THE SEAL C++ REFLECTION SYSTEM

S. Roiser*, P. Mato†
CERN, Geneva, Switzerland

Abstract

The C++ programming language has very limited built-in capabilities for reflection information about its types at run-time. In this paper we present a new reflection system for C++, which allows complete introspection of C++ types and their use at run time. This new reflection system has been developed in the context of the CERN/LCG/SEAL project in collaboration with the ROOT project.

A detailed description of the new model is given followed by an overview of the current implementation in use by the LCG SEAL and POOL projects.

INTRODUCTION

Reflection is the ability to programmatically inspect and use types for a given system. There are two types of reflection: compile-time and run-time. Compile-time reflection facilitates generic programming by obtaining properties of a specific type or relationship between types at the time the program is compiled. An example of a compile-time reflection library is Boost type_traits [1]. Run-time reflection makes use of information about the types when the program is running without prior knowledge of it when the program was compiled. The reflection information needs to be obtained and stored in memory for all types of the system which the user may want to interact with at run-time. This is the type of reflection we are interested in this paper. Run-time reflection information can be used by applications such as object persistency, multi-language bindings, interpreters, etc.

Some programming languages include the reflection capability as part of their standard specification. Examples of such languages are Java, C# or Python. For these languages, types are typically first class objects and therefore users can manipulate them as any other object of the system. The C++ language provides, on the contrary, very limited built-in reflection capabilities. Basically it is limited to the Runtime Type Information (RTTI), which is mainly used to specify the dynamic type of objects needed by dynamic cast operations and it provides also means of obtaining the type names. Therefore, if we want to give reflection functionality to C++ programs, we need somehow to extend the language. There are several approaches to extend C++ with reflection but these approaches often have the drawbacks of either being incorporated into a larger system (e.g. CORBA [2]) or the need of instrumenting the source code before the reflection information can be created.

A native extension of C++ with reflection information was proposed by Bjarne Stroustrup, called eXtended Type Information (XTI) which would have overcome the above problems. Unfortunately there was not much development on this topic recently.

GOALS

To overcome the mentioned shortcomings of C++, a new reflection system for C++ within the SEAL [3, 4] project has been developed. The goals of this reflection system are:

- Enhancement of native C++ with full reflection capabilities.
- Production of reflection information (dictionaries) in a non intrusive way. This means that no instrumentation of code is needed to generate the reflection information.
- Automated production of dictionaries. As little as possible user interaction should be required.
- The dictionary code that populates the in memory representation of the reflection information should be human readable. This shall for example ease tasks as debugging.
- The reflection model should follow closely the latest ISO/IEC standard for C++ [5].
- Minimal external dependencies should allow a lightweight and standalone system. The library providing the reflection API should be compilable without any external dependencies.
- Both the size of the dictionaries and the memory footprint of the reflection information in memory should be as small as possible.
- It should be possible to compile the system on several different platforms/compilers. For the LCG [6] project the minimum requirements are for the time being Linux(redhat 7.3 [7], Scientific Linux [8] / gcc 3.2, gcc 3.2.3), Windows (vc71), MacOSX (gcc 3.3).

THE REFLECTION MODEL

The designed reflection model (see Figure 1) tries to emulate the underlying model described in the C++ ISO standard as close as possible. As such, it will provide information for all the different language entities described in the standard (fundamental types, classes, enums, pointers, references, namespaces, etc.)

* stefan.roiser@cern.ch
† pere.mato@cern.ch

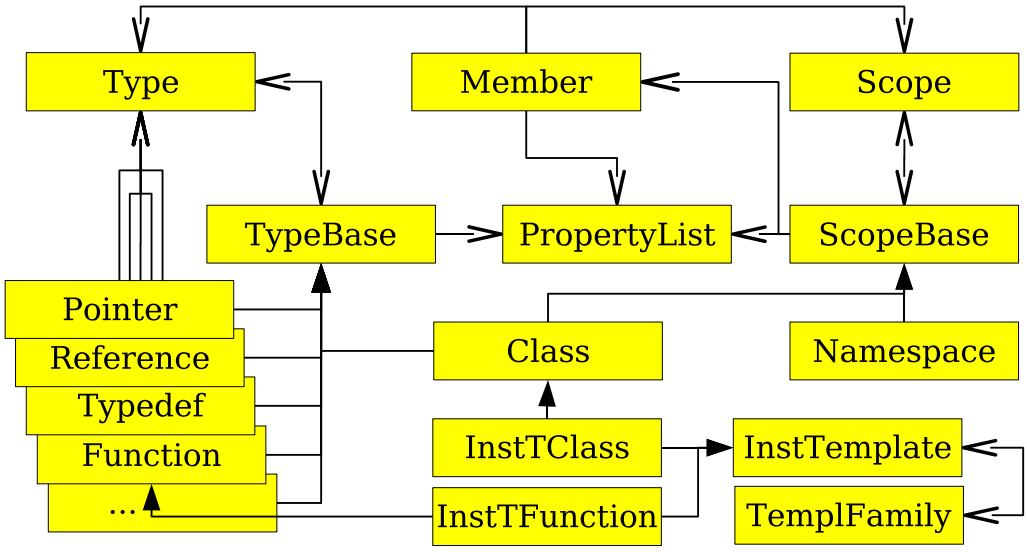

Figure 1: Simplified class diagram of the reflection model

One particularity of the current design is the distinction between the *Type* and *TypeBase* as well as between *Scope* and *ScopeBase*. This distinction implements the *state* pattern, such that *Type* and *Scope* can be created independently as needed before the concrete type is known and their address in memory will not change throughout the lifetime of a running program. This allows for an implicit forward declaration of types and scopes such that it is always safe to refer to a *Type* and to a *Scope* and the creation of the reflection information order does not play a role. Only concrete implementations of *Type* and *Scope* (deriving from *TypeBase* and *ScopeBase*) may change during the execution of the program. The *Class* type has a special position in the model as it is both a type and a scope.

As soon as a scope is defined deriving from *ScopeBase* it may also have members. In the model there is a distinction between *DataMember* and *FunctionMember* that is not shown in the Figure. A *Member* will itself live in a given *Scope* and be of a given *Type*.

Types, *Members* and *Scopes* may have *Properties* attached to them. These properties may contain information that is not part of the standard specification. Examples might be a class identifier or a description of a function or class.

Classes and *Functions* might be instantiations of templated classes or functions. In order to look up a templated type there is the *TemplateFamily* class which will allow such operations.

EXAMPLE USE CASES

A few use cases illustrate the capabilities of the reflection model and its proposed API (*Reflex* library).

To enter the model, it is possible to look up a certain type by either the string representation of the type (Listing 1, line 1-2) or its *typeid*. It will also be possible to loop over all available types. Once a type has been looked up one may retrieve information about the type such as the size of the type (Listing 1, line 3) or check whether the type is of a given implementation (Listing 1, line 4).

```
1  const Type * ty =
2          Type::byName("Particle");
3  size_t s = ty->sizeOf();
4  bool t = ty->isClass();
```

Listing 1: Types

In case the obtained *Type* is a *Class*, it is possible to convert to it (Listing 2, line 1) and examine for instance its base classes (Listing 2, line 2). The *Base* type will provide information such as the offset between the classes or whether the inheritance is virtual, public, protected, etc.

In addition to obtain information about a *Type*, it is also possible to interact with object instances. The first operation that is needed is to create and delete instances (Listing 2, line 3-4).

```
1  const Class * cl = ty->asClass();
2  const Base * base = cl->base(0);
3  Object obj = cl->construct();
4  obj->destruct();
```

Listing 2: Classes

A *Class* is also a *Scope* and as such may contain *Members*. The user may iterate over all the members (data member, function member or without distinction) of a *Class* or select a single one of them by providing an index (e.g. the first one, see Listing 3, line 1-2). To get or set the value of the data member of an object, one needs to provide the actual instance as an argument to the accessor function (List-

ing 3, line 3). It is also be possible to retrieve further information such as the offset of the data member relative to the beginning of the class (Listing 3, line 4) or its name or type.

```
1  const DataMember * dm =
2          cl->dataMember(0);
3  double d = *(double*) dm->get(obj);
4  size_t s2 = dm->offset();
```

Listing 3: Data Members

Function members can be retrieved in a similar way as data members (Listing 4, line 1-2). The characteristics of the function member such as the return type (Listing 4, line 3) or parameters, including types, names and default values can be examined. If an instance of a class is available, the function member can be invoked and the return value is returned (Listing 4, line 4).

```
1  const FunctionMember * fm =
2          cl->functionMember(0);
3  const Type * rt = fm->type()->
4          asFunction()->returnType();
5  void * ret = fm->invoke(obj);
```

Listing 4: Function Members

PRODUCING THE REFLECTION INFORMATION

The in memory reflection model of a running program needs to be filled with the information corresponding to required set of classes or types. This is typically done by executing specialized C++ code that uses the reflection model API, in particular the build part of the API, to generate and fill the information (Listing 5). The advantage of using compiled code to fill the reflection information is that the memory offsets to data members and pointers to functions (or stub functions) can be obtained in a portable manner without making any assumptions on the object memory layout. Currently we execute this specialized code implicitly while loading a number of sharable libraries, each of which is providing the reflection information for a number of classes for a given sub-system.

```
1  ClassBuilder<Particle>(Particle)
2     .addDataMember<double>(m_mass,
3        Offset(Particle,m_mass),PRIVATE)
4     .addFunctionMember<double(void)>
5        (mass,&stub_fun,0,0,PUBLIC);
6  }
```

Listing 5: Building reflection information

The reflection model filling code is generated starting from class or function declarations (header files) in non-intrusive manner using the command *lcgdict* (Figure 2). An invokation example of the command can be seen in Listing 6. This command uses internally a special front-end to the GCC compiler, called GCC_XML [9]. GCC_XML will produce an intermediate XML description of all entities contained in the header files and with a subsequent step (a python script) will process this XML representation and output a C++ file containing the code to fill the reflection information. By default dictionary information for all types contained in the input header files will be generated. If this is not wanted a *selection* XML file which contains the selection criteria for the desired classes and functions may be passed as an argument to the *lcgdict* command. The resulting C++ file will be compiled into a sharable library (called *dictionary* library).

A user may then use the reflection *Reflex* library, the unmodified user class library and the *dictionary* library in different number of ways in a user program. The libraries may be linked against the user main program or loaded at runtime.

```
lcgdict Particle.h -s sel.xml -I./include
```

Listing 6: lcgdict command

A simple example for a selection file to filter on specific types can be seen in Listing 7. The filtering can also be done on patterns of classes or on files or patterns of files.

```
1  <selection>
2    <class name=''Particle''/>
3    <class pattern=''MC*''/>
4  </selection>
```

Listing 7: selection file

STATUS AND OUTLOOK

The SEAL Reflection system consists of the following packages:

- Reflex is the new reflection library implementing the described model and API and superseeds the old libraries Reflection and ReflectionBuilder which are currently in use by POOL [10] and PyLCGDict)
- *lcgdict* command capable producing dictionary information for both the current and the new model.
- SealCLHEP, SealSTL, SealROOT are *dictionary* libraries for CLHEP, STL and ROOT [11] packages.

The SEAL Reflection system is currently used in:

- The LCG project for object persistency (POOL) uses the reflection information within its storage manager.
- PyLCGDict, a Python extension module developed within the SEAL project that provides dynamic python bindings to any C++ class for which the reflection information is available. This module will be superseded by PyReflex, which will be using the new model.

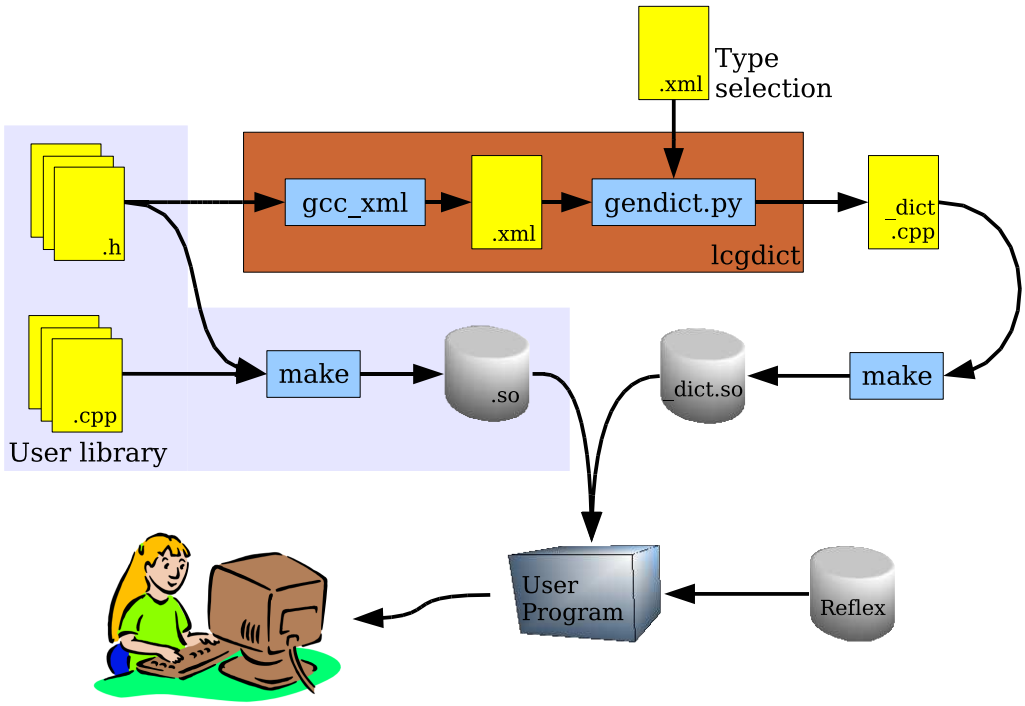

Figure 2: Producing reflection information

Future aspects of the development of the Reflex library are:

- Migration of the current meta classes in ROOT to use the Reflex API
- The current ongoing work items in Reflex are:
  - Provide means for memory allocation checking
  - Provide STL like iterators for Types, Scopes and Members
  - Complete implementation of reflection information for templated types and implementation of missing Builders (Enum, Union)

CONCLUSION

The Dictionary work package of the LCG/SEAL project provides libraries that enhance the C++ language with full reflection capabilities. The goal of this common effort is a common reflection system for both LCG and ROOT software.

The design of the libraries has been done in a way to guarantee a light and stand alone system minimizing dependencies on external software. During the design phase it was also taken care of to producing as small as possible memory footprints and library sizes.

The reflection information is generated in an automatic and non intrusive way using GCC_XML. By using the gcc_xml front-end to to the gcc compiler suite, it is guaranteed that for every program that can be compiled with gcc, also dictionary information can be generated.

The new Reflex library currently released provides reflection information already very close to the ISO/IEC standard for C++.

ACKNOWLEDGMENTS

We would like to thank the members of the LCG/SEAL team for their help and cooperation. We would also like to thank the ROOT team for their collaboration during the design phase of the Reflex library.

REFERENCES

[1] http://www.boost.org/libs/type_traits/

[2] http://www.corba.org

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

[4] Pere Mato et al. *SEAL: Common core libraries and services for LHC applications.* CHEP'04, San Diego, March 2003

[5] International Standardization Organization (ISO). *Programming languages - C++*. American National Standards Institute, New York, 1st edition, Sep 1998. Ref.No. ISO/IEC 14882:1998(E).

[6] http://lcg.web.cern.ch/LCG

[7] http://www.redhat.com

[8] http://www.scientificlinux.org

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

[10] http://lcgapp.cern.ch/project/persist

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