C++ Introspection and Object Persistency Through JIL

D. Lawrence, D. Abbott, V. Gyurjyan, E. Jastrzembski, C. Timmer, E. Wolin ,Jefferson Lab, Newport News, VA * †

Abstract

The JLab Introspection Library (JIL) provides a level of introspection for C++ enabling object persistence with minimal user effort. Type information is extracted from an executable that has been compiled with debugging symbols. The compiler itself acts as a validator of the class definitions while enabling us to avoid implementing an alternate C++ preprocessor to generate dictionary information. The dictionary information is extracted from the executable and stored in an XML format. C++ serializer methods are then generated from the XML. The advantage of this method is that it allows object persistence to be incorporated into existing projects with minimal or, in some cases, no modification of the existing class definitions (e.g. storing only public data members). While motivated by a need to store high volume event-based data, configurable features allow for easy customization making it a powerful tool for a wide variety of projects.

INTRODUCTION

Object persistence is a feature which is often needed when writing C++ programs. A program written in C++ is itself (usually) a collection of objects in memory. In order to save the state of the program or at least part of it, one needs a way to store the critical pieces of data in a file. The desirable way for a programmer to do this in an object-oriented programming environment is to simply write the objects themselves to a file. In other words, one would like to simply pass an object pointer to the I/O system without worrying about the details of the object's structure. Unfortunately, no facility exists in the C++ programming language or the standard C++ libraries to handle this. It is left to the individual developer to define the exact format of their files and use the low-level I/O routines to store and retrieve the object data. This has led to an enormous variety of file formats for storing C++ objects, most customized to a specific project. There are a number projects, however, that have tried to address the persistence issue in a general way. The vast majority of these deal with relatively small data sets and are therefore of limited use in large volume data handling as in experimental High Energy and Nuclear Physics(HENP).

There are existing packages that are suitable for HENP applications[1]. These, however, were not written to support external file formats[2]. The authors of the JLab Introspection Library (JIL) were interested in an object persistence package capable of supporting both the legacy format used by the CODA[1] data acquisition system while simultaneously supporting other formats. The JIL package provides this as a general use software package for making objects persistent while being designed to allow the end user to customized the underlying file format, if desired. Figure 1 shows a diagram of how JIL fits in to the overal process of object serialization.

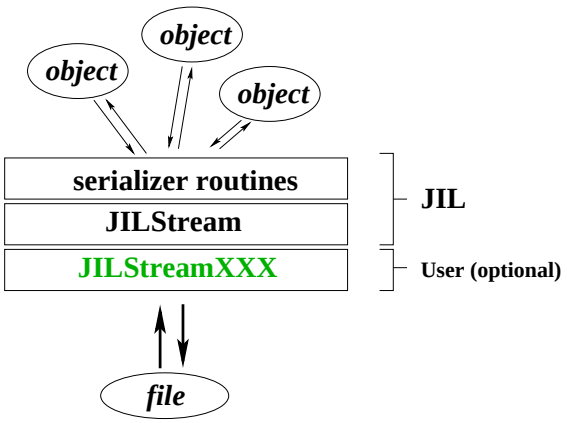


Figure 1: Schematic showing layers needed for object serialization and what parts JIL provides.

OBJECT SERIALIZATION

To store a C++ object, what one needs to do is store each of the data members of the object. Furthermore, it must be done in a way such that if the object is read back in on a computer with a different architecture, the proper byte-swapping,etc. is done to correct a change in endian-ness or word-length. In order to do this, the program must have information about the structure of the classes which it wishes to write out. The ability of a program to provide definitions of its own structures and classes is known as *introspection* or sometimes *reflection*. Here again, it would have been

* Data Acquisition Group, Jefferson Lab

[1] Reflex and ROOT[2]

[2] While it is possible to dig into the code and impose an external format in these open source projects, that was certainly NOT the original intent of the authors.

nice had the C++ language provided this information in some standard way, but alas, once again it is left to the poor programmer.

There are a few distinct approaches to providing introspection to C++ programs about their classes:

- **C++ parser/interpretor** Writing a fully functioning C++ parser/interpretor is a monumental task in and of itself. The advantage here is that a programmer gets to define C++ classes by writing C++ code. Another advantage to this is that one can essentially extend C++ by allowing users to insert comments or flags that help control specifics of how to the serialization should be done. For example, the end user could put in a flag that instructs the serializer NOT to store a certain data member with the object. The down side is that the parser has to be compatible with the compiler used to actually compile the code.

- **External file format** If the classes are defined in a well formed, easily parsed external file format (such as XML), then both the C++ class definitions and the serializers could be derived from the same source as is the case for the C++ parser/interpretor. JIL takes a similar approach, except that it derives XML from the C++ code.

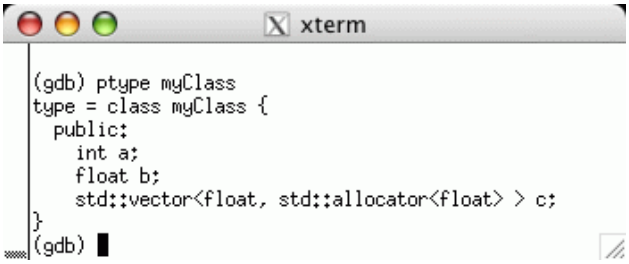
- **Hand Coded** Explicitly write code that specifies the class structure. Even though this provides the most flexibility, this is highly undesirable. The reason is that in addition to defining the class, the (de)serializer methods must contain essentially the same information. If the two pieces of code get out of sync, the result would be disastrous.

JIL uses a combination of the first two methods. It uses the GNU compiler to generate an object file with debugging symbols. It then uses the GNU debugger (gdb) to extract the class structure definitions from the object file. This is illustrated in the following example.

We start with a simple C++ class definition:

```
1
2 #include <vector>
3 using namespace std;
4
5 #ifndef _MY_CLASS_H_
6 #define _MY_CLASS_H_
7
8 class myClass{
9    public:
10
11       int a;
12       float b;
13       vector<float> c;
14
15       myClass(int a, float b, float c1, float c2):a(a),b(b){
16          c.push_back(c1);
17          c.push_back(c2);
18       }
19 };
20
21 #endif //_MY_CLASS_H_
22
```

This is compiled into an object file with debugging symbols using g++. The gdb program can then be used to extract the class definition using the *ptype* command as shown here:



The output of the *ptype* command looks very similar to the C++ source that orginally defined the class. The key difference, however, is that the output of gdb is in a predictible format. In a C++ file, the programmer can put several variables on one line, insert carriage returns as white space etc. As indicated above, writing a C++ parser is a difficult job given the large design space available in C++. The gdb output, on the other hand, is easily parsed. JIL does this by running gdb from within a perl script which then captures the output and parses it to determine the structures of the class(es) defined in the executable.

The output of the perl script is an XML representation of the class as shown here:

```
1
2 <JILDictionary>
3    <class name="myClass">
4       <typedef type="int" name="a" section="public"/>
5       <typedef type="float" name="b" section="public"/>
6       <typedef type="vector" name="c" section="public">
7          <typedef type="float"/>
8       </typedef>
9    </class>
10
11 </JILDictionary>
12
```

With the XML representation of the class(es) in hand, JIL can generate the *serializer/deserializer* methods. These are the C++ methods that convert an object, which is multi-dimensional [3] by nature, to and from a data buffer. The data buffer can be inserted into a file which is a one-dimensional entity. JIL does this using a second perl script and the ubiquitous XML::Parser module. The serializer and deserializer methods produced for the current example are shown here:

```
37
38 //----------------- myClass -----------------
39 JILBuffer& operator<<(JILBuffer &s, const myClass &c){
40
41    s << c.a;  // int   (public)
42    s << c.b;  // float   (public)
43    s << c.c;  // vector   (public)
44
45    return s;
46 }
47
48 JILBuffer& operator>>(JILBuffer &s, myClass* &c){
49    s >> c->a;  // int   (public)
50    s >> c->b;  // float   (public)
51    s >> c->c;  // vector   (public)
52
53    return s;
54 }
55
```

The conversion of atomic types(int, float, double, etc...)

[3] The data members of a class represent properties which have no particular order relative to one another not unlike orthogonal elements of a vector.

is handled inside the JILBuffer class. The methods for converting the complex classes are really global scope operators defined outside of the JILBuffer class. Thus, streaming an object to a JILBuffer results in each of its data members being streamed to the buffer. If the data member itself is a class, then that class's stream operator will be used to add it "inline" to the buffer.

In essence, the knowledge of the complex structure definitions are now contained in the serializer/deserializer methods. They have converted the problem into one of writing out and reading in a small set of fundamental data types.

THE JIL API

The part of the code that the JIL user will interact with most directly is the Application Programming Interface (API). The JIL API for writing and reading is probably best explained using the following two examples.

The code below shows a trivial example of how one would create a file and send objects to it. The API divides a file into *named sections*. A named section is a collection of objects that are associated with each other. Most commonly in HENP, a named section would be an event. One could, however, insert other types of named sections in the file with non-event based information such as slow controls readouts, scaler readouts, etc.. A named section can have any name and more than one section of the same name can be placed in the file.

In this example, the file is represented by the *JILStream* object. An object of type MyClass is streamed into the named section and the the named section is then streamed into the file. Note that one can add as many objects of differing (or the same) type to the section as needed before streaming the section to the file.

```
 1
 2 #include <iostream>
 3 using namespace std;
 4
 5 #include <JILStream.h>
 6 #include <JILNamedSection.h>
 7
 8 #include "myClass.h"
 9
10 int main(int narg, char *argv[])
11 {
12    JILStream jstream("test.jil","w");  // Open file for writing
13
14    myClass obj(3, 3.14, 10.0, 11.0);   // Create a user object
15
16    JILNamedSection jsection("Event"); // Create a new "Event" section
17
18    jsection<<obj;                     // Add user object to event
19                                       // (other objects can be added here)
20
21    jstream<<jsection;                 // Write event to the the file
22
23    return 0;
24 }
25
```

The API for reading objects in is a little different. This is partly because the program must "discover" from the file what types of named sections it has and the class and number of type of object it contains. In the code snippet below, the named section is read from the file by streaming it into the named section on line 16.

In the JIL API, the user is responsible for creating the objects. The `ReadObjects` method of the JILStream class will overwrite the data members with the values from the file. This was a design choice made over the alternative of generating the objects inside of JIL and passing the pointers back to the user. This choice was made because if JIL were to create the objects, it would need to do so by calling the "default" constructor for each object. This places a limitation on the design of classes one wishes to store. For instance, one common programming technique is to declare the default constructor of a class private in order to prevent its use, forcing users to use one of the non-default constructors of a class.

As shown in the example code below, the end user will create the objects and pass an STL vector of pointers to those objects in to the `ReadObjects` method. Note that ReadObjects is a template method so it knows what type of objects are being requested by the type of the *objects* variable.

```
 1
 2 #include <iostream>
 3 using namespace std;
 4
 5 #include <JILStream.h>
 6 #include <JILNamedSection.h>
 7
 8 #include "myClass.h"
 9
10 int main(int narg, char *argv[])
11 {
12    JILStream jstream("test.jil","r");  // Open file for reading
13
14    JILNamedSection jsection;           // Create a new  section
15
16    while(jstream>>jsection){           // Loop over events
17
18       vector<myClass*> objects;
19
20       int N = jilsection.GetNumObjects(objects);
21       for(int i=0; i<N; i++)objects.push_back(new myClass);
22
23       jilsection.ReadObjects(objects);
24
25       // ... use myClass objects in "objects" ...
26
27       for(int i=0; i<N; i++)delete objects[i];
28    }
29
30    return 0;
31 }
32
```

In a real, event-based analysis, one would likely not create the objects for every event as shown above. Rather, a pool of objects would be created once and recycled for every event, relieving the overhead incurred from repeated new/delete cycles.

PERFORMANCE

The performance of JIL, was compared to another object I/O package(ROOT[2]) known to be well optimized. Both packages have the ability to compress buffers automatically when writing/reading from a file. The tests, therefore, compared the rate at which JIL and ROOT could write and read event-like objects to and from a file in both compressed and uncompressed formats. The psuedo events consisted of randomly generated, digitized hits from a drift chamber and calorimeter such that the average event size was 5.5kB. Figures 2 and 3 show the total time taken to (de)serialize the events versus the number of events written to / read from the file. The (de)serialization times can be seen to be linear as a function of the number events in the file as one would expect. The slopes of the lines represent the reciprocal of the rates for each test. The important thing to note is that the rates are comparable to those achieved by ROOT and that in all cases, the (de)serialization times are small

compared to processing time needed to generate the data.

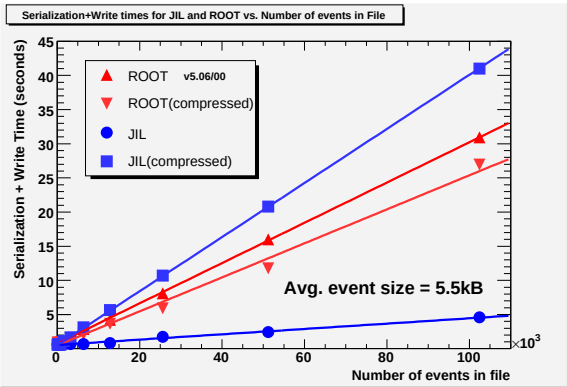


Figure 2: Total serialization time as a function of the number of events written for JIL and ROOT with compression turn on and off. Note that in these tests, the generated files were able to fit entirely in the cache so actual write times to disk are not included here.

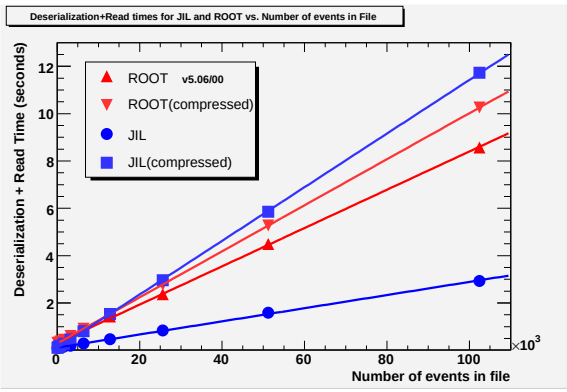


Figure 3: Total deserialization time as a function of the number of events read for JIL and ROOT with compression turn on and off.

FUTURE DEVELOPMENT

To add complete functionality to JIL, pointer tracking and schema evolution will need to be added.

Pointer tracking keeps track of the pointers of objects that have been written out and references them in the file. By keeping track of the pointers, the system is able to prevent an object from being written more than one (such as when the pointer to one object is kept as a data member in multiple objects). The object is written the first time it is encountered and references to it are written on subsequent encounters, saving both disk space and serialization time. It also enables one to "reconstruct" the web of pointer references between objects once they are read back in from the file.

Schema evolution keeps track of the class structure definitions in a file. This allows one to check that the definition of a class as compiled into the current executable is the same as what was used when the file was made. If not, attempts can be made to match up the parts that overlap giving some level of both forward and backward compatibility.

ACKNOWLEDGEMENTS

I would like to thank Rene Brun, Philippe Canal, and Stefan Roiser for some e-mails and conversations that helped give a more accurate representation of the abilities of ROOT.

REFERENCES

[1] http://www.jlab.org/CODA

[2] http://root.cern.ch