Portable reflection for C++ with Mirror

Author *Author@example.com*
University
Faculty

Abstract

Reflection and reflective programming can be used for a wide range of tasks such as implementation of serialization-like operations, remote procedure calls, scripting, automated GUI-generation, implementation of several software design patterns, etc. C++ as one of the most prevalent programming languages however, for various reasons, lacks a standardized reflection facility. In this paper we present Mirror - a portable library adding reflection to C++ with a command-line utility automating its usage. This library supports functional style static compile-time reflection and metaprogramming and also provides two different object-oriented run-time polymorphic layers for dynamic reflection.

Keywords: reflection, reflective programming, metaprogramming, design-pattern implementation

1. Motivation

There is a range of computer programming tasks which involve the execution of the same algorithm on a set of types defined by an application or on instances of these types, manipulating member variables calling of free functions or class member function in an uniform manner, conversion of data between the programming language's intrinsic representation and an external formats for the purpose of implementing the following:

- serialization or storing of persistent data in a custom binary format or in XML, JSON, XDR, etc. and deserialization or (re-)construction of class instances from external data representation formats (like those listed above), from the data stored in a RDBS, from the data entered by a user through a user interface,
- automatic generation of a relational schema from the application object model and object-relational mapping (ORM),
- scripting and remote procedure calls (RPC) / remote method invocation (RMI),
- object inspection and manipulation via a GUI and object access via web services (WS),
- visualization of application structure, data and the relations in the data,
- automatic or semi-automatic implementation of certain software design patterns,
- documentation or conceptual representation of a software system.

Several different aproaches to the implementation of such application features were proposed in the past. The most obvious and also usually the most error-prone is manual implementation.

Most of the operations listed above are inherently repetitive and basically require to process programming language constructs like types (either atomic or elaborate), containers, functions, constructors, class inheritance, class member variables, enumerated values, etc.) in a uniform way that could be easily algoritmized.

While it is acceptable (even if not very advantageous) for a design pattern [10] implementation to be made by a human, writing RPC/RMI related code is a task much better suited for a computer. Other tasks like creation of documentation or knowledge representation of a software system, require some human work, but can be partially automated.

This leads to the second, commonly used approach: preprocessing and parsing of the program source text by a (usually very specific) external program (documentation generation tool, interface definition language compiler, web service interface generator, a rapid application development environment with a form designer, etc.) resulting in additional program source code, which is then compiled and integrated into the final application binary.

While acceptable in some situations, this approach has also several problems: First it requres the external tools which may not fit well into the used build system or may not be portable between platforms, etc.; second, such tools are task-specific and many of them allow only a limited, if any, customization of the output.

Another approach to automation of these tasks is to use reflection, reflective programming, metaprogramming [1] and generic programming [2], as shown for example in [12], for the implementation of the *factory* design pattern (although the method described therein is somewhat limited because it requires that the constructed types have a default constructor).

The same approach is taken by several other reflection facilities, with the same limitations. Other examples of design pattern implementation with the help of reflection can be found in [17, 18, 34]. Reflection can also be employed to perform other tasks listed above as shown by projects like [21, 29, 30, 33] and others.

2. Introduction to reflection and related work

The term *reflection* describes the ability of a computer program to observe and possibly alter its own structure and/or its behavior. This includes building new or altering the existing data structures, doing changes to algorithms or changing the way the program code is interpreted [11]. Reflective programming is a particular kind of *metaprogramming*.

One of the advantages of reflection is that everything is implemented in a single programming language, and the human-written code can be closely tied with the customizable reflection-based code which is automatically generated by compiler metaprograms, based on the metadata provided by reflection.

Support for reflective programming is most common in high-level languages, often using a virtual machine, an interpreter or another such run-time environment, like JAVA, C, CLOS or Smalltalk [4, 5, 8] and less common or limited in lower-level statically typed languages like Objective C, Lisp or Scheme [11].

Today, C++ as one of the most popular [26, 28] multi-paradigm programming languages, lacks a direct support for reflection. One of the attempts for standardized reflection in C++ was made in [16], but the work seems to be on hold recently [15]. Another recent attempt to add "rich pointers" and reflection to C++ [3] looks more promising, but does not address several important aspects like strongly typed enumerations, template parameters, traversal of namespace members, etc.

There are several custom-built reflection systems, with varying degree of introspective and reflective capabilities, using various approaches.

The OpenC++ [27] is a programming toolkit that allows writing meta-level source-to-source transformation programs according to a meta-object-protocol (MOP) [5, 17] at the program pre-

processing/parse-time. The meta-level program is compiled by C++ compiler linked with an OpenC++ add-on. Unfortunately only some compilers are supported which makes the approach non-portable. Publications [19, 20] introduce another similar source-to-source translator named FOG, adding support for aspect-oriented programming and compile-time reflection to C++. More common are run-time reflection systems like, the CERN's Seal C++ reflection system [15], or those described in [8, 9, 13, 14, 22, 31]. There are also several static reflection facilities with some run-time features, like the one described in [32].

3. Mirror reflection utilities

3.1 Introduction and history

Mirror is a set of non-intrusive and portable compile-time and run-time introspection and reflection utilities for the C++ language, developed by the author. It provides reusable metadata reflecting C++ constructs like namespaces, types, classes, base class inheritance, member variables, class templates and their template parameters, free functions, etc., and implements an extensive set of metaprogramming tools for the traversal and usage of these metadata in compile-time functional and run-time object-oriented algorithms. Furthermore it provides several high-level tools, like the factory generator utility, described in greater detail in [7]. It adheres to the three principles attributed to well-designed reflective facilities [4]: *encapsulation*, *stratification* and *ontological correspondence*.

There are two versions of the library, both available for download under the Boost Software License (accessible at the following URL: `http://www.boost.org/LICENSE_1_0.txt`).

The first version [23], implemented in C++03, which is portable across all platforms having a conforming C++ compiler. One of the basic usages of this library - object serialization and marshalling in parallel applications using the MPI [25], is described in [6]. However, the development of this version has been discontinued and its use is deprecated.

The second version [24], which is currently in active development, is using multiple new features from the recent ISO C++ standard (C++2011). This version removes several design deficiencies discovered during the usage of the previous version and in addition contains also a run-time layer, built on top of the compile-time metaobjects, providing a polymorphic object-oriented interface for their usage in run-time reflective algorithms. Recently there has also been added a third - object-oriented compile-time layer and a type-erasure utility. This quaternary interface makes the libraries applicable in a wide range of scenarios. It is available for download from SourceForge.net at the following URL: `http://sourceforge.net/projects/mirror-lib/files/`.

3.2 Architecture

Mirror uses a hierarchic architecture where each layer provides services to the layers above as shown on Figure 1.

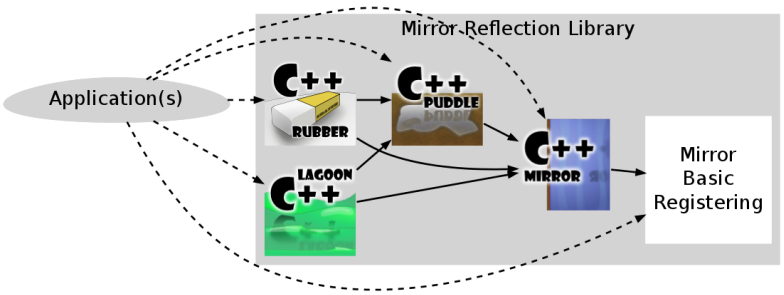

Figure 1: Architecture of the Mirror library

- *Registering and basic metadata*: Since standard C++ provides only a very limited set of metainformation to build upon, the basic programming language constructs like namespaces, types, classes, variables, functions, etc. need to be registered before they can be reflected. However Mirror tries to make the process of registering simple by providing a set of user-friendly registering macros and has the native and many of the other common types, classes, templates and namespaces pre-registered. This is the lowest layer and is used by the application only to register its components. This registering could be in future replaced by static meta-data provided by the compiler similar to the now standardized `type_traits`.
- *Functional compile-time layer* - **Mirror**: Built on the basic registered metadata. Suitable for generic metaprogramming similar for example to the standard `type_traits` library. Allows to write compile-time metaprograms which allow the compiler to generate efficient program code based on the metadata provided by reflection.
- *Object-oriented compile-time layer* - **Puddle**: Based on the functional layer, it provides an object-oriented interface which is more suitable for certain programming tasks and also provides some run-time features, but is still inherently static, allowing for extensive optimization by the compiler.
- *Object-oriented type-erasure utility* - **Rubber**: Based on the object-oriented compile-time layer; its purpose is to remove the exact types of the instances of the metaobject concepts provided by the lower layers and merge them into a single quasi-polymorphic type for each compile-time metaobject concept. This allows the type-erased metaobjects to be stored in standard containers (like vectors, maps, sets, etc.) and used in non-template functions (for example C++ lambda functions). Unlike the next layer, it does not employ virtual functions to achieve polymorphism. This utility is suited for situations where a combination of compile-time and run-time reflection is required.
- *Object-oriented run-time layer* - **Lagoon**: Based on top of the compile-time layer, it provides a run-time polymorphic interface, more suitable for run-time reflective programming, allowing the use of the provided metadata in a dynamic manner dependent on other data available only at run-time. One of its disadvantages is the performance penalty induced by virtual function calls and the inability of the compiler to inline such calls in many cases together with long compilation times. In the future this layer will allow to compile the metainformation into shared dynamic libraries separate from the applications and load them on-demand.

3.3 Goals and Features

The libraries are developed with the following goals in mind:

- *Reusability*: The metadata provided by Mirror is reusable in many situations and for many different purposes.
- *Flexibility*: Mirror and the additional layers built on top of it allow to access the provided metadata both at compile-time and run-time in a functional and object-oriented manner depending on the application needs.
- *Encapsulation*: Mirror and the additional layers provide interfaces for easy access to program metadata.
- *Stratification*: Mirror is non-intrusive and separates the meta-level from the base-level constructs it reflects. Things that are not needed are generally not compiled-into the final application.

- *Ontological correspondence*: The meta-level facilities correspond to the ontology of the base-level C++ language constructs which they reflect.

- *Completeness*: Mirror tries to provide as much useful metadata as possible, including various specifiers, iteration of namespace members and much more.

- *Ease of use*: Although Mirror allows to do very complicated reflective metaprogramming, simple things are kept simple.

- *Cooperation with other libraries*: Mirror can be used with the introspection facilities provided by the standard library and other libraries.

3.4 Metaprogramming utilities

The metaprogramming tools implemented by Mirror allow writing complex algorithms, using metadata describing the program, executed by the compiler at compilation time resulting in compile-time constants, types or program code chunks in intermediate form which are afterwards incorporated into the handwritten program code. The following program code snippet shows one possible usage: processing all members of the global scope filtering out only types which have the string 'long' in their name, resulting in a series of calls to a lambda function printing the names of the types to the standard output.

```
using namespace mirror;
std::cout << "Types having 'long' in their name:" << std::endl;
mp::for_each<
    mp::only_if<
        members<MIRRORED_GLOBAL_SCOPE()>,
        mp::and_<
            mp::is_a<
                mp::arg<1>,
                meta_type_tag
            >,
            cts::contains<
                static_name<mp::arg<1> >,
                cts::string<'l','o','n','g'>
            >
        >
    >
>(
    [](const rubber::meta_named_object& type)
    {
        std::cout << type.base_name() << std::endl;
    }
);
```

This example is rather synthetic, but there are also real-life scenarios where this feature could be used. If a consistent naming policy is defined for a large applications (like the names of all interfaces end with `Intf` or start with `I`, i.e. `ComponentIntf` or `IComponent`) special set of classes, functions, etc. can be selected and processed by a meta-program.

3.5 High-level utilities

As briefly mentioned above, Mirror, besides the metaobjects reflecting various language constructs, also provides several high-level utilities built on top of the basic metadata.

3.5.1 Factory generator

The factory generator utility allows to automate the implementation of the *factory* design pattern for types which are reflectable by the Mirror library. By factory we mean here a class, which can create instances of a *product* type, but does not require that the caller chooses the manner of the construction nor supplies the required parameters directly in the native data representation of C++.

Such generated factory examines the input which can be provided in an external data representation like XML, JSON, XDR, a simple scripting language, a dataset which results from a query to a RDBS, a user interface, etc., selects the constructor that is the best match for the available input data, converts the data from the external representation and calls the constructor. It may even construct some of the arguments recursively by the means of other, nested factories.

The advantage of this reflection-based approach is, that it separates (from the programmer's point of view) and then automatically combines the parts specific to the construction of a particular type from the general logic of the constructor selection, input data validation and conversion. This way any reflectable type can be constructed from any input data format for which the conversion logic is implemented.

The process how factories are generated from the metadata provided by Mirror and a set of user-specified templates is described in greater detail in [7],[24]. Example of a GUI created by factory which was generated by the Mirror's factory generator for a simple `tetrahedron` class with the following definition,

```
struct vector
 {
     double x,y,z;

     vector(double _x, double _y, double _z);
     vector(double _w);
     vector(void);

     // ... + other declarations
 };

 struct triangle
 {
     vector a, b, c;

     triangle(const vector& _a, const vector& _b, const vector& _c);
     triangle(void);

     // ... + other declarations
 };

 struct tetrahedron
 {
    triangle base;
```

```
    vector apex;

    tetrahedron(const triangle& _base, const vector& _apex);
    tetrahedron(
        const vector& a,
        const vector& b,
        const vector& c,
        const vector& d
    );

     // ... + other declarations
};
```

is shown on Figure 2. The automatically generated factory class creates a reusable GUI dialog window which allows the user to select any combination of constructors of the classes declared in the code shown above, and provides means to input the values necessary to construct an instance of the `tetrahedron` class.

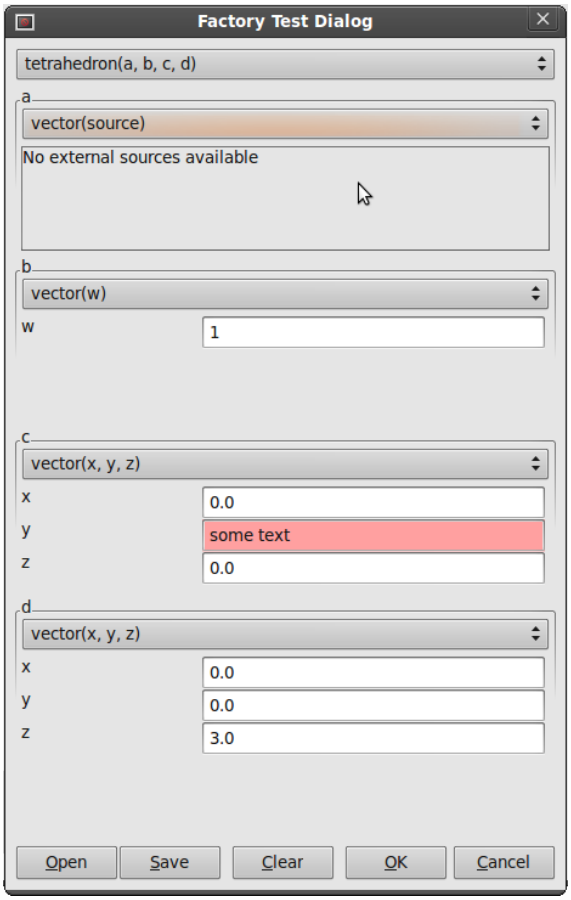

Figure 2: Example of a localized GUI created by an automatically generated factory.

It contains widgets for the input of the constructor parameters with atomic types (in this case `double`) with validators rejecting input of any text not convertible to a floating-point value and checking if all required data was provided. Upon clicking the OK button, the factory creates a new instance. This process can be repeated multiple times without the need to recreate the input dialog.

When the factory no longer required it takes care of properly freeing the resources associated with the dialog window.

Figure 3 shows another two dialogs created by an automatically generated factory, for a `person` class. In this example the support for localization is enabled (with the `en_US` and `sk_SK` locales) which results in more user-friendly dialogs by translating the basic C++ identifier names on the label widgets to a more human-readable form and adapting the input widgets to regional format.

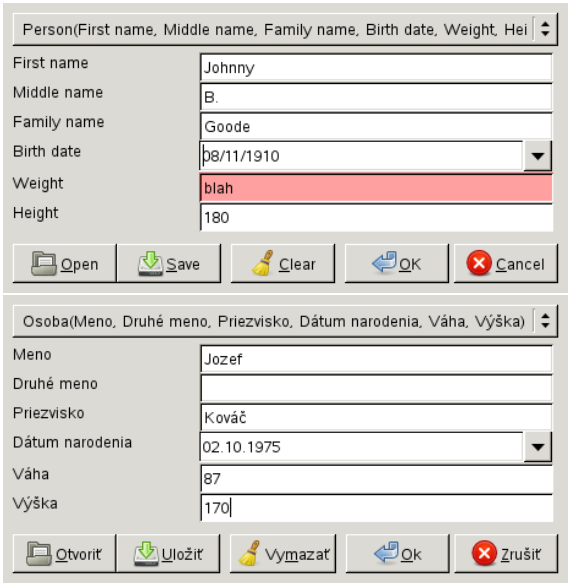

Figure 3: Example of a localized GUI created by a factory generated by the Mirror's factory generator.

3.5.2 Invoker generator

The invoker generator allows to create invoker classes which are able to call arbitrary reflectable functions, in the same manner as the generated factories call constructors. This feature is however still experimental.

3.6 External tools

The manual registering process can be tedious and error-prone in some situations. Although the registering macros use auto-detection and many things do not have to be specified explicitly, some changes in the base-level classes, like adding or removing of a member variable, constructor or a whole class, etc. require changes in the registering code.

If no special tweaking in the reflection of the base-level constructs (like hiding certain members or constructors of a class or specifying of getter/setter functions for a member variable) is required, then support for automatic reflection is desirable.

To provide this support is the aim of the *MAuReEn* (Mirror Auto-Reflection Engine) project also developed by the author, available at `http://gitorious.org/maureen`.

This tool parses the header files containing declarations of the base-level constructs to be registered (namespaces, types, classes, variables, etc.) and outputs the required registering code into header files as specifed by the user. It can be easily integrated into existing build systems like GNU Make, CMake, Boost.Build and others. It has a modular architecture which allows to

implement various methods of input file parsing and output file generation. The automatically generated registering code can be combined with hand-written registering code.

4. Conclusion and future work

As shown above, reflection can be a valuable programming tool. Even if excellent for compile-time metaprogramming, the C++ language misses a standard reflection facility. The aim of the Mirror library is to address this shortcoming and to provide portable reflection to C++. Mirror is still in development and there are several features that are planned for future releases, including but not limited to the following:

- Refactoring of the existing facilities for registering and reflection of free and class member functions with proper support for function overloads.
- An abstract factory generator that would combine the concrete factories generated by the existing factory generator utility into a single abstract factory.
- An object manipulator generator that would allow generating manipulators working on existing instances of various types. Such manipulators could perform tasks like GUI object inspection, serialization, etc. in a similar manner as the generated factories are used for object instantiation.
- An additional semantic-layer that would allow to tie conceptual data to the reflected program components, allowing the visualization of application's structure and data, generation of more user-friendly GUI/Web interfaces, access for agent oriented systems and automated reasoning.

References

[1] Abrahams, D; Gurtovoy, A. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004

[2] Alexandrescu, A. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2001.

[3] Berris, M. D.; Austern, M.; Crowl, L.: Rich Pointers. *ISO/IEC JTC1 SC22 WG21 N3340=120030*, 2012. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3340.pdf`

[4] Bracha, G; Ungar, D. Mirrors: Design Principles for Meta-level Facilities of Object-Oriented Programming Languages. *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 85 104, 2003.

[5] Chiba, S. A Metaobject Protocol for C++. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1995.

[6] Author. Support for object-oriented parallel programs for grids and clusters. *Proceedings of 2nd International Workshop on Grid Computing for Complex Problems*, Bratislava, Slovakia, pages 88-96, 2006.

[7] Author. Generating Object Factory Classes with the Mirror Reflection Library. *Journal of Information, Control and Management System*, Vol. 8, No. 2, ISSN 1336-1716, 2010.

[8] Chuang, T-R; Kuo, Y.S; WANG, C-M. Non-intrusive object introspection in C++. *Software-Practice and Experience*, issue 32, pages 191-207, 2002.

[9] Devadithya, T; Chiu, K; Lu, W. C++ Reflection for High Performance Problem Solving Environments. *In Proceedings of the 2007 spring simulation multiconference*, Volume 2, Norfolk, Virginia, USA, pages: 435-440, 2007.

[10] Gamma, E; Helm, R; Johnson, R; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Proffesional, ISBN 0-201-63361-2, 1995.

[11] Kirby, G.N.C. *Reflection and Hyper-Programming in Persistent Programming Systems*, Ph.D. Thesis, University of St Andrews, 1992.

[12] Kovacs, R. *Creating Dynamic Factories in .NET Using Reflection*. `http://msdn.microsoft.com/en-us/magazine/cc164170.aspx`.

[13] Madany, P. W; Islam, N; Kougiouris, P; Campbell, R. H. Reification and Reflection in C++: An Operating Systems Perspective. 1992.

[14] Madina, D; Standish, R. K. A system for reflection in C++. *Proceedings AUUG 2004: Always on and Everywhere*, 2004.

[15] Roiser, S; Mato, P. The Seal C++ Reflection system. CERN, Geneva, Switzerland.

[16] Stroustrup, B. XTI An Extended Type Information Library. `http://lcgapp.cern.ch/project/architecture/XTI_accu.pdf`.

[17] Tanter, E; Noyé, J; Caromel, D; Cointe, P. Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. *Proceedings OOPSLA'03*, Anaheim, California, USA, 2003.

[18] Tatsubori, M; Chiba, S. Programming Support of Design Patterns with Compile-time Reflection. *Proceedings OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, 1998.

[19] Willink, E. D; Muchnick, V. B. Weaving a Way Past the C++ One Definition Rule. *Proceedings of European Conference on Object Oriented Programming*. Lisbon, June 14, 1999.

[20] Willink, E. D. Preprocessing C++: Meta-Class Aspects. *Proceedings of the Eastern European Conference on the Technology of Object Oriented Languages and Systems*, TOOLS EE 99, Blagoevgrad, Bulgaria, June 1999.

[21] A C++ reflection-based data dictionary, `http://sourceforge.net/projects/crd/`.

[22] Metaclasses and Reflection in C++, `http://www.vollmann.ch/pubs/meta/meta/meta.html`.

[23] Mirror C++ reflection library documentation (C++98 version), `http://svn.boost.org/svn/boost/sandbox/mirror/doc/html/mirror.html`.

[24] Mirror C++ reflection library documentation (C++11 version), `http://kifri.fri.uniza.sk/~author/mirror-lib/html/`.

[25] MPI: *A Message-Passing Interface Standard*. MPI Forum, 2003: `http://www.mpi-forum.org/docs/mpi1-report.pdf`.

[26] Ohloh.net: compare languages tool. http://www.ohloh.net/languages/compare.

[27] OpenC++, http://opencxx.sourceforge.net.

[28] Programming Language Popularity, http://langpop.com/.

[29] Property Set Library (PSL), http://sourceforge.net/projects/psl/.

[30] QxOrm library, http://sourceforge.net/projects/qxorm/.

[31] Reflection for C++, http://www.garret.ru/cppreflection/docs/reflect.html.

[32] Static reflection in C++ using minimal repetition, http://www.enchantedage.com/cpp-reflection.

[33] Template Reflection Library, http://sourceforge.net/projects/trl/.

[34] The Visitor Pattern, http://www.oodesign.com/visitor-pattern.html.