C++ toolkits™

STL in Action: A Universal Streaming Service

By Graham Glass

Introduction

Now that STL is becoming an established part of every C++ toolkit, programmers are eager to add nonstandard extensions to STL that make it more powerful. In my last column, I described the concept of *helper algorithms*, which are a nonintrusive way to make STL algorithms easier to use. In this column, I describe a *universal streaming service* (USS) that allows STL containers and other classes to be stored into a file or sent across an interprocess communication mechanism. USS is a lightweight mechanism that allows processes to make objects persistent and transportable without incurring the expense and overhead of a full object database or object communication system such as CORBA.

For an entity to be streamed, it must be encoded into a format that allows the entity to be stored or transported and then later decoded back to its original state. In addition, the format should ideally allow the entity to be moved between different machines. Before the advent of object-oriented technology, most encoding of primitives (such as `chars`, `ints`, and `floats`) and composites (such as `arrays`, `unions`, and `structs`) was done using schemes such as XDR (External Data Representation), NDR (Network Data Representation) and ASN.1 (Abstract Syntax Notation). None of these encoding schemes were designed to work with C++. For example, they don't allow objects with virtual functions to be encoded automatically.

USS improves upon these older streaming approaches by allowing C++ objects to be easily encoded and decoded with very little programming effort. USS uses NDR, which is more time and space efficient than XDR, to encode the primitive parts of an object, and it adds additional machinery to support specific C++ object-oriented features.

USS has several qualities that make it ideal for use with STL and which give it an advantage over older object streaming designs:

- It supports advanced C++ language features such as templates, multiple inheritance, and virtual base classes.
- It is nonintrusive, allowing a class to be made streamable without requiring modification in any way.
- It is vendor independent, allowing objects from any vendor to be streamed.
- It is device independent, allowing streaming to I/O devices such ANSI IOStreams, sockets and pipes.
- It maintains the morphology of a network of objects.
- It is typesafe.
- It uses the familiar << and >> operators for object I/O.

This column is based on the work that I and other engineers performed for a product called Systems Toolkit*. This product contains a streaming system similar to the one described in this column and accommodates compiler limitations that otherwise would make such an approach impossible.

The purpose of this column is to describe the problems we encountered as we designed a streaming system that could handle STL elegantly, together with their solutions. The final approach is generic enough that it can be used to tackle many other problems of a similar nature.

The structure of the column is as follows:

- A description of the three main USS design goals
- Some examples of USS in action
- The steps that must be taken to enable a class for binary streaming
- A multipart description of how USS works
- A synopsis and summary

Goal #1: Be Nonintrusive

Before an object may be streamed, its class must be "binary-enabled"; that is, the class must know how its instances are to be written and read from a binary stream. Many of the older streaming designs require you to inherit from a common base class and override certain virtual functions to achieve this goal. For example, Tools.h++ requires you to inherit from `RWCollectable`, C++ Views require you to inherit from `Vobject`, and the MFC library requires you to inherit from CObject. This is an undesirable requirement because it ties your source code to that of another vendor. It also prevents you from adding streaming capability to classes from other vendors that do not adhere to your local policy, and STL is one such set of classes.

One workaround to this restriction is to create a new class that inherits from both the common root class and from the class that you wish to make streamable. This approach is not very pleasant, as it adds a new class for every class that you wish to stream and litters your source code with "streamable" versions of your original classes. We therefore decided that USS should adopt a nonintrusive policy that allows a programmer to externally associate streaming behavior with a class without having to modify the class itself.

Goal #2: Support Advanced Language Features

Many of the older streaming systems do not support the streaming of template classes or classes with virtual bases. This is especially noticeable by STL programmers, who use templates on a daily basis. USS supports these advanced language features in a manner that is described later in this column.

Goal #3: Be Vendor Independent

For USS to be useful to a wide range of programmers, it had to be vendor neutral. Its design allows objects from Recursion Software, Rogue Wave, Microsoft, and other vendors to be streamed using the same system.

Some Examples of the Universal Streaming Service

Before I describe the design of USS, I'd like to show you some examples of how it is used. These examples assume that you have binary-enabled a class `X` according to the following steps:

- If `X` is a nontemplate class with a header file and source file called `X.h` and `X.cpp` respectively, you have created a file called `X_b.cpp` that contains the binary streaming extensions for `X`. The object module associated with `X_b.cpp` must be linked into the final executable.

* See www.recursionsw.com

- If X is a template class with a header file called X.h, you have created an additional file called X_b.h that contains the binary streaming extensions for X.

Note that none of these steps require the header or source file of X to be modified in any way. The details of each step are described later in this column.

Instances of binary-enabled classes are streamed to and from an I/O device using an instance of a class called bstream (binary stream). This class is defined in a header file called "bstream.h."

When a bstream is constructed, it is associated with a particular I/O device via an intermediate *adapter* object. Adapters may be created for any kind of I/O device, including ANSI/ISO IOStreams, sockets and pipes. The function adapter_for() returns the appropriate adapter for a particular device.

When a primitive or object is written to a bstream, it is converted into a portable binary format and stored in an internal buffer. When a write operation completes, the internal buffer is flushed to the I/O device via the adapter. This technique allows the binary streaming mechanism to be fully decoupled from details of any particular I/O device. A similar buffering process occurs when a primitive or object is being read from a bstream.

Example 1: Streaming an Instance of a Nontemplate Class

The first example streams an instance of person to an ANSI/ISO fstream and then restores it. The code for person_b.cpp is presented later in this column. Please note that the data members are public purely for simplicity. In a commercial application, these attributes would be read and written via accessors. The memory addresses of the person p1 and his buddy are displayed to show that the pointers are restored correctly:

person.h

```
#include <iostream.h>
#include <string.h>

class person
  {
  public:
    person() : buddy_( 0 ) {}
    void print() const
      {
      cout
        << "Name = " << my_name_ << "@ " << this << endl;
      cout
        << "Buddy = " << buddy_->my_name_ << " @ "
        << buddy_ << endl;
      }
    string my_name_;
    person* buddy_;
  // Pointer to my buddy
  };
```

main.cpp

```
#include <fstream.h>
#include "bstream.h"
#include "person.h"

void write()
  {
  // Create an ANSI/ISO file stream.
  fstream file
    (
    "example1.bin",
    ios::out | ios::trunc | ios::binary
    );
  // Create a binary stream on the file stream.
```

```
    bstream stream( adapter_for( file ) );
    // Create two people, name them, and make them mutual
    // buddies.
    person* p1 = new person;
    p1->my_name_ = "Graham";
    person* p2 = new person;
    p2->my_name_ = "David";
    p1->buddy_ = p2;                  //make p2 a buddy of p1
    p2->buddy_ = p1;                  //make p1 a buddy of p2
    // Write p1 to the binary stream
    stream << p1;
    // Delete the people.
    delete p1;
    delete p2;
    }

  void read()
    {
    // Open the existing ANSI/ISO file stream.
    fstream file
      (
      "example1.bin",
      ios::in | ios::nocreate | ios::binary
      );
    // Create a binary stream on the file stream.
    bstream stream( adapter_for( file ) );
    person* p1;
    // Read p1 from the stream
    stream >> p1;
    person* p2 = p1->buddy_;
    // Print p1 and his buddy.
    cout << "p1 = ";
    p1->print();
    cout << "p2 = ";
    p2->print();
    // Delete the people.
    delete p1;
    delete p2;
    }

  int main()
    {
    write();         // Demonstrate writing to a binary stream.
    read();          // Demonstrate reading from a binary stream.
    return 0;
    }
```

output

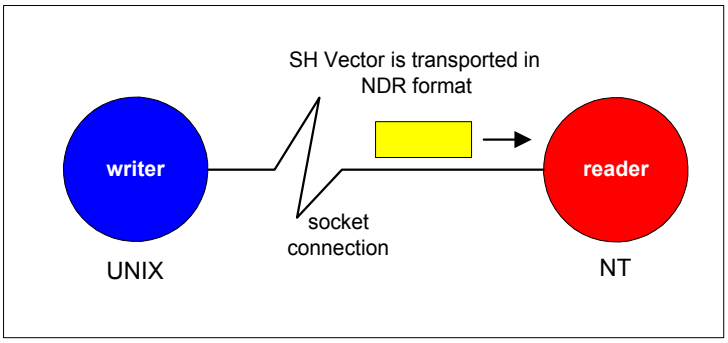
```
    p1 = Graham @ 0x85de8
    buddy = David @ 0x86e48

    p2 = David @ 0x86e48
    buddy = Graham @ 0x85de8
```

Example 2: Streaming an Instance of a Template Class

The second example streams an STL vector of integers between two processes using a socket connection. The program `writer.cpp` runs on a UNIX machine and sends a vector to the program `reader.cpp`, which runs on an NT machine, as illustrated by Figure 1.

Figure 1. Streaming an STL Vector



To stream a template class, the user must `#include` the binary-enhanced header file instead of the class's regular header file, and use the `STREAMABLE` macro family (described later) to register the particular instantiation of the template class with USS at startup. Note that the standard header file `vector.h` does not have to be modified in any way.

writer.cpp (runs on a UNIX machine)

```
#include "sockets.h" // Import socket classes.
// Use binary extension header instead of "vector.h"
// This file includes "vector.h" and then adds some binary
// extensions.
#include "vector_b.h"

// Register vector< int > with USS.
STREAMABLE_0( (vector< int >*) )

int main()
  {
  vector< int > v; // Create and populate an STL vector.
  v.push_back( 1 );
  v.push_back( 42 );
  // Connect to port 2000 on the machine called "NT".
  tcp_socket socket( socket_address( "NT", 2000 ) );
  // Create a binary stream on the socket.
  bstream stream( adapter_for( socket ) );
  stream << v; // Stream the vector to the socket.
  return 0;
  }
```

reader.cpp (runs on an Windows machine)

```
#include "sockets.h"       // Import socket classes.
// Use binary extension header instead of "vector.h"
#include "vector_b.h"

// Register vector< int > with USS.
STREAMABLE_0( vector< int >* )

int main()
  {
  // Create a socket server on port 2000.
  tcp_connection_server server( socket_address( 2000 ) );
  tcp_socket socket;
  // Accept an incoming socket connection.
  server.accept( socket );
  // Create a binary stream on the socket.
  bstream stream( adapter_for( socket ) );
```

```
    vector< int > v;
    stream >> v; // Read a vector< int > from the socket.
    // Display the vector's contents.
    for ( int i = 0; i < v.size(); i++ )
      cout << v[ i ] << endl;
    return 0;
    }
```

output

```
    1
    42
```

Binary-Enabling Classes

Before I describe the design of USS, I'll give you a guided tour of the extra header and source files that were required by the previous examples. Since the steps that you must perform to binary-enable your classes are slightly different for nontemplate and template classes, a separate section for each variation is presented next.

Binary-Enabling Nontemplate Classes

Here are the contents of the binary extension file of person, which is called `person_b.cpp`:

person_b.cpp

```
#include "person.h"
#include "bstream.h"

// Declare person as a class with zero base classes.
STREAMABLE_0( (person*) )

// Define code for writing a person to a binary stream.
void write( bstream& stream, const person& object )
  {
  stream << object.my_name_ << object.buddy_;
  }

// Define code for reading a person from a binary stream.
void read( bstream& stream, person& object )
  {
  stream >> object.my_name_ >> object.buddy_;
  }
```

In general, the contents of the binary extension file `X_b.cpp` for a class `X` should:

- Include `X.h` and `bstream.h`.
- Use the `STREAMABLE` macro (defined in `bstream.h`) to register `X` and its inheritance structure.
- Define the bodies of the nonmember `read()` and `write()` functions.
- Some versions of the `STREAMABLE` macro allow you to declare classes that inherit from one or more base classes.

Binary-Enabling Template Classes

Here are the contents of the binary extension file for vector, which is called `vector_b.h`:

vector_b.h:

```
#include "bstream.h"         // Binary streaming header.
#include <vector.h>          // Standard #include for vector.

// Define code for writing a vector to a binary stream.
template< class T >
void write( bstream& stream, const vector< T >& object )
  {
  stream << object.size();
```

```
  const T* i;
  for ( i = object.begin(); i != object.end(); ++i )
    stream << *i;
  }

// Define code for reading a vector from a binary stream.
template< class T >
void read( bstream& stream, vector< T >& object )
  {
  object.erase();                          // Clear the vector.
  vector< T >::size_type n;
  stream >> n;                       // Read its size.
  object.reserve( n );     // Reserve n elements in the vector.
  // Read each element.
  for ( vector< T >::size_type = 0; i < n; i++ )
    {
    T t;
    // Read element and add to the end of the vector.
    stream >> t;
    object.push_back( t );
    }
  }
```

In general, the contents of the binary extension file `X_b.h` for a template class `X` should:

- Include `X.h` and `bstream.h`.
- Define the bodies of the nonmember `read()` and `write()` template functions.

How It Works

Now that you've seen some examples of how USS can be used and what is necessary to binary-enable an individual class, it's time to examine how the system actually works. Like most designs, there's a lot more activity under the hood than you might first expect. The design is presented in seven parts, each of which builds on the previous parts.

The discussions make some assumptions about your compiler's support for the C++ language. It is assumed that the compiler supports RTTI and compiles templates as described in the current C++ language specification. There are workarounds for every known case where these assumptions are not true, but their description is beyond the scope of this column.

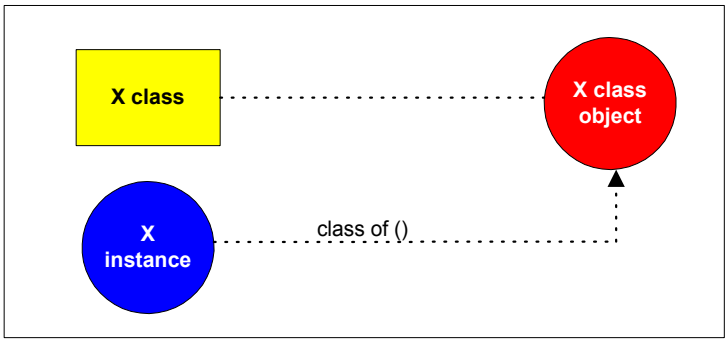
Part 1: Streaming Primitives

This is the easy part. When a primitive is written to a `bstream`, the `bstream` converts it into a portable NDR binary format that includes type information and stores the output in an internal buffer. This buffer is later flushed to the `bstream`'s adapter, which in turn sends the output to its associated I/O device. A similar process occurs when a primitive is read. The encoded type information is used to perform run-time type checking.

Part 2: Class Objects

The design relies on a concept that is common to the Smalltalk, Objective C and Java languages — that every class has an instance of a special “class object” that is created during system initialization. This class object contains metadata that describes key characteristics of the class, such as its name and pointers to its immediate base classes. The class inheritance structure information is important for dealing with virtual base class situations as described later in this article. The `STREAMABLE` macro family is responsible for the creation and initialization of class objects (see Figure 2).

Figure 2. Creation and Initialization of Class Objects



Note that every unique template instantiation is considered to be a new class. For example, a `vector<int>` would generate a separate class object from `vector<float>`. This is why the `STREAMABLE` macro must be used for every different instantiation of a vector.

When a class object is created, it is stored in a class map whose key is the RTTI value associated with the class that it represents. You may obtain the class object associated with a particular instance by using the nonmember template function `class_of()` which is defined in `bstream.h`. This function uses the RTTI value associated with an instance to index into the class map and locate its associated class object.

Part 3: Facets

Rather than require that the functions for writing and reading an instance be placed into a class, they are associated with the class by storing pointers to these functions in the class object. These “loosely coupled attachments” are known as *facets* of the class. The technique of extending a class's behavior by registering nonmember extensions with a separate object has been adopted by the `locale` class in the new Standard C++ Library.

Because each version of the `write()` and `read()` functions has a different signature, their addresses cannot be stored directly. An indirection scheme is therefore used. After the `STREAMABLE` macro has emitted code for creating and registering a class object, it emits the code for two static nonmember functions called `WRITE()` and `READ()` that are defined for class `X` as follows:

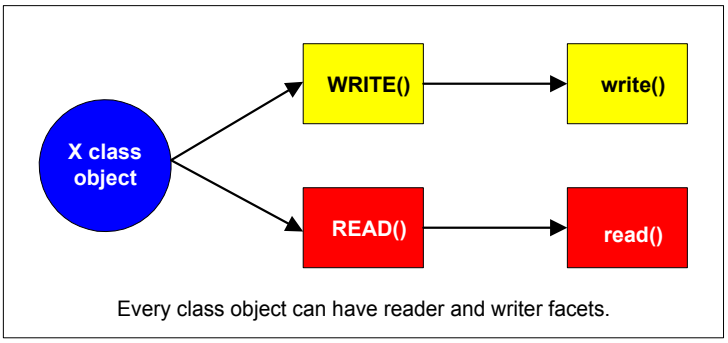
```
static void WRITE( bstream& stream, void* object )
  {
  // Call typesafe binary extension to write an X object.
  write( stream, * (X*) object );
  }

static void READ( bstream& stream, void* object )
  {
  // Call typesafe binary extension to read an X object.
  read( stream, * (X*) object );
  }
```

The macro then emits code that passes the class object the address of these `WRITE()` and `READ()` static nonmember functions. These functions can be retrieved from the class object at a later stage during a streaming operation and used to execute the class's typesafe `write()` and `read()` functions (see Figure 3). This approach is based on the External Polymorphism pattern by Doug Schmidt1.

[1] available at http://www.cs.wustl.edu/~schmidt/EuroPLoP-96.ps.gz

Figure 3. Read and Write Functions



Part 4: Streaming Objects

The streaming operators for objects are defined as template nonmember functions in `bstream.h`.When an object is streamed into a `bstream`, `operator<<` passes it the address of the object cast to a `void*` and a reference to its class object. The `bstream` first writes the name of the class to the stream so that type checking may be performed when the object is read. It then uses the class object to locate the class's nonmember `WRITE()` function and then calls it with the `bstream` and the object's address as parameters. The `WRITE()` function calls its associated typesafe `write()` function, which streams out the components of the object.

Similarly, when an object is streamed out of a `bstream`, `operator>>` passes the destination pointer/reference as a `void*` and a reference to the class object of the required type. The `bstream` reads the class name from the stream and performs dynamic type checking to make sure that the type of the required object is type-compatible with the type of the object that is about to be read. It then uses the class object to locate the class's nonmember `READ()` function and then calls it with the `bstream` and the object's address as parameters. The `READ()` function calls its associated typesafe `read()` function, which streams in the components of the object.

Part 5: Morphology

When an object containing pointers to other objects is written and then restored, its morphology is maintained. This means all of the objects it directly or indirectly points to are written exactly once, and the object pointers are correctly restored when the object is read.

The `bstream` object records the address and type of each object that is streamed to it. Once an object has been written once, subsequent writes of the same object cause a special reference to the object to be written instead of the object itself. These special references are detected during the reading process and are processed accordingly. This technique prevents objects that are referenced multiple times from being needlessly written more than once, and allows self-referential structures to be streamed without causing an infinite loop.

Part 6: Compatibility

Users of existing and less flexible streaming services such as those found in MFC and Tools.h++ may easily upgrade to USS. USS defines streaming operators that accept objects whose class is rooted in either `CObject` or `RWCollectable`. The `operator<<` uses the vendor's local streaming service to write the vendor's object into a memory buffer and then writes the buffer as an unformatted block of bytes. Similarly, `operator>>` reads the unformatted block of bytes into a memory buffer and then uses the vendor's local streaming service to decode the bytes into an object.

Part 7: Casting Issues

The casting operations described earlier are complicated when an instance of a class with multiple base classes is streamed in or out via a pointer or reference to one of its base classes. The following examples illustrate how

USS would react without its unique dynacast (dynamic casting) enhancements. Assume that the C class inherits from the A class and the B class.

In the following example, USS without dynacast would cast the address of the A subcomponent of the C object to a void*, and then cast it back to a C* in the WRITE() function associated with the C class. It would therefore attempt to read an entire C object from the area reserved just for its A subcomponent.

```
C c;
A& a = c;
stream << a;
```

In the next example, assume that a C object is being read from the stream. USS without dynacast would create a C object, cast its address to a void*, and then store this address into a B pointer. This is incorrect, since a pointer to a subcomponent of an object is generally not the same as a pointer to the containing object.

```
A* a;
a = new C;
stream << a; // Stream out a C object.
// _ other code goes here.
B* b;
stream >> b; // Stream in a C object.
```

Fortunately, both of the problems can be overcome in a portable manner. Dynacast performs correct up- and down-casting between any pair of type-compatible classes and will also generate run-time errors in the case of a cast mismatch. Both of the preceding examples therefore work correctly with USS. The description of this system is beyond the scope of this column, and will probably surface in a future column.

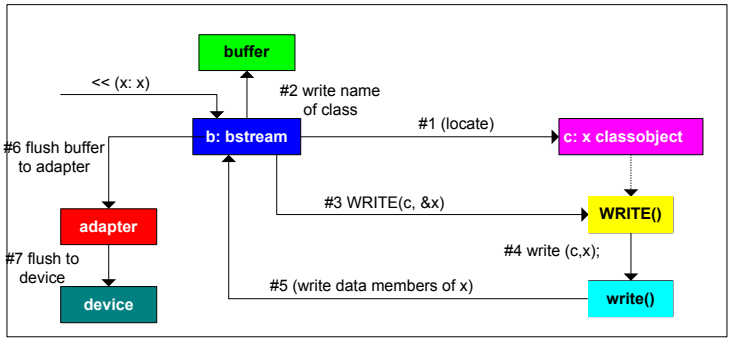
Synopsis

Here is a synopsis of the operations that occur when an instance of X is written to a bstream:

- The class object associated with the instance is located.
- The name of the class is written to the stream for later type checking.
- The WRITE facet of the class object is called with the raw address of the instance.
- The write() function associated with the X class is called with the typesafe address of the instance.
- The data members of the object are written to the stream.
- The contents of the bstream's NDR buffer are flushed to its adapter's I/O device.

Figure 4 illustrates these steps.

Figure 4. Synopsis of Operations



Similarly, here is a synopsis of the operations that occur when an instance of `X` is read from a `bstream`:

- The contents of the `bstream`'s NDR buffer are filled from its adapter's I/O device.
- The name of the class is read from the stream.
- The name is used to locate the incoming object's associated class object.
- Type checking is performed.
- The `READ` facet of the class object is called with the raw address of the target object.
- The `read()` function associated with the `X` class is called with the typesafe address of the target object.
- The data members of the object are read from the stream.

Summary

This column presented the design of a universal streaming service with the following characteristics:

- It uses facets to attach streaming facilities to a class in a nonintrusive manner.
- It uses RTTI to locate an instance's class object at run time.
- It integrates smoothly with other vendors' monolithic streaming services.

In general, the facets technique can be used to extend an existing class's behaviors without modifying the class in any way. In my next column, I'll describe some techniques for porting legacy applications to STL.

Recursion Software, Inc.
2591 North Dallas Parkway, Suite 200
Frisco, Texas 75034
1.800.727.8674 or 972.731.8800
www.recursionsw.com

RECURSION SOFTWARE, Inc.