A Case for Reflection[1]

Walter E. Brown[2], Philippe Canal[2], Mark Fischler[2], Jim Kowalkowski[2],
Pere Mato[3], Marc Paterno[2], Stefan Roiser[3], Lassi Tuura[3]
Document #: WG21/N1775 = J16/05-0035
2005-03-04

Contents

1 Introduction

1.1 Overview

The HEP (high-energy physics) community has long embraced C++ as its primary language for obtaining, saving to mass storage, restoring from mass storage, and analyzing large amounts of physics event data. Current experiments as well as experiments under development make extensive use of large bodies of code written in C++. Several *ad hoc* toolkits and libraries have been written to support the physicist user base in its efforts to process petabytes of scientific data.

We believe that many of these activities have been significantly more difficult than they might have been. We attribute much of this difficulty to the lack of a standard means to obtain information from a C++ language processor regarding a compiled piece of code, a facility historically termed *reflection*.

[1] This document arose from a three-day workshop held at Fermilab starting January 31, 2005. It summarizes the combined input of a representative sampling of knowledgeable members of the high-energy physics community.

[2] Affiliation: Fermi National Accelerator Laboratory, Batavia, Illinois, USA

[3] Affiliation: CERN (European Organization for Nuclear Research), Geneva, Switzerland

The purpose of this document is to identify and describe the features of a reflection mechanism that would satisfy the needs of our community and would support the tasks mentioned above.

After identifying the significant aspects of reflection mechanisms, this document will briefly describe many of the HEP applications (programs, components, tools, and design patterns) that would directly benefit from the availability of additional compiler-supplied information. We then present an overview of the capabilities that would make a reflection mechanism most useful for the described applications.

Finally, we present a representative survey of mature HEP-developed products which today depend on their own limited extra-linguistic implementations of C++ reflection capabilities.

1.2 Nomenclature

Broadly, a *reflection* mechanism is a means to obtain and exploit, under control of a compiled program, information that is traditionally known only to the compiler. More precisely, we define *reflection* as the aggregate of three sets of capabilities:

1. *Introspection*: Access to information that appears in the definitions of C++ entities [C++03, §3/3] such as functions, classes, and templates.
2. *Interaction*: Ability to invoke functions and to influence (and create) objects based on information acquired via introspection.
3. *Extension*: Ability to influence the behavior of a class (as opposed to an instance of a class), effectively altering the information that future introspection would deliver, without compiling additional code.

We note that conventional notions of introspection have focused on run-time availability of compile-time information. However, compile-time access to introspection information can be generally useful for tailoring code behavior via generic programming and metaprogramming techniques. The type traits facility specified in the forthcoming TR1 [Austern05, §4] is certainly a step in this direction, but is insufficient for our purposes. While it can provide compile-time answers to such queries as, "Is this data type integral?" it can't answer such questions as "Is an instance of this class usable as a niladic functor?" or "Does this class have a `debugPrint()` member?"

2 Executive summary

Prior published work on the subject of reflection includes [Adams04], [Forman99], [DosReis04], [Vollmann00], [Vollmann05], and [Vandevoorde03]. With some modifications, those or other approaches and systems can be utilized to satisfy many of the requirements we present in this paper. The hope is that C++0x will incorporate features along similar lines, thereby providing most of the reflection capabilities which current HEP applications employ but obtain *ad hoc*.

Our most basic need is the ability to represent a program's *types* and *functions* as data structures that are usable *outside* of that program[4]. External tools such as the Pivot [DosReis04] and `GCC_XML` [King] provide some of this functionality:

- A class library for the representation of a program's structure (the Pivot only),
- Tools for extracting this structural (implementation-independent) information from a body of source code, and
- A mechanism for saving this representation in an external format.

We also require implementation-dependent quantities, *e.g.*, the size, ordering, and padding of data members, in order to support our needs in creating a persistence mechanism. These are not today provided by either of these tools. Further, because these tools are inherently outside the compiler, they are more difficult to use, requiring a "two phase" build of an application (further discussed below).

Of next greatest importance is the ability to inspect and modify object data, based on the representations of the type of that object. When combined with the facility described above, the ability to inspect—and to modify—object data is important for saving and restoring object state. The ability to interact with an object through such a facility would decrease (or perhaps eliminate) our need for implementation-dependent information concerning the layout of class member data. The ability to make a persistence mechanism that is independent of C++ implementations is important to our community; it reduces the amount of domain-specific code that we must support.

We would next like to remove the need for the two-phase build[5] inherent in the use of any external tool. To do so requires that the data structures carrying the reflection information be created by the compiler, and made available (on demand) to the compiled application. Our community exerts considerable effort to support and maintain the code generation tools we currently use in the first build phase, and these tools are inadequate in that they understand only a subset of C++. Integration of such tools into the compiler would solve these problems.

Next in importance is the ability to access introspection information at compile time. The type traits facility [Austern05, §4] provides access to some type information, but it is not sufficient for such purposes as compile-time feature discovery, the ability to ask a class whether it supports a specific functionality and to adapt code accordingly. Lacking such facilities, our community, among others, currently forgoes numerous opportunities for early tuning of application code.

We would next like the ability to invoke functions based on reflection information. For example, such reflection information might constitute an interface to an external entity. This permits an application to make use of that external entity's services. Conversely, an application can publish its own capabilities for other applications to discover and exploit at run-time. (Similar interests are described in [Vollmann00].) This ability facilitates use of component programming in C++.

---

[4] Technically, of a *translation unit* rather than necessarily of an entire *program*.

[5] In the first phase of building, the reflection information is generated, and possibly new code is automatically written (and compiled) using that information. In the second phase of building, the generated code is used to build the user application.

Finally, we would like the ability to extend at run-time, with user-defined details, the information associated with an entity. Our community has termed this *annotation*, and would find this useful in providing application-specific hints to deal with dynamically-sized data in legacy types.

Should Vandevoorde's experimental *metacode* project [Vandevoorde03] prove to be feasible and achieve standardization in C++, we believe it would provide sufficient expressive power as to provide many of the above features directly, and to permit straightforward development of libraries to provide most or all of the remaining functionality.

3 How we use *reflection*

In this section, we describe categories of applications and programming techniques that employ reflection and that are important in physics (and very likely in other) contexts. We also indicate the specific reflection capabilities whose standardization would significantly improve the implementation of such applications.

3.1 Externalization

By *externalization* we mean the extraction of object state from a running program for its later reconstitution in a different context. Externalization is important for such applications as inter-process communication and long-term data storage. IPC is a prime use for *serialization*, the ability to march sequentially through each data member (including base classes) of an object and append each to a stream. Long-term data storage is the canonical use case for *persistence*, the ability to take an in-memory representation of an object and store it permanently, usually either directly in a file or in a database. Persistence may involve the use of serialization as well as more advanced mappings of objects to storage, *e.g.*, the use of several distinct storage streams.

Implementation of externalization requires the ability to obtain the static type information about an object and use it to traverse an object's public and non-public bases and members. Such per-member knowledge needs to include type, access, and offset information.

Externalization needs the ability to create objects, accommodating the possibility that an object may be not default-constructible. We also require the ability to recover on one platform objects that were externalized elsewhere. The object type and identity must thus be preserved across C++ implementations.

Support for *schema evolution* provides the ability to cope with changes, over time, in the definition of a persistent object's type. Current applications perform *ad hoc* matching, based on member names, in order to restore an object of some class.

Support for *transient data* provides the ability to reduce space requirements of persistence by storing only specified parts of an object's data. Any missing (*transient*) parts would be reconstructed, if needed, by other means. Since no persistence mechanism can *a priori* be expected to know which subobjects of a generic object may be transient,

there must be some means (often termed *annotation*) for a user program to provide such additional information. Since the persistence mechanism learns about the class via reflection, such an annotation capability should be coupled to reflection information.

Annotation can also be used to support persistence of an array whose extent is not deducible at compile-time. This facility is especially useful to provide persistence for types that are beyond a programmer's control.

We note that the same capabilities that enable externalization also make possible object browsing frameworks for visualization and data exploration. Physics analysis has historically made heavy use of such frameworks.

3.2 Algorithm tailoring

The ability to customize algorithms, *e.g.*, via template specialization, has proven a valuable feature of modern C++. Programming techniques such as *traits* and *policy classes* have been devised and have evolved in support of such customization in the context of generic code. While clearly useful, we view such techniques as workarounds, in significant part, to cope with the absence of reflection from C++.

Among the most common use cases is the desire to take advantage, in generic code, of a specific client class's features. If a class provides, say, a member `sort()`, a generic algorithm is typically better off to call it rather than `std::sort()`, for instances of that class. While numerous coding practices have been proposed to handle such a scenario, none has achieved widespread acceptance, and some have even (with some justification) been labeled "arcane."

A reflection mechanism that can report on the presence of a designated member function, coupled with other features previously articulated, can more uniformly support such use cases.[6]

3.3 Component programming

Component programming refers to the technique of discovering at run-time what capabilities are available from some system external to the current program's code, and then exploiting those capabilities. Reciprocally, the C++ application may wish to make known its own capabilities and interface so that other components can take advantage of its abilities.

Reflection can facilitate such applications. It is necessary to bind with only a single type, namely that of a reflection object, in order for an application to gain access to any other object that may be available within a running application. This allows the reflection objects' clients to use, without recompilation, existing and new services that

[6] While modern C++ can determine (via SFINAE application, for example) the presence of a specified member type, no coding technique has yet been devised to detect with well-formed code, at either compile-time or run-time, the presence of any other kind of specified member.

may become available. These features are particularly useful for applications that require human interaction when the combination of activities that may be performed is not known at compile time. In addition, a C++ application can publish its own interface in such a fashion as to permit the bridging of two computing environments.

To permit such use, the reflection machinery must be able to construct type-information objects and to access data and invoke available functions based on the contents of such objects. Further, the reflection machinery should be able to emit type-information objects describing its client application's capabilities.

Our community makes extensive use of the factory pattern [Gamma95] to instantiate an object whose type is determined at run-time. In the absence of reflection capability, this has required cooperation from the classes that define the types of objects to be thus instantiated. These candidate classes have been intrusively burdened to supply specific functionality for the factory to call, to "register" themselves before their instances can be created, or both. Support from a reflection mechanism would obviate the need for such intrusion. The primary requirement would be the ability for the mechanism to instantiate an object by invoking a class's constructor upon request.

4 Essential features and capabilities

In this chapter we provide an overview of the basic requirements for a reflection system as motivated by the HEP community's common uses described in the previous chapter. Most of these software needs are so important that the community's software specialists today implement the needed reflection capabilities themselves. Many or most of these home-grown implementations are neither very clean nor easy to use; nonetheless, their functionality is considered sufficiently important that good developers are willing to cope with anticipated maintenance headaches in order to get the needed capabilities.

We begin by articulating the basic operating principles envisioned for a reflection mechanism.

4.1 Basic principles

A reflections mechanism will be most valuable if:

1. The mere availability of reflection capabilities does not impact programs which make no use of reflection.
2. The mechanism is non-intrusive so that applications need not modify any class definition in any way in order to utilize reflection capabilities.
3. The mechanism is self-contained, requiring neither a separate pre-compilation step, nor run-time access to the compiled sources, nor the presence of the compiler at run-time.
4. The capabilities of reflection are sufficiently rich to support the important application areas we have previously identified.

We note that existing persistence mechanisms and other reflection-like tools written for HEP applications today manage to achieve most of the desired capabilities, but at the cost of being non-self-contained and sometimes intrusive (see §5 for an overview of two representative projects).

4.2 Introspection

Most of the requirements in this area pertain to the run-time acquisition of static type and type-related information regarding entities declared at namespace scope or at class scope. By *type-related* we mean to include such information as mutability, accessibility (in the case of sub-objects), and (in the case of template specializations) template parameters. By *static* we limit ourselves in this section to information that, coupled with knowledge regarding such compiler decisions as data and function placement, can be deduced by examining definitions and other declarations in the source code. For brevity, we occasionally refer to the ensemble of such extended type information as "Type information" (note the capitalization).

For entities that are composites, we will require the ability to iterate over the constituent parts and to obtain the Type information of each. Examples of such composites would include function and template argument lists (each argument is a constituent), classes (each member is a constituent), and overload sets (each function is a constituent). We would prefer to iterate separately over distinct kinds of constituents; for example, we would like to iterate over a class's data members separately from its function members.

Finally, as a special feature, introspection should make available at compile time sufficient information to tailor template code to take best advantage of a class's properties. As examples, it would be extremely useful to determine whether a class has a member function matching a given name and signature, or a data member of a given name and type, or (in general) all the attributes of any specified class member, such as the return type of a particular member function.

4.3 Interaction

The basic reflection requirements in this category involve the ability to access an object and to invoke a function, specifying the entity only at run-time. The objective of such requirements is to permit interaction, in a generic way, with object instances, without need to provide advance notice of the types to be involved in such interaction.

Interaction logically depends on introspection. As a particular example, the ability to call functions interactively would give us the ability to create objects by calling upon an appropriate constructor, after discovering via introspection what constructors of a particular type are available.

4.4 Extension

The most aggressive requirements on a reflection mechanism would be the ability to extend the type system in a dynamic fashion. In the absence of run-time invocation of compiler facilities, we would require extension only in the nature of annotation and of externally-based type information. As described earlier, such annotation can be used in conjunction with objects whose types have a dynamic component (such as dynamically-allocated or partially-used arrays).

5 Survey of related projects in the HEP community

This chapter provides a brief overview of two specific projects that members of the high-energy physics community have undertaken or contributed to in attempts to address the lack of reflection capabilities in C++98 and C++03.

5.1 CINT

Many high energy physicists work extensively with a data analysis framework called ROOT [Brun05]. To allow these users to express equations and algorithms interactively, and to link to complex analysis code modules, ROOT relies on an interpreter called CINT [Goto02]. Originally designed as a portable C interpreter, CINT was later enhanced to support much of C++. It is used in several ways:

- A "CINT script" can make use of compiled classes and functions.
- Compiled C++ code can use the CINT API to make callbacks to compiled or to interpreted functions.
- CINT also offers a gdb-like debugging environment for interpreted programs.

Each of these features requires introspection and interaction capabilities. For functions written in C and residing in shared object (or dynamically loadable) libraries, necessary information can be deduced from the compiled code in the library; the developers of CINT were able to find a utility (`dlsim`) to extract from the object code sufficient information to allow invoking the function.

However, C++ code is much richer in complexity; a C++ library's contents alone are insufficient to deduce the needed reflection information. Use of classes and their members necessitates an additional step to acquire the needed information: CINT processes the library's class definitions and function declarations in the source header files to generate C++ source files (called "rootcint dictionaries"). A dictionary can be compiled to form a module containing reflection information objects; this module can then be linked with the user program either dynamically or statically. The user code — or utilities such as the ROOT persistence mechanism — can make use of these reflection information objects.

The user can choose when to generate and compile the rootcint dictionary files. If the dictionaries are generated before static linking of the overall program, ROOT can

deliver persistence in a statically linked executable. By compiling dictionaries generated at run-time (and dynamically linking the resulting code along with the ordinary class code), ROOT can deliver interpretor-like control of C++ (compiled) classes.

In-memory reflection information objects produced by generating and compiling dictionaries comprise most of the introspection information and provide most of the interaction capabilities desired from static type information objects. ROOT uses this information to provide non-intrusive persistence, including schema evolution. CINT also provides a mechanism to call, at run-time, any of the functions described in these reflection information objects.

Implementing this system would have been simpler (and more robustly portable) if standard C++ provided the reflection capabilities outlined in this document. In the absence of such a standard reflection mechanism, CINT developers had to devise numerous workarounds:

- They had to define and implement their own reflection information class.
- They had to write and maintain a parser to read and understand C++ header files.
- They had to write a code generator to produce the rootcint dictionary files (which in turn contain definitions of reflection information objects).
- They had to devise a mechanism for compiling and linking the dictionaries.

All of these steps have caused major headaches whenever either the C++ language or the level of compliance attained by available compilers evolved. Despite heroic efforts, the CINT community has never been in a position to handle all C++ features, nor in a position to make every possible class persistent.

5.2 SEAL Reflex

SEAL Reflex [Roiser04] was developed by the HEP community in an attempt to remedy some of the deficiencies of the CINT parser and to provide full reflection information encompassing as much of the C++ ISO/ANSI standard as possible.

SEAL Reflex requires (as does CINT) generation, compilation, and linking of *dictionaries* in order to make reflection information available at run-time. Again, these in-memory reflection information objects provide introspection information (§4.2) and interaction capabilities (§4.3) meeting many of the requirements set forth above.

Because it relies on `GCC_XML` [King], SEAL Reflex guarantees that all of a user's header files can be properly parsed. `GCC_XML` is a front end to the GNU compiler, used to generate XML files representing the C++ entities. These XML files are then parsed by a code generator that produces the *dictionaries*.

Some of the desiderata guiding the design of SEAL Reflex are:

- Lightweight standalone system (no external dependencies).
- Automatic and non-intrusive generation of dictionary libraries.
- Emphasis on run-time performance and minimization of the memory footprint.
- Full C++ standard compliance.

Other than avoiding the need for a custom parser by relying on GCC_XML, the developers of SEAL Reflex faced all the same awkward implementation issues that CINT's developers encountered.

6 Conclusion

In this document, we have described the needs and representative requirements of the high-energy physics community for a general reflection mechanism to be incorporated into a future revision of the C++ language. We invite feedback from prospective users with similar needs, and encourage all members of the C++ community to contribute and discuss ideas regarding language-based and library-based solutions that meet these and related needs.

Bibliography

[Adams04] Adams, Arne. *Proposal to Include a Reflection Library into boost*. Online: http://www.arneadams.com/reflection_doku/index.html. See also relevant article in CUJ, Sept. 2004.

[Attardi] Attardi, Giuseppe and Antonio Cisternino. *Template Metaprogramming an Object Interface to Relational Tables*. Undated. Online: http://www.di.unipi.it/~attardi/Paper/Reflection01.pdf.

[Austern05] Austern, Matt. *Proposed Draft Technical Report on C++ Library Extensions*. N1745 = 05-0005. Jan. 17, 2005. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1745.pdf.

[Brun05] Brun, René, and Fons Rademakers. *The ROOT System Home Page*. July 10, 2004. Online: http://root.cern.ch.

[C++03] International Standards Organization. *Programming Languages — C++*, International Standard ISO/IEC 14882:2003(E).

[Chiba98] Chiba, Shigeru. *Open C++ Tutorial*. 1998. Online: http://www.csg.is.titech.ac.jp/~chiba/opencxx/tutorial.pdf.

[DosReis04] Dos Reis, G. and B. Stroustrup. *Representing C++ directly, completely and efficiently*. Unpublished draft, 2004.

[Forman99] Forman, Ira R. and Scott H. Danforth. *Putting Metaclasses to Work*. Addison-Wesley, 1999. ISBN 0-201-43305-2.

[Gamma95] Gamma, Erich *et al. Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 0-201-63361-2.

[Goto02] Goto, Masaharu, *et al. The CINT C/C++ Interpreter*. Jan. 8, 2002. Online: http://root.cern.ch/root/Cint.html; links from this page have been updated more recently.

[King] King, Brad. *GCC-XML*. Undated. Online: http://www.gccxml.org/HTML/Index.html.

[Roiser04] Roiser, S. and P. Mato. *The SEAL C++ Reflection System*. Computing in High Energy Physics (CHEP04), CH-Interlaken. Sept. 27–Oct. 1, 2004. Online: http://indico.cern.ch/materialDisplay.py?contribId=222&materialId=paper&confId=0.

[Vandevoorde03] Vandevoorde, David. *Reflective Metaprogramming in C++*. N1471 = 03-0054. April 18, 2003. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1471.pdf.

[Vollmann00] Vollmann, Detlef. *Metaclasses and Reflection in C++*. 2000. Online: http://www.vollmann.com/en/pubs/meta/meta/meta.html.

[Vollmann05] Vollmann, Detlef. *Aspects of Reflection in C++*. N1751 = 05-0011. Jan. 14, 2005. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1751.html.