

# Object Serialization in NMM

Motama GmbH, Saarbruecken, Germany  
(<http://www.motama.com>)

April 2010

Copyright (C) 2005-2010  
Motama GmbH, Saarbruecken, Germany  
<http://www.motama.com>

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being all sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found in the file COPYING.FDL.

THE DOCUMENT IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE DOCUMENT BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE DOCUMENT OR THE USE OR OTHER DEALINGS IN THE DOCUMENT.

## 1. Introduction

Whenever an application needs to save objects in a file or send it over a network it must serialize these objects to a formatted byte stream that stores all attributes. If the application reads the byte stream again it creates a corresponding object and sets the parameter or the object itself reads the attributes. If more than one object type is used a unique type-id is need to recreate the correct object from byte stream.

For i/o facilities C++ uses different kind of streams like Io-streams for standard input/output or fstreams for files. A stream offers operators (normally the operator<< to write and operator>> to read from stream) to read/write the generic data-types. These predefined operators can be extended with additional operators

for user defined data-types. A disadvantage of standard C++ streams is the lack of type information. Objects written to a stream lose any type information and if an object is recreated the stream can not be asked for type of the actual element. So the application developer must write some kind of 'magic number' to identify elements written to a stream. Therefore NMM uses so called ValueStreams which stores the type information of each element. These type information are used for a runtime type safe object serialization/deserialization.

Since the size of C++ data types depends on the corresponding platform, NMM provides platform independent definitions, that are called POD types, as an acronym for "plain old data", for all basic C++ data types and specifies the size for each data type. The size of a POD type is valid on all supported platforms. All POD types are defined in "nmm/nmm\_types.hpp". Please note that one must use these POD types in the context of object serialization. Otherwise a correct communication between different platforms can not be assured. Table 1 shows all POD types defined in NMM.

**Table 1. POD Types defined in NMM**

POD Type	Size of POD Type	Description
nmm_bool	8 bit	Stores a boolean
nmm_char	8 bit	Stores a signed character
nmm_uchar	8 bit	Stores a unsigned character
nmm_int16	16 bit	Stores a signed integer with 16 bits
nmm_int32	32 bit	Stores a signed integer with 32 bits
nmm_int64	64 bit	Stores a signed integer with 64 bits
nmm_uint16	16 bit	Stores a unsigned integer with 16 bits
nmm_uint32	32 bit	Stores a unsigned integer with 32 bits
nmm_uint64	64 bit	Stores a unsigned integer with 64 bits
nmm_float	32 bit	Stores a IEEE single-precision floating point number
nmm_double	64 bit	Stores a IEEE double-precision floating point number

In this context we use the following class as example to show the use of ValueStreams.

```
namespace MyNamespace {
    class MyStruct {
    public:
        MyStruct();
        MyStruct(const MyStruct&);
    };
}
```

```

        string&      getName ()           {return __name;}
const string&      getName ()           const {return __name;}

const list<nmm_float>& getValueList ()    const {return __value_list;}
list<nmm_float>&    getValueList ()       {return __value_list;}
nmm_int32         getCounter ()          const {return __counter;}
void              setCounter(nmm_int32 i) {__counter = i;}

private: //The attributes
string          __name;
nmm_int32       __counter;
list<nmm_float> __value_list;
};
}

```

In general it is sufficient that the attributes of a serializable class use POD types. However, it is recommended that the methods that provide access to these attributes also use the data types defined in NMM.

Section 2 describes the ValueStreams and their specific extensions. Section 3 shows the use of ValueStreams as well as some NMM specific classes that simplify object serialization.

## 2. ValueStreams

For a runtime type safe object serialization/deserialization NMM offers so called ValueStreams. Like C++ streams ValueStreams differ between input and output ValueStreams. Output ValueStreams or so called OValueStream are used for object serialization and input ValueStreams, so called IValueStreams, for deserialization. Both ValueStreams offer operators to read or write POD types but store additional type information in form of a TypeInfo-struct. This TypeInfo-struct stores the typename and its namespace as string. These type information are used to identify the type of the current element.

Please note that one must use POD types to read or write basic C++ data types from or to a Valuestream.

The rest of this section describes the extended interface of ValueStreams in comparison to normal C++ streams. Section 2.1 describes the extensions of OValueStream and section 2.2 of IValueStreams.

### 2.1. Extensions of OValueStream

Inside a formatted byte stream of several serialized objects a ValueStream must know the begin and the end of a object. Therefore OValueStreams offers the following two methods.

```

(1) void beginType(const TypeInfo&);
(2) void endType();

```

The first method initialize a new user defined type and expect a TypeInfo as parameter. All objects written to the ValueStream are member of the initialized type until the second method is called. An example in section 3.2 shows how to use these methods.

## 2.2. Extensions of IValueStream

As described in the introduction IValueStreams offers methods to read the type information of stored objects. Therefore IValueStreams offers the following two methods

```
(3) string getCurrentTypeInfo()      const;
```

Method (3) returns the TypeInfo-struct of the actual object and the. This method allows a runtime type check which is shown in section 3.2 .

The following methods allow access and navigation to the attributes of an object inside a ValueStream.

```
(5) bool goDown();
(6) void goUp();
(7) bool isEmpty() const;
(8) bool nextElement();
```

Method (5) allows access to the attributes of an object. If the return value is true the attributes inside the objects can be accessed, otherwise no internal attributes are available. Method (6) leaves the section with the internal attributes and skip to the next element.

Consider you have an IValueStream that stores two objects of type MyStruct, my\_struct1 and my\_struct2. If you call method (3) the IValueStream returns the TypeInfo of my\_struct1 which is **MyStruct** and the namespace **MyNamespace**. Method (5) allows the access to the internal attributes if return value is true. All described methods belong to the current position inside the stream, so the return value of method (3) is now **string** which is the type of MyStruct's first attribute `__name`. The IValueStream leaves the section with my\_struct1's attributes if method (6) is called and skip to the next element my\_struct2. All unread attributes of my\_struct1 are lost. So you should check with method (7) if all attributes are read. If you read all attributes of my\_struct1 the return value method (7) is true. After calling method(6) the return value of method(7) is false because the ValueStream stores my\_struct2 too. Method (8) allows to skip an element inside the stream. If you read a type from stream the ValueStream goes to the next element automatically so you must call this method if you skip one element. Section 3.1.2 shows an example of these methods.

## 3. Object Serialization with ValueStreams

Like in a classic object oriented manner the object itself is responsible for serialization and deserialization by implementing some abstract methods of a superclass. The lack of this approach is to integrate existing classes from libraries without changing the source code. Therefore ValueStream can be used in

two ways. The first one, described in section 3.1 , is to implement a C++ operator<< and operator >> for ValueStreams which is useful to use structs from existing libraries without changing their source code. The second one is to use some NMM specific base classes, described in section 3.2 , which simplify object serialization.

A reference implementation of both versions can be found in the current NMM example directory (nmm/example/serialize/).

## 3.1. Usage of C++ Operators

To reuse existing classes and structs in NMM you can implement the operator<< and operator>> for ValueStreams.

### 3.1.1. Operator<<

Writing an operator<< for an OValueStream is as simple as for a common ostream. The main difference is that you must initialize a new type with its typename and a namespace. The OValueStream offers the method **beginType(const TypeInfo&)** to start a user defined type and **endType()** to close it. The operator<< for the class MyStruct looks like:

```
OValueStream& operator<<(OValueStream& o_stream, const MyStruct& my_struct) {
    //Initialize a complex type with its typename MyStruct
    o_stream.beginType (TypeInfo ("MyStruct", "MyNamespace"));

    o_stream << my_struct.getName();           //Write the name
    o_stream << my_struct.getCounter();        //Write the counter
    o_stream << my_struct.getValueList();      //Write the list.

    o_stream.endType();                        //End the new type

    return o_stream;                          //return a reference to o_stream
}
```

Well thats it. Now we can serialize MyStruct with every ValueStream. Please note that one must use POD types to read or write basic C++ data types from or to a Valuestream. So if a class does not returns a POD type one must convert it to POD type manually before it is written to a OValueStream.

### 3.1.2. Operator>>

The corresponding operator>> for IValueStreams performs the type check and looks like:

```
IValueStream& operator>>(IValueStream& i_stream, MyStruct& my_struct) {
    /*Begin> Type checking */
    if (i_stream.getCurrentTypeInfo() != TypeInfo("MyStruct", "MyNamespace")) {
        cerr << "Invalid Type assignment" << endl;
    }
}
```

```

    // Throw an exception
    throw InvalidTypeInfoException(i_stream.getCurrentTypeInfo(),
                                  TypeInfo("MyStruct", "MyNamespace"));
}
/*End< */

/*Begin> Try to access the attributes */
if (!i_stream.goDown()) {
    cerr << "MyStruct has no attributes" << endl;
    i_stream.nextElement(); //Don't forget to skip this element.
    return i_stream;
}
/*End< */

/*Begin> read the attributes in the same order as they are written*/
i_stream >> my_struct.getName(); //reads the name

nmm_int32 counter;
i_stream >> counter; //reads the counter
my_struct.setCounter(counter);

i_stream >> my_struct.getValueList(); //reads the value_list
/*End<*/

i_stream.goUp(); //Don't forget to go up that calls nextElement() too.

return i_stream; //return a reference to the current ValueStream
}

```

Please note that one must use POD types to read or write basic C++ data types from or to a Valuestream. So if a class does not accept a POD type one must read it as a POD type and convert it manually to corresponding type.

### 3.1.3. Inheritance

This section describes the use of inheritance. Therefore we introduce a new class DerivedStruct that is derived from MyStruct and lives in the same namespace.

```

namespace NMM {
class DerivedStruct : public MyStruct {
public:
    DerivedStruct();
    DerivedStruct(const DerivedStruct&);

    nmm_uint32 getID() const { return __id; }
    void setID(nmm_uint32 id) { __id = id; }
private:
    nmm_uint32 __id;
}

```

### 3.1.3.1. C++ Operators<<

If we serialize the new class `DerivedClass` we must serialize the attributes from the base class `MyStruct` too.

```
OValueStream& operator<<(OValueStream& o_stream, const DerivedStruct& my_struct) {
    //Initialize a complex type with its typename DerivedStruct
    //and the namespace MyNamespace .
    o_stream.beginType(TypeInfo("DerivedStruct", "MyNamespace"));

    //enforce serialization of the base class.
    o_stream << static_cast<const MyStruct&>(my_struct);

    o_stream<< my_struct.getID();                //Write the id

    o_stream.endType();                          //End the new type

    return o_stream;                            //return a reference to o_stream
}
```

First we write the new type information. Then we use the `operator<<` from class `MyStruct` to serialize the attributes from the base class. This is similar to rewrite an `operator=` from a base class. In the new implementation you must call the `operator=` from the base class too. Then write the additional parameter.

### 3.1.3.2. C++ Operators>>

To deserialize the new class we first check the type information and call the `operator>>` from the base class.

```
IValueStream& operator>>(IValueStream& i_stream, DerivedStruct& my_struct) {
    /*Begin> Type checking */
    if (i_stream.getCurrentTypeInfo() != TypeInfo("DerivedStruct", "MyNamespace")) {
        cerr << "Invalid Type assignment" << endl;
        // Throw an exception
        throw InvalidTypeException(i_stream.getCurrentTypeInfo(),
                                   TypeInfo("DerivedStruct", "MyNamespace"));
    }
    /*End< */

    /*Begin> Try to access the attributes */
    if (!i_stream.goDown()) {
        cerr << "MyStruct has no attributes" << endl;
    }
}
```

```

    i_stream.nextElement(); //Don't forget to skip this element.
    return i_stream;

/*End< */

operator>>(i_stream, static_cast<MyStruct&>(my_struct)); //deserialize the base class

/*Begin> read the attributes in the same order as they are written*/
nmm_uint32 id;
i_stream >> id;          // reads the id
my_struct.setId(id);    // set the ID
/*End<*/

i_stream.goUp(); //Don't forget to go up that calls nextElement() too.

return i_stream; //return a reference to the current ValueStream
}

```

## 3.2. Class SerializedValue

To simplify the serialization process NMM uses the base class SerializedValue which is much more easy than writing the operators. The main idea is to register the attributes by a base class and this class is responsible for the serialization and deserialization of our attributes. To register different types, like int's, string's, etc, we need a common base class for all attributes which is the generic class TValue.

Important note: If you use the SerializedValue base class your class must offer a valid default constructor, copy constructor and assignment operator!!

The new declaration of class MyStruct using the SerializedValue as base class is now.

```

namespace MyNamespace {
class MyStruct : public NMM::SerializedValue {
public:
    MyStruct ();
    MyStruct (const MyStruct&);

    string&      getName ()           {return __name.getRValue ();}
    const string& getName ()           const {return __name.getRValue () ;}

    const list<nmm_float>& getValueList () const {return __value_list.getRValue ();}
    list<nmm_float>& getValueList ()         {return __value_list .getRValue () ;}
    nmm_int32      getCounter ()         const {return __counter.getRValue () ;}
    void           setCounter (nmm_int32 i)   {__counter = i;}
}

```

```

        //virtual CopyConstructor
        MyStruct*    copy()          const { return new MyStruct(*this); }
        MyStruct&   operator=(const MyStruct&); //In this example you can use the C++ gene

        // stores the type information of the new class in a static member
        static const NMM::TypeInfo MyStruct_type_info

private:
    void registerValues();

private: //The attributes
    NMM::TValue<string>          __name;
    NMM::TValue<nmm_int32>       __counter;
    NMM::TValue<list<nmm_float> > __value_list;
};
}

```

The difference between the old and new declaration of MyStruct are the TValue-Wrapper for the attributes. It is not necessary to implement the assignment operator in this example because C++ does the job for us. Additionally we introduce the static member MyStruct\_type\_info which stores the type information of the new class. The new declaration inherits from SerializedValue and encapsulates all attributes in TValue 's. All classes must offer a virtual copy-constructor which is implemented using the standard copy constructor. The new private method registerValues() is responsible for the registration of the attributes by the base class and must be called in all constructors. The implementation of this method is very simple.

```

void MyStruct::registerValues() {
    insertArgument(&__name);          //register __name
    insertArgument(&__counter);       //register __counter
    insertArgument(&__value_list);    //register __value_list
}

```

This new method must be called in all constructors.

```

MyStruct::MyStruct()
    //forward the type information to the base class
    : NMM::SerializedValue(MyStruct_type_info)
{
    registerValues();
    //...
}

MyStruct::MyStruct(const MyStruct& my_struct )
    : NMM::SerializedValue(my_struct)
{
    registerValues();
    *this = my_struct; //use C++ generated assignment operator.
}

```

```

}

//Initialize the static member
const MyStruct::MyStruct_type_info("MyStruct", "MyNamespace");

```

### 3.2.1. Inheritance

This section describes the use of the class `SerializedValue` and inheritance. In this case we must tell the base class `SerializedValue` our new type information and the new attributes.

```

namespace NMM {
class DerivedStruct : public MyStruct {
public:
    DerivedStruct();
    DerivedStruct(const DerivedStruct& ds)

    DerivedStruct*   copy()           const { return new DerivedStruct(*this);}

    nmm_uintu getID()                 const { return __id; }
    void setID(nmm_uint32 id)         { __id = id; }

    static const NMM::TypeInfo DerivedStruct_type_info // stores the type information of th
private:
    void registerValues();

    TValue<nmm_uint32> __id;
}

```

First we implement our `registerValues()`-method which adds the internal id

```

void DerivedStruct::registerValues() {
    insertArgument(&__id);           //register __id
}

```

In the constructor we must set the type information in the base class `SerializedValue`, but the constructor of class `MyStruct` accepts no type information. But we can use the protected member method `setTypeInfo`. The following code demonstrates one possible implementation of the constructor.

```

DerivedStruct::DerivedStruct()
{
    /* The class MyStruct has no Constructor
    * that accepts an TypeInfo-struct. So we use
    * the protected method setTypeInfo.
    */
    setTypeInfo(DerivedStruct_type_info);
}

```

```

    /* register the attributes */
    registerValues();
    //...
}

DerivedStruct::DerivedStruct(const DerivedStruct& ds)
    : MyStruct(ds)
{
    /* The class MyStruct has no Constructor
     * that accepts an TypeInfo-struct. So we use
     * the protected method setTypeInfo.
     */
    setTypeInfo(DerivedStruct_type_info);

    /* register the attributes */
    registerValues();
    *this = ds;
}

//Initialize the static member
const DerivedStruct::DerivedStruct_type_info("DerivedStruct", "MyNamespace");

```

That's it. As you can see the usage of the base class SerializedValue simplifies the serialization process. You should use this class instead of writing the operators, if possible!

## 4. ValueFactories and Dynamic Object Allocation

In the examples above, we hide an important problem of object serialization and especially deserialization; the use of pointers. Pointers are normally used to store the address of a dynamically allocated and/or polymorphic object. Instead of a reference a pointer in C++ can change the object pointing to or point to no object, a so called null pointer. The polymorphy of C++ objects is another problem that must be handled by the deserialization process. A pointer can represent only a specific subset of the object's type but to instantiate the object we must know the concrete type. And before we deserialize an object we must create an instance of this object.

So this chapter describes how to solve these problems. Section 4.1 describes and demonstrates how to serialize a pointer to an object. Section 4.2 describes how to deserialize such an object using a so called ValueFactory which can instantiate objects from a given TypeInfo-struct.

### 4.1. Serialization of a Pointer

The serialization process of pointer is simple. If the pointer stores the address to an object, we dereference the pointer and serialize the object. But if the pointer does not store the address of an object, we need

an extension to handle so called NULL-pointers. Therefore we reserve a specific type with typename "NULL" to serialize a NULL-Pointer.

If we register attributes by our base class SerializedValue, it expects that the address of registered attributes never change. But pointers can change the address pointing too. So we need an update mechanism to inform the base class that some addresses changes. To simplify the use of pointers with our base class SerializedValue, NMM offers a TValue template spezialisation for pointers. This spezialisation is similar to a smart pointer that is a wrapper class for pointer and allows the same use of pointers like a normal type. This TValue-Wrapper is allocated as an object and stores the pointer. This object is registered by our base class and serialize a stored object or writes the NULL-TypeInfo if the pointer is NULL. Important Note! This TValue-specialization can only handle types which inherits the base class Value (SerializedValue inherits Value too). You can not write something like TValue<int\*>. You must use a pointer to an TValue<nmm\_int32\*> and write TValue<TValue<nmm\_int32\*>> my\_int\_pointer.

To show the use of pointers we extend our example with a new class which inherits MyStruct and stores an int pointer and a MyStruct pointer.

```
namespace MyNamespace {
class MyNewStruct : public MyStruct {
public:
    MyNewStruct();
    MyNewStruct(const MyNewStruct&);

    nmm_int32*  getIntPointer()  { if( __int_pointer.getValue() )
                                return __in_pointer.getValue().getPValue();
                                return 0;
                                }
    const nmm_int32*  getIntPointer() const { if( __int_pointer.getValue() )
                                                return __in_pointer.getValue().getPValue();
                                                return 0;
                                                }

    MyStruct*      getMyStructPointer()      {return __my_struct_pointer.getValue();}
    const MyStruct*  getMyStructPointer() const {return __my_struct_pointer.getValue();}

    //virtual CopyConsturctor
    MyNewStruct*  copy()      const  { return new MyNewStruct(*this); }

    static const NMM::TypeInfo MyNewStruct_type_info;

private:
    void registerValues();

private: //The attributes
    // pointer to an TValue<nmm_int32> (which does not inherits Value)
    NMM::TValue< TValue<nmm_int32> *> __int_pointer;
    // pointer to the MyStruct object (which inherits Value)
    NMM::TValue<MyStruct*>      __my_struct_pointer;
};
}
```

Now lets show how to implement the Constructor, CopyConstructor and assignment operator.

```
const NMM::TypeInfo MyNewStruct::MyNewStruct_type_info("MyNewStruct", "MyNamespace");

MyNewStruct::MyNewStruct() {
    setTypeInfo(MyNewStruct_type_info); //set the new real typename
    registerValues(); // register all attributes
}

MyNewStruct::MyNewStruct(const MyNewStruct& my_struct )
    : MyStruct(my_struct)
{
    registerValues();
    *this = my_struct; //use C++ generated assignment operator.
}

void MyNewStruct::registerValues() {
    insertArgument(&__int_pointer); //register __int_pointer
    insertArgument(&__my_struct_pointer); //register __my_struct_pointer
}
```

As you can see nothing changes. If you use pointer as attributes you should implement an assignment operator for a correct memory management of the pointer.

## 4.2. Deserialization of a Pointer

No lets focus the problem of deserialization of pointers. In our example in section 3 all attributes are allocated if an instance of MyStruct is created. If we use pointers these attributes must be created dynamically before the attribute can be deserialize and we must allocate the correct type. Our IValueStream stores the correct type information but we need an mechanism to allocate an corresponding object. Therefore NMM uses a ValueFactory that can allocate an object from a given TypeInfo-struct. All these things are done automatically by the base class SerializedValue and the TValue-class. The developer must only instantiate a so called ValueConstructor for the new class. This ValueConstructor registers them self with given type information by the ValueFactory and must be instantiated as a global attribute in the .cpp file. There are two different ValueConstructr's. The TValueConstructor must be used if you NOT use the class SerializedValue and the TConstructor if you use it.

```
/* If you implement the operators you must write */
TValueConstructor<MyStruct> my_struct_factory_object(MyStruct::MyStruct_type_info);

/* If you use the SerializedValue base class, you must write */
TConstructor<MyStruct> my_struct_factory_object(MyStruct::MyStruct_type_info);
```