Reflection support by means of template metaprogramming

Giuseppe Attardi, Antonio Cisternino

Dipartimento di Informatica, corso Italia 40, I-56125 Pisa, Italy
{attardi,cisterni}@di.unipi.it

Abstract. The C++ language has only a limited runtime type information system, which doesn't provide full reflection capabilities. We present a general mechanism to support reflection, exploiting template metaprogramming techniques. Two solutions are presented: a static one where metaclass information is only available at compile time to produce class specific code; and a dynamic one where metaclass objects exist at runtime. As a case study of technique we show how to build an object interface to relational database tables. By just annotating a class definition with meta information, such as storage attributes or index properties of fields, a programmer can define objects that can be stored, fetched or searched in a database table. This approach has been used in building a high-performance, full text search engine.

1 Introduction

When building generic components, capable of handling a variety of object types, not yet known, the programmer is faced by a number of possibilities:

1. produce a library totally unaware of the specific kind of objects used by applications of the library. This is typical of C-based libraries, where parameters are passed as arrays of raw bytes (i.e. `void*`). The application programmer must supply custom code required for converting back and forth the parameters between the library and the application.
2. provide a multiple variant API, for a certain programming language, which includes one special function for each basic type, used to inform the component of each specific supplied parameter [5, 6, 8, 10]. Consider for instance database interfaces like ODBC, graphic libraries like OpenGL. The component in this case has more information available on parameters and can perform optimizations and provide higher-level services. The programmer must write sequences of invocations whenever complex parameters are involved.
3. extend the application programming language with ad-hoc primitives for interacting with the component, and supply a code generator or preprocessors which performs source to source code transformations, producing specific code for each application. For instance embedded-SQL allows inserting SQL-like statements within an ordinary program: a preprocessor translates embedded SQL into suitable database system calls.

4. develop a special purpose language for programming the component: e.g. Macromedia Lingo, PL/SQL [10], etc.
5. exploit reflection [1]. Using reflection the library can inspect the parameter types and optional traits and determine the proper way to handle them, e.g. converting them or handling them with the most appropriate specific code.

Reflection is the most advanced and powerful solution, since it does not involve external tools from the programming language, relieves the application programmer from low level interface coding, and enables a variety of domain optimization by the component developer. Unfortunately reflection is not generally available in most programming languages: most notable exceptions are Java, C#, CLOS, Smalltalk.

There are a few proposals for extending C++ with support for reflection. In [14, 15, 16] keywords are added to the language for specifying the location of meta-information. A preprocessor generates C++ source code containing the appropriate classes that reflects program's types. In [13] a similar approach is presented which avoids the use of special keywords but still uses a preprocessor.

OpenC++ instead extends the C++ compiler providing support for reflection during program compilation [17].

As pointed out in [12] the ability of handle reflective information at compile time leads to more efficient and usable reflection-oriented programs. Nevertheless the capability of accessing meta-information at runtime is fundamental for supporting dynamic binding.

We present how to implement the technique of reflection in C++ by means of template metaprogramming, which allows executing code at compile time which accesses type information without involving a preprocessor. It is supported in standard C++ and can be used with any recent C++ compiler.

While reflection is normally available and used by a program at run time, our approach provides reflection support also to metaprograms at compile time. This allows generating a specific version of the program or library, optimized to the special kind of parameters in the application.

As a case study of the use of reflection, we show how to build an object interface to a relational database table. The metaclass for objects to be stored in the table can be annotated with custom information about methods and attributes, useful for instance to express specialized traits, like indexing properties and size for a database column.

2 C++ Template Metaprogramming

C++ supports generic programming through the *template* mechanism, which allows defining parameterized classes and functions. Templates together with other C++ features constitute a Turing-complete, compile-time sublanguage of C++. C++ can be considered as a two-level language [2] since a C++ program may contain both static code, which is evaluated at compile time, and dynamic code, which is executed at runtime. Template meta-programs [2] are the part of a C++ source that is executed during compilation. Moreover a meta-program can access information about types not generally available to ordinary programs – with the exception of the limited facilities provided by the Run Time Type Identification (RTTI) [3].

Template metaprogramming exploits the computation performed by the type checker to execute code at compile time. This technique is used mostly for code selection and code generation at compile time. Its applications are mainly code configuration, especially in libraries, and code optimization [2].

Partial evaluation can be used to produce optimized code, specialized for a particular combination of the arguments, allowing, for example, the development of generic libraries that are specialized for each particular type used in an application.

In the examples below we use some useful meta-functions for testing types and perform other common metaprogramming tasks [9], for instance, `If<Condition, Then, Else>`, `IsClass<T>`, `IsPointer<T>`, `IsConst<T>`, `Equals<T, U>`, `IsA<T, U>`.

We exploited metaprogramming for providing an introspection facility for C++.

3 C++ Reflection

Throughout the paper we will use the following example of a C++ class, named `DocInfo`, which contains information about a collection of documents. The class definition is enriched with meta-information, as follows:

```
class DocInfo {
  char const*    name;
  char const*    title;
  char const*    type;
  int            date;

  META(DocInfo,
       (VARKEY(name, 2048, Field::unique),
        VARFIELD(title, 2048),
        VARFIELD(type, 32)),
        KEY(date));
};
```

`META` is a macro which exploits template metaprogramming for creating a metaclass for the class, as described later. Such annotation is the only effort required to a programmer for being able to use reflection on a certain class. The type of each attribute is in fact deduced from the class definition, using template metaprogramming. Macros `VARKEY` and `VARFIELD` allow supplying attribute traits: `2048` and `Field::unique` for instance express storage properties of the attribute in our application: in particular the maximum byte size and the kind of indexing for the column.

Using introspection capabilities we can define the template class `Table` that implements a persistent table containing objects of a specified class:

```
Table<DocInfo> table("db/table");
```

The table can be queried through a cursor created on the results of the query, which behaves essentially like an iterator on a list of objects.

Reflection involves creating a metaclass cointaining information about each class. We describe how to provide interospection and intercession capabilities both to metaprograms, through a metaclass type (*static reflection*), and to programs, through metaclass objects present at runtime (*dynamic reflection*).

We limit our discussion to how to handle reflection on attributes of classes: constructors and methods may be dealt similarly.

Static reflection

In static reflection, the metaclass of a particular class (`Foo`), is defined by a corresponding class (`MetaFoo`). The latter class stores information about the described class in static members. For instance, let us consider the following class:

```
class Foo {
  int attr1;
  char* attr2;
};
```

The associated metaclass is the following:

```
struct MetaFoo {
 typedef ctype class;
 typedef Reflection::End _curr_type0;
 static char* const name = "Foo";
 // Field attr1
 struct attr1_name { static char* const name = "attr1"; };
 typedef Reflection::Field<attr1_name, class,
  (int)&(((class*)0)->attr1), int, _curr_type0> _curr_type1;
 // Field attr2
 struct attr2_name { static char* const name = "attr2"; };
 typedef Reflection::Field<attr2_name, class,
 (int)&(((class*)0)->attr2), char*, _curr_type1> _curr_type2;
  typedef _curr_type2 attributes;
};
```

The metaclass describes the attributes and methods using a template list. An attribute is described by the following class:

```
template<class name_, class cont, int offs, class h_,
         class t_ = End,
         class traits_ = Reflection::defaultTraits>
struct Field {
  typedef cont memberof;
  typedef h_ attr_type;
  typedef t_ Tail;
  typedef traits_ traits;
  enum { size = sizeof(attr_type), offset = offs };
  static char * const name = name_::name;
};
```

A class field is described by suitable template parameters: `t_` represents the tail of the list, `End` the end of the list, `h_` the type of the attribute which is stored in a `typedef` within the class, `cont` is the class containing the field.
Without going into the details of the implementation, we just mention that the meta-class provides informations like these:

```
MetaFoo::class;                     // the type Foo
MetaFoo::name;                      // the string "Foo"
MetaFoo::attributes::type;          // type char*
MetaFoo::attributes::name;          // the string "attr2"
```

The static metaclass can be used either by meta-functions or by standard code. In the former case all computation is performed at compile time, while in the latter case types and constants defined in the metaclass can be used at runtime.

The generic template metafunction `Serialize` for instance produces code for serializing an object of a given type. This is the base case for the function:

```
template <class T>
struct SerializeBase {
  static void serialize(T *p, byte* buffer) {} };
```

`SerializeBase` is called for the `End` type terminating the recursion through the list of attributes of a class. The general case is the following:

```
template <class T>
struct Serialize {
  static void serialize(typename T::memberof *p,
                        byte* s) {
    typedef Reflection::If<
      !Reflection::Equals<typename T::Tail,
                        typename Reflection::End>::VAL,
      Serialize<typename T::Tail>,
      SerializeBase<typename T::memberof> >::VAL next;
    next::serialize(p, s);

    typename T::type *m =
        (typename T::type *)((char*)p + T::offset);
    *(T*)m = *(T*)s;
  }
};
```

Method `serialize()` generates the appropriate sequence of assignments for storing each field of the object, by recurring over the list of attributes of the metaclass. At each step, `serialize()` gets called on the type of the next attribute. This generates the serialization code for the following attributes. The current attribute gets serialized by knowing the base pointer of the object and the attribute offset. An object `foo` can be serialized to a location `dest` as follows:

```
Serialize<MetaFoo::attributes>::serialize(foo);
```

Static reflection does not handle subtype polymorphism: a metaprogram can only know the formal parameter type, not the actual type.

Dynamic reflection

Dynamic reflection uses an instance of class `MetaClass` for describing a class. Each class holds a static reference to its metaclass. The `META` construct for class `Foo` produces the following code:

```
class Foo {
  int attr1;
  char* attr2;

  typedef Foo _CLASS_;
  static Field* _createFields() {
    return &(
      createField("attr1", (int)&((_CLASS_*)0)->attr1,
        0, Field::unique, &((_CLASS_*)0)->attr1),
      createField("attr2", (int)&((_CLASS_*)0)->attr2,
        1024, Field::None, &((_CLASS_*)0)->attr2),
      ); }
  static MetaClass metaClass;
};
MetaClass Foo::_metaClass("Foo", Foo::_createFields());
```

Method `createFields()` builds the list of fields for the class. Function `create-Field()` creates an object of class `Field` with the specified attributes: the name of the attribute, its offset, the size of the field, whether the field must be indexed and a pointer to the attribute used to determine the type with template metaprogramming. The `operator,()` for class `Field` has been overloaded to create a linked list of Fields. The metaclass is initialized with its name and the list of fields. The same approach can be extended to methods and constructors.

Function `createField()` uses template metaprogramming to build the proper `Field` object and it is defined as follows:

```
template <class T>
inline Field& createField(char_t const *name, size_t
  offs, size_t maxLength,
  Field::IndexType indexType, T*) {
  MetaClass* mc = If<isClass<T>::VAL,
                     getMetaClass<T>,
                     noMetaClass>::VAL::get();
  return
    If< isPointer<T>::VAL,
        FieldBuilder<VarField<deref<T>::VAL> >,
        If< isClass<T>::VAL,
           CompositeBuilder<T>,
           FieldBuilder<FixedField<T>
           > >::VAL
        >::VAL::factory(name, offs, maxLength, indexType, mc);
}
```

Class `Field` is abstract and can be specialized for different kinds of fields, in particular: `VarField`, `FixedField` and `CompositeField`. `FixedField` is used to represent an attribute of fixed size such as a number or a pointer. `VarField` is used to represent a variable length type such as a C string. `CompositeField` is

Template classes derived from `Field` provide a method `store()` for storing the field of an object in a table row. Here is the case for `FixedField`:

```
template <class T>
byte* FixedField<T>::store(byte*& row, byte* src) {
    *(T*)row = *(T*)src;
    return row + sizeof(T);
}
```

The serialization of an object `foo` of class `Foo` is performed by the static method `serialize()` in `Foo`'s metaclass `Foo::metaClass`:

```
Foo::metaClass.serialize(row, &(byte*)foo);
```

which is defined as follows:

```
byte* MetaClass::serialize(byte*& row, byte* src) {
  for (Field* fd = columns; fd != NULL; fd = fd->next)
    row = fd->store(row, src + fd->offset);
  return row;
}
```

This method simply iterates over the list of fields and for each field calls its virtual `store()` method.

Static vs. dynamic reflection

When using dynamic reflection, having a metaclass is sufficient to manipulate objects of the corresponding class, hence it is possible to define classes dynamically assembling the field descriptions and other information. For instance, the metaclass for `Foo` can be created like this:

```
MetaClass metaFoo("Foo",
   createField("attr1", 0, 0, Field::unique, (int*)0,
    createField("attr2", 4, 1024, Field::None,
                (char**)0)));
```

Our framework provides class `AnyObject` to represent instances produced from such metaclasses, and class `DynamicTable` for using them in tables:

```
   AnyObject any(metaFoo);
   any.field<int>(0) = 5;
   any.field<char*>(1) = "value for attr2";
   DynamicTable table("/tmp/foo", metaFoo);
   table.insert(&any);
```

`DynamicTable` is just a variant of class `Table` and defines the same tables, provided the same metaclass is used. For instance an SQL interpreter needs to use the `DynamicTable` interface in order to access a table created with C++ classes, since the classes it will use are not known at compile time.

Certain methods used with dynamic reflection involve some runtime computations for interpreting the metaclass information, while with static reflection the body of such methods is expanded at compile time into code specific for the class. For example, the dynamic version of method `serialize()` iterates through the fields of the class and calls methods for storing each field. Instead the static version of `serialize()` consists of a sequence of store operations of the appropriate type for each field: there is no iteration nor invocation of virtual methods.

On the other hand dynamic reflection can use virtual methods, which cannot be dealt instead with static reflection.

Both solutions suffer for a minor drawback: namespace pollution, since they introduce classes or types (e.g. `MetaFoo`, `Foo::_CLASS`) that might conflict with names present in the user program.

4 Case study: a relational object table

An object oriented interface library to a relational table must be capable of storing objects of any class in rows of a relational table. Therefore the library must know the structure of the class of the objects in order to perform serialization when storing the object. The table schema is often extracted from the database itself, which was created separately or by means of SQL constructs like “`create table`”. For fetching or updating objects from a table, the library needs only to provide methods for accessing the individual fields of a row: the programmer must know and specify the type of each field being accessed and he is also responsible of storing values of the correct type into the fields of the object. Table schema definition and table usage are independent operations, of which the compiler is totally unaware: none of the information involved in such operations is available to it, in particular type information.

On the other hand, if the programming language used to implement the interface library supports introspection [1], the library can exploit it for determining the attributes of a class and their types. Through intercession [1] the library is then capable of modifying the object’s attributes by accessing the description of the class.

An interface library built with introspection can provide a higher-level interface to programmers, relieving them from the burden of reconstructing an object fetched form a table or supplying detailed information about the class of the object.

We present the design for such an object interface to relational tables. The interface goes beyond the ability to store flat objects, corresponding to relational rows, and allows storing composite objects, containing other objects. A full object-oriented database can be build with limited effort on top of this interface.

The relational table interface has been inspired by GigaBase [7], but exploits metaprogramming in order to produce a suitable metaclass, capable of handling for instance various kinds of attribute traits. The interface has been used in implementing IXE, a fully featured, high performance class library for building customized, full-text search engines.

Class `Table` implements a relational table stored on disk on a Berkeley Database [4]. The template parameter to this class defines the structure of the table and must provide a metaclass through the `META` construct. Various kind of indexing can be specified through attributes traits, including inverted indexes and full-text indexes.

A program can load the data into the table as follows:

```
Table<DocInfo> table(table_file);
DocInfo aDocinfo(…);
table.insert(aDocInfo);
```

A query on the table can be performed as follows:

```
Query query(query_string);
QueryCursor<DocInfo> cursor(table, query);
while (cursor.hasNext()) {
  DocInfo docInfo = cursor.get();
  // use docInfo
}
```

Differently from traditional interfaces to database systems [5, 6, 8], here the cursor returns a real object, built from database row data using reflection. The cursor is capable of accepting complex boolean queries, involving full-text searches on full-text columns and other typical SQL conditions on other columns.

IXE uses dynamic reflection, which is required for dynamic class creation, a necessary feature for building an SQL interpreter. In future versions of the library we will combine static and dynamic reflection to exploit the efficiency of static reflection.

5 Conclusions

We have presented a general technique based on template metaprogramming for supporting reflection in C++. Metaprogramming is crucial to the solution since it allows accessing type information from the compiler and inaccessible otherwise.

We have shown how to use reflection to define a generic component for storing objects in a relational table. The component can be specialized to any class of objects. Such component has been used in developing the search engine library IXE. The application programmer can insert C++ objects directly into the table, without any conversion. Search the table is done through a cursor interface that allows scanning the results returned as C++ objects. The IXE library has proven effective in building several customized search engines and its performance is superior to similar commercial products.

Future work includes combining static and dynamic reflection. Static reflection would be the preferred choice with fall-back on dynamic reflection when sufficient type information is not available.

References

1. R.G. Gabriel, D.G. Bobrow, J.L. White, *CLOS in Context – The Shape of the Design Space*. In *Object Oriented Programming – The CLOS perspective*. The MIT Press, Cambridge, MA, 1993, pp. 29-61.
2. K. Czarnecki, U.W. Eisenacker, *Generative Programming – Methods, Tools, and Applications*. Addison Wesley, Reading, MA, 2000.
3. B. Stroustrup, *The Design and Evolution of C++*. Addison Wesley, Reading, MA, 1994.
4. Sleepycat Software, *The Berkeley Database*, http://www.sleepycat.com.
5. MySQL, *MySQL*, http://www.mysql.com.
6. Microsoft, *ActiveX Data Objects*, http://msdn.microsoft.com/library/psdk/dasdk/adot9elu.htm.
7. K.A. Knizhnik, *The GigaBASE Object-Relational database system*, http://www.ispras.ru/~knizhnik.
8. Sun Microsystems, Java Database Connectivity, http://java.sun.com/.
9. Petter Urkedal, *Tools for Template Metaprogramming*, http://matfys.lth.se/~petter/src/more/metad.
10. R. Sunderraman, *Oracle8™ Programming: a primer*, Addison-Wesley, MA, 2000.
11. P. J. Plauger, A. Stepanov, M. Lee, D. Musser, *The Standard Template Library*, Prentice-Hall, 2000.
12. J. Malenfant, M. Jaques, and F.-N. Demers, *A tutorial on behavioral reflection and its implementation*. Proceedings of the Reflection 96 Conference, Gregor Kiczales, editor, pp. 1-20, San Francisco, California, USA, April 1996.
13. Tyng-Ruey Chuang and Y. S. Kuo and Chien-Min Wang, *Non-Intrusive Object Introspection in C++: Architecture and Application*. Proceedings of the 20th Int. Conference on Software Engineering, IEEE Computer Society Press, pp. 312-321, 1998
14. Peter W. Madany, Nayeem Islam, Panos Kougiouris, and Roy H. Campbell, *Reification and reflection in C++: An operating systems perspective*. Technical Report UIUCDCS-R-92-1736, Dept. of Computer Science, University of Illinois at Urbana-Champaign, March 1992.
15. Yutaka Ishikawa, Atsushi Hori, Mitsuhisa Sato, Motohiko Matsuda, J. Nolte, Hiroshi Tezuka, Hiroki Konaka, Munenori Maeda, and Kazuto Kubota, *Design and Implementation of metalevel architecture in C++ – MPC++ approach*. Proceedings of the Reflection 96 Conference, Gregor Kiczales, editor, pages 153-166, San Francisco, California, USA, April 1996.
16. B. Gowing and V. Cahill, *Meta-Object Protocols for C++: The Iguana Approach*. Proc. Reflection '96, San Francisco, California, 1996, pp. 137-152.
17. Shigeru Chiba. *A metaobject protocol for C++*. Conference Proceedings of Object-Oriented Programming Systems, Languages and Applications, pp. 285-299, ACM Press, 1995.