Streaming with C++ is brilliant

Christian Koberg

21 Feb 2018 CPOL

Rate this: ★★★★★ 4.96 (44 votes)

The basics, snares, pitfalls and the bright side of C++ streaming standard.

Download Concepts.zip - 38.2 KB

Download Exercises.zip - 154.5 KB

Download Finals.zip - 104.9 KB

Download Dokuments.zip - 147.6 KB

We have to take some hurdles, but the result is fantastic.

First we see some basics. Then I'll show the snares and pitfalls that must be circumvented. But then we do the decisive step to the bright side of C++ streaming standard, that nothing compares to it.

This Tutorial deals with the following topics:

Catchwords

- Streaming C++ STL
- Memory Stream
- Binary Stream
- cout & Managed Code, Managed Objects
- StringStream
- Template
- Mixing Native and Managed Types

Motivation

I started this work, because on one hand I had a datastorage in native C++ and on the other a UI in .NET. How could the Objects easy interoperate, and ideally in a general way?

Having weighed the pros and cons, I chose the binary representation in a Memory-Stream.

The objective is the exchange of object-data between applications on the same system. No handling of different codepages or MSB/LSB question.

To exemplary exchange Name, Account, Count either-way. Like (in pseudo-code):

Hide Copy Code

```
Stream << System::String
      << System::Double
      << System::Int32.
Stream >> std::string
      >> std::double
      >> std::int.
```

just quick and safe and duplex.

An early decision I made: Because `System::String` always is WideChar, I choose `std::wstring` on the other side. To convert between std::string or `std::wstring` is not part of my Binary-Stream.

Synonyms

Just a word on the many synonyms that could be confused sometimes.

Native (high quality) C++ Standard <-> Managed (MS) C++ Code; namespace System <-> namespace std; ref class <-> (native) class; String <-> string; CLR <-> STL; .NET ATL MFC <-> STL;

even "safe" <-> "unsafe" code (an idiotic MS promotion jingle).

Mostly I will talk about Ref-Class and Native-Class or CLR vs. STL. I don't use ATL or MFC here.

Table of content

Downloads

You can download Dokuments.zip, with a detailed description of each library and executable plus a tutorial in .html files.
The sources are structured in Finals, Exercises and Concepts.

Binary Stream

Hide Shrink Copy Code

```
//t2_bytes1.cpp

#define WIN32_LEAN_AND_MEAN

#include <iostream>
#include <string>

using namespace std;

class Bytes
{
    unsigned char *pusc;
    bool initialized;
    unsigned char *ppos;
public:
    void init(int size);
    void free();
    void startreading();
    Bytes();
    ~Bytes();
    // ++++++ shift operators for writing '<<' and reading '>>' ++++++
    Bytes& operator<<(int &val);
    Bytes& operator>>(int &val);
    Bytes& operator<<(double &val);
    Bytes& operator>>(double &val);
    Bytes& operator<<(wstring &val);
    Bytes& operator>>(wstring &val);
};

void main()
{
    cout << "++++++ store data to Bytes-Stream ++++++\n";
    //                ..create BinStream
    Bytes binstream;
    int bsize;
    //                ..data to be stored
    int a = 123;
    //                ..calculate the Size of BinStream you need
    bsize = sizeof(a);
    binstream.init(bsize);
    //                ..shift data out
    binstream << a;
    //                ..BinStream holds the data now
    //                ..decide the receiver of the data
    int b;
    //                ..read data into receiver
    binstream.startreading();
    binstream >> b;
    //                ..free memory, it's useless further on
    binstream.free();
    cout << "int b=" << b << endl;
    //                ..next data
    double d = 456.789;
    bsize = sizeof(d);
    binstream.init(bsize);
    binstream << d;
    //                ..read data into new object
    double e;
    binstream.startreading();
    binstream >> e;
    binstream.free();
    cout << "double d=" << d << endl;
    //                ..now the W-string
    wstring ws(L"a String of Wide-Chars");
    //                ..size of variable type =
    //                ..    size of counter + (size of element * counter)
    bsize = sizeof(int) + (sizeof(wchar_t) * (int)(ws.length()));
    binstream.init(bsize);
    binstream << ws;

    wstring xs;
    binstream.startreading();
    binstream >> xs;
    binstream.free();
    wcout << "wstring xs=\"" << xs << "\"" << endl;
}
// ++++++ Bytes Implementation ++++++
void Bytes::init(int size)
{
    pusc = new unsigned char[size];
    ppos = pusc;
    initialized = true;
}
void Bytes::free()
{
    delete [] pusc;
    pusc = NULL;
    ppos = pusc;
    initialized = false;
}
void Bytes::startreading()
{
    ppos = pusc;
}
Bytes::Bytes()
    :pusc(NULL), initialized(false), ppos(pusc)
```

```
{}
Bytes::~Bytes()
{
    //        ..release the resource if not freed yet
    if (initialized) free();
}
Bytes& Bytes::operator<<(int &val)
{
    int *pdest = (int*)ppos;
    *pdest++ = val;
    ppos = (unsigned char*)pdest;
    return *this;
}
Bytes& Bytes::operator>>(int &val)
{
    int *psrc = (int*)ppos;
    val = *psrc++;
    ppos = (unsigned char*)psrc;
    return *this;
}
Bytes& Bytes::operator<<(double &val)
{
    double *pdest = (double*)ppos;
    *pdest++ = val;
    ppos = (unsigned char*)pdest;
    return *this;
}
Bytes& Bytes::operator>>(double &val)
{
    double *psrc = (double*)ppos;
    val = *psrc++;
    ppos = (unsigned char*)psrc;
    return *this;
}
Bytes& Bytes::operator<<(wstring &val)
{
    int wsize = (int)val.length();
    //            ..first store the size
    *this << wsize;
    wchar_t *pdest = (wchar_t*)ppos;
    const wchar_t *psrc = val.c_str();
    while (*psrc != L'\0')
        *pdest++ = *psrc++;
    ppos = (unsigned char*)pdest;
    return *this;
}
Bytes& Bytes::operator>>(wstring &val)
{
    int wsize;
    //            ..first read the size
    *this >> wsize;
    val = wstring((wchar_t*)ppos, wsize);
    ppos += wsize * sizeof(wchar_t);
    return *this;
}
```

Example: T3_Object1

Preface: Generally we have two kinds of fundamental data: Fixed-Length, Variable-Length.

Fixed-Length I store unpacked with their size, Variable-Length I store with its count of elements at the beginning and the elements afterwards, no delimiters.

If we have it done for C++ Standard-Class we go to the "save" "managed" "MS" Ref-Class.

Oops-a-daisy

Now an obstacle, I anticipated. because `System::String` is not implemented bidirectional. There are many ways to build `System::String` from string, wstring, C-string, but reverse? Why no `System::String.c_str()` conversion; stupid!

Finally I found P`trToStringChars()` in `<vcclr.h>` and `pin_ptr<>`, `interior_ptr<>` in `namespace cli`. After processing big-data from the Internet with the eye and brain.

It's implemented in *StringConvert.cpp* and we use it here now.

We have changes in the syntax by declaring an Object-Reference with '%' instead of '&'.

If a managed object resides on the Garbage-Collected-Heap the type is suffixed with '^' and members are referenced by pointer '->' systax.

Datatypes `System::Int32` and `System::Double` have a direct compatibility to the corresponding standard int, double.

`System::String` can be converted as described above.

Example T4_Bytes2

A great hurdle

Basic System::-Types work with stream, but with a Ref-Class-Object we get the friends-problem.

error C3809: 'RSimple': a managed type cannot have any friend functions/classes/interfaces

"Is this the end _ _ _", taa taa tataa?

The "do nothing" operation

Don't worry, there are still other possibilities to store the data to the stream and keeping the Members private. Let's see.

A popular one is a void method that takes a reference of the stream as parameter and puts the members onto the stream. Quite good, but it's a rupture of the streaming syntax of continued shift operations.

I think about a method, which takes a stream-reference to put the members on it and returns the reference as return-value too.

The method might look like:

Hide Copy Code

```
stream& Class::Out( stream& )
stream& Class::In( stream& ).
```

and the call would be like:

Hide Copy Code

```
stream << Class::Out(stream)
stream >> Class::In(stream).
```

But be aware, that it comes to a new syntactical situation, which has no common behaviour and defaults sometimes to what you wouldn’t expect.

If the method returns with stream & it leads to the constellation of

Hide Copy Code

```
stream << stream
```

which means in global function declaration

Hide Copy Code

```
stream& operator<<( stream&, stream& )
```

or as method of our own stream-class

Hide Copy Code

```
stream& stream::operator<<( stream& )
```

And for operator>> likewise.

We must say, what the compiler has to do now. The intuitive behaviour is appended the right to the left, or extract from left to right. But in our case we get back the identic stream, we gave to the method before. There is nothing left to append. All happened inside the method. It's important to understand this short demonstration:

Hide Shrink Copy Code

```
//demonstrateStr.cpp

#define WIN32_LEAN_AND_MEAN

#include <string>
#include <iostream>
#include <sstream>
using namespace std;

//        ..the compiler must see this declaration before it implements any
//        ..expression with this stream. Because otherways he takes his
//        ..default behavior you would'nt expect.
//        ..Comment the following declaration out, to see the effect(/defect).
stringstream& operator<<(stringstream &left, stringstream &right);

class Mclass
{
    string member;
public:
    Mclass():member("Mclass::string"){}
    stringstream& Out(stringstream& str)
    {
        str << member;
        return str;
    }
};
stringstream& operator<<(stringstream &str, Mclass &obj)
{
    str << obj.Out(str);
    return str;
}
void main()
{
    Mclass mc;
    stringstream as, bs;
    as << "+++++";
    bs << "''''''";

    bs << mc;
    cout << bs.str() << endl;
    bs << as;
    cout << bs.str() << endl;
    cout << ".. as" << as.str() << endl;
/*    * Likewise to streaming out '<<' the same principals apply to
    * streaming in '>>'! But they are'nt that easy to visualize
    * as for streaming out.
*/
}
stringstream& operator<<(stringstream &left, stringstream &right)
{
    /*
        if you don't use left- << or right >> schift of
        different streams of the same type in your application,
        just write:
        return left;
    */
    if (&left == &right)
    {
        cout << " + ident +\n";
        return left;
    }
    else
    {
        cout << " + different +\n";
        left << right.str();
    }
    return left;
}
```

As we in our stream control all operations, we don't allow shifting between two objects and always return the left object.

"We are unstoppable!" :-D Lets implement that pattern:

Hide Shrink ▲ Copy Code

```
stream& MyClass::Out( stream &str )
{
      str << this->member ...;
      return str;
}
stream& MyClass::In( stream &str )
{
      str >> this->member ...;
      return str;
}
stream& stream::operator<<( stream &right )
{
      return *this;
}
stream& stream::operator>>( stream &right )
{
      return *this;
}
stream& operator<<( stream &str, MyClass &obj )
{
      str << obj.Out(str);
      return str;
}
stream& operator>>( stream &str, MyClass &obj )
{
      str >> obj.In(str);
      return str;
}
```

Now the approach of the former example works with "save-" "managed-" "MS-" code too. Yahooo!

The absolute advantage of this pattern is, to shift objects private members in a class-method without having the stream as a friend. It works with any stream :-)

Are we ready now? Rather we mastered the first step to a coming Safari (through oasis and desert), but we can build on that. But now we get the efforts of the level.

What I want to point out!

Once more about the "do nothing" operation. Because managed Ref-Classes reject the "friend" declaration, we have to use an alternate pattern of streaming Class-Members, as shown above.

I want to lead you to the point now. With that little example T4_Bytes2 we are in the good situation, to get a clear view of the conditions.

We have implemented the complete interface of a stream by ourself. And(!!) we introduced full streaming-behaviour to Ref-Classes by omitting the common "friend" syntax and implementing an alternative. And here we are free of some sideeffects of a precarious implementation of STL-streams (at least in VC2005), I will show you later!

Now we can go to the declaration of our own stream in Bytes.h. Here are the two lines of declaring the "do nothing".

If you comment them out

Hide Copy Code

```
//BytesR& operator<<(BytesR &right); //do nothing
//BytesR& operator>>(BytesR &right); //do nothing
```

and compile the file t4_bytes2.cpp now, the compiler shows you the two statements

Hide Copy Code

```
str << obj->Out(str);
str >> obj->In(str);
```

where this functionality is requested.

More about this find her in the chapter "Effects or Defects?"

A comfotable interface for classes

On the previous page, we explored the technical principals of storing and retrieving a binary datagram of an object's data to and from a Memory-Stream. And how it can be implemented for Common-C++ classes, and MS-Managed-C++ Ref-Classes as well.

Here now we look at the refinement to a usable utility.

The intention is, to get a package of an object and to put it, like on the network, on the inner-application control flow. The stream should be a message of an object, not a storage, as streams usually are seen.

Like on the network, a message that is sent, but not consumed, is worthless. This perspective will impact on some implementation details later.

Usability

Now we must put some comfort to the usability. As you imagine, the chosen solution works very strict and some mistakes could happen. First of all, we deal with rough memory and have to check the boundaries. Just as important is the usability. E.g. wrong size calculated, the types calculated, written or read differ, you see. Especially with binary data, the user needs quick and strong notice of what could have failed.

Therefore: First the size of the binary representation is calculated and fixed. The stored data must exactly meet this size. To store more is forbidden and less is an error too.

It's impossible to retrieve data from a stream, that's not proper built.

Not consuming all the stored data is an error.

And inhibit typos like this:

Hide Copy Code

```
Stream >> Name
      << Account
      >> Count;
```

Syntactically correct but rare intended. I decide it to be an error.

Requests

As previously shown, there are some shortcomings so far, which raise the following requests.

The read/write access is controlled with two different Constructors and during the read/write operations.

If the data is received in a wrong sequence, bricked results are produced and must be controlled optically in the source.

To keep the footprint of the Binary-Stream as lean as possible the implementation gets two parts:

One that holds the data to be dragged around, and another is a helper, that does the work of write, read and control. And there is an Interface to the user defined (UD-)classes that can be inherited. It defines the at least required methods of the class and provides services to it.

Due to the syntactical differences of native- and managed code, there are two helpers and two interfaces.

The "Native-" components are kept pure native and compile in such projects without /clr-switch.

Designing the interface

Now we look at the interfaces here:
The principals i discuss for Standatd-C++. /CLR is a little bit different, we will look at it later.
What are our elements?

1. we have a message-envelope to retrieve the data, to drag it around and to deliver it.

Hide Copy Code

```
class Bytes;
```

2. the helper that fills and empties the envelope.

Hide Copy Code

```
class BinaryN;
```

3. an interface to the UD-class.

Hide Copy Code

```
class INbinary;
```

4. the user's class, inheriting the interface.

Hide Copy Code

```
class NEntity : public INbinary;
```

Supplied syntax

The statements, I want to supply (in pseudo-code):

1. produce the data-message

Hide Copy Code

```
Bytes << NEntity; //shift out
```

2. deliver the data-message

Hide Copy Code

```
Bytes >> NEntity; //shift in
```

3. produce the data-message by function

Hide Copy Code

```
Bytes foo(); //function produces a Bytes-object as returnvalue
```

4. dragging around the data-message

Hide Copy Code

```
Bytes = foo(); //assigning a function's result
```

5. deliver the data-message by function

Hide Copy Code

```
void foo( Bytes& ); //function takes Bytes-object as reference
```

Class layout

This leads to the following layout of the classes:
The binary stream Bytes

Hide Shrink ▲ Copy Code

```
class Bytes
{
    friend class BinaryN;
    friend class INbinary;
//          ..Pointer holds the resource
    unsigned char *pusc;
//          ..Working pointer to the actual position
    unsigned char *ppos;
//          ..If flagged, destructor must release the resource
    bool initialized;
//          ..Is set by the helper BinaryN's reading '>>' action,
//          ..to tell that the message is delivered and must be deleted.
    bool toDelete;
//          ..Class must know the size of resource
    int size;
//          ..Initialized by NEntity
    void init(int sz);
//          ..Release the resource
    void free();
//          ..Set class to initial state
    void nulling();
//          ..Move resource by assignment- or copy-operation
    void movesecure(Bytes &dest, Bytes &src);
//          ..Point of throwing errors
    void error(string e);
public:
    Bytes& operator=(Bytes&);
//          ..Accepts a NEntity that inherits from INbinary
    Bytes& operator<<(INbinary&);
    Bytes& operator>>(INbinary&);
//          ..Instantiates an empty envelope
    Bytes();
//          ..Const copy-constructor is requested by STL-containers
//          ..(e.g. queue<>) and used by local stack too.
//          ..The stack would be satisfied by assignment or
//          ..nonconst copy too, but STL container requires it definitely.
    Bytes(const Bytes&);
//          ..At the end of life :-))
    ~Bytes();
};
```

The helper of Bytes for storing and retrieving

Hide Shrink ▲ Copy Code

```
class BinaryN
{
//          ..Pointer to Bytes-object
    Bytes *theMem;
//          ..Pointer to the data-buffer
    unsigned char *mem;
//          ..Working pointer controls the end-position
    unsigned char *pend;
//          ..enum to set the state of allowed operations
//          ..e.g. write-constructor sets it to 'o_put' ,
//          ..read-constructor to 'o-get' , etc...
    enum Operation{o_put, o_get}op;
//          ..Default-constructor forbidden
    BinaryN();
//          ..Throws an error, if the actual state doesn’t allow the requested operation
    void allowed(Operation o);
//          ..Unique point of errors
    void error(string e);
public:
//          ..Write-constructor for put '<<' operations
    BinaryN(int sz, Bytes &abuf);
//          ..Read-constructor for get '>>' operations
    BinaryN(Bytes &abuf);
    ~BinaryN();
//          ..Method to free the resource
    void deleteMem();
//          ..Shift-in, shift-out for standard types
    BinaryN& operator<<(unsigned __int32 &val);
    BinaryN& operator>>(unsigned __int32 &val);
    BinaryN& operator<<(int &val);
    BinaryN& operator>>(int &val);
    BinaryN& operator<<(wstring &val);
    BinaryN& operator>>(wstring &val);
    BinaryN& operator<<(double &val);
    BinaryN& operator>>(double &val);
//          ..The "do nothing", as described in the previous chapter
    BinaryN& operator<<(BinaryN &right);
    BinaryN& operator>>(BinaryN &right);
//          ..Static methods to help NEntity to calculate its actual size
    static __int32 sizeBytes(const wstring &val);
    static __int32 sizeBytes(const unsigned __int32 &val);
    static __int32 sizeBytes(const int &val);
    static __int32 sizeBytes(const double &val);
};
```

Interface-class, a user-defined (UD-)class (e.g. NEntity) should inherit for easy interaction with Bytes-stream.

Hide Copy Code

```
class INbinary
{
    friend class Bytes;
//          ..Puts NEntity's data to Bytes-stream
    void storeBytes(Bytes&);
//          ..Gets NEntity's data from Bytes-stream
    void retrieveBytes(Bytes&);
protected:
//          ..A derived class can store its class-name here
//          ..for easier identification in Debugger
    string className;
public:
//          ..Methods, that must be defined by NEntiy
    virtual BinaryN& Out(BinaryN &str) = 0;
    virtual BinaryN& In(BinaryN &str) = 0;
    virtual const __int32 mySize() const = 0;
    virtual wostream& Out(wostream &str) = 0;
};
```

A UD-class inheriting the interface.

Hide Copy Code

```
class NEntity : public INbinary
{
//          ..Class members
    wstring ss;
    double sd;
public:
//          ..Class methods
    NEntity(wstring s = L"", double d = 0);
    void Text(wstring val);
//          ..Required interface methods
    //+++++++ INbinary Interface ++++++
//          ..Calculate the actual size of class-members
    const __int32 mySize() const;
//          ..Put the members to stream
```

```
    BinaryN& Out(BinaryN &str);
//          ..Get the members from stream
    BinaryN& In(BinaryN &str);
//          ..Show class' content
    wostream& Out(wostream &str);
};
//          ..Declaration of left-shift '<<' operator 'wostream << NEntity;'
wostream& operator<<(wostream &str, NEntity &obj);
```

Special lines of code

In the Bytes-sources you find some special lines of code, I want to explain here, if they are new to you:

Hide Copy Code

```
#define DEBUG_Bytes_Class
#undef DEBUG_Bytes_Class

#ifdef DEBUG_Bytes_Class
      static int count;
public:
      int name;
private:
#endif

#ifdef DEBUG_Bytes_Class
      name = count++;
      cerr << "...\t empty\t_" << name << endl;
#endif
```

At the start of Bytes' header-file there is a #define, defining a macro-name, immediately followed by an #undef of the same macro-name which invalidates the previous #define. Does that make sense?

It is intended as a switch, helping to debug the sequence of important methods.

If the #undef line is commented out //#undef ... the previous definition stays active. It remains active until the end of the actual compilation, or until another #undef of that macro-name occurs. If the compiler meets an now #ifdef with that macro-name, it is true and additional lines of code are added to the class' declaration and to the definition.

In my case, it's a static counter and an int called 'name'. The counter is incremented by each construction of a new object and gives each instance a unique name. When an instance now executes methods, it always tells its name and the method it's executing. So i had on the Console-output a protocol: who is doing what, when. I like this method to analyse complex situations.

Details of implementation

To create a writer, Bytes-object must be empty.

To create a reader, Bytes-object must hold data exact of the calculated size.

The assignment- and the copy-operation hands over the data-buffer to the left side and the right-side-pointer is NULLed. Therefore the const right-hand object of Copy-constructor must be modified. To do this, the constness of the source-object is casted away by calling the method movesecure(...):

Hide Copy Code

```
movesecure(*this, const_cast<Bytes&>(src) );
```

Differences of CLI implementation

CLI introduces its new type-syntax.

A declared managed class ref class CAny; can be defined in two ways; CAny^ A and CAny B.

A reference to a managed class is declared with '%' instead of '&'.

This results in differences in the declarations, but the implementations are very similar.

Likewise to the standard-class interface, we have a managed-class interface

Hide Copy Code

```
ref class IRbinary;
```

and a UD-managed-class which inherits that interface

Hide Copy Code

```
ref class REntity : public IRbinary;
```

But in class Bytes there is no interface to Ref-Class `IRbinary`, although it could be declared like this:

Hide Copy Code

```
friend ref class IRbinary;
Bytes& operator<<(IRbinary^%);
Bytes& operator>>(IRbinary^%);
Bytes& operator<<(IRbinary%);
Bytes& operator>>(IRbinary%);
```

The reason is the request of full compatibility to C++ standard (native) code, to compile without CLR, which would be claimed here.

But the shift operators "<<, >>" can be defined globally as well. So i declared the namespace `Bytes_Ref` and put them there.

Example BytesInterface shows that implementation.

The plan is realized now. We have a stream, that takes and delivers a binary datagram of an object, that works for standard class and managed class as well.

Are we ready now? Maybe, but we are appetized for more !;-)

Due to the construction, the Binary-Stream is limited to one object. May be an application likes to ship more than one object at once.

Streaming all around

On the previous pages we have done the basics so far. Now we start with streaming all around.

The backbone Memory-Stream will be STL-StringStream and we will use Filestream for persistence.

Supplied syntax

The goal is, to put on or get from a stream a sequence of objects in binary representation.

Syntactically (in pseudo-code):

Hide Copy Code

```
stream << CAny << ...
stream >> CAny >> ...
```

Implementation

Because we've done much work previously, we can bring in the harvest now :-)

For user-defined (UD) classes like CAny we have a uniform interface to our binary representation and add this task there.

As shown in chapter 1, we need three elements for each to do it.

Hide Copy Code

```
stream& operator<<( stream &left, stream &right );
stream& CAny::Out( stream& );
stream& operator<<( stream &str, CAny &obj );

stream& operator>>( stream &left, stream &right );
stream& CAny::In( stream& );
stream& operator>>( stream &str, CAny &obj );
```

The first belongs globally to the App, the other both to the UD-class `CAnny` that inherits from interface-class `INbinary`.

It's implemented in `StreamDoNothing`, `INbinary`, `IRbinary`.

The syntax to put on filestream is `fs << str.str();` , to get from str << fs.rdbuf();.

Example ByteStream.

Conclusion

Now we have come really very far. Let's see what we've got.

We stream to / from a memory-stream (STL-stringstream here):

Hide Copy Code

```
str << ob1 << ob2;
str >> ob1 >> ob2;
```

We stream the members of a complex class to / from the memory-stream (Bytes).

Hide Copy Code

```
str << name << account << Simple::Out(str) << date.Out(str);
str >> name >> account >> Simple::In(str) >> date.In(str);
```

We write to file `fs << str.str();`.

We read from file `str << fs.rdbuf();`.

Streaming all around. Isn’t that fine !:-D

From a basic type to a complex object, to a sequence of objects, to a Memory-Stream, to persistence, to any STL-stream, for Standard-C++ and MS-Managed-C++, and duplex in both directions!

Why am I still a little dissatisfied? First of all I struggled with the STL-stream implementation, and additionally the usability needs some refactoring.

Alternate pattern

Streaming of managed objects with STL-stream.

Intention

Here I want to show the principals of streaming managed objects by STL-Stream in detail, because it's not without pittfalls.

Ostream for managed types

The standard pattern of streaming objects, by having the stream as a friend of the class, does not work for managed Ref-Class. This leeds to the necessity of an alternative, which is used allready in former examples and is explained in detail here.

Hide Shrink Copy Code

```
//t1_pattern.cpp
/*
    this Example shows an alternate Pattern of Cooperation of
    Object and Stream,
    demonstrated with wostream.
*/
#define WIN32_LEAN_AND_MEAN        // Exclude rarely-used stuff from Windows headers

#include <iostream>

using namespace std;
using namespace System;

//++ this declaration of "do nothing" must be seen by the compiler first,
//++ before it implements any expression with that stream.
//++ otherwise it could implement a default-behavior, you would'nt like.
wostream& operator<<(wostream &left, wostream &right); //do nothing

class Native
{
    double member;
public:
    Native(double d = 0): member(d)
    {}
    //                ..this ist the common Pattern, how Object ant Stream cooperate
    //                ..having the Stream as a Friend of Class and
    //                ..Stream has Access to the private Members
    friend wostream& operator<<(wostream &str, Native &obj);
};
wostream& operator<<(wostream &str, Native &obj)
{
    str << "Native::member=" << obj.member;
    return str;
}
ref class Managed
{
    Double member;
public:
    Managed(Double d): member(d)
    {}
    wostream& Out(wostream &str)
    {
        str << "Managed::member=" << member;
        return str;
    }
};
//                      ..the common Pattern does not work with /CLI Classes
//                      ..because Managed-Classes don't know 'friend"-Declaration.
//                      ..But there is an Alternative to have a Class-Method Out(Stream&)
//                      ..that accesses the private Data and to tell
//                      ..the Stream to call this Method.
//                      ..the Crux is, to implement the 'Do-Nothing'-Function (as I call it).
wostream& operator<<(wostream &str, Managed %obj)
{
    str << obj.Out(str);
    return str;
}
wostream& operator<<(wostream &str, Managed^ %obj)
{
    str << obj->Out(str);
    return str;
}
wostream& operator<<(wostream &left, wostream &right) //do nothing
{
    return left;
}

void main()
```

```
{
    Native n(123.456);
    wcout << n << endl;

    //            ..if you haven't seen Streaming with Managed Objects yet,
    //            ..here it is!
    Managed^ m = gcnew Managed(789.123);
    wcout
        << m << endl
        << *m << endl;
}
```

It is rich commented and shows the difference of common and alternate pattern with an exemplary implementation.

Alternate pattern with nested classes

This example is to show all the elements of alternate pattern in declaration, definition and usage clear and separated for nested classes.

Example Ostream

Alternate pattern for output and input

Here the alternate pattern is implemented for nested classes for output and input with std::stringstream.

Example Stringstream

For comparison purposes is here an implementation with the standard pattern.

Example TraditionalStream

Syntax pitfall

Now we see one of the traps of STL-stream implementation, one could step in.

And if you look at the example, you see that it's going wrong without any hint by the compiler. It just implements a default behaviour, that's not very useful here.

That's the reason too, why the previous two examples explained the proper syntax that verbose, because you must control it optically, manually in your source, without any help by compiler or linker.

Hide Shrink ▲ Copy Code

```
//demonstrateStr.cpp

#define WIN32_LEAN_AND_MEAN

#include <string>
#include <iostream>
#include <sstream>
using namespace std;

//        ..the compiler must see this declaration before it implements any
//        ..expression with this stream. Because otherways he takes his
//        ..default behavior you would'nt expect.
//        ..Comment the following declaration out, to see the effect(/defect).
stringstream& operator<<(stringstream &left, stringstream &right);

class Mclass
{
    string member;
public:
    Mclass():member("Mclass::string"){}
    stringstream& Out(stringstream& str)
    {
        str << member;
        return str;
    }
};
stringstream& operator<<(stringstream &str, Mclass &obj)
{
    str << obj.Out(str);
    return str;
}
void main()
{
    Mclass mc;
    stringstream as, bs;
    as << "+++++";
    bs << "'''''";

    bs << mc;
    cout << bs.str() << endl;
    bs << as;
    cout << bs.str() << endl;
    cout << ".. as" << as.str() << endl;
/*    * Likewise to streaming out '<<' the same principals apply to
    * streaming in '>>'! But they are'nt that easy to visualize
    * as for streaming out.
*/
}
stringstream& operator<<(stringstream &left, stringstream &right)
{
    /*
        if you don't use left- << or right >> schift of
        different streams of the same type in your application,
        just write:
        return left;
    */
    if (&left == &right)
    {
        cout << " + ident +\n";
        return left;
    }
    else
    {
        cout << " + different +\n";
        left << right.str();
    }
    return left;
}
```

In the following, we see more of them

Effects or defects of STL-Stream implementation

As we have already seen in previous Chapters, the alternate pattern of cooperation of stream and object consists of three elements (in pseudo-code):

A) The "do-nothing" (as I call it)

declared as a method of stream-class

Hide Copy Code

```
stream& stream::operator<<( stream& );
stream& stream::operator>>( stream& );
```

or declared globally

Hide Copy Code

```
stream& operator<<( stream&, stream& );
stream& operator>>( stream&, stream& );
```

B) The class-methods

that access the class-members

Hide Copy Code

```
stream& AnyClass::Out( stream& );
stream& AnyClass::In( stream& );
```

C) The stream-to-class operator

that leads the stream to the class-method

declared globally

Hide Copy Code

```
stream& operator<<( stream&, AnyClass& );
stream& operator>>( stream&, AnyClass& );
```

or declared as stream-class method

Hide Copy Code

```
stream& stream::operator<<( AnyClass& );
stream& stream::operator>>( AnyClass& );
```

The global declaration works for every class, and has no impact to stream's interface at all.

A short, but educative tour you find in

Hide Shrink ▲ Copy Code

```
//demonstrateStr.cpp

#define WIN32_LEAN_AND_MEAN

#include <string>
#include <iostream>
#include <sstream>
using namespace std;

//        ..the compiler must see this declaration before it implements any
//        ..expression with this stream. Because otherways he takes his
//        ..default behavior you would'nt expect.
//        ..Comment the following declaration out, to see the effect(/defect).
stringstream& operator<<(stringstream &left, stringstream &right);

class Mclass
{
    string member;
public:
    Mclass():member("Mclass::string"){}
    stringstream& Out(stringstream& str)
    {
        str << member;
        return str;
    }
};
stringstream& operator<<(stringstream &str, Mclass &obj)
{
    str << obj.Out(str);
    return str;
}
void main()
{
    Mclass mc;
    stringstream as, bs;
    as << "+++++";
    bs << "''''''";

    bs << mc;
    cout << bs.str() << endl;
    bs << as;
    cout << bs.str() << endl;
    cout << ".. as" << as.str() << endl;
/*    * Likewise to streaming out '<<' the same principals apply to
    * streaming in '>>'! But they are'nt that easy to visualize
    * as for streaming out.
*/
}
stringstream& operator<<(stringstream &left, stringstream &right)
{
    /*
        if you don't use left- << or right >> schift of
        different streams of the same type in your application,
        just write:
        return left;
    */
    if (&left == &right)
    {
        cout << " + ident +\n";
        return left;
    }
    else
    {
        cout << " + different +\n";
        left << right.str();
    }
    return left;
}
```

More examples in the previous chapters.

A closer look at the streaming syntax

Besides the "do-nothing" another centrepiece of the alternate pattern is using methods in a stream-expression.

So let's examine it in detail in

Example LeftToRight

The streaming operators '<<' and '>>' have left-to-right associativity. They are said, but let's see the truth.

Some details of the construction of the example.

It has the following elements:

- a simple class (class Txt)
- three streams

Hide Copy Code

```
std::ostream
class CMyQstr
std::stringstream
```

- three signatures of function/method

Hide Copy Code

```
stream& Txt::Out( stream& ) /...::In(...)
stream& strout( stream&, Txt& ) /...strin(...)
string txtXout( Txt& ) /...txtin(...)
```

- a global `std::string` Gst it collects the out-streamed elements in the sequence they are evaluated
- three implementations, one per stream

Hide Copy Code

```
Ostream.cpp
QueueStream.cpp
StringStream.cpp
```

If you look at `main()` function, you see, that only tree functions are active, the others are all commented out.

It's because this example should be done step by step, following this guided tour.

Further on we will do some modifications of the source, to get a clear view on the effects.

If you execute the first 3 functions

Hide Copy Code

```
oput1(..) oput2(..) oput3(..)
```

you get the following output:

Hide Copy Code

```
********** Studies of '<<', '>>' syntax **********
      exploring the left to right associativity
            ********** Ostream **********
            ----- oput1 -----
stored:  1a2b3c
read:  1a2b3c
            ----- oput2 -----
stored:  c3b2a1
read:  c3b2a1
            ----- oput3 -----
stored:  cba123
read:  c3b2a1
```

stored: means, how the elements are put on stream, and

read: shows, how the expressions have been evaluated.

Although the elements three times are put on stream in the same order, they have been treated different each time.

`oput1()` put objects on stream and the elements have been evaluated and stored in left-to-right order.

`oput2()` put the objects on stream per function, like by the alternate pattern, and the objects have been evaluated and stored in right-to-left order

`oput3()` put the objects on stream by two different functions. The expressions have been evaluated in right-to-left order, but on stream there are two different orders. The values of the function with return-value stream& it took first and in the evaluated right-to-left order. The values of the function with return-value string it took second and in left-to-right order, as they are written in the statement.

Now lets do some modifications of the source, to see some effects of that behaviour.

A) We decorate the output of `oput1()` with some literal constants. The values should be outputted like ,1',a' and so on.

The function `ostream& operator<<(ostream &str, Txt &obj)` in source *Txt.cpp* has a commented line. Remove the comment of that line and put the comment to the one above.

On execution you get the output

Hide Copy Code

```
----- oput1 -----
stored:
1,'a,'2,'b,'3,'c,'
read: 1a2b3c
```

It's not what we wanted, but understandable, since we know, that it stores the return values of type stream& first, and the others afterwards.

One way to solve it, is to split the streaming-statement into

Hide Copy Code

```
str << ",";
str << obj.Out(str) << "'";
```

B) Next we look at the first output and input stream `CMyQstr`.

In `main()` we uncomment the 3 blocks of `MyQueueSream`.

Past execution you see a new output-line got:.

The lines read: and stored: show, that on writing it behaves like ostream, and line got: shows, that on read it restores the values in the correct order, despite of the written sequence.

C) Next we explore std::stringstream.

At first we uncomment the 1. block of STL-StringStream.

On execution the output shows, that it evaluated the expressions like the other both, and line stored: shows the values, each followed by the separator blank.

D) Next we modify the source of sput1(..). As first element of output, we add the output of "-" and write << "-" just before << z1.

Past execution we look at the output and wonder what had happened.

It evaluated like before, but on stream it stored the ostream implementation instead of the stringstream one.

The fact is: Putting the character "-" (or any other basic type) on stringstream corrupted the stringstream to iostream and then the compiler implemented ostream instead of stringstream function. Basic types are corrupting stringstream to iostream!

The workaround is to end the corrupted statement and start a new one like

Hide Copy Code

```
str
    << "-";
str << z1 ...
```

What would you say: Is it a bug or a feature (of VS2005, at least)?

But that's not the end of the "features" ;-)

E) We uncomment the next block.

Past execution we see, it behaves likewise CMyQstr.

F) From the 3. block, we uncomment the first two lines, the heading and sput3(..).

As result we see an extra sort of the expressions of the output statement.

It inspected the expressions of the statement and sorted them by return-value before executing them in right-to-left order. The resulting it sorted like `CMyQstr` or ostream.

G) We uncomment the rest of the last block now. And we go to *StringStream.cpp* to uncomment the implementation of sget3(..) too.

Now we get a compilation error C2679:. What's the matter?!

Hide Copy Code

```
str
      >> txtin(z1)
      >> strin(str, ta)
```

```
        >> txtin(z2)
        ...
```

The error-message refers to the second expression >>strin(..) and means, it can't continue the statement past txtin(..).

The fact is:

`Function txtin(..)` returns a `std::string` which corrupts stringstream to iostream. Therefore the compiler is looking for a function like

Hide Copy Code

```
stringstream& operator>>( istream&, stringstream& );
```

You can try it, if you declare that signature at the beginning of the source file. You'll see, the compiler is satisfied. But it's impossible to implement that operation well.

But you can implement a do-nothing of `std::istream` instead.

Hide Copy Code

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

Do it in *StreamDoNothing.h* and you'll see, it works.

But it's not a good solution, to feed the compiler with wrong stream implementations. The better workaround is, to remove the bad istream declaration again and to put the corrupting expressions into a separate statement and continue with a new statement afterwards. **Don't forget: You must do the output likewise!**

Hide Copy Code

```
str
        >> txtin(z1)
        >> txtin(z2)
        >> txtin(z3)
        ;
str
        >> strin(str, ta)
        >> strin(str, tb)
        >> strin(str, tc)
        ;
```

But there is still another major problem with STL-stringstream: It has no `clear()` or `truncate()` method, at least in VC2005. So it grows to infinite, if you use it repeatedly. Deadly for long-running applications. There are two workarounds, if you are dealing properly with internals of its implementation. There is a particularized discussion about in example `ByteStream` and `ByteStreamT`.

Now we have seen, how streams work with functions and methods.

And we have found workarounds of all that hurdles, brought about this hellish implementation of STL-streams. Preventable problems that let some despair.

More Examples

Some examples about using methods in streaming-expressions.

OstrFormatter, Manipulator

There is a better way

But the more beautiful part is coming now.

We turn the page on the problematic STL-streams and see how comfortable it is, to develop a stream on one's own.

- no stream-corruption
- no delimiters

- no problem with whitespace in strings
- no growth to infinite, no dealing with internals

Streams by ones own

The first stream by our own we did in example LeftToRight, in class CMyQstr(.h .cpp).

The best choice at all for a stream-like behaviour (first in - first out) is datastructure Queue, because it grows on write (push) and shrinks on read (pop) automatically.

Class `CMyQstr` is kept simple, just streaming `std::string` type, but its declaration shows all elements, a stream needs.

Hide Copy Code

```
class CMyQstr
{
	vector<string> qstring;
	vector<string>::iterator iter;
public:
	CMyQstr();

	CMyQstr& operator<<(string &v);
	CMyQstr& operator>>(string &v);

	//the Stream provides the "Do Nothing" Operator by itself.
	CMyQstr& operator<<(CMyQstr &right); //do nothing
	CMyQstr& operator>>(CMyQstr &right); //do nothing
};
```

And the declaration of a streamable class:

Hide Copy Code

```
class MyUDC
{
	string st;
public:
	MyUDC(char v[]);
	MyUDC(string &v = string(". "));
	//              ++ the interface to streams ++
	CMyQstr& Out(CMyQstr &str);
	CMyQstr& In(CMyQstr &str);
	ostream& Out(ostream &str);
};
ostream& operator<<(ostream &str, MyUDC &obj);
CMyQstr& operator<<(CMyQstr &str, MyUDC &obj);
CMyQstr& operator>>(CMyQstr &str, MyUDC &obj);
```

A stream for nested Objects

Example MyStreamX shows a stream for the three basic datatypes string, integral, float and how it works from basic type to nested classes.

Here you can see how pretty and handsome streaming is made by C++, if it's not destroyed by a hellish implementation:

Hide Shrink Copy Code

```
class COrder
{
	string header;
	CArticle article;
	CPerson person;
public:
	// ...

	CMyStream& Out(CMyStream &str);
	CMyStream& In(CMyStream &str);
};
CMyStream& operator<<(CMyStream &str, COrder &obj);
CMyStream& operator>>(CMyStream &str, COrder &obj);

CMyStream& COrder::Out(CMyStream &str)
{
	str
		<< header
		<< article
		<< person
		;
	return str;
}
CMyStream& COrder::In(CMyStream &str)
{
	str
		>> header
		>> article
		>> person
		;
	return str;
}
```

No stream corruption, no delimiter struggling.

Now we have seen the convenience of a tidy stream and make ours ready to use with likely templates.

Wrapper instead of inheritance

Streaming all around. That's fine, I think.

But to inherit the interface has a significant impact on class' design that dissatisfies me. Each class must inherit, to become binary streamable and if you derive from them, you easy get it twice in the new class. Additionally the importance of the streaming interface is just for a short time, right to the end or right to the beginning of an object's lifetime.

To circumvent these complications, a wrapper seems the better way. And it looks really smart ;)

The wrapper-class takes a reference to class `CAnyN` to `binSerial` and redirects each call of `INbinary`'s interface to `CAnyN`'s implementation.

Implementation for Native-Class

Hide Shrink Copy Code

```
typedef CAnyN TYPE;
class BinStreamN: public INbinary
{
	TYPE &binSerial;
	BinStreamN(); //no Default-Constructor
public:
	BinStreamN(TYPE &objRef)
		:binSerial(objRef)
	{
		className = "BytesWrapper";
	}
	BinaryN& Out(BinaryN &str)
	{
		return binSerial.Out(str);
	}
	BinaryN& In(BinaryN &str)
	{
		return binSerial.In(str);
	}
	const __int32 mySize() const
	{
```

```
            return binSerial.mySize();
      }
      wostream& Out(wostream &str)
      {
            str << "Cname=\"" << StringConvert::wstr(className) << "\" ";
            return binSerial.Out(str);
      }
};
wostream& operator<<(wostream &str, BinStreamN &obj)
{
      return obj.Out(str);
}
```

Declaration for Ref-Class

Hide Copy Code

```
ref class CAnyR;
ref class BinStreamR: public IRbinary
{
      CAnyR ^binSerial;
      BinStreamR(); //no default-constructor
public:
      BinStreamR(CAnyR^ %objRef);
      virtual BinaryR& Out(BinaryR &str) override;
      virtual BinaryR& In(BinaryR &str) override;
      virtual const __int32 mySize() override;
      virtual wostream& Out(wostream &str) override;
};
wostream& operator<<(wostream &str, BinStreamR^ %obj);
wostream& operator<<(wostream &str, BinStreamR %obj);
```

As you see, the implementation is the same for any class. Therefore it's a proper candidate to become a template. We will do it later on.

Let's go to see how to do with templates.

Template

When I started with templates, I found many in-depth articles, but I was still struggling with the syntax-basics.

So I'll not give a talk about the syntax, but show concrete implementations of the basics.

First of all the naming Function-Template or Template-Function.

Like class and object, where object is an instance of a class, Template-Function is a by compiler implemented Function-Template. Class-Template and Template-Class likewise.

At the beginning a function-template. example TemplateFoo.

Deriving a template, I always start with `typedef`, because I look at a template as a typedef for multiple types. And as an advantage, I don’t have to deal with template syntax at the beginning, i am still inexpertly in detail.

So I write:

Hide Copy Code

```
typedef int TYPE;
void show(const TYPE &val)
{
      cout << "TYPE: " << val << endl;
}
```

and I test it with different types like

Hide Copy Code

```
int i(123);
show(i);
```

If it works, I substitute the `typedef` with `template`

Hide Copy Code

```
template<typename TYPE>
void show(const TYPE &val)
{
      cout << "TYPE: " << val << endl;
}
```

and the template is done :-)

Note: `typename` is an `alias` of `class` in this context.

As next a **specialisation** of that template, because a string must be treated different form fixed-length types more often than not.

You declare it to be a template and write an implementation with the concrete type:

Hide Copy Code

```
template<>
void show(const string &val)
{
      cout << "string: " << val << endl;
}
```

Also important is the information, that for templates the declaration and the definition must be in a header-file, because the definition must be seen by compiletime. Otherwise the linker will report an error.

As next **template overloading**:

Hide Copy Code

```
void show(const char val[])
{
      cout << "char[]: " << val << endl;
}
```

is not another specialisation of function-template `show(..)`, because the variable has changed from being a reference to an array.

Here you put the declaration

Hide Copy Code

```
void show(const char val[]);
```

to the header-file and the implementation to the .cpp file.

On call of a function-template some or all types can be defined explicitly too.

For example:

Hide Copy Code

```
template<class T1, class T2>
void foo(T2 value)
{
      T1 conversionType;
      ...
};
```

will be called for `std::double` and `System::Double`

Hide Copy Code

```
Double amount;
foo<double>( amount );
```

Bingo! That's quite all, i had to know about templates, to do all the rest you'll see.

With class-templates the same rules apply.

A class can have method-templates too.

And the concrete writing of a class-template's declaration separate from the class-template's definition, you'll see in the following.

Class-method-template

Example MyStream shows a Queue as StringStream-like implementation.

The stream is generalized to all built-in types by method-templates and by template overloading.

It shows the streaming syntax for nested and inherited classes.

Note: The reference to a parent `ref class CManaged;` would look as

Hide Copy Code

```
(CManaged%)*this
```

Class-template

Example QueueTemplate implements a typesafe Queue-stream for all standard types and user-defined (UD-)classes.

I developed the UD-classes and the queue-stream-template parallel and had uncertainties here and there.

So I started developing the template with `typedef`.

The declaration in file *CObjStreamT.h*

Hide Copy Code

```
typedef COrder TYPE;
class CObjStream
{
      queue<TYPE> *bq;
public:
      CObjStream& operator<<(TYPE&);
      CObjStream& operator>>(TYPE&);
      CObjStream();
};
and the definition in CObjStreamT.cpp
CObjStream& CObjStream::operator<<(TYPE &obj)
{ ... }
CObjStream& CObjStream::operator>>(TYPE &obj)
{ ... }
CObjStream::CObjStream()
: bq(new queue<TYPE>), op(o_put)
{}
```

When I switched to template, I had to change the syntax.

All occurrences of the `class-name` as a type-definition and as scope-definition had to be changed to `class-name<TYPE>`.

And the class declaration and each method implementation had to be prefixed with `template<class TYPE>`, that it looked like

Hide Copy Code

```
template<class TYPE>
class CObjStream
{
      queue<TYPE> *bq;
public:
      CObjStream<TYPE>& operator<<(TYPE&);
      CObjStream<TYPE>& operator>>(TYPE&);
      CObjStream();
};

template<class TYPE>
CObjStream<TYPE>& CObjStream<TYPE>::operator<<(TYPE &obj)
{ ... }
template<class TYPE>
CObjStream<TYPE>& CObjStream<TYPE>::operator>>(TYPE &obj)
{ ... }
template<class TYPE>
CObjStream<TYPE>::CObjStream()
: bq(new queue<TYPE>), op(o_put)
{}
```

And the implementation had to be moved form .cpp-file to header-file.

The stream's interface has grown and very quick i took the help of some macro-definition.

You see it at the beginning of the header-file:

Hide Copy Code

```
#define CObjStream_TEMPLATE_DEBUG
#undef CObjStream_TEMPLATE_DEBUG //to DEBUG comment this line out

#if defined CObjStream_TEMPLATE_DEBUG
//                                +++ DEBUG-ON mode +++
typedef COrder TYPE;
#define CObjStream_TEMPLATE CObjStream
#define TEMPLATE_class
#else
//                                +++ TEMPLATE-ON mode +++
#define CObjStream_TEMPLATE CObjStream<TYPE>
#define TEMPLATE_class template<class TYPE>
#endif //DEBUG
and just to the end of header-file
#if !defined CObjStream_TEMPLATE_DEBUG
#include "CObjStreamT.cpp"
#endif //DEBUG
```

With that definitions it switches in DEBUG-ON to version typedef and in TEMPLATE-ON to version template.

That is very helpful, because without templates many compilers bring better error-messages.

Try it by yourself :-)

With templates we've got the last chapter to put that all together.

Now we'll make the BinaryStream smooth and handsome.

Interface, Wrapper, Template

Four major tasks are left to be done, to get a final.

- we have to enhance the interface of UD-classes to our stream
- the stream must get an interface to accept all built-in types
- the safety against misusage must be improved. The user needs strong indications, if he is acting with wrong types or in the wrong sequence.
- we need more than one datagram. We strive for a container that takes any number of any objects.

In this chapter here we will do the interface.

In chapter 2 (example BytesInterface) we built the interfaces (two; one for standard and second for managed class) and it does all we need. What could be wrong with it?

Well, the discussion is, that it has a strong implication to class-layout. Especially for managed class, that knows just single inheritance. If it is derived from another class, it can't inherit from the interface-class too. In short, we are looking for a wrapper that's loose coupled to the UD-class. And it should be a template, that fits to any UD-class.

A first discussion of that subject we did in chapter "Wrapper instead of inheritance".

Here we see that class modified as wrapper-class-template for standard-C++ UD-objects.

Hide Shrink ▲ Copy Code

```
template<class CANYN>
class BinStreamN: public INbinary
{
      CANYN &objReference;
      BinStreamN(); //no Default-Constructor
public:
      BinStreamN(CANYN &objRef)
      :objReference(objRef)
      {
            className = "BinStreamN";
      }
      BinaryN& Out(BinaryN &str)
      {
            return objReference.Out(str);
      }
      BinaryN& In(BinaryN &str)
      {
            return objReference.In(str);
      }
      const __int32 mySize() const
      {
            return objReference.mySize();
      }
      wostream& Out(wostream &str)
      {
            str << "Cname=\"" << StringConvert::wstr(className) << "\" ";
            return objReference.Out(str);
      }
};
template<class CANYN>
wostream& operator<<(wostream &str, BinStreamN<CANYN> &obj){ return obj.Out(str); }
```

The wrapper inherits the Ibinary interface-class, takes an object-reference and redirects all interface-calls to the referenced object's implementation.

The syntax to instantiate a concrete wrapper and giving it an object-reference is:

for standard class on local stack:

Hide Copy Code

```
CNative objnative;
BinStreamN<CNative> wrnative( objnative );
```

for standard class on heap:

Hide Copy Code

```
CNative *objnative = new CNative();
BinStreamN<CNative> wrnative( *objnative );
```

for managed ref-class on GC-heap

Hide Copy Code

```
CRefclass^ objrefclass = gcnew CRefclass();
BinStreamR<CRefclass> wrrefclass( objrefclass );
```

for managed ref-class on local stack (alike)

Hide Copy Code

```
CRefclass objrefclass;
BinStreamR<CRefclass> wrrefclass( %objrefclass );
```

Example _Common implements that wrapper-class-template.

With this user-interface (UI) of UD-class to our Binary-Stream I'm satisfied. The UI-class is loose coupled to the UD-class, design and lifetime of UD-class is independent of UI.

You act with the UD-class as usual:

Hide Copy Code

```
CNative *objnative = new CNative (); //         instantiate an UD-object
BinStreamN<CNative> wrnative(*objnative ); //     wrapper binds to UD-object
bstream >> wrnative; //            UD-object gets the data by wrapper
objnative->Text(L"!:-O  ;-))  :) :D  XD :p"); // modify the UD-object
bstream << wrnative; //      stream-out the UD-object by wrapper
```

We are ready to finalize now!

Finals

The remaining three tasks we'll do here.

1. we need more than one datagram. We strive for a container that takes any number of any objects.
2. the stream must get an interface to accept all built-in types
3. the safety against misusage must be improved. The user needs strong indications, if he is acting with wrong types or in the wrong sequence.

Final of ByteStream

Example BytesInterface

It's similar to example ByteStream expanded by the Queue-Stream and improvements.

To accomplish task A), I add a Queue-Stream to class Bytes by deriving.

Hide Copy Code

```
class BytesQueue : public Bytes
{
      queue<Bytes> *bq;
      ...
```

Class Bytes acts with the UD-object and datagram and BytesQueue manages these datagrams.

BytesQueue additional provides an interface to STL-streams to deal with any of them. STL-StringStream and STL-FileStream are exemplary implemented.

Hide Copy Code

```
    //++ backbone-interface of BytesQueue ++
    void push();
    void pop();
    stringstream& Out(stringstream &str);
    stringstream& In(stringstream &str);
};
stringstream& operator<<(stringstream &str, BytesQueue &obj);
stringstream& operator>>(stringstream &str, BytesQueue &obj);
```

To accomplish task B), the helpers to class `Bytes`, `BinaryN` and `BinaryR` get method-templates to `operator<<` and `operator>>` and template-overloads to the three string-types `std::string`, `std::wstring` and `System::String`.

The method `sizeBytes(..)` must become a template too with template-overloads to the string-types.

To accomplish task C), each storage of a basic type is preceded by a typeidentity, that marks the start of the type's datagram and the type of the data that follows. The typeidentity is not unique to all different types. For example an `__int64` can be stored and read by a `double`, because both have a type-size of 8 bytes. But it's impossible to intermix types of different size or `string` with `wstring`.

Keep in mind: If you are mixing standard-C++ and managed-C++, there is no corresponding type to `std::string` in managed code. `System::String` corresponds to `std::wstring` only.

And the selfmade `ByteStreamT` is ready to use now.

The outstanding advantage of ByteStreamT

With its functional principles it can deal with objects holding any kind of binary data, for example images, audio streams and so on.

Final ByteStreamT consists of the following files:

Hide Copy Code

```
BinaryN (.h .cpp)
BinaryR (.h .cpp)
BinStreamN (.h)
BinStreamR (.h)
ByteQueueStream (.h .cpp)
Bytes (.h .cpp)
BytesR (.h .cpp)
Hexdump (.h .cpp)
INbinary (.h .cpp)
IRbinary (.h .cpp)
StreamDoNothing (.h .cpp)
StringConvert (.h .cpp)
```

The following files belong to the UD-application:

Hide Copy Code

```
ByteStreamT (.cpp)
Complex_N (.h .cpp)
Complex_R (.h .cpp)
Date_N (.h .cpp)
Date_R (.h .cpp)
Simple_N (.h .cpp)
Simple_R (.h .cpp)
```

Final MyStream

The example of MyStream has been shown already in the chapter about templates.

Its advantages are:

- generally applicable
- it acts STL-stringstream-like without having its implications
- you haven't to deal with spaces in strings or empty strings it solves that problem for you
- persisted files are human-readable

Final MyStream consists of the following files:

Hide Copy Code

```
CMyStream (.h .cpp)
StreamDoNothing (.h .cpp)
```

The following files belong to the UD-application:

Hide Copy Code

```
CArticle (.h .cpp)
COrder (.h .cpp)
```

```
CPerson (.h .cpp)
CShipAdvanced (.h .cpp)
CShippment (.h .cpp)
CTransaction (.h .cpp)
MyStream (.cpp)
```

Final StringRstream

StringRstream is an exemplary implementation that shows how to stream managed objects and System:: types with STL-streams.

It demonstrates the technique and has all essential elements.

Final StringRstream consists of the following files:

Hide Copy Code

```
RStream (.h .cpp)
StreamDoNothing (.h .cpp)
StringConvert (.h .cpp)
StrString (.h .cpp)
```

The following files belong to the UD-application:

Hide Copy Code

```
CArticle (.h .cpp)
COrder (.h .cpp)
CPerson (.h .cpp)
CShippment (.h .cpp)
CTransaction (.h .cpp)
StringRstream (.cpp)
```

I wish you much fun and success with streaming all around.

;-))

Feedback

Please let me know your experience with any of the provided articles and sources, or any questions about.
And it would be interesting if other compilers have implemented the same or different behavior with STL-streams.

Sources

Binary Stream

T2_Bytes1	The principals of class Bytes (= my Binary-Stream)
declares + implements the basics of class Bytes main() is a rich commented example of using the interface of Bytes.	
T3_Object1	Bytes-Stream with objects, implementing the common pattern of having the stream as friend of the class.
declares + implements a UD-class with it's interface to class Bytes. Class Bytes is moved to seperate header and implementation. main() shows the usage.	
T4_Bytes2	Bytes-Stream with managed Ref-Objects, implementing the alternative pattern without having the stream as friend of the class.
declares + implements a UD-managed ref-class with its interface to class Bytes.	

Because it's impossible for managed classes to have a friend-declaration, an alternative pattern is implemented for the cooperation with a stream.
The details to the alternative pattern are shown in chapter "alternative Pattern".

Interface for classes

BytesInterface	How Bytes works Bytes as a datagram of an object
that implements the full interoperation of standard- and managed UD-classes. Bytes-Stream gets two new interface-classes BinaryX and IXbinary. BinaryX does the writing and reading of data-members, IXbinary provides the interface that UD-class needs to Bytes-Stream, that can be inherited by a UD-class.	

Streaming all around

ByteStream	Bytes with StringStream-Backbone a Set of object-datagrams
implements an interface of Bytes-Stream to STL-StringStream, to get a Set of object-datagrams	

alternate Pattern

T1_pattern	Ostream 'wcout' for managed types
explains the necessity of the pattern and demonstrates its implementation.	
Ostream	New pattern with Ostream; Classes with Inner-Classes
exemplary implementation of the pattern with nested classes	
Stringstream	New pattern with StringStream
exemplary implementation of the pattern for output- and input-stream StringStream.	
demonstrateStr	Syntax pitfall
demonstration of 'do nothing', or not?	
TraditionalStream	Traditional pattern with StringStream
an implementation of example "Stringstream" with the traditional pattern.	

Syntax of STL-Stream effects or defects?

demonstrateStr	Syntax pitfall
the missing link in Stream-Syntax?	
LeftToRight	Exploring the left to right associativity of different Streams. Syntax exploration, pitfalls
Left to Right isn't allways left-to-right. To get the full experience, see the article.	
OstrFormatter	Syntax exploration
an example about methods in stream-statements	
Manipulator	Stream Manipulator for Native & Managed Code
manipulates its parameter before output and past input. Implemented in StrString(.h .cpp)	

Own Streams

CMyQstr(.h .cpp)	in project LeftAndRight
simple Example with the core-elements a stream needs.	
MyStreamX	a Stream develloped on one's own
Exercise with Basic-Types and nested classes shows a technique to serialize Queues or arrays. shows a general way to persistence via STL-StringStream and STL-FileStream.	

Template

TemplateFoo	Example: Function-Template
a Function-Template, Template-Specialization, Template-Overloading	
MyStream	Queue as StringStream-like implementation
works StringStream-like without the StringStream-complications, generalized by template-methods,	
QueueTemplate	Typesafe Queue-Stream for all std-types and UD-classes
implements a typesafe Queue-Stream. Exemplary implementation of a template-class with spezialisation.	

Wrapper, Template

_Common	Bytes & Interfaces & template-class as wrapper of IN-/IRbinary ByteStreamT uses that templated interface.
implements an interface wrapper-class template, that takes a reference of UD-class. It has least implications for the design of UD-class.	

Finals

ByteStream	BytesQueue; uses the templated interface _Common
Bin-Queue as container for UD-Classes a stream that is a Set of binary Object-Datagrams. For all standard and managed basic-types and UD-classes. With a generalized interface to STL-Stream for persistence.	
MyStream	Queue as StringStream-like implementation
a container for all standard and managed basic-types and UD-classes. With a generalized interface to STL-Stream for persistence. Generally applicable, files are human-readable.	
StringRstream	Managed objects and System::Types streamed to and from STL-StringStream ;-D
Even managed types and objects are streamable with STL-Streams.	