as that specified by the `-prefix-enum` flag. Note that for backwards compatibility reasons this flag has no effect when a single global C++ enum is used to represent `xs:enumeration` values.

-prefix-enumeration-name YYY
: Sets the prefix used for the C++ enum types associated with `xs:enumeration` facets to YYY. By default it is the same as that specified by the `-prefix-enum` flag. Note that for backwards compatibility reasons this flag has no effect when a single global C++ enum is used to represent `xs:enumeration` values.

-prefix-fraction-digits YYY
: Set the prefix of the fraction_digits_{name} variables to YYY.

-prefix-get YYY
: Set the prefix of the methods named get_{name}(...) to YYY{name}(...).

-prefix-getenum YYY
: Set the prefix of the methods named getenum_{name}(...) to YYY{name}(...).

-prefix-getorder YYY
: Set the prefix of the methods named getorder_{name}(...) to YYY{name}(...).

-prefix-in YYY
: Specifies that the methods to retrieve range information for use in C++11 range-based for loops be prefixed by YYY.

-prefix-inner YYY
: Specifies that all inner class names are prefixed by YYY.

-prefix-insert YYY
: Set the prefix of the methods named insert_{name}(...) to YYY{name}(...).

-prefix-insertenum YYY
: Set the prefix of the methods named insertenum_{name}(...) to YYY{name}(...).

-prefix-isnil YYY
: Set the prefix of the methods named isnil_{name}(...) to YYY{name}(...).

-prefix-isset YYY
: Set the prefix of the methods named isset_{name}(...) to YYY{name}(...).

-prefix-nil YYY
: Set the prefix of the nil_{name} variables to YYY.

-prefix-present YYY
: Set the prefix of the present_{name} variables to YYY.

-prefix-set YYY
: Set the prefix of the methods named set_{name}(...) to YYY{name}(...).

-prefix-setenum YYY
: Set the prefix of the methods named setenum_{name}(...) to YYY{name}(...).

-prefix-setnil YYY
: Set the prefix of the methods named setnil_{name}(...) to YYY{name}(...).

-prefix-size YYY
: Set the prefix of the methods named size_{name}(...) to YYY{name}(...).

-prefix-storage YYY
: Set the prefix of the main stotage variable (m_{name}) to YYY.

-prefix-unset YYY
: Set the prefix of the methods named unset_{name}(...) to YYY{name}(...).

-release-switch
: Uses a switch statement when releasing members of a choice. This results in more code, but is more portable.

-root YYY
: Specifies that if the root class is generated, which by default is derived from the main schema's namespace prefix, or called `c_root`, then it should be called `c_YYY`. Note that the `-prefix-class` flag can be used to change the prefix of the class name.

-root-from-xsd-tns-prefix
: Specifies that if the root class is generated, which by default is derived from the main schema's namespace prefix, or called `c_root`, then it's name should be derived from the namespace prefix associated with the schema's target namespace. Note that the `-prefix-class` flag can be used to change the prefix of the class name.

-root-from-xsd-file
: Specifies that if the root class is generated, which by default is derived from the main schema's namespace prefix, or called `c_root`, then it's name should be derived from the filename of the main schema. For example, if the main schema file is called `mySchema.xsd`, then the root class will be called `c_mySchema`. Note that the `-prefix-class` flag can be used to change the prefix of the class name.

-no-root-class
: Prevents generation of a root class in the case where there is more than one global element.

-snippets
: Causes LMX to output markers between which you can add snippets of your own code. See 3.14 - Augmenting Generated Classes With Your Own Code for more information.

-stdafx
: Causes the line `#include "stdafx.h"` to be written into the generated code.

-soap
: Generates additional code for when the generated results are used for SOAP transactions using LMX's `c_soap` SOAP wrapper template class. See 3.11 - Use with Web Services for more information.

-suffix-attribute YYY
: Specifies that all attribute names are suffixed by YYY.

-suffix-element YYY
: Specifies that all element names are suffixed by YYY.

-suffix-type YYY
: Specifies that all type names are suffixed by YYY.

-suffix-group YYY
: Specifies that all group names are suffixed by YYY.

-suffix-inner YYY
: Specifies that all inner class names are suffixed by YYY.

-suffix-anonymous YYY
: Specifies that all anonymous class names are suffixed by YYY.

-no-summary
: The error output of LMX uses the format used by Microsoft®* Visual Studio to allow easy integration. To allow standalone operation, LMX will produce a summary of the errors and warnings encountered (e.g. 0 error(s), 0 warning(s)). However, when used within an IDE, some IDEs produce this summary themselves. Setting this flag tells LMX not to produce an error summary so that only one error summary is produced. (*Microsoft is a Registered Trademark of Microsoft Corporation.)

-tframework
: Tells LMX to create a separate .cpp file containing test code. See 3.32 - Test Framework Generation for more information.

-no-unmarshal
: Disables the generation of the methods associated with unmarshaling XML to C++ objects. See also -no-marshal.

-user-base
: Selecting this option results in all generated classes ultimately deriving from a user implemented base class called `lmx::c_user_base`, defined in the `USER: USER_BASE` section of `lmxuser.h`. This allows you to store common information in the generated class objects.

-no-warn-ur-type
: Some texts recommend not using types specified as anyType or the ur-type. After all, this gives no useful information to the validator. Consequently, by default, LMX gives a warning when a type is not specified for an item (thus defaulting to the ur-type). This flag switches off the warning generated in this situation.

-werror
: Causes warnings detected by LMX to be treated as errors when the final program return code is evaluated.

Flags to support additional testing:

-eframework
: Similar to `-tframework`, but generates more testing code.

-test-mem
: When `-eframework` is used, this option generates code to ensure that memory is not lost when out of memory errors occur.

-no-tmain
: When `-eframework` is used, this option changes the name of the `main()` function. This permits the inclusion of the test code into a larger piece of code.

3.9 - Error Codes

The marshal and unmarshal functions return an error value of type `enum elmx_error`. The error codes (whose integer values are given in the brackets) have the following meanings:-

ELMX_OK (0)
: No errors - everything is OK.

ELMX_ALL_CARDINALITY_VIOLATION (2)
: The cardinality constraints of an xs:all construct have been violated. One possibility is that multiple instances of an element are present.

ELMX_ATTRIBUTE_ALREADY_READ (3)
: Two (or more) attributes with the same name appear in the same start tag.

ELMX_ATTRIBUTES_IN_ANONYMOUS_COMPOSITOR (4)
: The read XML contained attributes in the start tag of an anonymous compositor. This should not occur.

ELMX_BADLY_FORMED (5)
: The XML is badly formed.

ELMX_BAD_CHAR_ENCODING (6)
: Either the character encoding specified in the XML document, or the character encoding deduced from the opening sequence of bytes in the document are not supported by the parser.

ELMX_BAD_CHOICE (7)
: A bad choice has been selected when you wrote data to the generated objects. Most likely a choice was not selected when it should have been.

ELMX_BAD_COMMENT (8)
: The format of an XML comment (e.g. <!--...-->) is invalid.

ELMX_BAD_DTD (9)
: The DTD in an XML document could not be read correctly.

ELMX_BAD_ELEMENT_END (10)
: An end tag was either expected and was not found, or the name in the end tag did not match the name of the element in the start tag.

ELMX_BAD_END_OF_START_TAG (11)
: Failed to find a correct end of start tag when looked for.

ELMX_BAD_END_TAG (12)
: The format of an end tag is not syntactically correct from an XML viewpoint.

ELMX_BAD_XML_DECL (13)
: There is a problem with the XML decl spec. For example, it may be syntactically incorrect, in the wrong place or multiple decl specs may appear.

ELMX_BAD_XML_VERSION (14)
: The XML version is not known.

ELMX_CHAR_ENCODING_MISMATCH (15)
: The sequence of bytes read from a data source are not legal for the character encoding specified in the XML document.

ELMX_ELEMENT_NOT_FOUND (16)
: A sought element (typically the document element) was not found.

ELMX_ENTITY_NOT_FOUND (41)
: The XML contained an entity that does not have a definition.

ELMX_EXTERNAL_ENTITY (17)
: A reference to an external entity has been found. The parser does not resolve references to external entities.

ELMX_FRACTION_DIGITS_EXCEEDED (18)
: The maximum number of fraction digits specified for a value by a schema have been exceeded.

ELMX_LENGTH_TOO_LONG (19)
: The length of a string or binary value is too long compared to the constraint set in the schema.

ELMX_LENGTH_TOO_SHORT (20)
: The length of a string or binary value is too short compared to the constraint set in the schema.

ELMX_MANDATORY_ELEMENT_MISSING (21)
: A mandatory element (minOccurs != 0) was not found.

ELMX_NILLED_ELEMENT_NOT_EMPTY (40)
: The body of a nilled element was not empty.

ELMX_NO_FILE (1)
: Unable to open file.

ELMX_NO_PATTERN_MATCHED (22)
: The data value does not match any of the pattern facets specified for the item.

ELMX_NOT_WELL_FORMED (23)
: The document is not well formed in the XML sense.

ELMX_OCCURRENCE_ERROR (46)
: When marshalling, the number of occurrences of a element was not in the valid range.

ELMX_RECURSIVE_ENTITY_DEFINITION (24)
: An entity definition is recursive, and hence cannot be successfully expanded.

ELMX_REQUIRED_ATTRIBUTES_MISSING (25)
: One or more required attributes are missing.

ELMX_TOO_MANY_ITEMS (26)

The number of instances of an element exceeds the constraint set by the schema's relevant maxOccurs attribute.

ELMX_TOTAL_DIGITS_EXCEEDED (27)
: The maximum number of total digits specified for a value by the schema have been exceeded.

ELMX_TYPE_NOT_SET (45)
: An element that required its type to be specified using xsi:type did not have an xsi:type attribute.

ELMX_UNABLE_TO_READ_ATTRIBUTE_VALUE (28)
: Unable to read an attribute's value.

ELMX_UNABLE_TO_READ_ELEMENT_VALUE (29)
: Unable to read an element's value.

ELMX_UNDEFINED_ERROR (42)
: Used as an initialization value that can be later overwritten with a valid error code.

ELMX_UNEXPECTED_ALL_EVENT (30)
: Typically, an unknown element occurred in an xs:all construct.

ELMX_UNEXPECTED_ATTRIBUTE (39)
: An unknown attribute has been encountered.

ELMX_UNEXPECTED_ELEMENT_EVENT (31)
: Typically, an unknown element occurred in an xs:sequence or xs:choice construct.

ELMX_UNEXPECTED_EMPTY_ELEMENT (32)
: An element was empty when it should not have been.

ELMX_UNEXPECTED_ENTITY (33)
: An external entity was not declared as either PUBLIC or SYSTEM.

ELMX_UNEXPECTED_EOM (34)
: The end of the message (or file) was encountered before the XML content was successfully parsed.

ELMX_UNKNOWN_XSI_TYPE (44)
: The type declared for an element using xsi:type is unknown.

ELMX_USER_DEFINED_1 (1000) to ELMX_USER_DEFINED_9 (1008)
: Error codes reserved for use by the user. These can be used when errors are returned from snippet event handlers.

ELMX_VALUE_BAD_FORMAT (35)
: A value such as an integer, binary value, float and so on was not of a suitable format.

ELMX_VALUE_DOES_NOT_MATCH_FIXED_CONSTRAINT (43)
: The value of an element or attribute that is declared as fixed differs from the fixed value.

ELMX_VALUE_EXCEEDS_MAX (36)
: The value in the XML document is greater than the maximum specified by the schema.

ELMX_VALUE_EXCEEDS_MIN (37)
: The value in the XML document is less than the minimum specified by the schema.

ELMX_VALUE_NOT_ENUMERATED (38)
: The read value could not be matched against any of the allowed enumeration values.

3.10 - Generating makefiles

LMX has the ability to generate a makefile fragment suitable for inclusion into your own makefiles. This is invoked using the `-makefile` command-line flag. The generated makefile fragment will be named `<output root>.makefile`.

Additionally, if there is no file named `makefile` in the project output directory, then LMX will generate an example makefile that demonstrates a possible use of the generated makefile fragment. You can then modify this file according to your requirements as LMX will not overwrite it if it is already present.

An example of the generated makefile fragment is shown below. It contains a number of makefile variables specifying the files used in the project. Note that it does not include any actual build instructions.

```
MYPROJECTCPPS = \
    myproject.cpp
MYPROJECTTESTCPP = myproject-main.cpp
MYPROJECTOBJS = $(MYPROJECTCPPS:.cpp=.o)
MYPROJECTTESTOBJ = $(MYPROJECTTESTCPP:.cpp=.o)
MYPROJECTHS = $(MYPROJECTCPPS:.cpp=.h)
```

The expectation is that this makefile fragment will be included in another makefile written by the developer that contains the actual build instructions, for example:

```
include myproject.makefile

LMXLIB = -llmx # Modify using LMX -makefile-lib flag

.PHONY: all test clean

all: libmyproject.a

test: myproject-test

libmyproject.a: $(MYPROJECTOBJS)
     -rm libmyproject.a
     ar rcs libmyproject.a $(MYPROJECTOBJS)

myproject-test: $(MYPROJECTTESTOBJ) libmyproject.a
     $(CXX) -o myproject-test $(MYPROJECTTESTOBJ) libmyproject.a $(LMXLIB)

%.o: %.cpp $(MYPROJECTHS)
     $(CXX) -c $(CXXFLAGS) $(LMXCXXFLAGS) $(LMXFLAGS) $      <-o $@

clean:
     rm $(MYPROJECTOBJS)
```

You can use the `-makefile-lib` flag to specify the name of the library containing the LMX Supporting Software Runtime Source Code. For example, `-makefile-lib lmx710` will include a `-llmx710` in the

makefile.

Note that in order to build the test project, you will need to specify the `-tframework` command-line option, or select the corresponding checkbox in the GUI interfaces when running LMX so that the necessary test `.cpp` is generated.

3.11 - Use with Web Services

LMX supports a number of features that help developing web services. Specifically, LMX has a `c_soap` template class that can be used to assemble the various components of a SOAP based web service, such as the specific SOAP header, body and fault detail types.

LMX also has a `c_winhttp` class which provides HTTP client functionality for SOAP and REST style client operations on the Windows platform. (A similar HTTP component for Linux platforms is due to be included in a future release.)

Additionally LMX provides features for querying web service messages, such as that described in 3.28 - mustUnderstand / Comprehension Required.

3.11.1 - Handling SOAP Messages

To associate the various SOAP header, body and fault detail types that a SOAP messages can have, LMX provides the `c_soap` template class. This is defined in `lmxsoap.h`. The code for this is built into the binary versions of the LMX Supporting Software or available in source code form in the Professional edition. The example at http://codalogic.com/lmx/download/?lmx-soap-example.zip shows a number of ways of using the SOAP functionality.

The `c_soap` template takes up to three parameters, these being the class representing the SOAP bodies, the class representing the SOAP headers and the class representing the fault detail. It is thus defined as:

```
template< class Tbody, class Theader = c_soap_empty, class Tfault_detail = c_soap_empty >
class c_soap
{...}
```

For cases where a fault detail is required, but a header is not, a derivative of `c_soap`, called `c_soap_no_header`, is defined as follows:

```
template< class Tbody, class Tfault_detail = c_soap_empty >
class c_soap_no_header
{...}
```

Unless stated otherwise, anything that applies to `c_soap` also applies to `c_soap_no_header`.

These classes store `std::vector`s of pointers to the respective body and header types, and a separate fault class. The fault class stores a pointer to a fault detail object if required. The `bodies()` and `headers()` methods return a reference the relevant `std::vector` of pointers and by using `std::vector` operations such as `push_back()` and `at()` the overall SOAP message can be constructed or interrogated as required.

`c_soap` also contains a class for storing SOAP fault information, called `c_soap_fault`. This latter class contains a `std::vector` of pointers to objects of the `Tfault_detail` template argument type. `c_soap_fault` also contains the SOAP fault code, fault string and fault actor details if required, for which there are appropriate getter and setter methods, e.g. `get_fault_code()`, `set_fault_code()`, `get_fault_string()`, `set_fault_string()`, `get_fault_actor()`, and `set_fault_actor()`. As the fault actor is optional, it's presence can be tested for using the `isset_fault_actor()` method.

By default, the `c_soap` object deletes the objects that its `std::vector` containers point to when it is destructed. If this behavior is not required, then the `do_delete_on_destruct( false );` method should be called before the object is destructed. If it is desired to have the `c_soap` object delete some objects that are pointed to but not all, then the `std::vector` operations can be used to remove pointers to those objects that it is desired to preserve prior to destructing the `c_soap` object.

If it is desired to marshal the SOAP message specified by a constructed `c_soap` instance, then one of `c_soap`'s `marshal()` methods can be called. There are a number of such methods which allow marshaling to files, or memory for example.

Similarly, if a SOAP XML message is to be unmarshalled into the corresponding C++ objects, then one of `c_soap`'s `unmarshal()` methods can be called.

Finally, `c_soap` can also store a `SOAPAction`, accessible via the `get_soap_action()` and `set_soap_action()` methods (located in `c_soap_base`). While SOAPAction is not part of a SOAP message's XML contents, it can be convenient to group it together with the other SOAP information.

3.11.2 - HTTP Operations for Web Services

LMX includes the `c_winhttp` class (declared in `lmxwinhttp.h`) to provide HTTP client functionality for the Windows platform. A derivation of `c_winhttp` called `c_secure_winhttp` provides secure HTTPS client functionality. Unless stated otherwise, anything that applies to `c_winhttp` also applies to `c_secure_winhttp`.

To allow customization of this functionality to your particular needs, these are provided in source code form in both the Standard and Professional Editions of LMX. Similar HTTP/HTTPS client functionality for the Linux platform is planned for a future release.

LMX does not provide HTTP server side functionality as the expectation is that a regular HTTP server such as Apache or Microsoft IIS will provide this. LMX generated code can then be plugged into one of those architectures in an appropriate way. (However, please let us know if you have different requirements.)

LMX HTTP (and HTTPS) operations consist of two main parts; creating a connection to a server and then performing operations using the connection. The example mentioned above that is located at http://codalogic.com/lmx/download/?lmx-soap-example.zip also includes examples of using the `c_winhttp` class's functionality.

The server to connect to is specified using `c_winhttp`'s constructor, which is:

```
c_winhttp( const std::string &ar_host,
           size_t a_port = k_default_port,
           bool a_is_secure = false );
```

Or that of `c_secure_winhttp` who's constructor is:

```
c_secure_winhttp( const std::string &ar_host,
                  size_t a_port = k_default_https_port );
```

The `ar_host` argument contains the name of the host being contacted. For example, if the desired operation URI is `http://example.com:8080/my/soap/service` then `ar_host` should be set to `example.com`.

`c_winhttp` allows a number of different types of web service operation to be performed once a connection has been established. The following sections describe each of these operations.

3.11.2.1 - SOAP Message to Message Operations

Message to message SOAP operations are performed using the following method:

```
int soap_rpc( const std::string &ar_path,
              const tlmx_uri_string &ar_soap_action,
              const std::string &ar_request_xml,
              std::string *ap_response_xml );
```

This operation sends the XML encoded SOAP message contained in `ar_request_message` to the server, and places the reply into the `std::string` pointed to by `ap_response_xml`.

The `ar_path` argument contains the path part of the HTTP URI. For example, if the desired operation URI is `http://example.com:8080/my/soap/service` then `ar_path` should be set to `/my/soap/service`.

The `ar_soap_action` argument contains the `SOAPAction`.

The method returns `0` if the operation is successful and non-`0` otherwise. The `get_error_description( int a_error )` method can be used to convert an error code into a text based error description.

3.11.2.2 - SOAP Object to Object Operations

What LMX refers to as object to object SOAP operations are performed using the following method:

```
s_winhttp_error soap_rpc(
                    const std::string &ar_path,
                    const c_soap_base &ar_request,
                    c_soap_base *ap_response );
```

This operation marshals the objects aggregated by the `c_soap` object referenced by `ar_request` into an XML message, and sends this to the server. The XML message returned from the server is unmarshalled and placed in the `c_soap` object pointed to by `ap_response`.

The `SOAPAction` for the operation is retrieved from the `c_soap` object referenced by `ar_request`.

The `ar_path` argument has the same meaning as in the message to message operation case.

The method returns an instance of `s_winhttp_error`. As the method performs both LMX and HTTP operations, this is an aggregate object capturing the various possible error conditions. If `s_winhttp_error`'s `is_ok()` method returns `true` then the operation was successful. If the operation was unsuccessful, the operation phase at which the error occurred (e.g marshalling, HTTP transaction or unmarshalling) can be determined by inspecting `s_winhttp_error`'s `phase` member. Depending on the phase at which the error occurred, either the `lmx_error` or `http_error` member can be inspected to find out more specific information.

3.11.2.3 - Simple SOAP Operations

When a SOAP operation consists of only a single body part, without SOAP headers or faults, then a Simple SOAP Operation can be performed using the following template function:

```
template< class Tin, class Tout>
s_winhttp_error simple_soap(
                    const std::string &ar_host,
                    size_t a_port,
                    const std::string &ar_path,
                    const tlmx_uri_string &ar_soap_action,
                    Tin *ap_in,
                    Tout **app_out = LMXNULL,   // Allows an empty SOAP body in the reply
                    bool a_is_secure = false )
```

Or, if it is not necessary to change the default port, the following template function:

```
template< class Tin, class Tout>
s_winhttp_error simple_soap(
                    const std::string &ar_host,
                    const std::string &ar_path,
                    const tlmx_uri_string &ar_soap_action,
                    Tin *ap_in,
                    Tout **app_out = LMXNULL,   // Allows an empty SOAP body in the reply
                    bool a_is_secure = false )
```

This is *not* a member of `c_winhttp`, but uses its services.

In these functions, the `ar_host`, `a_port`, `ar_path`, `ar_soap_action`, and `a_is_secure` have the same meanings as described elsewhere in this section.

The `ap_in` argument takes a pointer to an object corresponding to the template argument `Tin` (whose code is generated by LMX) that, when marshaled, will form the body of the SOAP request.

If the operation is successful, the function creates an unmarshalled object of type template argument `Tout` on the heap. To be able to access this object, the calling function must have a pointer to a `Tout` object. A pointer to this pointer is then supplied to `simple_soap` so that `simple_soap` can set its value. Note that if no SOAP body is returned, the pointer to `Tout` in the calling function may be set to 0.

The return value of the function is an instance of `s_winhttp_error`, and is interpreted as described in the object to object operation case above.

3.11.2.4 - RESTful Operations

REST style operations can be performed using the following `c_winhttp` member method:

```
int rest_query( const std::string &ar_path,
                const std::string &ar_query,
                std::string *ap_response_xml );
```

If the URI for the REST operation was `http://example.com:8080/my/service?what=rest` then `ar_path` should be set to `/my/service` and `ar_query` should be set to `what=rest`. (The host is set when the `c_winhttp` object is constructed.)

The XML response to the query is placed in the `std::string` pointed to by `ap_response_xml`. This can subsequently be unmarshalled using LMX generated code.

If the method returns `0` then the operation was successful. A non-`0` value indicates an error. If an error has occurred, a description of the error can be obtained by calling the `get_error_description( int a_error )` method.

3.12 - Naming of Methods and Variables

LMX allows extensive customization of the variable and method names used in the generated code. There are two parts to setting the names and variables in LMX; setting the prefixes and setting the base names. Both of these operate independently.

The prefixes of methods and variables are configured using project configuration flags (also accessible via the Windows UI). For example, if you wish the functions that are named by default as `getenum_YYY()` to be `getEnumYYY()` then the project flag '`-prefix-getenum`' should be set to '`getEnum`', either directly or via the Windows UI. The prefixes set in this way have project wide scope. See the `-prefix-???` flags in section 3.8 - Command-line Flags for the set of prefixes that can be changed.

The base names of variables and methods (e.g. the `YYY` part in the examples above) can be automatically converted from the form specified in the schema to CamelCase or underscore_separated either via the GUIs or using the `-naming camel` or `-naming underscore` flag.

The base names of variables and methods can also be set by names files. LMX automatically performs name collision avoidance, but if a schema is updated there is a small chance for a name to be assigned to a different element/attribute in the new code to the one it was in the previous version. The names files address this problem. (Note that in many scenarios such naming issues will not be a problem between versions of a schema, so this feature need not be used in all cases.)

When the `-names-out` option is specified, LMX will generate a names file for each schema used in a project. By default, the name of the names file will be the name of the schema appended with '`.lmxnames-`*`out`*`.xml`'. For example, if the name of the schema file is '`mySchema.xsd`', the generated names file will be called '`mySchema.xsd.lmxnames-out.xml`'. To avoid accidental overwriting, the appended text on the names file needs to be modified to '`.lmxnames-`*`in`*`.xml`' before it can be used as an input names file. An input names file will be read (for example, when processing the new version of the schema) if the '`-names-in`' option is chosen.

Additionally, it is also possible to specify an alternative directory in which the names files are stored. This is done using the `-names-dir YYY` flag; where YYY is the name of the directory (of folder) storing the names files. This can allow the same schema to have different names files for different projects. In this case the name of a names file is made up of the directory specified by the `-names-dir` flag, and the base name of the schema file, suffixed by either the extension `.lmxnames-out.xml` or `.lmxnames-out.xml`. For example, if `-names-dir foo` is specified and the schema file name is `bar/mySchema.xsd`, then the in names file would be `foo/mySchema.xsd.lmxnames-in.xml`.

The names file also allows you to customize the names used in the code. If you are not able to use the WinLMX version of LMX, the easiest way to do this is to run the code generator with the '`-names-out`' option first, re-name the names file extension to '`.lmxnames-`*`in`*`.xml`', and edit the names in the file to your convenience. When the naming is complete, run the code generator with the '`-names-in`' option specified.

The schema for the names file is as follows:

```
<xs:schema targetNamespace="http://xml2cpp.com/lmx-customize.xsd"
                elementFormDefault="qualified"
                xmlns="http://xml2cpp.com/lmx-customize.xsd"
                xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:clt="http://codalogic.com/xsdtypes">

    <xs:element name='lmxCustomize'>
        <xs:complexType>
            <xs:sequence>
                <xs:element name="element" type="item"
                                minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="attribute" type="item"
                                minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="group" type="item"
                                minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="attributeGroup" type="item"
                                minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="type" type="item"
                                minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>

            <xs:attribute name="version" type="xs:token" default="1.0">
                <xs:annotation><xs:documentation>
                    Use major.minor format
                </xs:documentation></xs:annotation>
            </xs:attribute>

        </xs:complexType>
    </xs:element>


    <xs:complexType name="item">
        <xs:sequence>
            <xs:element name="element" type="item"
                                minOccurs="0" maxOccurs="unbounded">
                <xs:annotation><xs:documentation>
                    For naming child elements of complex type/complex
                    content.
                </xs:documentation></xs:annotation>
            </xs:element>

            <xs:element name="attribute" type="item"
                                minOccurs="0" maxOccurs="unbounded"/>
```

```
        </xs:sequence>

        <xs:attribute name="name" type="xs:string">
            <xs:annotation><xs:documentation>
                Optional.  The name of the item specified in the schema.
                It may be absent if it is an anonymous type.
            </xs:documentation></xs:annotation>
        </xs:attribute>

        <xs:attribute name="asVarName" type="clt:asciiString">
            <xs:annotation><xs:documentation>
                Optional - The name of the item when it is used as a
                variable's name, as in: 'c_type var_name;'  It must
                be locally unique, but need not be globally unique.
            </xs:documentation></xs:annotation>
        </xs:attribute>

        <xs:attribute name="asTypeName" type="clt:asciiString">
            <xs:annotation><xs:documentation>
                Optional.  The name of the item when it is used as a
                C++ type, e.g. 'c_type name;'  It must be globally unique.
            </xs:documentation></xs:annotation>
        </xs:attribute>

    </xs:complexType>
</xs:schema>
```

3.13 - Adapting LMX to Your Environment

In some cases you may find that the various classes and types included with LMX are not what you want. LMX has been designed so that it is easy to change the types and classes used.

The majority of configuration options are specified in the `lmxuser.h` file. Each user configuration section is marked with C++ comments of the form `USER: A_SECTION`.

There are a number of ways in which you can customize the features controlled by the `lmxuser.h` file. For example, you can:

1. Modify `lmxuser.h` directly. This is the least recommended method because it means if a new version of LMX is issued, then you may have to update a new copy of `lmxuser.h`, potentially introducing bugs through human error.
2. If only minor configuration is required, and the changes needed are of the more common type, the build environment (such as project files or make files) can arrange for the appropriate `#defines` to be set to get the desired configuration.
3. If more substantial configuration is required, then it is suggested that the various `#defines` are included in their own `.h` file and that `.h` file in turn includes `lmxuser.h`. You can then use LMX's `-include-h` flag (or "Advanced"->"Include Files in Generated.h Files..." menu option in WinLMX) to include the file in the generated code.
4. As an alternative to the previous option, you can include your configuration changes in the files called `lmxuser-def.h` and `lmxuser-defs-end.h`. `lmxuser.h` will pull in the file `lmxuser-def.h` if your project specifies the definition `LMX_WANT_USER_DEFS` and pull in `lmxuser-defs-end.h` if your project specifies `LMX_WANT_USER_DEFS_END`.

The last two methods are appropriate when complete sections of `lmxuser.h` need to be changed. It will be observed that each section is included in its own conditional compilation guard block. If a particular feature can not be changed using the `#define`s that LMX supports, then copy the entire section into the separate include file mentioned in the latter methods, and edit it accordingly. Then `#define` the appropriate guard value so that the default section in `lmxuser.h` is ignored.

3.13.1 - Modifying Schema Type to C++ Type Mapping

XML schema type to C++ type mapping is specified in the `USER: BASIC_TYPES` and `USER: COMPOUND_TYPES` sections of the lmxuser.h file. The type mapping is achieved using simple C++ `typedef` statements to avoid you having to learn additional LMX code-generator configuration options. Changing the types used is simply a matter of modifying the `typedef` statements. For example, if you wish to use your own date class, you could define a mapping of:

```
typedef c_my_date          tc_date;
```

The following table describes the relationship between the schema types, the C++ `typedef` name used to represent it, and the default C++ type that the `typedef` name is mapped to.

XSD Type	C++ Intermediate Type	Default C++ Type
boolean	bool	bool (This type can not be customized)
byte	tlmx_int8	signed char
short	tlmx_int16	short
int	tlmx_int32	int
long	tlmx_int64	For MSVC: __int64 For GCC: long long
unsignedByte	tlmx_uns8	unsigned char
unsignedShort	tlmx_uns16	unsigned short
unsignedInt	tlmx_uns32	unsigned int
unsignedLong	tlmx_uns64	For MSVC: unsigned __int64 For GCC: unsigned long long
integer, nonPositiveInteger, negativeInteger, nonNegativeInteger, positiveInteger	tc_big_int	c_big_int
decimal	tc_decimal	c_decimal
float	tlmx_float	float
double	tlmx_double	double
duration	tc_duration	c_duration
dateTime	tc_datetime	c_datetime
time	tc_time	c_time
date	tc_date	c_date
gYearMonth	tc_gyearmonth	c_gyearmonth
gYear	tc_gyear	c_gyear
gMonthDay	tc_gmonthday	c_gmonthday
gDay	tc_gday	c_gday
gMonth	tc_gmonth	c_gmonth
hexBinary, base64Binary	tc_binary	c_binary
string, normalizedString, language, token, Name, NCName, ID, IDREF, IDREFS, ENTITY, ENTITIES, NMTOKEN, NMTOKENS, NOTATION	tlmx_unicode_string	std::wstring
anyURI	tlmx_uri_string	std::wstring
QName	tc_qname	ct_qname<tlmx_uri_string>

Table 5: XSD to C++ Type Mappings

Similarly `tlmx_uri_string` controls the type of string used for URIs. By default this is mapped to `std::wstring`, but if you wish it to be mapped to `std::string` then arrange for the #define variable `LMX_NARROW_URI_STRINGS` to be 1. If `LMX_NARROW_URI_STRINGS` is not specified, then it is automatically given the same value as `LMX_NARROW_UNICODE_STRINGS`.

Note that by default `xs:float` is mapped to a C++ `float`. Such a single precision floating point number has about 7.5 decimal places of precision; a precision that can not be readily expressed using C++'s output

converters. Consequently, LMX outputs C++ `floats` using 7 decimal places of precision (as opposed to 8 places which typically results in sub-optimal formatting), thus losing the ability to express the entire precision of the type. Therefore, if you require the full precision of the `xs:float` type it is recommended that you map it to C++'s `double` type.

In addition to changing the C++ types used to represent the LMX types, LMX needs to know whether each integer type is a type native to your C++ compiler (e.g. VC++'s `__int64` is considered native), or a special class. This is done using the `USER: INT_NATIVE_FOREIGN` section of the `lmxuser.h` file. This section contains a number of macros specific to each integer type that either maps to a native value or a foreign value. You should change the mapping of these macros according to your platform requirements.

See also 3.23 - Pattern Facet Handling Customization.

3.13.1.1 - Choosing Between Wide and Narrow Unicode Strings

Of particular note is the mapping of `tlmx_unicode_string`. By default this is mapped to `std::wstring`. If you wish to use 'narrow' strings to store parsed strings, you can arrange your project C++ build configuration to define the #define variable `LMX_NARROW_UNICODE_STRINGS` to be 1. Alternatively, you can directly modify the mapping in `lmxuser.h` to be `std::string`. All other changes will be automatically taken care of by the C++ compiler. Or you can map `tlmx_unicode_string` to some other string class.

3.13.1.2 - Input/Output Converters

As part of changing the type mappings used, it may be necessary to define additional input and output converters. These are defined in the `USER: TEMPLATE_INPUT_CONVERTERS`, and `USER: TEMPLATE_OUTPUT_CONVERTERS` sections of lmxuser.h.

All types are converted from their string form to their type form via the templated function `v_to_o` (short for 'value_to_object'.). If your type (such as a date class) has an `operator =` member that can convert a std::string value to appropriate internal values, then the default `v_to_o` template function will take care of this and no additional work is required. If your class does not support this, then you should provide a template specialization of `v_to_o` that will do the conversion. Examples of such specializations are shown in the `USER: TEMPLATE_INPUT_CONVERTERS` section. One option for such conversion that eases parsing is to first convert the input to an appropriate LMX class, and then use the public members of that class to populate your class.

For output, the LMX code generator arranges for each parameter to be wrapped in an instance of an `as_xml` io manipulator. These manipulators are defined in the `USER: TEMPLATE_OUTPUT_CONVERTERS`. If your class or type has an io operator (i.e. <<) that generates data in a suitable form for output in an XML document, then the default template will be sufficient and no additional specialization is required. If this is not the case, then a specialization of the output template is required. For example, a specialization for your type might appear as:

```
template <>
std::ostream & operator << ( std::ostream & ar_os,
                             const c_as_xml<c_my_date> &ar_v )
{ ... }
```

Within the template specialization function, the quantity `ar_v.p` is a pointer to the item to be output. You can look at some of the other specializations to see how to write a specialization.

3.14 - Augmenting Generated Classes With Your Own Code

LMX allows you to augment the generated classes with your own code. Specifying the `-snippets` flag will cause LMX to output a pair of comments towards the beginning of each generated class in a .h file. The markers have the form:

```
//:snippet:start YYY
//:snippet:end YYY
```

where YYY is the name of the class in which the markers are included. You can add your own code between the markers. When code is re-generated, LMX first reads the original .h file and extracts the code between the snippet markers. The code is then put back when the new code is generated. Non-header code such as method definitions should be put in a separate .cpp file. You should not change the generated .cpp files.

As a precaution, it is recommended that you have a backup of the original .h file (for example, in a version control system) before re-generating code to replace it.

If you wish to augment a class with lots of code it may be appropriate to consider defining a separate class, and then including the declaration of an instance of that class in the snippets section. This further separates your code and the generated code. To do this it may be necessary to use the `-include-h` to inform LMX of a header file defining your class, and implement an `on_lmx_construct()` snippet event handler (see 3.14.1 - Augmenting Generated Classes with Snippet Event Handlers below) that can inform your separate class of the `this` pointer (or a reference) of the generated class. For example, in this case the snippets section for a class might appear as:

```
//:snippet:start YYY
c_my_details_class my_class;
void on_lmx_construct() { my_class.set_parent( this ); }
//:snippet:end YYY
```

As described above this procedure requires editing the generated `.h` file. If you do not wish to do this, the `-file-ext-snippets` flag allows the specification of the file extension for files that store snippets. For example, if `-file-ext-snippets .lmxsnips` is specified, LMX will look in `foo.lmxsnips` for snippets rather than in `foo.h`.

3.14.1 - Augmenting Generated Classes with Snippet Event Handlers

The behavior of the LMX generated classes can be augmented by what are called snippet event handlers. When generating code, LMX looks for the declaration of specially named functions in each class's snippets section. If the name of a special function is found, then code is generated to call that function. The defined snippet event handler functions are:

```
void on_lmx_construct();

void on_lmx_copy_construct( const ??? & r_rhs );

void on_lmx_swap( ??? & r_rhs );
```

```
void on_lmx_destruct();

bool on_lmx_eq( const ??? & r_rhs ) const;

bool on_lmx_is_occurs_ok() const;

lmx::elmx_error on_lmx_marshal_start( lmx::c_xml_writer & r_writer,
                                      const char * p_element_name ) const;

lmx::elmx_error on_lmx_marshal_end( lmx::c_xml_writer & r_writer ) const;

// Currently return code for on_lmx_marshal_start_tag() is ignored
lmx::elmx_error on_lmx_marshal_start_tag( lmx::c_xml_writer & r_writer ) const;

lmx::elmx_error on_lmx_marshal_body_start( lmx::c_xml_writer & r_writer ) const;

lmx::elmx_error on_lmx_marshal_body_end( lmx::c_xml_writer & r_writer ) const;

lmx::elmx_error on_lmx_alt_marshal_attribute_XYZ(
          lmx::c_xml_writer & r_writer,
          const lmx::ct_typed_validation_spec< ??? > & r_item_spec ) const;

lmx::elmx_error on_lmx_alt_marshal_element_XYZ(
          lmx::c_xml_writer & r_writer,
          const lmx::ct_typed_validation_spec< ??? > & r_item_spec ) const;

lmx::elmx_error on_lmx_unmarshal_outer_start( lmx::c_xml_reader & r_reader );

lmx::elmx_error on_lmx_unmarshal_outer_end( lmx::c_xml_reader & r_reader );

lmx::elmx_error on_lmx_unmarshal_start( lmx::c_xml_reader & r_reader,
                                        const std::string & r_name );

lmx::elmx_error on_lmx_unmarshal_start( lmx::c_xml_reader & r_reader,
                                        const std::string & r_name );

lmx::elmx_error on_lmx_unmarshal_end( lmx::c_xml_reader & r_reader );

lmx::elmx_error on_lmx_unmarshal_attributes_check(
                                      lmx::c_xml_reader & r_reader );

lmx::elmx_error on_lmx_unmarshal_body_start( lmx::c_xml_reader & r_reader );

lmx::elmx_error on_lmx_unmarshal_body_end( lmx::c_xml_reader & r_reader );

lmx::elmx_error on_lmx_unmarshal_attribute_XYZ( lmx::c_xml_reader & r_reader );

lmx::elmx_error on_lmx_alt_unmarshal_attribute_XYZ(
          lmx::c_xml_reader & r_reader,
          const lmx::ct_typed_validation_spec< ??? > & r_item_spec );

lmx::elmx_error on_lmx_unmarshal_element_XYZ( lmx::c_xml_reader & r_reader );

lmx::elmx_error on_lmx_alt_unmarshal_element_XYZ(
          lmx::c_xml_reader & r_reader,
          const lmx::ct_typed_validation_spec< ??? > & r_item_spec,
          const lmx::s_event_map event_map[] );

lmx::elmx_error on_lmx_alt_unmarshal_xs_any(
          lmx::c_xml_reader & r_reader,
          const lmx::s_event_map ac_event_map[] );
```

The snippet event handler functions are called in the following situations:

on_lmx_construct()
: Called immediately after the code generated to initialize the object. This allows initializing snippet data.

on_lmx_copy_construct()
: Called in the generated copy constructor.

on_lmx_swap()
: Called in the generated swap method.

on_lmx_destruct()
: Called immediately before the code generated to destroy an object. This is intended to give the opportunity to do special handling of the data before it is destroyed and destroy any data introduced through snippets.

on_lmx_eq()
: Called in the generated equality testing methods called by the generated `operator ==()` method.

on_lmx_is_occurs_ok()
: Called in place of the generated version of is_occurs_ok() at the start of the marshalling operation.

on_lmx_marshal_start()
: Called at the start of the object's main top-level marshaling method. It can be used to implement additional user defined checks on the data before it is marshaled.

on_lmx_marshal_end()
: Called at the end of the object's main top-level marshalling method just prior to the final `return` statement.

on_lmx_marshal_start_tag()
: Called while outputting attributes in an element's start tag. It can be used to output additional attributes. (Note that the return code from this function is currently ignored. This situation may change in future however.)

on_lmx_marshal_body_start()
: Called at the start of an elements `marshal_body()` method.

on_lmx_marshal_body_end()
: Called at the end of an elements `marshal_body()` method. It can be used to output additional elements.

on_lmx_alt_marshal_attribute_XYZ()
: Allows you to specify alternative marshalling behavior for attribute XYZ.

on_lmx_alt_marshal_element_XYZ()
: Allows you to specify alternative marshalling behavior for element XYZ.

on_lmx_unmarshal_outer_start()

Called at the start of the outer-most, top-level object's main unmarshaling method.

on_lmx_unmarshal_outer_end()
: Called at the end of the outer-most, top-level object's main unmarshaling method.

on_lmx_unmarshal_start()
: Called at the start of the object's main top-level unmarshaling method.

on_lmx_unmarshal_end()
: Called at the end of the object's main top-level unmarshalling method just prior to the final `return` statement. It can be used to implement additional user defined checks on the data immediately after it is unmarshaled (for example co-occurrence constraints).

on_lmx_unmarshal_attributes_check()
: Called in the `unmarshal_attributes_check()` method.

on_lmx_unmarshal_body_start()
: Called at the start of an elements `unmarshal_body()` method.

on_lmx_unmarshal_body_end()
: Called at the end of an elements `unmarshal_body()` method.

on_lmx_unmarshal_attribute_XYZ()
: Called when the attribute stored in the C++ variable m_XYZ has been unmarshalled and stored in the object.

on_lmx_alt_unmarshal_attribute_XYZ()
: Allows you to specify alternative unmarshalling behavior for attribute XYZ.

on_lmx_unmarshal_element_XYZ()
: Called when a new instance of the element has been unmarshalled and is stored in the C++ variable m_XYZ.

on_lmx_alt_unmarshal_element_XYZ()
: Allows you to specify alternative unmarshalling behavior for element XYZ.

on_lmx_alt_unmarshal_xs_any()
: Allows you to specify alternative unmarshalling for an element corresponding to an xs:any specification.

It will be noticed that the marshal and unmarshal event handlers return an `lmx::elmx_error` value. If the value returned is not `lmx::ELMX_OK` then the marshaling or unmarshaling operation is terminated and the error code returned. To provide custom error feedback, the `lmx::elmx_error` enumeration contains enumeration values `ELMX_USER_DEFINED_1` to `ELMX_USER_DEFINED_9`.

Similarly, the marshal and unmarshal event handlers take as an argument a reference to the marshaling operations `lmx::c_xml_writer` object, or the unmarshaling operations `lmx::c_xml_reader` object. If the `-alt-xml-writer` or `-alt-xml-reader` command-line flags are used to change the names of these two latter objects, then a reference to the specified objects will be passed. This gives the opportunity to pass additional data into the event handlers.

One use of the `on_lmx_alt_unmarshal_element_XYZ()` snippet event handler is to enable the user to ignore particular elements. The `elmx_error c_xml_reader::ignore_element(...)` method is provided to help with the implementation of such a handler. For example:

```
lmx::elmx_error c_MyElement::on_lmx_alt_unmarshal_element_ChildElement(
            lmx::c_xml_reader & ar_reader,
            const lmx::s_event_map ac_event_map[] )
{
    return ar_reader.ignore_element( ac_event_map );
}
```

3.15 - DTDs and Namespaces

DTDs do not explicitly support XML namespaces. However, LMX can extract namespace information from DTDs via attribute definitions that contain the `xmlns` prefix and have a #FIXED specification. For example, to declare that the namespace prefix '`ns`' should be associated with the namespace '`http://xml2cpp.com/example`', use the following attribute definition:

```
<!ATTLIST ns:MyElement xmlns:ns CDATA #FIXED "http://xml2cpp.com/dtd-test.html">
```

To set the default namespace, use:

```
<!ATTLIST MyElement xmlns CDATA #FIXED "http://xml2cpp.com/dtd-test.html">
```

3.16 - Debugging and Handling Errors

LMX is designed to be flexible about how run-time errors are handled. It also includes features to aid debugging.

There are three categories of error that can occur. These are parse errors, write errors, and class access errors. Any error that occurs in one of these categories is first passed to a category specific error handling function. The functions are `c_xml_reader::handle_error`, `c_xml_writer::handle_error` (both in lmxparse.h), and `lmx_error` in lmxuser.h respectively. These functions allow you to tailor the error handling to your requirements.

3.16.1 - Reporting Errors

LMX includes a number of functions to help report errors to a user.

To convert an error code to a description, the global `get_error_description` function can be called. The prototype for this function is:

```
const char *get_error_description( elmx_error a_error_code );
```

The error code is entered in `a_error_code` and the return value points to a const char string describing the error.

c_xml_reader and c_xml_writer both include a get_error_message function. The prototypes for the functions are:

```
virtual std::string & c_xml_reader::get_error_message( std::string *ap_message );
virtual std::string & c_xml_writer::get_error_message( std::string *ap_message );
```

An error message, including the name of the element or attribute in question, and the line number (if applicable) will be inserted with a description into the string pointed to by ap_message. The function returns a reference to *ap_message. The function is virtual to allow user customization of the message format.

The line number that the error occurred on can be obtained by calling:

```
int c_xml_reader::get_line_no();
```

c_xml_writer does not keep track of line numbers due to its use of iostreams, and hence has no corresponding function.

c_xml_reader and c_xml_writer allow the element or attribute name applicable to an error to be obtained by calling:

```
const std::string & c_xml_reader::get_error_item_name();
const std::string & c_xml_writer::get_error_item_name();
```

3.16.2 - Changing the Error Handling Behavior

As supplied, the above functions support the 'return code' paradigm for reporting errors and the code generated by LMX allows the return code to be propagated back to the original calling function.

You can select exception-based error reporting by setting the LMX_USE_USER_VALIDATE_EXCEPTIONS and LMX_USE_EXCEPTIONS pre-processor constants to 1. The constants are defined in the USER: CTRL_DEFS section of lmxuser.h. LMX_USE_USER_VALIDATE_EXCEPTIONS selects exceptions when validating user access to the objects and LMX_USE_EXCEPTIONS selects exceptions when marshaling and unmarshaling. If exception-based error reporting is selected, exceptions are reported using instances of the c_lmx_exception and c_lmx_reader_exception classes defined in lmxinternals.h. c_lmx_reader_exception is used to signal exceptions from c_xml_reader, and c_lmx_exception is used for all other exceptions. There public interface is as follows:

```
class c_lmx_exception
{
public:
        c_lmx_exception( elmx_error a_error_code );
        elmx_error get_error_code() const;
        virtual const char *what() const throw();
};

class c_lmx_reader_exception : public c_lmx_exception
{
public:
        c_lmx_reader_exception( elmx_error a_error_code, int a_line );
int get_line() const;
};
```

If you wish to use an alternative error reporting strategy, then define new classes that derive from `c_xml_reader` and `c_xml_writer` and define appropriate `user_handle_error` methods.

3.16.3 - Conditional Error Handling

The error handling structure used by LMX allows you to conditionally ignore errors. For example, if the XML being parsed has been human generated, then for certain errors you may wish to continue parsing and reporting errors after the first error has occurred. This can be done by modifying the `c_xml_reader::user_handle_error(...)` and `c_xml_writer::user_handle_error(...)` methods to return `lmx::ELMX_OK` rather than the error that occurred. As `c_xml_reader::user_handle_error(...)` and `c_xml_writer::user_handle_error(...)` are virtual methods an alternative strategy is to define classes that extend `c_xml_reader` and `c_xml_writer` and define appropriate `user_handle_error(...)` methods therein.

Note that care should be taken when adopting this strategy as it is not possible to proceed after certain errors have occurred (such as an unexpected end of message). Hence it is recommended that your code explicitly defines the errors that are being ignored rather than having a default of ignoring all errors and defining those that are not ignored.

3.16.4 - Collecting Debug Error Information when using Convenience Methods

The convenience methods for unmarshaling and marshaling (See 2.3 - Unmarshaling (simple form) and 2.4 - Marshaling (simple form)) are intentionally light-weight, and in their simplest form only return an error code when an error occurs.

However, to aid debugging, in Debug mode the global `debug_error` object, which has the type `s_debug_error`, is set with any information about the latest error.

If use of a global object for debugging is not suitable, then the convenience methods can also be called with an instance of `s_debug_error` that will be populated with any error information.

`s_debug_error` is declared in `lmxinternals.h` and has the following declaration:

```
struct s_debug_error
{
    elmx_error error;
    std::string item_name;
    const char * p_prog_file;
    int prog_line;
    int xml_line;

    LMX_PDECL s_debug_error();
    LMX_PDECL void clear();
    LMX_PDECL void set(
                elmx_error error_in,
                const std::string & r_item_name_in,
                const char * p_prog_file_in,
                int prog_line_in,
                int xml_line_in = -1 );
```

```
    LMX_PDECL const char * get_description() const;
    LMX_PDECL const char * get_description(
                const s_custom_error_description * p_custom_descriptions ) const;
    LMX_PDECL std::string to_string() const;
    LMX_PDECL std::string to_string(
                const s_custom_error_description * p_custom_descriptions ) const;
    LMX_PDECL std::ostream & to_stream( std::ostream & r_os_in ) const;
    LMX_PDECL std::ostream & to_stream(
                std::ostream & r_os_in,
                const s_custom_error_description * p_custom_descriptions ) const;
};

LMX_PDECL std::ostream & operator << (
                std::ostream &ar_os,
                const s_debug_error &ar_debug_error );

LMX_PDECL extern s_debug_error debug_error;

#define LMX_OUTPUT_DEBUG_ERROR( xstream ) ...
```

The `to_string()` method returns the combined debug information as a suitable message in a string.

The `to_stream()` method outputs the combined debug information in a suitable message to the specified stream.

The variants with the `const s_custom_error_description *` parameter allow setting custom mappings of LMX error codes to descriptions. This is intended specifically for the LMX error codes of the form `ELMX_USER_DEFINED_?`.

Mappings are specified by an array of `structs` of the form:

```
struct s_custom_error_description
{
    elmx_error      code;
    const char *    p_description;
};
```

For example:

```
lmx::s_custom_error_description custom_error_descriptions[] = {
        { lmx::ELMX_USER_DEFINED_1, "Out of range" },
        { lmx::ELMX_USER_DEFINED_2, "A must be present with B" },
        { lmx::ELMX_OK, LMXNULL }
        };
```

Note the end of the array is identified by the entry `{ lmx::ELMX_OK, LMXNULL }`.

If you want to disable `debug_error` being used in your code during Debug mode, then arrange for the #define `LMX_DEBUG_CAPTURE_ERROR` to have the value `0`. Conversely, if you want to force `debug_error` to be in your code, even in Release mode (but see below), then arrange for the #define `LMX_DEBUG_CAPTURE_ERROR` to have the value `1`.

In Debug mode, or when `LMX_DEBUG_CAPTURE_ERROR` is set to 1, the `LMX_OUTPUT_DEBUG_ERROR( xstream )` macro will output the error information to the stream `xstream`. In Release mode and when `LMX_DEBUG_CAPTURE_ERROR` is not set to 1 the `LMX_OUTPUT_DEBUG_ERROR( xstream )` macro has no effect.

The output operator function allows code such as:

```
std::cout << debug_error;
```

To access other debug information just access the data members directly.

If additional flexibility is required for error reporting, it is suggested that the Advanced forms of marshaling and unmarshaling be used (See 3.5 - Marshaling (advanced forms) and 3.2 - Unmarshaling (advanced forms)).

3.16.5 - Debugging Support

The common error handling functions provided by LMX aid debugging support. Setting a break point in the above-mentioned functions can facilitate detecting problems during parsing, writing or accessing the class members. It is then a simple matter of tracing back through the stack to see why the error is being reported.

LMX also aids the development process when your code accesses the generated classes. You can choose whether facet validation is done when you write data to a generated member functions by defining the compiler definition `LMX_USER_VALIDATE` in the `USER: CTRL_DEFS` section of the lmxuser.h file. If this directive is defined, any such errors that occur will first be reported to the `lmx_error` function in lmxuser.h, and then optionally the return code of that function is returned to the caller. Alternatively, an exception can be thrown, or an `assert` statement can be added to `lmx_error`. This feature supports the 'fail fast' paradigm and allows errors to be detected close to where they occur.

LMX also generates functions called `is_occurs_ok()` and `check()` for each class. These test whether the minimum cardinality requirements for the class have been satisfied. They thus allow you to test whether you have set a sufficient number of the class members for the class to be valid. They test the cardinality constraints for both elements and attributes. One way to use these functions is in an `assert` statement once you have set all the appropriate members, e.g.:

```
c_NAME & name_ref = top.get_NAME();
name_ref.set_CHILD( 12 );
name_ref.set_CHILD2( 1.2 );

assert( name_ref.is_occurs_ok() );
```

See 2.5.15 - Run-time Checking for more information.

3.17 - Adding External Character Set Transcoders

The LMX XML parser natively supports UTF-8, UTF-16 (BE & LE), UCS2 (BE & LE), ISO-8859-1 and US-ASCII. If support for additional character sets is required, then these can be configured externally.

Examples of how to configure external transcoders are shown in the installed files `include/lmx-transcoder-example.h` and `src/lmx-transcoder-example.cpp`.

There are two interface classes used to support external transcoders:

A `const` instance of a class implementing the `lmx::c_external_transcoder_factory` interface is used to create instances of the classes that actually do the character transcoding. When a new transcoder

object is required LMX calls the interface's `get_transcoder( c_read & r_reader_in, c_error & r_error_in, c_get_as_utf8::e_encoding_mode initial_encoding_mode_in, const char * p_encoding_name_in )` method. This should return a pointer to a transcoder object, or `NULL` if no suitable transcoder is found. LMX will delete the returned transcoder object when it is no longer required.

The class implementing the `lmx::c_external_transcoder_factory` interface can be specified to LMX on a permanent basis using the static `c_xml_reader::set_default_external_transcoder_factory()` method or on a per-parsing session basis by calling the `set_external_transcoder_factory()` method on the relevant instance of the `c_xml_reader` class (or derivative thereof).

Each transcoder should implement the `lmx::c_external_transcoder` interface class and implement the `c_external_transcoder::s_result read()` method. The `c_external_transcoder::s_result` object includes the character just read in the `c` member variable, and optionally a pointer to a NULL-terminated array of UTF-8 encoded bytes that LMX should read before asking the transcoder for new characters in the `mp_utf8_expansion` member variable. If no such additional UTF-8 bytes are present then `s_result.mp_utf8_expansion` should be set to NULL. If there is no input available then the transcoder class should return `c_read::k_eof` in `s_result.c`. If an error occurs then this should be signaled to the `c_error` error class referenced by `r_error_in` and `c_read::k_eof` returned in `s_result.c`.

3.18 - Adding Extra Namespace Information

When a document contains `xs:anyAttribute` or `xs:any`, it may be necessary to provide extra namespace information to the unmarshaling and marshaling operations.

If a section of xs:any XML has been extracted while unmarshaling it is possible that the start tag of the value part of the instance does not include appropriate namespace information. Without suitable namespace information it is not possible to unmarshal its contents using another module of unmarshaling code. To remedy this, `c_xml_reader` allows additional namespace information to be added to it prior to initiating unmarshaling. The functions that support this are:

```
void c_xml_reader::add_namespace_mappings( const c_any_info & r_any_info );
void c_xml_reader::add_namespace_mappings( const c_namespace_context & r_namespace_context );
void c_xml_reader::add_namespace_mapping( const std::string & r_ns_prefix, const std::string
void c_xml_reader::add_default_namespace_mapping( const std::string & r_namespace );
```

Where:

`r_ns_prefix`
: is the namespace prefix to be assigned to the namespace.

`r_namespace`
: is the actual namespace URI, e.g. "http://codalogic.com/example".

`add_default_namespace_mapping( namespace );` is effectively an alias for `add_namespace_mapping( "", namespace );`.

An example of adding extra namespace information is:

```
lmx::c_xml_reader l_reader( l_low_reader );
l_reader.add_namespace_mapping( "alt", "http://codalogic.com/example" );
c_MyElement l_item;
lmx::elmx_error l_error = l_item.unmarshal( l_reader );
```

Equally, when writing XML that contains `xs:anyAttribute` or `xs:any` additional namespace information may need to be added to the `c_xml_writer` class. This is supported by the following methods:

```
void c_xml_writer::add_namespace( const std::string & r_ns_prefix, const std::string & r_namespace );
void c_xml_writer::add_default_namespace( const std::string & r_namespace );
```

Where:

`ar_ns_prefix`
: is the namespace prefix to be assigned to the namespace.

`ar_namespace`
: is the actual namespace URI, e.g. "http://codalogic.com/example".

Again, `add_default_namespace( namespace );` is an alias for `add_namespace( "", namespace );`.

An example of their use is:

```
lmx::c_xml_writer l_writer( l_out_stream );
l_writer.add_namespace( "alt", "http://codalogic.com/example" );

l_item.marshal( l_writer );
```

3.19 - Adding Schema Location Information

You can specify to the `c_xml_writer` class that you want schema location information to be output in the generated XML. This will cause either the `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation` attributes to be output in the marshaled code. This is done using the following methods:

```
void c_xml_writer::add_schema_location( const std::string & r_ns_uri, const std::string & r_location
void c_xml_writer::add_no_namespace_schema_location( const std::string & r_location );
```

Where:

`ar_ns_uri`
: is the namespace URI associated with the schema, e.g. "http://codalogic.com/example".

`ar_location`
: is the location hint for the actual schema file, e.g. "http://codalogic.com/schemas/example.xsd".

`add_schema_location()` specifies the location for a schema that has a namespace URI associated with it.

`add_no_namespace_schema_location()` specifies the location of a schema that has no namespace associated with it.

Multiple calls can be made to `add_schema_location()` to specify the locations of multiple schemas with namespaces.

Only one call to `add_no_namespace_schema_location()` may be made.

If `add_schema_location()` is called, then `add_no_namespace_schema_location()` must not be called, and vice versa.

3.20 - Disabling Output of XML Namespaces

When generating XML fragments that will be inserted by other means into other XML instances it may be appropriate to not have the generated XML include the `xmlns` namespace attributes. This can be achieved by calling the `disable_ns_map_output()` method on an instance of a `c_xml_writer` object. For example:

```
c_MyObject myObject;
std::ostringstream sos;
lmx::c_xml_writer writer( sos );
writer.disable_ns_map_output();   // N.B.
myObject.marshal( writer );
```

This requires using the advanced forms of marshalling as described in 3.5 - Marshaling (advanced forms).

3.21 - Adding an XMLDecl to the XML output

By default when using the Convenience methods for marshalling (2.3 - Unmarshaling (simple form)), the XMLDecl is output (the part that looks like `<?xml version="1.0"?>`). This can be configured on a global basis using the `no_xml_decl_with_convenience_methods()`, `c_xml_writer::set_convenience_options( c_xml_writer::t_writer_options )` and `c_xml_writer::get_convenience_options()` methods, where. `c_xml_writer::t_writer_options` is defined as:

```
typedef int t_writer_options;
static const t_writer_options no_xml_decl;
static const t_writer_options include_xml_decl;
static const t_writer_options include_standalone;
```

The `c_xml_writer::t_writer_options` can be combined by using the C++ OR operator (e.g. `include_xml_decl | include_standalone`). An XMLDecl is output when the `include_xml_decl` option is included. The standalone attribute can be included in the XMLDecl by specifying the `include_standalone` option. Use of the `include_standalone` option implies `include_xml_decl`, and so it is not necessary to specify `include_xml_decl` when selecting the standalone attribute. To disable output of an XMLDecl, use `no_xml_decl_with_convenience_methods()` or `set_convenience_options( c_xml_writer::no_xml_decl )`. Using `c_xml_writer::set_convenience_options()` and `c_xml_writer::get_convenience_options()` requires prior #inclusion of `lmxparse.h`, whereas using `no_xml_decl_with_convenience_methods()` does not.

To enable instance by instance configuration of whether the XMLDecl is output it is recommended to use the advanced forms of marshalling (3.5 - Marshaling (advanced forms)) rather than the above methods.

By default `c_xml_writer` does not output the optional XMLDecl.

You can instruct `c_xml_writer` to output the XMLDecl by using the constructor that takes a `t_writer_options` argument. This has the form:

```
c_xml_writer( std::ostream &ar_os,
              t_writer_options a_options,
              const char *ap_tab = "\t",
              const char *ap_nl = "\n" );
```

Thus an example of how to construct an instance of `c_xml_writer` that outputs the XMLDecl is:

```
lmx::c_xml_writer l_writer( l_out_stream,
                            lmx::c_xml_writer::include_xml_decl );
```

An example of selecting an XMLDecl that contains the standalone attribute is:

```
lmx::c_xml_writer l_writer( l_out_stream,
                            lmx::c_xml_writer::include_standalone );
```

3.22 - Handling Multiple Schemas

3.22.1 - Handling Multiple Independent Schemas

If your project includes multiple independent schemas (or independent sets of schema) then we recommend generating the code for each independent schema set separately and instructing LMX to generate the code into separate C++ namespaces using the `-cns YYY` flag or via the WinLMX "Generated Code C++ Namespace" group box on the "C++ Names & Namespaces" tab. For example, if you have schemas `s1.xsd` and `s2.xsd` you could run LMX and instruct it to generate code for `s1.xsd` into C++ namespace `s1` (using `-cns s1`) and then run LMX again and instruct it to generate code for `s2.xsd` into the `s2` C++ namespace (using `-cns s2`). In your C++ files that #include the generated code you can use statements such as "`using namespace s1;`" and "`using namespace s2;`" to avoid qualifying each use of a variable with a namespace prefix.

While the above is our recommended method for handling this situation, if it is not acceptable in your environment, you can use the `-prefix-declash YYY` flag to avoid name clashes when you come to link your generated files. In this scenario you could use `-prefix-declash s1_` when generating code for `s1.xsd` and `-prefix-declash s2_` when generating code for `s2.xsd`.

3.22.2 - Multiple Schemas that Share Common Schemas

A common scenario is when two or more schemas share one of more common imported or included schemas. In this case problems occur if the schemas are compiled separately due to duplicate code being generated.

To address this problem, all schemas that share common schemas should have code generated for them in a single LMX compilation operation. As an example, assume that `schema1.xsd` and `schema2.xsd` both share definitions from `common.xsd`. Rather than compiling `schema1.xsd` and `schema2.xsd` separately, `schema1.xsd` and `schema2.xsd` should be compiled together. Using the command-line

interface, this can be done using a DOS command-line such as:

```
lmx  schema1.xsd  ++  schema2.xsd  +  common.xsd  outbase
```

Or a Linux command-line such as:

```
linmx  schema1.xsd  ++  schema2.xsd  +  common.xsd  outbase
```

From 2.1.2 - Code Generation Using the Windows (DOS) Command Line Version it can be seen that the + sign informs LMX of a schema file that contains additional definitions that need to be imported. However, global elements in such a schema file are not treated as possible root elements for the XML instances. The ++ sign similarly informs LMX of a schema file that contains additional definitions, but in this case the global elements are considered to be possible root elements for a valid XML instance. Hence, specifying ++ `schema2.xsd` on the command-line will ensure that the global elements in `schema2.xsd` will be treated as valid XML instance root elements, giving the desired effect.

The same results can be achieved using WinLMX. In this case, on the `Schema Files` tab, enter `schema1.xsd` in the `Base Schema File` edit box. Enter `schema2.xsd` and `common.xsd` in the `Additional Imported Schema Files` section using the relevant `Add...` button. Then select `schema2.xsd` in the listview and press the `Edit...` button. In the dialog box that appears, check the `Treat as Global Schema` checkbox and press `OK`. When the dialog box closes a small globe symbol will be shown next to `schema2.xsd`'s name. This signifies that global elements in the schema will be treated as valid root elements in XML instances. Compile as usual.

3.23 - Pattern Facet Handling Customization

LMX includes a default regular expression library for pattern facet validation (which is built into the binaries).

You can disable testing pattern facets by arranging for the `#define LMX_VALIDATE_PATTERNS` to be set to `0`. In this case all input strings will be considered to be valid against their schema patterns.

LMX also allows you to specify an alternative pattern-matching library. This is configured by setting the `tc_pattern` `typedef` in lmxuser.h. Simply modify the `typedef` statement so that your pattern-matching class is used instead of the default supplied with LMX. Your pattern-matching class must include as a minimum the following public interfaces:

```
class c_my_pattern_class
{
public:
    c_my_pattern_class( const char *ap_pattern );
    bool is_match( const std::string &ar_v ) const;
    bool is_match( const std::wstring &ar_v ) const;
};
```

In both cases the function `is_match()` should return `true` if the string in `ar_v` matches the pattern specified by `ap_pattern` when the class was constructed, and `false` otherwise. The pattern specified by `ap_pattern` is UTF-8 encoded.

Note that the `is_match( const std::wstring &ar_v )` variant is used for facet testing when you set data using the generated classes accessor methods. If you do not use facet testing in this phase (see `LMX_USER_VALIDATE` in 3.16.5 - Debugging Support), then you don't need to implement this member.

Alternatively, if you use accessor side facet testing, but don't want to test pattern facets, you can arrange for this function to always return `true`.

A simple approach to implementing the `is_match( const std::wstring &ar_v )` method is to use LMX's `convert()` functions to convert the wide string to a narrow string and then call `is_match( const std::string &ar_v )`.

If you only have simple pattern facets and you don't want a full regular expression engine in your application (perhaps due to platform constraints), then one approach is to define a class that, based on the string that is passed to it in its constructor (specifying the desired pattern), sets a pointer to a function that will test for matches against that pattern. The class will then call the function pointed to by the pointer when a test is required. The code below demonstrates such an example:

```
#include <cassert>
#include <cstdlib>
#include <iostream>

// Utility functions called by the match functions
// (Note: C++'s isdigit() and isupper() are not so good on 8-bit chars...)

// N.B. A reference to the pointer is passed in so the pointer is updated
// in the calling function
size_t scan_digits( const char * & arp_string )
{
    size_t l_size;
    for( l_size = 0; *arp_string >= '0' && *arp_string <= '9'; ++l_size )
        ++arp_string;
    return l_size;
}

size_t scan_upper_letters( const char * & arp_string )
{
    size_t l_size;
    for( l_size = 0; *arp_string >= 'A' && *arp_string <= 'Z'; ++l_size )
        ++arp_string;
    return l_size;
}

size_t scan_range( const char * & arp_string,
                   size_t (*pf_test)( const char * & arp_string ),
                   size_t a_min,
                   size_t a_max=~0 )
{
    size_t l_n_matches = pf_test( arp_string );
    if( l_n_matches >= a_min && l_n_matches <= a_max )
        return true;
    return false;
}

// The functions implementing the matches we want

bool is_ip_address( const std::string &ar_v )
{
    const char *lp_v = ar_v.c_str();
    for( size_t l_i=0; l_i<3; ++ l_i )
        {
        if( ! scan_range( lp_v, scan_digits, 1, 3 ) ||
                *lp_v++ != '.' )
            return false;
        }
```

```
    if( ! scan_range( lp_v, scan_digits, 1, 3 ) ||
            *lp_v != '\0' )
        return false;
    return true;
}

bool is_sku( const std::string &ar_v )
{
    const char *lp_v = ar_v.c_str();
    if( scan_range( lp_v, scan_upper_letters, 3, 3 ) &&
            *lp_v++ == '-' &&
            scan_range( lp_v, scan_digits, 2, 2 ) &&
            *lp_v == '\0' )
        return true;
    return false;
}

// The actual class doing the matching
class c_my_simple_patterns
{
private:
    // Pointer to function that will do the match
    bool (*pf_matcher)( const std::string &ar_v );

public:
    c_my_simple_patterns( const char *ap_pattern ) : pf_matcher( 0 )
    {
        // Select the right match function based on the input pattern
        if( strcmp( ap_pattern, "([0-9]{1,3}\\.){3}[0-9]{1,3}" ) == 0 )
            pf_matcher = &is_ip_address;

        else if( strcmp( ap_pattern, "[A-Z]{3}-[0-9]{2}" ) == 0 )
            pf_matcher = &is_sku;

        else
            assert( 0 );
    }
    bool is_match( const std::string &ar_v ) const
    {
        // Just call the right match function
        if( pf_matcher )
            return pf_matcher( ar_v );
        return false;
    }
    bool is_match( const std::wstring &ar_v ) const
    {
        std::string l_narrow_string;
        //lmx::convert( &l_narrow_string, ar_v ); // Include lmxuser.h for this
        return is_match( l_narrow_string );
    }
};

// main() function demonstrating usage
// (Note that LMX usually generates all the code to generate the match class
// instances and invoke the tests.)
int main(int argc, char* argv[])
{
    c_my_simple_patterns l_ip_addr( "([0-9]{1,3}\\.){3}[0-9]{1,3}" );
    std::cout << "Is IP 10.0.0.1 (true): " <<
                 l_ip_addr.is_match( "10.0.0.1" ) << "\n";
    std::cout << "Is IP 1000.0.0.1 (false): " <<
                 l_ip_addr.is_match( "1000.0.0.1" ) << "\n";
    std::cout << "Is IP 10.0.0.1.0 (false): " <<
```

```
                l_ip_addr.is_match( "10.0.0.1.0" ) << "\n";
    std::cout << "Is IP 10.0.0. (false): " <<
                l_ip_addr.is_match( "10.0.0." ) << "\n";
    std::cout << "Is IP 10.0.0 (false): " <<
                l_ip_addr.is_match( "10.0.0" ) << "\n";

    c_my_simple_patterns l_sku( "[A-Z]{3}-[0-9]{2}" );
    std::cout << "Is SKU ABC-12 (true): " <<
                l_sku.is_match( "ABC-12" ) << "\n";
    std::cout << "Is SKU ABCD-12 (false): " <<
                l_sku.is_match( "ABCD-12" ) << "\n";
    std::cout << "Is SKU ABC-12A (false): " <<
                l_sku.is_match( "ABC-12A" ) << "\n";
    std::cout << "Is SKU ABC.12 (false): " <<
                l_sku.is_match( "ABC.12" ) << "\n";
    std::cout << "Is SKU 1AB-12 (false): " <<
                l_sku.is_match( "1AB-12" ) << "\n";

    return 0;
}
```

3.24 - Custom Pattern Facet Output Formatting

Occasionally an `xs:pattern` facet implies XML formatting of a data type that differs from the default representation. For example, an integer type may require leading zeros.

To accommodate this you can use the `-pattern-out` flag. The format of this flag is `-pattern-out PATTERN FUNCTION`, where `PATTERN` is the `xs:pattern` specified in the schema and `FUNCTION` is the name of the function you have specified to output the type correctly.

If the data type has multiple `xs:patterns` specified for it then all the `xs:pattern` values should be concatenated together using the `|` character as a separator into one pattern string to form the flag's `PATTERN` argument. This may require quotes to be used on the command-line.

3.25 - XML Output Format Customization

LMX allows you to control the format of the generated XML. Currently you can control the character(s) used at the end of a 'line', and the character(s) used for indenting. This capability allows you to control whether you want the most compact XML format, or a format that is more attractive and readable when displayed in a viewer/editor.

The format can be controlled at compile-time, globally at run-time and per-unmarshalling process at run-time.

The default format can be defined at compile-time in the `USER: XML_WRITER_FORMAT` section of `lmxuser.h` by setting the `LMX_WRITER_DEFAULT_TAB`, `LMX_WRITER_DEFAULT_NL`, `LMX_WRITER_DEFAULT_ATTRIBUTE_TAB` and `LMX_WRITER_DEFAULT_ATTRIBUTE_NL` #defines to appropriate values.

`LMX_WRITER_DEFAULT_TAB` and `LMX_WRITER_DEFAULT_NL` control the formatting of elements and `LMX_WRITER_DEFAULT_ATTRIBUTE_TAB` and `LMX_WRITER_DEFAULT_ATTRIBUTE_NL` control the formatting of attributes.

If not explicitly set, the values of `LMX_WRITER_DEFAULT_TAB` and `LMX_WRITER_DEFAULT_NL` are defaulted to `"\t"` and `"\n"` respectively.

If not explicitly set, the values of `LMX_WRITER_DEFAULT_ATTRIBUTE_TAB` and `LMX_WRITER_DEFAULT_ATTRIBUTE_NL` are defaulted to the values of `LMX_WRITER_DEFAULT_TAB` and `LMX_WRITER_DEFAULT_NL` respectively.

Setting all values to `""` will result in the smallest XML. Setting `LMX_WRITER_DEFAULT_TAB` to `"  "` and `LMX_WRITER_DEFAULT_NL` to `"\n"` will result in shallower nesting.

The default format can be defined at run-time using the `c_xml_writer` static methods `c_xml_writer::set_default_tab( const std::string & ar_default_tab )`, `c_xml_writer::set_default_nl( const std::string & ar_default_nl )`, `c_xml_writer::set_default_attribute_tab( const std::string & ar_default_ attribute_tab )` and `c_xml_writer::set_default_attribute_nl( const std::string & ar_default_ attribute_nl )` methods.

`set_default_attribute_tab()` and `set_default_attribute_nl()` control the default formatting of attributes. For backwards compatibility reasons, if the element and attribute formatting characters are the same when `set_default_tab()` and `set_default_nl()` are called, they will alter both the element and attribute formatting characters. Conversely, if the element and attribute formatting characters are different at the time the methods are called then they will only change the respective element formatting characters. For this reason it may be appropriate to ensure that `set_default_tab()` and `set_default_nl()` are called prior to `set_default_attribute_tab()` and `set_default_attribute_nl()` if independent formatting is required.

If you do not wish to allow these two methods to be effective then you can set the define `LMX_WRITER_LOCK_FORMAT` to `1`.

The formatting is controlled per-unmarshalling process via the constructor of the `c_xml_writer` class. This mechanism overrides the defaults set by the above methods. The prototype for this is:

```
c_xml_writer( std::ostream & ar_os,
              const char * ap_tab = LMXNULL,
              const char * ap_nl = LMXNULL,
              const char * ap_attribute_tab = LMXNULL,
              const char * ap_attribute_nl = LMXNULL );
```

The format is specified by setting `ap_tab`, `ap_nl` `ap_attribute_tab` and `ap_attribute_nl` to point to suitable character strings.

If `ap_attribute_tab` is absent, then, if present, the value of `ap_tab` is used to control attribute formatting. If `ap_tab` is absent, then the default attribute formatting character sequence is used.

Similarly, if `ap_attribute_nl` is absent, then, if present, the value of `ap_nl` is used to control attribute formatting. If `ap_nl` is absent, then the default attribute formatting character sequence is used.

As above, for the most compact representation, you may wish to construct the writer using the arguments:

```
c_xml_writer l_writer( os, "", "" );
```

If the tab character nests the XML output too deeply, you might wish to choose the following form:

```
c_xml_writer l_writer( os, "  ", "\n" );
```

Note that not all character combinations used in the constructor will result in legal XML being generated.

To make use of per-unmarshalling process formatting feature you will need to use the advanced forms of unmarshalling as described in 3.5 - Marshaling (advanced forms).

3.26 - Specifying Micro Formats

A micro format is a simple type that has multiple values encoded into a single string. XML Schema's xs:dateTime family of types are micro formats, encoding multiple integer types such as the year, date of month and hour into a single string. You might choose to define a GPS coordinate or complex number (with real and imaginary parts) as a micro format.

You can tell LMX to treat a simpleType xs:string as a micro format using the -micro-format flag. This has the form:

```
-micro-format PATH CLASS
```

Here, `PATH` specifies the item(s) in the schema to which the micro format applies. It has the format:

```
/? (<name>/)* <name>
```

Note that only type specifications, either as an independently defined simpleType or a simpleType definition embedded in an element definition, can be assigned a micro format. An element that links to a separately defined simpleType can not have a micro format specification that is independent of the simpleType to which it links.

`CLASS` is the fully C++ namespace-qualified name of the C++ class used to represent the micro format.

An example micro format specification might be:

```
-micro-format position c_gps_coordinate
```

By default, it is assumed that the class that represents the micro format has a static method of the form:

```
static bool is_valid( const std::string & ar_v );
```

This will be used to test whether received XML values conform to the micro format. If such a method does not exist then you should specify a template specialization for the lmx::s_is_valid< T > template class located in the `MICRO_FORMAT_VALIDATORS` section of `lmxuser.h` to do this; for example:

```
template<>
struct s_is_valid< c_gps_coordinate >
{
    static bool is_valid( const std::string & ar_v )
        { return ...; }
};
```

You may wish to specify this in a separate `.h` file in which case you can use the `-include-h` flag to specify the file.

It is also assumed by default that that the micro format class has an assignment operator (`operator =`) that takes a `const std::string` reference to convert the XML input into the object, and an output stream operator (`operator <<` or non-class equivalent) that will generate appropriate XML. If this is not the case then see 3.13.1.2 - Input/Output Converters.

3.27 - Allowing Unknown Items in an XML Instance

A commonly requested feature is to allow auto-versioning of XML schemas. This allows code generated using older versions of an XML schema to accept XML instances that contain attributes and elements specified in a newer version of an XML schema, without generating errors. This enables the XML format to be extended, or versioned, over time as a productï¿½s feature set grows, in a simple and effective way.

LMX can generate code using two modes of versioning:

The first mode is enabled using the `-autover` flag. This will ignore any unknown attributes. In the case of an `xs:sequence` construct it will ignore any unknown child elements appended to the end of a sequence of elements. However, it will not ignore unknown elements positioned within the known set of elements. In the case of an `xs:choice` construct, an unknown element is allowed (and ignored) in place of the elements specified in the original XML schema.

The second mode is enabled using the `-autover2` flag (and is accessible in the GUI interfaces under the `Advanced -> Non-Standard Features...` menu option). This mode enhances the first mode by removing the restriction that the unknown elements must be located at the end of an `xs:sequence` construct, so that any unknown child element within an `xs:sequence` construct will be ignored. Its behavior when handling attributes and `xs:choice` constructs is the same as for the first mode.

To enable the code to be selective about which attributes and elements it ignores, the `c_xml_reader` class has the following two virtual functions, which are called when an unknown attribute or element is found when one of the auto-versioning modes is enabled:

```
virtual elmx_error user_found_unknown_attribute(
                        const std::string & r_attribute_name_in,
                        const c_string_stack & r_element_name_stack_in,
                        const char * p_prog_file_in,
                        int prog_line_in )
```

and:

```
virtual elmx_error user_found_unknown_element(
                        const std::string & r_element_name_in,
                        const c_string_stack & r_element_name_stack_in,
                        const char * p_prog_file_in,
                        int prog_line_in )
```

Where:

`r_attribute_name_in`
: The name of the unknown attribute, including namespace prefix.

`r_element_name_in`
: The name of the unknown element, including namespace prefix.

`r_element_name_stack_in`
: Reference to a class that contains the name of the element in which the unknown attribute or element is located. To convert this into a string like "/foo/bar", do: `r_element_name_stack_in.to_string()`.

`p_prog_file_in`
: The name of the C++ program file associated with the unknown attribute or element.

`prog_line_in`
: The line number within the C++ program file associated with the unknown attribute or element.

Parsing will continue if `ELMX_OK` is returned, and terminate (yielding the specified error code) if any other error code is returned.

3.28 - mustUnderstand / Comprehension Required

Some specifications allow extensions to specify that they must be understood, or the XML instance should not be parsed. There are a number of ways to do this. One way is to include an attribute in the extension indicating that such comprehension is required. SOAP takes this approach. To accommodate this approach, LMX provides the `has_must_understand` function. The prototype for this function is:

```
elmx_must_understand has_must_understand( c_xml_reader & ar_reader,
            const std::string &ar_must_understand_namespace,
            const std::string &ar_must_understand_name,
            t_must_understand_test_function a_function = bool_must_understand );
```

The return code for the function is of the type:-

```
enum elmx_must_understand { EMU_OK, EMU_ERROR, EMU_MUST_UNDERSTAND };
```

Where:-

EMU_OK
: indicates that a must understand attribute was not found, and no errors were encountered

EMU_ERROR
: indicates that an error was encountered during parsing

EMU_MUST_UNDERSTAND
: indicates that an attribute specifying that understanding is required was found.

The parameters to `has_must_understand` have the following meaning:-

ar_reader
: is a low level reader object as used in 2.3 - Unmarshaling (simple form)

ar_must_understand_namespace
: is the namespace of the attribute specifying must understand

ar_must_understand_name

is the local name of the attribute specifying must understand

a_function
: is an optional pointer to a function that indicates whether the attribute value specifies that comprehension is required or not. If this parameter is not present, then it is assumed that the attribute value is of type xs:boolean. The prototype for this function is:

  elmx_must_understand <must understand function name>(const std::string & ar_value);

  It should return `EMU_MUST_UNDERSTAND` if the extension must be understood and `EMU_OK` otherwise.

An example for SOAP might be:-

```
if( has_must_understand( reader,
           "http://www.w3.org/2002/12/soap-envelope",
           "mustUnderstand" ) == EMU_OK )
    // Process the block
```

Note that it is very important to call the `has_must_understand()` function at the correct place during XML instance processing, otherwise undesired results will occur. The blocks that contain `mustUnderstand()` attributes are expected to occur as part of `xs:any` elements contained in an instance. If information about an instance fragment contained in `xs:any` is understood, then it should be parsed without calling `has_must_understand()`, even if it has further `xs:any` elements that may not be understood. When a fragment from an `xs:any` element is found which is not understood, then `has_must_understand()` can be called to determine whether it must be understood. If it must be understood, then the whole XML instance must be discarded, otherwise the fragment can be ignored and processing can continue.

3.29 - Finding an XML Instance's Namespace

If your code must interpret XML instances from multiple schemas, you may wish to find the namespace of the root element of the XML instance in order to determine which parser needs to be used to unmarshal the instance. LMX provides the `get_instance_namespace` function for this purpose. The function has the following prototype:-

```
bool get_instance_namespace( c_xml_reader & ar_reader, std::string *ap_namespace_out );
```

Where:

ar_reader
: is a low level reader object created as in the unmarshaling examples above (see 2.3 - Unmarshaling (simple form))

ap_namespace_out
: is a pointer to a string where the namespace is to be placed if it is found.

The function returns `true` if the namespace is found, and `false` otherwise.

For example:-

```
std::string instance_namespace;
if( lmx::get_instance_namespace( reader, &instance_namespace ) )
{
    std::cout << "The instance namespace is: " << instance_namespace << "\n";
}
```

3.30 - Setting a Decimal Value With a Float

C++ provides no native support for the XML schema `xs:decimal` type. A convenient way to interface with such values is via C++ float values (although note that rounding errors may cause undesirable results in some situations). The LMX `c_decimal` class provides methods to read decimal values as floats. However, using a float to set a `c_decimal` type may require knowledge of the fraction digits facet which an instance of `c_decimal` will not have. To allow setting a `c_decimal` value with a fraction digits facet via a float LMX generates an extra function in this case of the form:-

```
set_NAME( double in );
```

As such a method may not be required in all builds, the code for this is generated in a conditionally compiled block. The code is included in the compilation if the #define:

```
LMX_WANT_DECIMAL_FLOAT_SET
```

is set to 1 in the `USER: CTRL_DEFS` section of `lmxuser.h` (the default case) and excluded otherwise.

3.31 - XML Billion Laughs Mitigation

The XML Billion Laughs attack is an attack that allows a small number of DTD entities to expand into a large amount of text (See https://en.wikipedia.org/wiki/Billion_laughs_attack). This can be mitigated against in LMX at compile-time using the `LMX_MAX_ENTITY_SIZE #define`, on a runtime global basis using the static `c_xml_reader::set_default_max_entity_size( size_t )` method and on a per unmarshal object basis using the `c_xml_reader::set_max_entity_size( size_t )` method.

The default maximum entity size is 1 million bytes. Note that for reasons of efficiency the size of an entity is only measured when starting to add a new entity. Thus entities may be returned that are larger than the maximum specified but this is not considered counter to mitigating this kind of attack.

To disable all DTD entity inclusions, set the limit to `0` using any of the above mechanisms.

Reading the values set by the above functions can be done using the respective methods `c_xml_reader::get_default_max_entity_size()` and `c_xml_reader::get_max_entity_size()`.

3.32 - Test Framework Generation

To help with evaluation and testing, LMX can generate a separate .cpp file containing code to exercise the generated marshaling and unmarshaling code and a `main` function. The name of this file is `<output root>-main.cpp`. The generated test code unmarshals to objects a specified XML instance from a file, assigns the object to a new object (using both assignment and construction) and then marshals the result back

out to a named file. If run under Windows, the test code will also check for memory leaks that occur during the test process - there should be none though. You can then compare the two files to check that the round tripping has been successful. For XML instances that do not use multiple namespaces or entity substitutions, the `xmlcomp.exe` program supplied in the evaluation support suite can be used to automate the comparison.

3.33 - Strategies for Increasing Code Flexibility

One aspect of using LMX generated code is that the code you write can end up being very sensitive to changes in the re-generated code after making small changes to a schema. This section describes some ways to write code that is more flexible.

One option is to generate code using the `-no-nested-classes` flag. If a schema has nested complex types, this effectively flattens it out. So rather than ending up with classes like:

```
class c_foo {
    class c_bar {
        class c_baz {};
}; };
```

you end up with classes like:

```
class c_baz {};
class c_bar {};
class c_foo {};
```

This avoids naming types such as `c_foo::c_bar::c_baz`.

Avoiding long chains of accessors also helps. Instead, first taking a pointer to a type at the end of a long chain of accessors makes the code easier to maintain. So instead of:

```
item.get_foo().get_bar().get_baz( 0 );
item.get_foo().get_bar().get_baz( 1 );
```

do:

```
c_bar * p_bar = &item.get_foo().get_bar();
p_bar->get_baz( 0 );
p_bar->get_baz( 1 );
```

Even better is to put the 'baz' code in a separate function, e.g.:

```
handle_baz( item.get_foo().get_bar() );
```

Making handle_baz() a template can make the code even more generic. For example:

```
template< class T >
void handle_baz( T * p_t )
{
    p_t->get_baz( 0 );
    p_t->get_baz( 1 );
}
```

Then it doesn't matter what type handle_baz() is called with.

3.34 - LMX Extensions

LMX supports some additional built-in data types. These types are defined in the namespace `http://codalogic.com/xsdtypes`. Note that use of these additional built-in types is suitable for internal proprietary schema, but is not recommended for schema that are made public.

The type `asciiString` defines a string that consists only of ASCII characters, and by default is mapped to std::string. The existence of this type allows easy co-existence of Unicode and ASCII values using a more natural representation in C++ code. An example of how to define the type is:

```
<xs:attribute xmlns:clt="http://codalogic.com/xsdtypes"
                  name="OwnerName" type="clt:asciiString"/>
```