

# A PERSISTENT CROSS-PLATFORM CLASS OBJECTS CONTAINER FOR C++ AND WXWIDGETS

MICHAL BLIŽŇÁK<sup>1</sup>, TOMÁŠ DULÍK<sup>2</sup>, VLADIMÍR VAŠEK<sup>3</sup>

Department of Applied Informatics<sup>1,2</sup>, Department of Automation and Control Engineering<sup>3</sup>

Faculty of Applied Informatics, Tomas Bata University

Nad Stráněmi 4511, 760 05, Zlín

CZECH REPUBLIC

bliznak@fai.utb.cz<sup>1</sup>, dulik@fai.utb.cz<sup>2</sup>, vasek@fai.utb.cz<sup>3</sup>

*Abstract:* This paper introduces a new open-source cross-platform software library written in C++ programming language which is able to serialize and deserialize hierarchically arranged class instances and their data members via XML files. The library provides easy and efficient way for processing, storing and managing complex object-oriented data with relationships between object instances. The library is based on mature cross-platform toolkit called wxWidgets and thus can be successfully used on many target platforms such as MS Windows, Linux or OS X. The library is published under open source licence and can be freely utilised in both open source and commercial projects. In this article, we describe the inner structure of the library, its key algorithms and principles and also demonstrate its usage on a set of simple examples.

*Keywords:* Data, class, persistence, container, serialization, XML, tree, list, C++, wxWidgets, wxXmlSerializer, wxXS

## 1 Introduction

The ability to store complex data processed by software applications is one of the most important features provided by various programming frameworks or software libraries. Most of them already offer some suitable technology, such as various types of configuration files, integrated XML parsers/builders or database layers. Unfortunately, majority of these technologies are designed to store only raw data (e.g. via database layers) or they require a lot of additional programming to process complex data types. Typical and most difficult case is the implementation of persistent storage for class instances and their hierarchy. In high-level programming languages like Java or Python, it is easy to solve this by using serialization, however, there are virtually no options for serialization in lower-level programming languages like C++.

If the requirement is to implement a simple storage for raw data, the best solution is probably to use a suitable database system/layer. Nowadays database servers and software libraries like JDO in Java [2] are able to store even very complex data via methods for mapping objects to database records. Moreover, some of the current database systems, for example SQLite database [3] or Firebird embedded [4], can be linked to an application as libraries allowing storing data to local filesystem or a database server.

Unfortunately, all of these database technologies lack the ability to preserve the hierarchical relations

between object class instances. It means the user can store data records but cannot define their hierarchy (who is the parent and who is the child, etc).

The goal of this paper is to introduce a new simple software library called wxXmlSerializer [8] (shortly wxXS) which fills the gap in the nowadays offer of available data persistence technologies. The wxXS is designed for storing not only raw data, but also their hierarchical relationship.

## 2 What the wxXmlSerializer is

Generally, the wxXS is a cross-platform software library written in C++ programming language based on wxWidgets [1] which offers a functionality needed for creation of **persistent hierarchical data containers** able to store various **C++ class instances** (can be regarded as complex **data records**). wxXS allows users to easily serialize hierarchically arranged class instances and their data members to an XML structure and deserialize them later. Currently supported data types serializable by the wxXS are:

- Generic data types such as: bool, char, int, long, float, double
- Most frequently used wxWidgets data types: wxString, wxPoint, wxSize, wxRealPoint, wxPen, wxBrush, wxFont, wxColour,
- wxArrayString, array of wxRealPoint

values, arrays of common generic data types and list of `wxRealPoint` values

- Dynamic or static instances of the serializable base class itself or its derivatives.

Moreover, the library architecture allows user to extend built-in list of supported data types by any other data type. A new data type can be added by implementing a new I/O handler, which is relatively easy piece of code composed of code macros provided by the library and small amount of manual programming.

The `wxXS` library can be used for wide range of application scenarios, e.g.:

- simple saving and loading of the program settings/configurations,
- as a persistent dynamic linked list of class instances encapsulating application data,
- as a persistent dynamic n-ary tree-based data container with methods needed for comfortable management of its items (useful for the software applications managing their data in tree controls, applications working with diagrams, etc.).

### 3 Used technologies

`wxXS` library was created as an add-on to well known cross-platform software library `wxWidgets` [1]. `wxWidgets` gives the programmers single, easy-to-use API for writing their applications on multiple platforms that still utilize the native platform controls and utilities. By linking the application with the appropriate library for the target platform (Windows/Unix/Mac, others coming shortly) using almost any popular C++ compiler, the application adopts the look and feel appropriate for that platform. On top of great GUI functionality, `wxWidgets` supports network programming, streams, clipboard and drag and drop, multithreading, image loading and saving in a variety of popular formats, database support, HTML viewing and printing, and much more [1].

`wxXS` library uses the **streams**, **XML** and **RTTI** classes provided by the `wxWidgets` so it can be used only together with this library. However, this “disadvantage” is balanced by the fact, that these crucial technologies are maintained and improved continuously by the wide and reliable open-source community. Moreover, the license policy [6] used by the `wxWidgets` library does not restrict the programmer in any way so the applications and derivatives based on the `wxWidgets` can be distributed

both as an open-source and/or commercial applications. `wxXS` itself is created under the same license so it can be used absolutely freely for any purpose, even for commercial projects.

### 4 The Library Structure

`wxXS` consists of three main classes encapsulating its basic functionality. It includes also several auxiliary classes encapsulating the I/O functionality for various data types and implements typed data containers used by the library. Now let's take a look to the purpose of the three main library classes which are:

- **wxXmlSerializer** class
- **xsSerializable** class
- **xsProperty** class

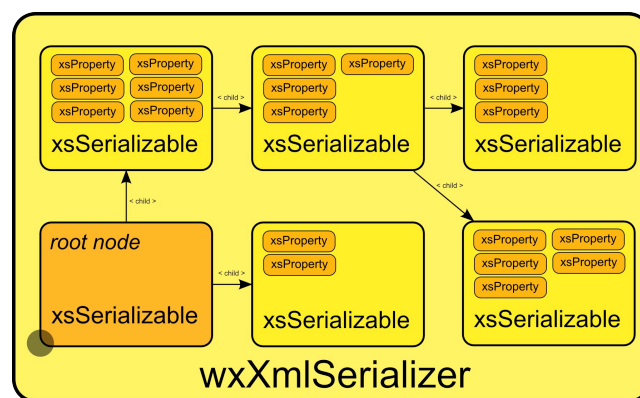


Figure 1: The library structure

**wxXmlSerializer** class is the main **data manager** class and implements the common data container functionality. Its member functions allow user to manage instances of serializable classes (encapsulated by the `xsSerializable`) and provide the I/O functionality like serialization and deserialization of stored serializable class objects. This class can be used as it is or as a base class for various derivations enhancing its built-in functionality.

**xsSerializable** class is the base class for so called “**serializable**” classes (i.e. classes manageable by the `wxXmlSerializer` class). It provides functionality needed for hierarchical arrangement of serialized class instances (every class instance includes linked list of another `xsSerializable` class instances, i.e. its *children*), I/O operations and it also holds information about serialized data members (instances of `xsProperty` class).

**xsProperty** class encapsulates a single data members (**properties**) of a serialized class object,

which is an instance of `xsSerializable` class. It stores information about memory address of the data member, its data type, default value, a name of the data member in the XML structure and flag telling whether the property should be serialized or not.

I/O and data conversion operations provided by the `xsSerializable` class are performed via the **`xsPropertyIO`** I/O handler class and its derivatives, which are responsible for conversion of serialized property values to/from its string representation and for reading and writing of this textual information from/to the XML structure.

Except built-in support for common data types the user can simply create new I/O handler class by using set of code macros defined in the library headers. This powerful feature will be discussed in more details later.

Relationship between the main library's classes is shown on figure 2.

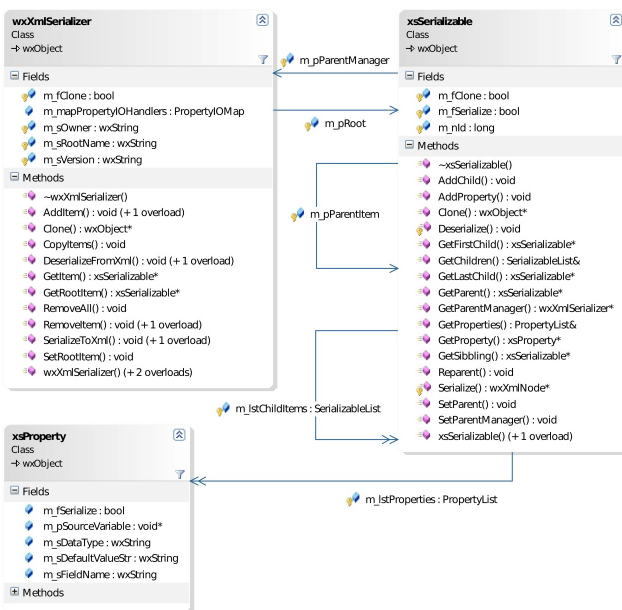


Figure 2: Relationship of main library's classes

## 5 wxXS: User Guide

wxXS library can be used in many ways but the main idea is following: the user can create classes derived from **`xsSerializable`** base class and then define which of the class members will be serialized and which not. For serialization of these class instances, it is necessary to add them to a data manager, which is an instance of **`wxXmlSerializer`** class. There is also one instance of **`xsSerializable`** class called a **root item** included in the data manager class as its member object. All the other serialized objects added to the data manager are inserted into the linked list of

the root item. Serialized class data members called properties are encapsulated by an **`xsProperty`** class

The `xsProperty` class instances can be created in several ways:

- using universal macros  
`XS_SERIALIZE(member, field)` OR  
`XS_SERIALIZE_EX(member, field, defval)`
- using one of defined macros designed for particular data type (e.g.  
`XS_SERIALIZE_LONG(member, field)` or  
`XS_SERIALIZE_LONG_EX(member, field, defval)`
- using the function  
`xsSerializable::AddProperty(xsProperty *property)`

The argument *member* is the name of data member, which should be serialized and the argument *field* is a name used for its identification in the output XML structure. The macros must be placed somewhere in a class implementation code (typically in a constructor).

Macros with suffix “\_EX” in their names allow user to define default property value. In this case, the property is serialized only if its current value differs from the default one. This approach leads to smaller size of the output XML structure because only changed property values are serialized.

Now let's illustrate these mechanisms on a simple console application which serializes simple class instance and its member data to an XML file.

The first needed step is declaration of a serializable class encapsulating an application data. Of course, also basic headers files provided by `wxWidgets` and `wxXmlSerializer` libraries have to be inserted into source code.

### Example 1:

```
// wxWidgets main header file
#include <wx/wx.h>

// wxXmlSerializer main header file
#include
"wx/wxmlserializer/XmlSerializer.h"
////////////////////////////////////////
// SerializableObject class
////////////////////////////////////////

class SerializableObject : public
xsSerializable
{
// RTTI must be provided by the class
DECLARE_DYNAMIC_CLASS(SerializableObject);
// constructor
```

```

SerializableObject();
// destructor
virtual ~SerializableObject() {}

// protected data member
wxString m_sTextData;

private:
// private data member
static int m_nCounter;
};

```

The implementation of the serializable class declared above is straightforward as well:

```

// constructor
SerializableObject::SerializableObject()
{
// initialize member data
m_sTextData = wxString::Format(
wxT("'SerializableObject' class instance No.
%d"), m_nCounter++ );

// mark the data members which should
// be serialized
XS_SERIALIZE( m_sTextData, wxT("text") );
}

// static data members
int SerializableObject::m_nCounter = 0;

// implementation of RTTI for serializable
// class
IMPLEMENT_DYNAMIC_CLASS( SerializableObject,
xsSerializable );

```

Now the data manager (serializer) class must be defined to handle instances of the SerializableObject class. In this example, the standard wxXmlSerializer class is used. Generally, there are two ways how to handle the serializable class objects by the serializer class:

- one serializable objects can be set as a root node of the serializer,
- several serializable objects can be appended to a default root node included in the serializer or to another already managed serializable objects..

These quite different approaches differ in the way the stored serialized class instance can be handled and accessed. For better understanding, both of these ways are discussed bellow.

Let us use the first mentioned way to serialize the Settings class object to a file called "data.xml" stored on a harddrive:

```

////////////////////////////////////
// The application's entry point

int main( int argc, char ** argv )
{

```

```

// Create a serializer object.
wxXmlSerializer xml_IO;

// Initialize the serializer.
xml_IO.SetSerializerOwner( wxT("Sample") );
xml_IO.SetSerializerRootName( wxT("data") );
xml_IO.SetSerializerVersion( wxT("1.0.0") );

// Create a serialized settings class
// object with its default values.
SerializableObject *m_pData = new
SerializableObject();
if( m_pData )
{
// Insert the object into serializer as
// its root node.
xml_IO.SetRootItem(m_pData);

xml_IO.SerializeToXml( wxT("data.xml"),
xsWITH_ROOT );
}
}

```

A content of the output XML file is now:

```

<?xml version="1.0" encoding="utf-8"?>
<data owner="Sample" version="1.0.0">
  <data_properties>
    <object type="SerializableObject">
      <property name="text"
type="string">'SerializableObject' class
instance No. 0</property>
    </object>
  </_properties>
</settings>

```

The stored XML file can be loaded back in a simple way:

```

if( wxFileExists( wxT("data.xml") ) )
{
// Load settings from file
m_XmlIO.DeserializeFromXml( wxT("data.xml
") );
}

```

After deserialization, loaded data can be accessed in a very simple way using a function called wxXmlSerializer::GetRootNode(), which returns a pointer to the serializer's root node (in our example to a class object encapsulating the stored data).

The second possible way of managing the stored serializable objects is illustrated in the next example. In this case the serializable class instances are stored in a list encapsulated by the root serializer's node. Except this example, slightly modified source code presented later shows how the serializable class instances can be arranged into a tree structure as well.

So, let us create a few of these nodes, add them to the serializer object and store the serializer content to a disk file. Note, that the serializable class used in

examples 2 and 3 is the same like introduced in example 1.

### Example 2:

```

////////////////////////////////////
// The application's entry point

int main( int argc, char ** argv )
{
    // create instance of XML serializer
    wxXmlSerializer xml_IO;

    // first, create set of serializable
    // class objects and add them to the
    // serializer
    for( int i = 0; i < 5; i++ )
    {
        // Add all new class objects to the
        // serializer's root so the instances
        // will be arranged into a list. Note
        // that each serializable object
        // can be assigned as a child to another
        // one so the objects could be arranged
        // into a tree structure as well.
        xml_IO << new SerializableObject();
    }

    // store the serializer's content to an
    // XML file:
    xml_IO.SerializeToXml( wxT("data.xml") );

    // clear the serializer's content:
    xml_IO.RemoveAll();

    // now, re-create list of stored class
    // instances from XML file (data.xml):
    xml_IO.DeserializeFromXml( wxT("data.xml")
);

    // finally, print out content stored in
    // loaded class instances:
    SerializableList::Node *node =
xml_IO.GetRootItem()->GetFirstChildNode();

    while( node )
    {
        SerializableObject pObj =
(SerializableObject*)node->GetData();

        wxPrintf( pObj->m_sTextData << wxT("\n") );

        node = node->GetNext();
    }

    return 0;
}

```

A content of the output XML file created by previous code is:

```

<?xml version="1.0" encoding="utf-8"?>
<root owner="" version="">
  <object type="SerializableObject">
    <property name="id"
type="long">1</property>
    <property name="text"
type="string">'SerializableObject' class
instance No. 0</property>

```

```

</object>
  <object type="SerializableObject">
    <property name="id"
type="long">2</property>
    <property name="text"
type="string">'SerializableObject' class
instance No. 1</property>
  </object>
  <object type="SerializableObject">
    <property name="id"
type="long">3</property>
    <property name="text"
type="string">'SerializableObject' class
instance No. 2</property>
  </object>
  <object type="SerializableObject">
    <property name="id"
type="long">4</property>
    <property name="text"
type="string">'SerializableObject' class
instance No. 3</property>
  </object>
  <object type="SerializableObject">
    <property name="id"
type="long">5</property>
    <property name="text"
type="string">'SerializableObject' class
instance No. 4</property>
  </object>
</root>

```

Screenshot of the application from Example 2 is shown on figure 3.

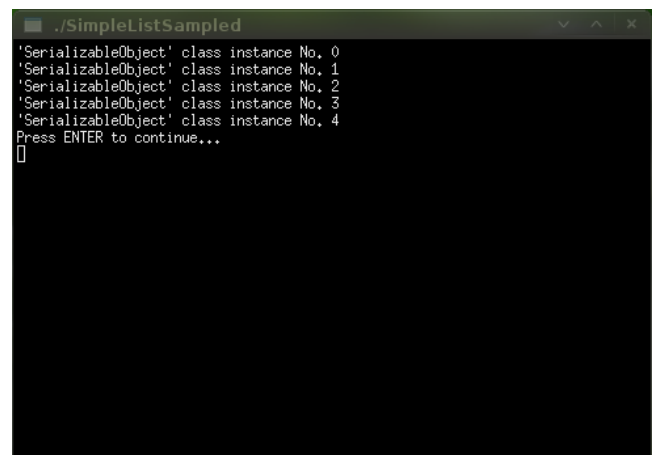


Figure 3: Application from Example 2.

As can be seen from the source code, an overloaded operator '<<' was used for adding the serializable class instances into the serializer's root node. For this task also more universal function `wxXmlSerializer::AddChild()` can be used as follows:

```

xml_IO.AddItem( (xsSerializable*)NULL, new
SerializableObject() );

```

The first function argument determines which already managed serializable class instance will be a parent of new class object specified by the second parameter. If

the parent is NULL, then the new appended object will be assigned directly to the serializer's root node. It is obvious that any n-ary tree structure can be constructed in this way. Following modified source code shows how a tree structure consisting of serializable class instances can be created and serialized by the library.

### Example 3:

```

////////////////////////////////////
// Auxilary functions

xsSerializable* MakeTree( xsSerializable
*parent, int levels )
{
    if( levels > 0 )
    {
        levels--;
        // add new instances of serializable
        // object to given parent and pass these
        // new instances as parents for
        // recursive call of this function
        // ( operator << returns pointer to
        // newly added object ):
        MakeTree( *parent << new
SerializableObject(), levels );
        MakeTree( *parent << new
SerializableObject(), levels );

        // also member function of
        // xsSerializable class can be used for
        // this task as follows:
        //MakeTree( parent->AddChild( new
SerializableObject() ), levels );
    }

    return parent;
}

void PrintTree( xsSerializable *parent, int
level )
{
    level++;

    // iterate through children list of given
    // parent:
    SerializableList::Node *node = parent-
>GetFirstChildNode();

    while( node )
    {
        SerializableObject *pObject =
(SerializableObject*) node->GetData();

        // print info about processed object:
        for( int i = 1; i < level; i++ )
        {
            wxPrintf( wxT(" ") );
        }

        wxPrintf( pObject->m_sTextData << wxT("\n") );

        // process the object's children:
        if( pObject->HasChildren() )
        {
            PrintTree( pObject, level );
        }
    }
}

```

```

        node = node->GetNext();
    }
}

////////////////////////////////////
// The application's entry point

int main( int argc, char ** argv )
{
    // create instance of XML serializer
    wxXmlSerializer xml_IO;

    // first, create set of serializable class
    // objects and add them to the serializer
    // (add a root item of a tree of
    // serializable class objects to the
    // serializer
    xml_IO << MakeTree( new
        SerializableObject(), 3 );

    // also member function of wxXmlSerializer
    // class can be used for this task as
    // follows:
    // (first NULL argument means that the
    // object is added directly to the
    // serializer's root):
    // xml_IO.AddItem( (xsSerializable*)NULL,
    MakeTree( new SerializableObject(), 3 ) );

    // store the serializer's content to an
    // XML file:
    xml_IO.SerializeToXml( wxT("data.xml") );

    // clear the serializer's content:
    xml_IO.RemoveAll();

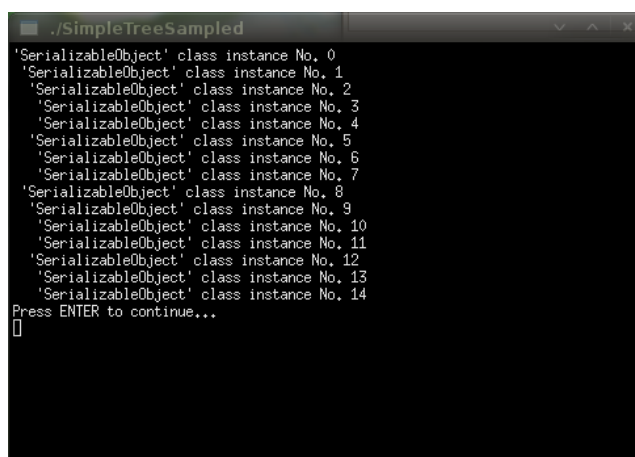
    // now, re-create stored class instances
    // from the XML file (data.xml)
    xml_IO.DeserializeFromXml( wxT("data.xml")
);

    // finally, print out info about loaded
    // clas instances:
    PrintTree( Serializer.GetRootItem(), 0 );

    return 0;
}

```

An possible output of the application discussed in Example 3 could be seen figure on 4.



```

./SimpleTreeSampled
'SerializableObject' class instance No. 0
'SerializableObject' class instance No. 1
'SerializableObject' class instance No. 2
'SerializableObject' class instance No. 3
'SerializableObject' class instance No. 4
'SerializableObject' class instance No. 5
'SerializableObject' class instance No. 6
'SerializableObject' class instance No. 7
'SerializableObject' class instance No. 8
'SerializableObject' class instance No. 9
'SerializableObject' class instance No. 10
'SerializableObject' class instance No. 11
'SerializableObject' class instance No. 12
'SerializableObject' class instance No. 13
'SerializableObject' class instance No. 14
Press ENTER to continue...

```

Figure 4: Application from Example 3.

Note, that after successful deserialization the stored class instances can be accessed by set of member functions declared in `sxsSerializable` class like:

- `sxsSerializable::GetParent`,
- `sxsSerializable::GetFirstChild`,
- `sxsSerializable::GetLastChild`,
- `sxsSerializable::GetSibbling`,

etc. In addition, various member functions of `wxXmlSerializer` class can be used. For more information about all available data handling functions supported by the `wxXS` library please see the reference documentation available at [8].

## 6 Extending The `wxXmlSerializer`

There are many built-in data types supported directly by the `wxXS` but aside of them, also custom data types can be processed by the library. For this case, the `wxXS` provides set of code macros and classes suitable for a creation and registration of user-defined I/O handlers.

In the following example a new I/O handler suitable for serialization/deserialization of `wxColourData` class is created and used. Note, that the example application is one of sample projects distributed together with the library source code. It is highly recommended to study reference documentation as well as the supplied sample projects for full understanding of discussed topics.

So, first of all, the programmer must declare new I/O handler class and define code macros used for marking of a serialized data members. This should be done in appropriate header file. The declaration code can be as follows:

### Example 4:

```

////////////////////////////////////
// Declaration of new I/O handler
////////////////////////////////////

// Declaration of a class
// 'xsColourDataPropIO' encapsulating the
// custom property I/O handler for
// 'wxColourData' data type.
XS_DECLARE_IO_HANDLER(wxColourData,
xsColourDataPropIO);

// Code macros which create new serialized
// wxColourData property
#define XS_SERIALIZE_COLOURDATA(x, name)
XS_SERIALIZE_PROPERTY(x, wxT("colourdata"),
name);

#define XS_SERIALIZE_COLOURDATA_EX(x, name,
def) XS_SERIALIZE_PROPERTY_EX(x,

```

```

wxT("colourdata"), name,
xsColourDataPropIO::ToString(def));

```

Let us discuss the code listed above in more details. The macro `XS_DECLARE_IO_HANDLERS` declares a new class called `xsColourDataPropIO` suitable for processing of data members with data type `wxColourData` which is a class provided by the `wxWidgets` used for data transfer between an application and the color picker dialog. User-defined macros `XS_SERIALIZE_COLOURDATA` and `XS_SERIALIZE_COLOURDATA_EX` can be later used in the implementation code to mark `wxColourData` class members in the similar way as the `XS_SERIALIZE` macro was used in the previous examples. Note that the text string "colourdata" must be a unique identifier used for identification of this data type in serialized XML structure.

Now see the implementation code:

```

// Define custom data I/O handler
XS_DEFINE_IO_HANDLER(wxColourData,
xsColourDataPropIO);

// Two following static member functions of
// the data handler class MUST
// be defined manually:

// wxString xsPropIO::ToString(T value) ->
// creates a string representation of the
// given value:

wxString
xsColourDataPropIO::ToString(wxColourData
value)
{
    wxString out;

    out << xsColourPropIO::ToString(
        value.GetColour());

    for(int i = 0; i < 16; i++)
    {
        out << wxT("|") <<
            xsColourPropIO::ToString(value.
                GetCustomColour(i));
    }
    return out;
}

// T xsPropIO::FromString(const wxString&
// value) -> converts data from
// given string representation to its
// relevant value:

wxColourData
xsColourDataPropIO::FromString(const
wxString& value)
{
    wxColourData data;

    if(!value.IsEmpty())
    {

```

```

    int i = 0;
    wxStringTokenizer tokens(value,
        wxT("|"), wxTOKEN_STRTOK);

    data.SetColour(xsColourPropIO::FromString(
        tokens.GetNextToken()));

    while(tokens.HasMoreTokens())
    {
        data.SetCustomColour(i,
            xsColourPropIO::FromString(
                tokens.GetNextToken()));
        i++;
    }
    return data;
}

```

The most of the implementation effort is hidden in the XS\_DEFINE\_IO\_HANDLER macro. Here, the programmer must manually create only two static functions responsible for conversion of processed data value to its string representation and vice versa. These static functions are then internally used by core library classes for serialization and deserialization but they can be used also for any other purposes. For example, in the code above you can see similar static functions called xsColourPropIO::FromString() and xsColourPropIO::ToString() defined by built-in I/O handler designed for processing of wxColour data members.

Now, let's declare and define a class containing member data of type wxColourData and mark it for serialization via previously user-defined I/O handler:

```

////////////////////////////////////
// Settings class declaration
////////////////////////////////////

class Settings : public xsSerializable
{
public:
    // RTTI and xsSerializable::Clone()
    // function must be provided
    XS_DECLARE_CLONABLE_CLASS(Settings);
    // constructor
    Settings();
    // copy onstructor needed by default
    // implementation of
    // xsSerializable::Clone() function
    Settings(const Settings &obj);
    // destructor
    virtual ~Settings();

    // public data members
    wxColourData m_colourData;
};

////////////////////////////////////
// Settings class implementation
////////////////////////////////////

XS_IMPLEMENT_CLONABLE_CLASS(Settings,
xsSerializable);

```

```

Settings::Settings()
{
    // set default values of data member
    m_colourData.SetColour(*wxBLUE);

    for(int i = 0; i < 16; i++)
    {
        m_colourData.SetCustomColour(i,
            wxColour(i*16, i*16, i*16));
    }

    // serialize colour data
    XS_SERIALIZE_COLOURDATA(m_colourData,
        wxT("colordlg_content"));

    // this version of mark macro causes the
    // data will be serialized only if its
    // current value differs from the default
    // one (the last macro parameter):

    //XS_SERIALIZE_COLOURDATA_EX(m_colourData,
        wxT("colordlg_content"), m_colourData);
}

Settings::Settings(const Settings &obj)
{
    // set default values of adata member
    m_colourData = obj.m_colourData;

    // serialize colour data everytime
    XS_SERIALIZE_COLOURDATA(m_colourData,
        wxT("colordlg_content"));

    // this version of mark macro causes the
    // data will be serialized only if its
    // current value differs from the default
    // one (the last macro parameter):

    //XS_SERIALIZE_COLOURDATA_EX(m_colourData,
        wxT("colordlg_content"), m_colourData);
}

Settings::~Settings()
{
}

```

Note that new macros XS\_DECLARE\_CLONABLE\_CLASS and XS\_DECLARE\_CLONABLE\_CLASS were used in the code. These macros differ from DECLARE\_DYNAMIC\_CLASS and IMPLEMENT\_DYNAMIC\_CLASS macros in such way that they implement also Clone() function for the class, which can be used for retrieving the exact copy of the class instance. This function is further used by wxXmlSerializer::CopyItems() member function and its copy constructor so a whole serializer's content can be copied in a single program line.

The last step needed for proper initialization of the new I/O handler class is its registration. It should be done as soon as possible, typically in the application initialization code. For registration of the I/O handler, the XS\_REGISTER\_IO\_HANDLER macro can be



used as shown in the following code taken from mentioned sample project:

```

////////////////////////////////////
// CustomDataSampleApp class
////////////////////////////////////

bool CustomDataSampleApp::OnInit ()
{
    // load application settings if the
    // configuration file exists, otherwise
    // create new settings class object with
    // default values

    // initialize serializer (m_XmlIO class
    // member)

    m_XmlIO.SetSerializerOwner (wxT ("CustomData
SampleApp"));

    m_XmlIO.SetSerializerRootName (wxT ("setting
s"));

    m_XmlIO.SetSerializerVersion (wxT ("1.0.0"));

    // register new property I/O handler
    // 'xsColourDataPropIO' for data type with
    // name 'colourdata'

    XS_REGISTER_IO_HANDLER (wxT ("colourdata"),
xsColourDataPropIO);

    // create serialized settings class object
    // manually with default values
    m_pSettings = new Settings ();
    // insert settings class object into
    // serializer as its root node
    m_XmlIO.SetRootItem (m_pSettings);

    if ( wxFileExists (wxT ("settings.xml")) )
    {
        // load settings from configuration file
        m_XmlIO.DeserializeFromXml (wxT ("settings
.xml"));
    }
    // do some other initialization step ...

    return true;
}

```

The application settings could be saved to an XML file later by code like this:

```

int CustomDataSampleApp::OnExit ()
{
    // serialize settings to XML file
    m_XmlIO.SerializeToXml (wxT ("settings.xml")
, xsWITH_ROOT);

    return 0;
}

```

Figure 5 shows a main application frame of sample project mentioned above which demonstrates a creation of user-defined data handlers.

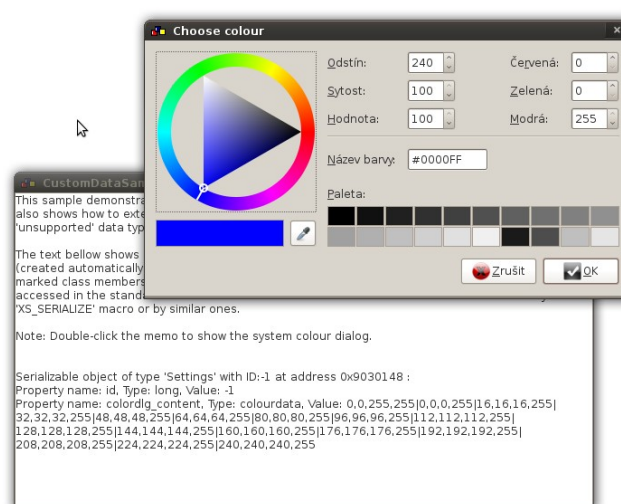


Figure 5: Sample project demonstrating user-defined data handlers

## 7 Conclusion

The aim of the paper was to introduce a new cross-platform software library wxXmlSerializer (wxXS) suitable for easy and elegant creation of serializable hierarchical data containers using the wxWidgets library and C++ programming language. The library is available as an open-source project and can be used for both commercial and open-source software applications.

The wxXS was already successfully used as a technological background for various projects such as the wxShapeFramework [7] cross-platform graphics library or the UML code generation tool called CodeDesigner developed at the Tomas Bata University.

The library features described in this document are only a tiny fraction of comprehensive functionality provided by the library. For more information about its usage and abilities please see the library reference or code examples available at the Source Forge web site (<http://www.sourceforge.net/wxxs>).

## 8 Acknowledgements

This work was supported by the Ministry of Education of the Czech Republic under grant No. MSM7088352102.

## 9 References

- [1] J. Smart, K. Hock, S. Csomor, *Cross-Platform GUI Programming with wxWidgets*, Prentice Hall PTR, 2006

- [2] Java Data Objects (JDO) at Sun Developer Network (SDN), 2008:  
<http://java.sun.com/jdo/http://java.sun.com/jdo/>
- [3] SQLite database home website, 2008:  
<http://www.sqlite.org/>
- [4] Firebird database home website, 2008:  
<http://www.firebirdsql.org/>
- [5] wxWidgets home website, 2008:  
<http://www.wxwidgets.org/>
- [6] wxWidgets license documents, 2008:  
<http://www.wxwidgets.org/about/newlicen.htm>
- [7] wxShapeFramework library website, 2008: <http://sourceforge.net/projects/wxsf>
- [8] wxXmlSerializer library website, 2008:  
<http://sourceforge.net/projects/wxxs>