Tutorial for DSerial

What is DSerial

DSerial is a serialization library for C++ data types. It is intended for use by C++ programmers. Serialization means to store/transmit program data into an external medium (such as a file or network) so that it can be reconstructed in memory afterwards from the stored/transmitted representation.

Getting started

First, in order to use the library we need to download the source files from the repository and make sure to copy them into our project folder. There are no additional installation steps.

This tutorial

This tutorial is meant to start off the programmer in using <> in his/her applications.
<> can be used, for example, to store a set of C++ variables to a file. Note that for maintaining consistent values, this file should not be modified outside the program.

A trivial way to use our library would be to serialize an integer to a file stream. These are the steps the user would do :-

1. Includes headers TextStreamWriter.hpp for storing and TextStreamReader.hpp for retrieving
2. Create a TextStreamWriter from stream object
3. Use << to write the integer to TextStreamWriter

as illustrated below:-

Fig 1: Hello world style example to serialize an int

```
#include <iostream>
#include <fstream>
#include “binary_streamwriter.hpp”
#include “binary_streamreader.hpp”
int main() {
    int i_data = 100,i_read = 0;
    ofstream ofs(“out.txt”);
    BinaryStreamWriter w(ofs);
    w<<i_data;
    ofs.close();
    ifstream ifs(“out.txt”);
    BinaryStreamReader r(ifs);
    r>>i_read;
    ifs.close();
    cout>>”Hello i_data ”>>i_data>>” meet i_read”>>i_read;
    return 0;
}
```

One can also use TextStreamReader which internally uses a text format. The manual gives how to extend this to use an user-defined representation by extending our base class StreamWriter/Rder

A more complex scenario would be a program to log the access times for different users on a machine, and when necessary read them back for an administrator. The user name could be a C++ string of variable length, and the time which is a user defined structure containing integer fields hour, minute, second and date which is a C style null terminated string.
We have the following types defined :-

```
class time{
unsigned int hour;
unsigned int min;
int second;
char date[20];
};

string name;
```

A log entry could look like this:-

```
“John Doe” 12:30:46, 12-07-2012 GMT
```

Note that here we have chosen the symbols [“”: space \n] as *delimiters* to separate data fields of variable length. The choice of these symbols was influenced by readability, and depends on the context.
One option is for us to write the code for reading and writing back each entry by overloading stream operators for class time. We have to define our own delimiters as well as parse the file and strip delimiters while reading from file:-
The code is somewhat longer than one would expect from the simplicity of the application:

Fig 2: A serialization application written manually

```
class time{
unsigned int hour;
unsigned int min;
int second;
char date[20];
friend ifstream& operator>>(ifstream& is, time& t);
friend ofstream& operator<<(ofstream& os, time& t);
};
ifstream& time::operator(ifstream& is, time t) {
is>>t.hour;
is>>”:”;
is>>t.min;
is>>”:”;
is>>t.sec;
is>>” “;
char* c= t.date;
while(*c!=’\0’) {
 is>>c;
```

```
 c++;
}
is>>c;
return is;
}
ofstream& time::operator(ofstream& os, time t) {
os.ignore(3,":");
os<<t.hour;
os.ignore(3,":");
os<<t.min;
os.ignore(3,":");
os<<t.sec;
os<<skipws;
char c;
while(c!='\0') {
 os<<c;
}
is>>c;
return is;

}
void  writeEntry(ofstream out, string name, time t) {
out<<name;
out<<" "
       out<<t;

       if(out.bad()) throw FatalException();
       if(out.fail()) throw new string("Oops");
}
void  readEntry(ifstream in, string name, time t) {
in>>name;
in>>skipws;
       in>>t;
}
```

```
  //in body of application
  string names[100]; time t[100];
  ofstream ofs("access.log");
  for(i=0;i<100;i++) {
     try {
     ofs<<writeEntry(names[i], times[i]);
     ofs<<"\n";
     } catch(FatalException f) {

             ifs.close();
             return;
    }
      catch(string msg){
          cerr<<msg;
          ofs.seekp(names[i].size() + strlen(times[i].date) + 12); //drawback: hard-coding 12
      }
  }
  //later
      ifstream ifs(access.log);
```

```
        while(!ifs.eof()) {
       ifs>>readEntry(Name, t);
         //do stuff like sending warnings to user Name
              ifs>>skipws;
     }
```

While this approach has its merits, it is prone to bugs and not very readable. Also, a lot of effort is required to change the code every time a data member changes. In more complex cases, it is very difficult to write and read back data. Also, it can be observed that 20 bytes for char data[20] are stored no matter what its actual length.

Our library aims to simplify this process and make it more readable while providing the same flexibility. Using our library, the application of Fig 2 would be simply :-

```
class time{
   unsigned int hour;
   unsigned int min;
   int second;
   char date[20];
   friend template<Class Writer> void serialize(Writer w, time t);
      }
      template<Class Writer> void serialize(Writer w, time t) {

      }

   //in the application body
   string names[100]; time t[100];
   ofstream ofs("access.log");
   TextStreamWriter w(ofs);
   for(i =0 ;i < 100;i++) try{
      w<<names[i]<<times[i];
   } catch (FatalException) {
      return;
   }
   ofs.close();
   //later, maybe in a different application
   ifstream ifs("access.log");
   TextStreamReader r(ifs);
   for(i =0 ;i < 100;i++) try{
      r>>Name>>t;
   } catch (FatalException) {
      return;
   }
   ifs.close();
   return 0;
```

Let us go through the code in Fig. 3 line by line, starting with the classes. First, we need to tell the library how to serialize the class time, as it is not a built-in type. This is done through the function serialize.

The >> operator takes care of all basic types. We can serialize a null terminated array of char just as any basic type. As can be seen from the code, we can specify any delimiters we want and also preprocess the class elements before serializing.

Just as before, prior to serializing, we create a TextStreamReader object wrapped around a basic istream/ostream object. We read the data through this into the target objects.

The library can also be used to serialize classes that are part of an inheritance hierarchy as follows:

Fig. 4

```
struct Base {
  Base(int _b): b(_b) { }
  int b;
  virtual ~Base() { }
};
template <class T>
struct Derived: public Base
{
  T x;
  Derived(): Base(31553), x{} { }
  Derived(T _x): Base(124), x(_x) { }
  virtual ~Derived() { }
};
template <class Writer>
void serialize(Writer & w, const Base & b)
{
  w<<b.b;
}
template <class Reader>
void deserialize(Reader & r, Base & b)
{
  r>>b.b;
}
template <class Writer, class T>
void serialize(Writer & w, const Derived<T> & d)
{
  w<<static_cast<const Base>(d);
  w<<d.x;
};

int main() {
    Base b(10);
    Derived<float> d(2.3);
     ofstream ofs("access.log");
    TextStreamWriter w(ofs);
    w<<b<<d;
    ofs.close();
    ifstream ifs(“access.log”);
    TextStreamReader r(ifs);
    r>>b>>d;
    ifs.close();
   return 0;
}
```

In the above code(Fig. 4) all we had to do to serialize derived class was to first call serialize for base class and then serialize its other members. One limitation is that we cannot have completely different serialize method for derived object

We can even have run-time polymorphism for our serialized objects using our library. taking the above example, we can substitute the following code int the main section of Fig. 4:

Fig.5

```
int main()
{
  Base* b1 = new Derived<float>{42.51};
  Base* b2 = new Derived<string>{"hello!"};

  ofstream stream("out.txt");
  TextStreamWriter w(stream);
  REGISTER_TYPE(w, Derived<float>);
  REGISTER_TYPE(w, Derived<string>);

  w<<b1;                  // w<<*(Derived<float>)*(b1)
  w<<b2;

  ifstream readstream("out.txt");
  TextStreamReader r(readstream);
  REGISTER_TYPE(r, Derived<float>);
  REGISTER_TYPE(r, Derived<string>);

  Base* br1;
  Base* br2;
  r>>br1;
  r>>br2;
//close streams if you care
  return 0;
}
```

The only new thing in this code from the previous versionis REGISTER_TYPE(). This is a macro which registers the appropriate type with the appropriate StreamReader/StreamWriter object. We have to register all possible derived types that can be inherited directly or indirectly from a polymorphic base pointer. This has to be done before both serialization and deserialization applications.Note that support for polymorphic serialization relies on RTTI which must be supported by your compiler. If not available, remove types.hpp from the PATH and compile again. Also disable if high performance is desired as RTTI and our library calls lead to additional runtime overhead.

References

For a more detailed description read our manual or the documentation(README.html) provided with the library.