Reflex - Reflection for C++

S. Roiser*, P. Mato†, CERN, Geneva, Switzerland
P. Canal‡, FNAL, Batavia, IL 60510, USA

Abstract

Reflection is the ability of a programming language to introspect and interact with it's own data structures at runtime without prior knowledge about them. Many recent languages (e.g. Java, Python) provide this ability inherently but it is lacking for C++. This paper describes a software package, Reflex, which provides reflection capabilities for C++. Reflex has been developed in the context of the LCG Applications Area at CERN. The package tries to comply fully to the ISO/IEC standard for C++ which was taken as the main design guideline. In addition it is light, stand alone and non intrusive towards user code. This paper focuses on the user API of the package and its underlying design, the way to generate reflection information from arbitrary C++ definitions and recent additions. Reflex has been adapted by several projects at CERN e.g. POOL, COOL. Recently Reflex started to be integrated with the ROOT data analysis framework where it will strongly collaborate with the CINT interpreter. An overview of the plans and developments in this area will be discussed. An outlook to possible further modifications e.g. IO/Persistency, Python bindings, plugin management will be given.

INTRODUCTION

Reflection is the ability of a programming language to introspect with and interact on its own data structures without prior knowledge about them. Many modern programming languages provide this feature inherently (e.g. Java, Python).

REFLECTION

The C++ programming language provides very limited capabilities for type introspection called the Runtime Type Information (RTTI). RTTI provides basically a string representation which is mangled according to the compiler implementation and a unique address per type in memory. The work presented in this paper introduces a library and tools which enhance C++ with runtime reflection functionality.

The Reflex Library

In order to provide full reflection information for C++ the Reflex library [1, 2] was developed. Reflex was designed with the following goals:

- Enhance C++ with full runtime reflection capabilities
- Non intrusive, fully automatic generation of reflection information
- Generation of reflection information from arbitrary C++ definitions
- ISO/IEC 14882 [4] standard for C++ as main design guideline
- Light and stand alone package without external dependencies
- Available on multiple platforms/compilers (linux, macos, win32, ...)
- Reflection system and the dictionaries have small memory footprints

Reflex provides three different levels of reflection.

- The **introspection** level allows retrieval and investigation of classes, scopes, members, etc. (see Table 1).
- With the **interaction** level it is possible to interact on the retrieved information. This will allow construction of types, invocation of functions or retrieval and setting of data member values (see Table 2).
- Using **modification** will allow the change of dictionary information at runtime. This can be useful to add e.g. new function members to a scope which are defined in another language which binds to the C++ reflection system.

Reflex has 8 API classes (see Fig. 1) which provide the full functionality of the package.

- A ***Type*** is an abstraction of a C++ type. In Reflex the following *Type*s are defined: *Pointer, PointerToMember, Array, Function, Fundamental, Union, Enum, Class, ClassTemplateInstance, Typedef*. Depending on its concrete representation *Type* will provide different functionality. E.g. a *Type* representing a class/struct can be instantiated or destructed, or will know about its sub types (e.g. inner classes). A *Type* representing a function can be introspected for its parameter or return types. A pointer can be dereferenced, an array can be queried for its size etc.

* stefan.roiser@cern.ch
† pere.mato@cern.ch
‡ pcanal@fnal.gov

Table 1: Example Introspection

```
using namespace ROOT::Reflex;

Type cl = Type::ByName(‘‘Particle’’);

if ( cl.IsClass() ) {
  for (MemberIterator mi = cl.DataMember_Begin(); mi != cl.DataMember_End(); ++mi) {
    std::cout << mi->Type().Name(SCOPED) << ‘‘ ‘‘ << mi->Name() << ‘‘; ‘‘;
    if (mi->PropertyListGet().HasKey(‘‘comment’’)) {
      std::cout << mi->PropertyListGet().PropertyAsString(‘‘comment’’);
    }
    std::cout << std::endl;
  }
}
```

Table 2: Example Interaction

```
Type cl = Type::ByName(‘‘Particle’’);

Object obj = cl.Construct();

Object ret = obj.Invoke(‘‘myFunction’’);

for (MemberIterator mi = cl.FunctionMember_Begin(); mi != cl.FunctionMember_End(); ++mi) {
   if ( mi->Name() == ‘‘myFunction’’ ) {
     ret = mi->Invoke(obj);
   }
}

obj.Destruct();
```

- A ***Scope*** is an abstraction of a C++ scope. In Reflex the following scopes are defined *Namespace, Class, Union, Enum, ClassTemplateInstance*. A *Scope* will provide information about its declaring scope, its sub scopes and sub types and its members. As several *Scope*s are also *Type*s an automatic conversion of *Scope* to *Type* was implemented, e.g. the functionality of the class which is a *Type* and a *Scope* at the same time can be retrieved.
- A Reflex ***Member*** contains information about a C++ member. These can be *FunctionMember, FunctionMemberTemplateInstance, DataMember*. A member will know about its declaring scope and the its type. A member representing a function member has a function *Type* associated. Data members can have any other type. In case of a function member the parameter names and default values can be introspected. A data member's value can be retrieved or set in conjunction with an object while the function member can be invoked and its return value retrieved.
- An ***Object*** is an abstraction of a generic C++ object. An object represents a concrete instantiation of a *Type*. It contains the information about its type and location in memory. For convenience some forwarding functions have been implemented with the *Object* such as invocation of member functions, get/set of data members or destruction from memory.
- The ***Base*** contains information about the inheritance of C++ classes. Information contained here is the offset between classes, the types of base classes and other qualifiers.
- A ***PropertyList*** can be attached to *Type*s, *Scope*s and *Member*s. *Propertylist*s shall contain all information that is not part to the C++ standard but still worth attaching to the reflection information (e.g. documentation). *Propertylist*s are containers of key, value pairs, where the key is a string and the value an object of any type [3].
- Two more extra classes are representations of ***TypeTemplate***s and ***MemberTemplate***s to support templates. These contain the dictionary information about template families of template instance types.

All API classes are used with a by value semantics. They are very small objects (approx. sizeof(void*) + sizeof(int)).

Reflex provides a state pattern [5] for *Scope*s and *Type*s. This will allow seamless loading and unloading of dictionary information. The API objects will always stay

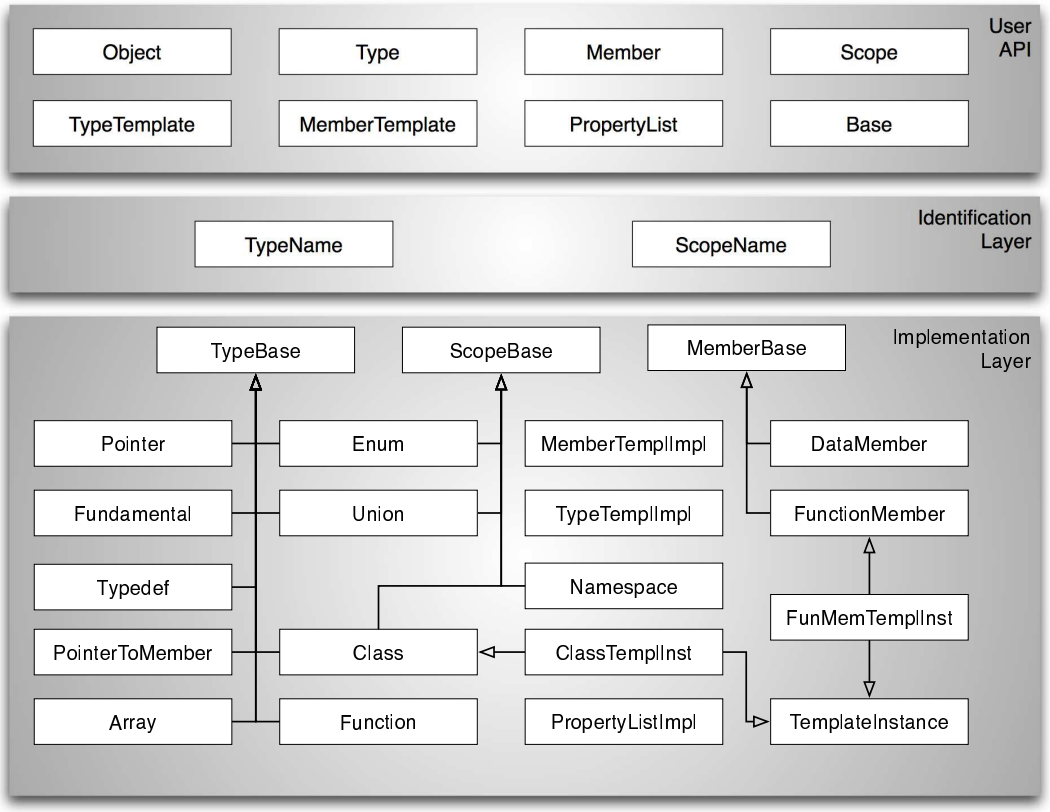

Figure 1: The Reflex UML diagram

valid pointing to an immutable object in the identification layer(see Fig. 1) while the underlying representation in the implementation layer, containing its full information, can be loaded un- or reloaded at any time.

Producing Dictionary Information

The production of reflection information is done in a non intrusive manner . The generation is based on an external tool called gccxml [6] which is a back-end to the gcc compiler. Gccxml produces an XML description of the C++ definitions that it parses which then can be further used for the production of dictionary source code. This parsing of the XML representation and production of the source code is done via a python script (These two steps are invoked via a single command “genreflex”). The dictionary sources, in C++, will then be compiled into a sharable library and subsequently loaded, filling the Reflex data structures.

The output of gccxml will be the XML representation of all C++ definitions in the input files. This output can be quite big because of other definitions included and many of these definitions are not wanted because otherwise dictionaries would be duplicated. For this reason the dictionary generation step allows the selection of wanted C++ definitions. Currently this selection is done via XML which specifies e.g. for which classes, functions etc. the dictionary source code shall be generated. Reflex also provides a simple mechanism to select C++ definitions implicitly in C++ code itself. This mechanism will be further developed in the future.

REFLEX IN ROOT

End of 2005 the Reflex library and tools have been moved from the SEAL [7, 8] project to ROOT [9]. As such Reflex is available as a ROOT module.

The ROOT project is built on top of a C++ parser and interpreter called CINT. CINT currently uses it’s own dictionary system in order to interpret C++.

Goals

With the move of Reflex into the ROOT project the goal was to merge the dictionary system of Reflex into CINT. This goal was setup and agreed in a mini workshop in May 2005 at CERN. Once this goal is achieved several advantages for ROOT will be accomplished.

- The memory allocation of dictionaries will be smaller.
- Dictionaries for all kinds of C++ definitions can be generated, as Reflex relies on the gcc compiler, through gccxml, on the source code parsing.
- LCG users will only load one Reflex dictionary into memory.

The plan to achieve this goal is divided into 3 differnt steps:

- As a first step the dictionary source code produced with CINT shall produce code which is compliant with the Reflex API. This first step will allow to load native ROOT dictionaries into Reflex and as long as step 2 is not available to load them back into CINT through Cintex at runtime.
- In a second step the internal dictionary structures of CINT will be converted to the Reflex ones.
- The last step will be the adoption of CINT users to the new Reflex API. Such users are e.g. inside ROOT the "meta" package, responsible for I/O, the PyROOT package which provides the python interface to ROOT or THtml which also uses dictionary information to produce the ROOT documentation.

Producing Dictionary Information

Usually inside ROOT dictionary source code is produced invoking a program called rootcint which will further invoke CINT to parse C++ definitions, select some via a Linkdef header file, and subsequently produces dictionary source code. Until now the dictionary source code produced was compliant to CINT and could be loaded via the CINT API into it's own data structes.

Changing the CINT dictionary system into the Reflex one, required the production of Reflex dictionary source code via rootcint. This can be done either by calling CINT or genreflex, using gccxml, for parsing the C++ definitions.

Using CINT as source code parser If CINT is wanted to be used for parsing C++ definitions rootcint shall be invoked with an option "-reflex". Further the CINT parser will parse the passed header file definitions and produces the dictionary source code.

Using gccxml as source code parser If gccxml is wanted for parsing C++ definitions, rootcint has to be invoked with an option "-gccxml". Rootcint will then internally invoke genreflex, the python script, and pass the types selected via Linkdef to it. The genreflex step will further invoke gccxml and parse the output XML definitions as it is done in a standalone version and subsequently produce dictionary source code.

Cintex

Reflex dictionaries can be converted into CINT dictioanaries using another module inside ROOT called Cintex. Cintex will automatically convert Reflex data representations into CINT ones. Once the merge of the Reflex and CINT dictioanary system is completed, this package will not be needed anymore.

CONCLUSION

The Reflex library and tools, which enhance C++ with reflection capabilities, are available as a standalone package [1] or as a ROOT module [9]. The Reflex library is deployed and used by several projects (e.g. POOL [10, 11].

Reflex will be integrated into the ROOT project and will replace the CINT reflection system. As such it will enhance the reflection capabilities of ROOT and provide better compliance with C++ and less memory consumption.

REFERENCES

[1] http://cern.ch/reflex

[2] Roiser, S. and Mato, P.; The SEAL C++ Reflection System; http://indico.cern.ch/materialDisplay.py?contribId=222&sessionId=6&materialId=paper&confId=0; oct 2004; 2004 Conference for Computing in High Energy and Nuclear Physics (CHEP)

[3] http://www.boost.org/doc/html/any.html

[4] International Standardization Organization (ISO); Programming languages – C++; Americal National Standards Institute; dec 2003; New York; 2nd ed.; Ref.No. ISO/IEC 14882:2003(E)

[5] Gamma E. and Helm R. and Johnson R. and Vlissides J.; Design Patterns, Elements of Reusable Object-Oriented Software; Addison-Wesley Publishing Company; sep 1995; Reading, MA; 4th. ed; ISBN 0-201-63361-2

[6] http://www.gccxml.org

[7] http://cern.ch/seal

[8] Chytracek, R and Generowicz, J and Lavrijsen, W. and Marino, M. and Mato, P. and Moneta, L. and Roiser, S. and Tuura, L. and Winkler, M; Status of the SEAL Project; http://seal.cern.ch/documents/seal_acat.pdf; dec. 2003; IX International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT), KEK

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

[10] http://cern.ch/pool

[11] Düllmann, Dirk et al.; The LCG POOL Project - General Overview and Project Structure; http://www.slac.stanford.edu/econf/C0303241/proc/papers/MOKT007.PDF; mar 2003; 2003 Conference for Computing in High Energy and Nuclear Physics (CHEP)