C++ and Data

Axel Naumann[*]
CERN
E-mail: axel.naumann@cern.ch

Philippe Canal
Fermilab
E-mail: pcanal@fnal.gov

High performance computing with a large code base and C++ has proved to be a good combination. But when it comes to storing data, C++ is a problematic choice: it offers no support for serialization, type definitions are amazingly complex to parse, and the dependency analysis (what does object A need to be stored?) is incredibly difficult. Nevertheless, the LHC data consists of C++ objects that are serialized with help from ROOT's reflection database and interpreter CINT. The fact that we can do it on that scale, and the performance with which we do it makes this approach unique and stirs interest even outside HEP. I will show how CINT collects and stores information about C++ types, what the current major challenges are (dictionary size!), and what CINT and ROOT have done and plan to do about it.

XII Advanced Computing and Analysis Techniques in Physics Research
November 3-7, 2008
Erice, Italy

[*]Speaker.

1. Prerequisites for Storing Data

An integral part of physics is the processing of detector signals to convert them into physics results. Physics analyses are usually statistical evaluations based on a large ensemble of occurrences of physics interactions, demanding large amounts of data. Storing the data in a persistent way is this a necessary ingredient of any physics experiment. The data to be stored is calculated within the experiments' programs. It is most efficient to allow these programs to write their data in their own format. Most of these programs are nowadays written in C++, and C++ represents data in objects. What is needed is thus serialization of C++ objects, e.g. to store petabytes of data of the LHC experiments[1].

To store data means to store all the primitive values of objects, to handle objects that are contained in other objects, to handle references between objects, and to be able to reconstruct this network of objects into memory. While much of this poses challenging issues, we will focus on the determination of the data to be stored, based on the classes' layout. The description of classes and their members is called *reflection*, any serialization framework needs to extract it from the class definitions.

Many languages provide a means to ask objects what their type is, and given that type what its constituents are. This mechanism is called *introspection*.

When reading data back from persistent storage, the objects' primitive data members must be filled into the memory allocated for these objects. This requires the classes' constructors to be accessible from the serialization framework. There is no defined byte ordering for multibyte datatypes (endianess). The data read from persistent storage might thus have to undergo a byte shuffling step before it can be mapped into memory.

2. Language Integration of Reflection

Of course all of these basic ingredients: reflection, introspection, and endianess, should be handled by the programming language. As a matter of fact, many languages do:

- Java requires classes to implement the `Serializable` interface to allow serialization; introspection and reflection are available for any Java type, see listing 1. This makes Java a very powerful language for type description. By design it hides the internal representation of objects; the data layout in memory is inaccessible, which poses problems for advanced serialization frameworks like ROOT: algorithms like data member ordered serialization, selection of I/O subsets, and alike require access to the raw memory layout.

- Python supports simple serialization with `pickle`, see listing 2. Introspection is an integral part of the language because an object's type is usually only known at runtime.

- C++ objects for classes written with Microsoft's .NET extensions ("managed C++") can be stored if the class has the `[Serializable]` attribute. This opens a powerful combination of C++ and raw I/O with built-in language and compiler support as shown in listing 3.

ANSI C++, on the other hand, does not offer any of these features. There are several libraries that try to add reflection support to C++[2],[3]; we will focus on CINT[4] and especially Reflex[6], both used within the context of the ROOT data analysis framework[7].

```
import java.lang.reflect.*

public static void main(String[] args) {
  Class kl = Class.forName("Klass");
  Field members[] = kl.getDeclaredFields();
  for (int i = 0; i < members.length; i++) {
    Field m = members[i];
    System.out.println("Type:␣" + m.getType()
      + "␣Name:␣" + m.getName());
  }
}
```

Figure 1: Introspection with Java.

```
import cPickle
class C: field = 'a␣field'
c=C()
if 'field' in c.__class__.__dict__: print c.field
...
cPickle.dump(c, file, -1)
```

Figure 2: Introspection and serialization with Python.

3. Adding Reflection to C++

For C++ to support reflection, the types used in the C++ code must be known. Because C++ does not offer reflection and introspection facilities, the type description is usually based on external libraries. They all apply a typical combination of tools to handle reflection:

- **templates** are used for the type meta description, see listing 4,
- the **typeid** operator is used to access the type information of an instance at runtime, see listing 5,
- **preprocessor macros** are employed to induce the meta description's scaffolding around a type definition as shown in listing 6.

Some of these description tools require changes in the class definitions; listing 6 is an example. This is called an *intrusive* type description; it is always desirable to force as little changes to type definitions as possible. As an example, Reflex does not require any changes to type definitions; it is non-intrusive. This is not common at all; e.g. `boost::serialization`[9] requires classes to define a function to be called for serialization, which in turn must enumerate all data members that are to be serialized.

```
using namespace System;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;

[Serializable]
public class Klass {
  public:
    Double fMember;
    Double func();
};

Klass k;
Type t = typeof(k);
Binder binder = null;
object[] args = null;
t.InvokeMember("func",
  BindingFlags.InvokeMethod,
  binder, k, args);

k.fMember=42.;
Stream out = File.OpenWrite("out.bin");
BinaryFormatter bf=new BinaryFormatter();
bf.Serialize(out, k);
out.Close();
```

Figure 3: Introspection and serialization with .NET managed C++.

3.1 Dictionary Generation

The persistent description of types and their members is called a *dictionary*. There are several options of creating dictionaries:

- **Patched Compiler:** One can use a patched compiler, that instead of compiling C++ source extracts the reflection data and passes it on to the reflection system, which in turn stores it. This approach is chosen with GCCXML[8], a patched version of GCC, and genreflex[6], Reflex's default reflection generator.

- **Custom Parser:** Instead of using a compiler's parser one can create a dedicated one, again extracting the reflection data and converting it into a persistent format. Because this custom parser is under the user's control (unlike an official compiler), it is much easier to influence what the parser makes available, e.g. whether typedefs are replaced by their underlying type or not, or whether template are instantiated or not. Many of these features are actual issues with the *Patched Compiler* option. CINT is an example of a custom parser creating its own

```
template <typename T>
struct IsPointer {
  enum { Value = 0; };
};
// partial specialization for pointers
template <typename T>
struct IsPointer<T*> {
  enum { Value = 1; };
};

template<typename TYPE>
void f() {
  if (IsPointer<TYPE>::Value)
  ...
}
```

Figure 4: The use of templates for type meta description.

```
struct Type {}; // represents types
std::map<type_info, Type> mapTypeInfo;
... // fill the map
class Base {
public:
  // polymorphic base class
  virtual ~Base();
};
Type GetType(const Base* base) {
  // determine the instance's type
  return mapTypeInfo[typeid(*base)];
}
```

Figure 5: The use of the typeid operator for runtime type extraction.

dictionaries. Its disadvantage is duplication of effort: it needs to maintain a C++ parser, even though high performance, high quality parsers are available within compiler suites.

- **Debug Symbols:** Usually, reflection data is needed for libraries of compiled code. If this library is built with debug symbols, a description of the types and members it defines is available. A tool can be written that converts this debug information to reflection data. The disadvantage of this approach is again the lack of control of the information made available, and the platform dependence of the debug symbol format (DWARF, COFF, etc).

- **Manually Maintained:** Instead of generating dictionaries, it can be seen as the developer's

```
#define CLASSDECL(NAME) \
  static const char* ClassName() {\
    return #NAME; }\
  static Type GetType() {\
    return mapTypeInfo[typeid(NAME)]; }

class Klass: public Base {
public:
  Klass();
  CLASSECL(Klass);
};

bool IsSameKlass(Klass* k, Base* base) {
  return (k->GetType() == GetType(base));
}
```

Figure 6: An example of preprocessor macros used to inject type description for introspection.

```
void dict_Klass {
  // inject into the reflection database:
  ClassBuilder("Klass").
    // constructor:
    AddFunction("Klass");
}
```

Figure 7: Simplified excerpt of reflection data stored in a Reflex dictionary.

responsibility to maintain the type description in parallel with the type itself. She would annotate the code such that a C++ meta description of the type exists, which is simply a hand-tailored version of a dictionary. This of course has the huge disadvantage of a manually maintained system: it will become out of sync with the code it describes, inconsistencies will have to be tracked down, and it creates an extra burden for the developer. `boost::serialization` is one of the examples for this approach.

4. Dictionaries

The power of a reflection and serialization library depends to a large extend on the content and format of the dictionary it uses. This comes in several aspects: the features it supports and the C++ entities it can describe, the size it uses in memory and on disk for this description, and the time it takes to generate it.

4.1 Dictionary Content

The major task of a dictionary is the description of types. Whatever its format, it must contain

```
void ctor_stub_Klass(void* ret, ...) {
  *((Klass**)ret) = new Klass();
}
Type t("Klass");
t.SetCtor(ctor_stub_Klass);
```

Figure 8: Simplified example of a constructor stub.

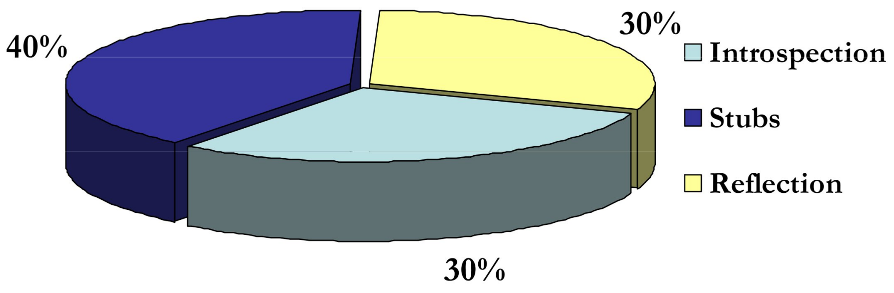

Figure 9: The parts of a typical CINT dictionary and their sizes on disk.

the name of a type it describes, its enclosing scope, the members (data, function, and nested types). Reasonable implementations also add extended C++ type attributes like base classes for classes, whether a class is abstract, the storage class of members etc. Dictionaries used for serialization must also allow to deduce the memory layout of its data members which includes the memory layout of a class's base classes within the derived class. For CINT and Reflex, this data is written as C++ code. The code fills its data into the reflection database when called; the code is paraphrased in listing 7.

As we have shown, constructors must be accessible for serialization to be able to write to a well defined and initialized memory space. This in turn means that the dictionary must allow calling access to a types constructor, given the reflection's representation of the type. In the example of listing 5 it would mean that `Type` implements e.g. a `CreateInstance()` function which calls the type's constructor. For CINT and Reflex this is implemented by a *wrapper*, also called *stub*, as shown in listing 8. These stubs are C++ code and must be compiled; they allow the transition between description and C++ code.

It is an obvious extension to implement stubs for all public functions, and thus to allow all public functions to be called through the dictionaries. This enables interpreters to transform strings like "`Klass::staticFunc()`" into a library call.

4.2 Dictionary Size

A major concern is the size of these dictionaries: they enumerate every single type that is selected as relevant for serialization or interpretation, all its members, and they implement call stubs for all available functions. Fig. 9 shows the contributions of the different parts making up a CINT dictionary.

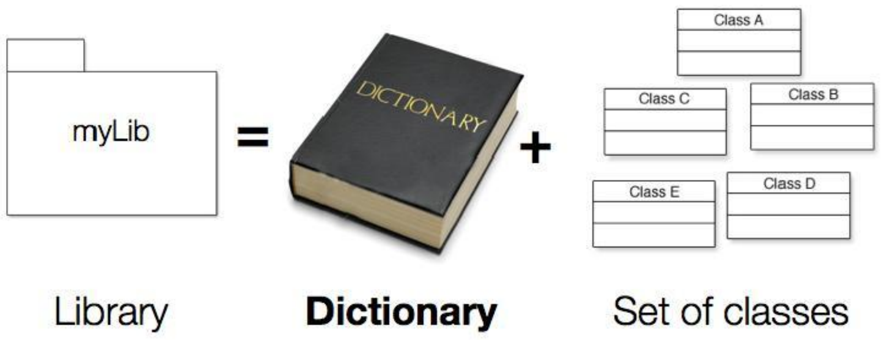

Figure 10: A dictionary enhances a library with reflection data.

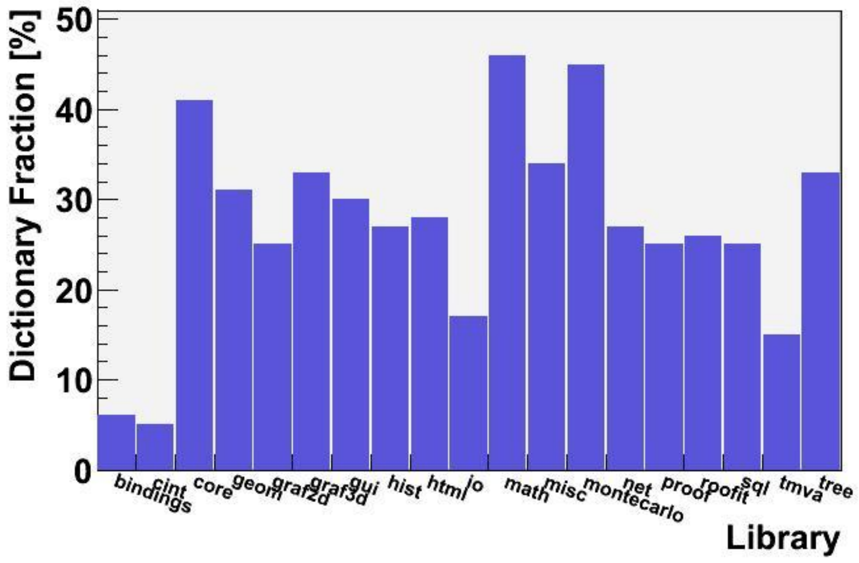

Figure 11: Relative size of dictionaries in ROOT's libraries.

The CINT and Reflex dictionaries are usually compiled and linked against the original library, to create an instrumented library as shown in fig. 10. For typical cases, dictionaries approximately double the original, non-instrumented library in size. The contribution of dictionaries in ROOT's libraries is depicted in fig. 11. For code defining a lot of templates the fraction of the dictionary is often much larger than that. This is caused by the library usually not containing any object code because the instantiation is delayed and part of the code using the templates. In these cases (common for many of the LHC experiments), the dictionaries seem to have a larger impact. It is thus more useful to compare the dictionary size to the size of the instantiated template object code.

Templates pose another problem: Each template's function is a separate symbol that needs to be instantiated when used, so it can be called or referenced. The dictionary thus needs a separate entry for every template instantiation. The needed instantiations are often difficult to predict; it is a common procedure to generate dictionaries for a wide variety of template instantiations, and for all available functions. This, too, increases the the library size due to an increased dictionary size. A possible solution is outlined in ch. 5.2.

5. Dictionary Optimizations

There are several optimizations possible to reduce the size of the dictionaries, both on disk and in memory. Some of these optimizations have already been applied, some will require more fundamental changes.

5.1 Stubs

A considerable amount of space is used by the function stubs. We have identified and implemented two improvements: re-using virtual functions' stubs, and using symbol names.

5.1.1 Virtual Functions

CINT created stubs for every reimplementation of virtual functions. Instead one can re-use a base class's stub: the object's vtable will determine which class's function reimplementation will be called. This was implemented already in ROOT version v5.20. Is reduces the dictionary for ROOT itself size by a few percent.

5.1.2 Library Symbols

Instead of using stubs to translate function names to function calls one can use the functions' symbols. ROOT has implemented this feature as an option; for now only on Linux 32 bit with GCC 3.3.

For this algorithm to succeed, a library's available symbol names must be determined, and function names must be matched to symbol names. Once the symbol name is known, an artificial call stack must be created, such that the function can be called. Each of these steps is highly platform specific (generally referred to as the ABI); it is not only different for Linux and Windows, but even for Linux ia32 and x86_64, or GCC 3.3 and GCC 4.1.

5.2 Automatic Template Instantiation

Traditionally, dictionaries were generated for any templates instantiation that was foreseen to be used. This created a considerable overhead: many instantiations were available but in fact not used by any code. When code tries to access (e.g. for serialization or interpretation) a templated class, CINT is looking this type up in its collection of dictionaries. We have extended CINT to react to a lacking dictionary for a templated class by generating it: If the member of a class `Klass<MyArg>` is accessed, and if CINT knows the header files defining `Klass` and `MyArg` (e.g. because of `#include` statements), CINT will tell ROOT to create a dictionary for it using ROOT's automatic library builder ACLiC[5].

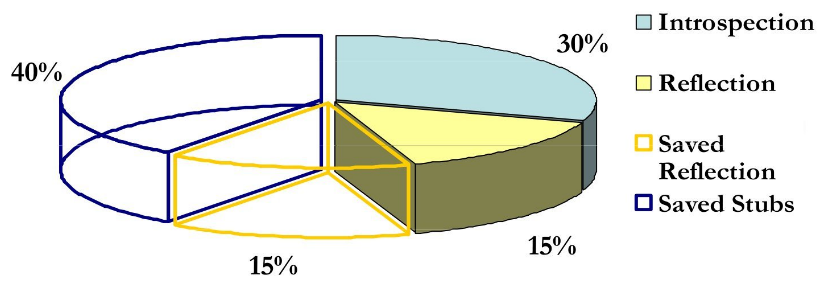

Figure 12: Size reduction of dictionaries in ROOT's libraries (prototype).

5.3 Alternative Parsers

We are planning to implement a new C++ interpreter in the context of LLVM[10], using its C++ front-end clang[5]. This front-end allows access to the reflection data collected by the parser through a C++ interface. It keeps the available information as closely to the source as possible: typedefs are not replaced, line numbers are kept, and because it is part of a parser used for a compiler front-end it even allows to request template instantiations.

It is thus an ideal candidate for a next-generation C++ parser and reflection generator.

5.4 Alternative Dictionary Formats

Using clang, the reflection data does not necessarily have to be written as a C++ file which gets compiled, linked, and dlopened. Instead one could use the compiler's precompiled header (*PCH*)feature as the persistent format. This will also reduce the build time and the memory used: the dictionary as a C++ source file contains all names as strings in the data section of the library. They get copied into the heap for use by the reflection datatabase. This means that each type and member name exists twice in memory.

If a PCH is used instead, it can be loaded as a binary file, the database can be filled, and the file then closed again: there is no second copy of strings in the data section of the library.

Even without clang and LLVM one can reduce the dictionary size drastically by writing the reflection database into a binary file. ROOT offers a powerful serialization interface which can also be used to write out the C++ objects representing the available reflection data. As with the PCHs, only those parts will occupy memory that are actually used. The file can be deleted from memory as soon as its (relevant) content has been read into memory.

Combining all of that, ROOT can achieve a considerable improvement in dictionary sizes, both on disk and in memory. A prototype has already been implemented; its results are shown in fig. 12.

6. Conclusion and Outlook

We have shown the intricacies of storing C++ objects: C++ is a difficult language for data persistence, among other reasons due to its lack of language features for reflection, introspection, and automatic object-level serialization. External libraries exist that can enable serialization even with C++.

We have presented how CINT and Reflex (and thus ROOT) gather, store, and access reflection data from header files. The size of these dictionaries is a considerable contribution to the total size of an instrumented library, and is thus subject to ongoing improvements. We have identified several gradual improvements in this area; space reductions around 50% are realistic.

We have proposed a more fundamental improvement involving a new interpreter / compiler project based on LLVM. It is expected to reduce build times, disk and memory size of the dictionary.

Given the complexity of this endeavor and the perfection by which data is handled in High Energy Physics it comes to no surprise that these reflection tools are very visible outside HEP. They are sought after by external companies and should be seen as a part of High Energy Physics' contribution to society. Taking into account the need for solid reflection systems within HEP, combined with the experience gained with data storage, and the impact we have outside of HEP shows that reflection is a key ingredient that deserves continued attention and improvements.

References

[1] M. Lamanna, *The LHC computing grid project at CERN*, NIM A **534**, Issues 1-2, Pages 1-6. Proceedings of *IXth ACAT, 2004*, doi:10.1016/j.nima.2004.07.049

[2] T. Devadithya, K. Chiu, W. Lu, *XCppRefl. C++ Reflection for High Performance Problem Solving Environments*. In *Proceedings of High Performance Computing Symposium (HPC 2007).* Norfolk, Virginia, March 25-29, 2007. See also http://www.extreme.indiana.edu/reflcpp/

[3] K. Knizhnik, *CppReflection: Reflection Package For C++*, http://www.garret.ru/cppreflection/docs/reflect.html

[4] M. Goto, *C++ Interpreter - CINT*, CQ publishing

[5] A. Naumann and P. Canal, *The Role of Interpreters in High Performance Computing*, in proceedings of *ACAT 2008, Sicily*, PoS(ACAT08)065

[6] S. Roiser. *The SEAL C++ Reflection System.* In *Computing in High Energy and Nuclear Physics (CHEP),* September 2004. See also http://root.cern.ch/drupal/reflex

[7] R. Brun and F. Rademakers, *ROOT - An Object Oriented Data Analysis Framework*, in *Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996*, Nucl. Inst. & Meth. in Phys. Res. A **389** (1997) 81-86. See also http://root.cern.ch

[8] B. King, Kitware, *GCC-XML, the XML output extension to GCC*, http://gccxml.org

[9] R. Ramey: BOOST C++ Libraries: Serialization. See also http://www.boost.org/doc/libs/1_38_0/libs/serialization/doc/index.html

[10] Ch. Lattner and V. Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*