STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIX**, Number 2, 2014

C++ COMPILE-TIME REFLECTION AND MOCK OBJECTS

GÁBOR MÁRTON AND ZOLTÁN PORKOLÁB

Abstract. Reflection is an important tool for serializing objects, creating mock objects for testing, creating object relational mappings and many other cases. Without standardized C++ compile-time reflection these tasks are repetitive and error-prone. Therefore the ISO C++ started a study group (SG7) to examine the possibilities of compile-time reflection in C++. With compile-time reflection it would be possible to have a generic library for serialization or for object relational mappings.

There are several potential notions about how to approach this kind of reflection, like introducing high-level new language elements like `static for`, or creating library interfaces which are hiding compiler intrinsics for each specific reflection subtask.

In this paper an alternative C++ compile-time reflection approach is discussed in favor of finding a generic solution for this task. The approach is based on introducing new library elements. Under the hood these library element implementations have to be compiler specific intrinsics (compiler specific expressions). With these expressions, variables and functions could be declared and defined from results of reflection queries.

1. Introduction

Reflection is the ability of a program to inspect or modify its own structure. In other words, reflection is referred as the meta information associated with programming structures like types and functions. For example, in case of a class type this meta information can provide the names and types of the class' fields. It is said that a code is doing introspection, if it is observing its own state and structure. Also, when a code is capable of modifying its structure or state it is called intercession.

There are several uses of reflection. For instance it is used for serializing objects, implementing language bindings, creating object relational mappings (ORM) and implementing unit test frameworks with mock objects.

Received by the editors: August 24, 2014.

2010 *Mathematics Subject Classification.* 68N15.

1998 *CR Categories and Descriptors.* D.3.3 [**Programming Languages**]: Languages Constructs and Features – *Classes and objects*.

Key words and phrases. C++, Reflection, Unit test.

Compared to other mainstream programming languages, C++ is lagging behind in reflection. In this paper we analyze and summarize current C++ reflection capabilities and researches about compile-time reflection. Based on our analysis we introduce a new approach of compile-time reflection. This approach could be used to implement generic proxy objects and mock objects for unit test frameworks.

This article is organized as follows: in Section 2 we describe reflection fundamentals, summarize other languages reflection capabilities, display current C++ capabilities and describe how generic problems like serialization is solved without a mature and native C++ reflection. Later in Section 3 we display the current researches of C++ compile-time reflection, where the important proposals are summarized. We present and analyze our new reflection approach for proxy and mock objects in Section 4. Future work is discussed in Section 5. Then our paper concludes in Section 6.

2. Reflection Fundamentals

2.1. **Compile-time and Runtime Reflection.** Compile-time reflection is about getting information which is internal to the compiler during the compilation process. Based on this information, the compiler's internal abstract syntax tree (AST) can be modified. Usually this modification is not more than adding new nodes to the AST. This can be done either by normal language elements (i.e. by adding a new function) or by using some compiler intrinsics.

Runtime reflection is happening during the program's execution time. Usually runtime reflection is implemented with the use of runtime meta objects. This means, there is a meta object associated to each real object. During runtime these metaobjects provide all the information and methods which are needed to achieve the reflection. These metaobjects are always part of the final executable, therefore making its size bigger. Runtime reflection works also with objects whose exact type is not known during compile time (i.e. dynamic polymorphic types).

Runtime reflection has a few drawbacks compared to the compile-time reflection: the executable's size will be bigger even if not all runtime objects are reflected, and our program will perform slower in runtime. There are languages where this is affordable, but in C++ the performance is a critical viewpoint, therefore runtime reflection is not a real option for C++. On the other hand, compile-time reflection is not working with objects of dynamic polymorphic types.

2.2. **Reflection in Other Languages.** Managed languages like Java and C# have a very strong and well developed *runtime* reflection system.

In Java it is possible to query a class' name, package info, superclass, implemented interfaces, methods, fields and annotations through the `Class` object.

It is even achievable to get information about private members. Regarding to methods, one can get all the parameters' type and the return type. It is also feasible to call one reflected member function (without knowing the exact name of it). Java reflection can be used to list an enum class' enumeration values as well. C# has similar reflection capabilities to Java [6, 7].

Scala provides both *runtime* and *compile-time* reflection. The compile-time reflection is realized in the form of macros, which provide the ability to execute methods that manipulate abstract syntax trees at compile time. Scala uses the so-called `Universe` to set up runtime or compile-time reflection. It is accomplish-able to control the set of entities that we have reflective access to, by the so-called `mirror` [8].

The D programming language provides *compile-time* reflection through the `Traits` extension. It has very similar properties to the C++ type traits library, but a little bit more can be achieved with it [9].

2.3. **Standardized Reflection in C++ at 2014.**

2.3.1. *RTTI.* Run-Time Type Information and `dynamic_cast` expressions can be used together to determine the dynamic type of an object of a polymorphic class. Under the hood `dynamic_cast` might use similar or common implementation details to the `typeid` operator which results a `type_info` object. Objects of class `type_info` can be compared, so the same polymorphic types will have the same objects. Since C++11, `hash_code` can be used which returns a value which is identical for the same types. Also since C++11 `type_index` is existing, which is a wrapper around a `type_info` object, that can be used as an index in associative and unordered associative containers. RTTI can be considered as a runtime reflection in C++, however the reflected meta data is simply not enough to execute higher level reflection tasks [1, 10].

2.3.2. *Type Traits.* Type traits are type related queries and type modifications, which can be executed during compile-time. Most of the queries are returning with a boolean value or with a simple integral value. Examples:

- `is_integral` checks if a type is integral type
- `is_same` checks if two types are the same
- `rank` obtains the number of dimensions of an array type
- `remove_reference` removes reference from the given type

Type traits are reflecting meta data of types, but with the help of them it can only be decided whether a type has a specific property or not. Higher level reflection tasks, like querying names of all the fields of a class are impossible with them [1, 10].

2.4. **C++ Without Standardized Compile Time Reflection.** In this subsection we discuss current C++ techniques which are widely used in industrial environments. We demonstrate through examples, why the life of a C++ programmer is harder without built-in compiler support for static reflection.

2.4.1. *Serialization.* There is a boost serialization library which can be used both intrusively and non-intrusively [11]. The following example demonstrates the non-intrusive method (the original class is not modified.)

```
struct gps_position
{
    int degrees;  int minutes; float seconds;
};
namespace boost { namespace serialization {
template<class Arch>
void serialize(Arch& ar,gps_position& g,const unsigned int ver) {
    ar & g.degrees;
    ar & g.minutes;
    ar & g.seconds;
} }} // namespace boost::serialization
```

We can see that for each and every new class a new template specialization is needed to be written. If there had been static reflection, then serialization could be solved in a generic way.

2.4.2. *Unit Test Mock Frameworks.* The following code snipet demonstrates an abstract class (`Turtle`) and its mock class. The mock can be used everywhere, where the original type appears as an interface. The mock class is created by Google's mocking framework, gmock [12].

```
class Turtle {
  virtual ~Turtle() {}
  virtual void PenUp() = 0;
  virtual void PenDown() = 0;
  virtual void Forward(int distance) = 0;
  virtual void Turn(int degrees) = 0;
  virtual void GoTo(int x, int y) = 0;
  virtual int GetX() const = 0;
  virtual int GetY() const = 0;
};
class MockTurtle : public Turtle {
 public:
  MOCK_METHOD0(PenUp, void());
  MOCK_METHOD0(PenDown, void());
```

```
  MOCK_METHOD1(Forward, void(int distance));
  MOCK_METHOD1(Turn, void(int degrees));
  MOCK_METHOD2(GoTo, void(int x, int y));
  MOCK_CONST_METHOD0(GetX, int());
  MOCK_CONST_METHOD0(GetY, int());
};
```

It is an obvious drawback, that each and every function is needed to be defined by a macro. If the interface (the abstract class in this case) is a subject of change, then the mock class needs to be updated with the same change frequency. If there was static reflection, then mock classes could be programmed in a generic way, and they could be created by the compiler.

2.4.3. *Static Reflection.* There are workarounds for the missing static reflection though. The below example demonstrates how to add meta data manually, without the compiler's help.

```
// Your existing struct
struct Foo { int i; bool j; /* ... */ };
// "Foo" as a Boost.Fusion sequence
BOOST_FUSION_ADAPT_STRUCT(Foo, (int, i) (bool, j))
struct Action {
    template<typename T>
    void operator()(T& t) const {
        // do whatever you need, e.g. serialize
    }
};
void usage() {
    Foo foo;
    boost::fusion::for_each(foo, Action{});
}
```

Here, the Boost.Fusion library is used [13], but there are several other similar libraries for this purpose. Someone, who is building a generic object relational mapping library, might end up using something similar to this. Note that, when `Foo` is changing, the manually provided meta data is needed to be changed, so again one conceptual change requires at least two change in the editor.

3. Related Work

The ISO C++ Committee (WG21) is organized into several subgroups. The Reflection Study Group (SG7) started its work at the fall of 2013 with the paper N3814 – Call for Compile-Time Reflection Proposals [14]. In this paper the most important compile-time reflection use cases are enumerated and the C++ community is asked to provide proposals to introduce compile-time reflection into the language. The use cases are:

(1) Generation of common functions like equality operators, serialization functions. (Note that this implies the enumeration of class members.)
(2) Type transformations like Struct-of-Arrays.
(3) Compile-time context information (replacing assert).
(4) Enumeration of other entities (namespaces, enums, etc).

3.1. **Low level intrinsics - N3815.** In response to N3814, N3815 had been written to give a proposal about compile-time reflection of enumeration lists. N3815 proposes to add three Property Queries to the Metaprogramming and Type Traits Standard Library that provide compile-time access to the enumerator-list of an enumeration type [15]. Specifically:

- std::enumerator_list_size<E>: the number of enumerator-definitions in the enumerator-list of E.
- std::enumerator_identifier<E,I>: the identifier from the I'th enumerator-definition.
- std::enumerator_value<E,I>: the value from the I'th enumerator-definition.

There is a reference implementation for N3815 in clang, done by Christian Kaeser [22]. This contains clang specific compiler intrinsics with which the above mentioned Property Queries can be served.

In this reference implementation amongst the enumerator related intrinsics there are other intrinsics implemented for querying member fields of a class:

- record_member_field_count<A>: the number of fields in A.
- record_member_field_identifier<A, I>: the identifier from the I'th field of A.
- object_member_field_ref<A, a, I>: the reference of the I'th field in object a, where a is an instance of A.

Note: the implementation uses __ prefixes for each intrinsics. With the above three intrinsics, the *Generation of common functions* problem can be solved either with recursive templates or with variadic templates [16]. This implementation can be easily extended for example to query the number of methods [23].

Until the time of writing this paper there was no such proposal for querying meta data of classes with a "size + index" interface. On the other hand it is highly expectable, that in june 2014 in Rapperswil Andrew Tomazos will give such a proposal [17].

3.2. **All In One - typename<>(N3951).** N3951 proposes to gather the meta data at once, without a "size + index" interface [18]. From a type T, obtain static typed reflection adding 2 language constructs:

(1) An instruction `typename<T>...` that expands members identifiers of type `T` into a variadic template. Each type of n-th element of `typename<T>...` is a const char* and each n-th value is the identifier of n-th member of `T`, expressed in UTF-8 encoded;
(2) An instruction `typedef<T>...` that expands members of type `T` into a variadic template (in the same order of `typename<T>...`). Each n-th type of `typedef<T>...` is the type of the n-th member of `T` and each n-th value is a pointer to n-th member of T, or a value if member is a `constexpr` member or enum item; `typename<T>...` and `typedef<T>...` could be implemented in terms of the N3815 related lower level "size + index" reflection traits as a library. Therefore it is likely that the "size + index" related proposals will be accepted finally.

3.3. **Exposing the AST - N3883.** There is a proposal which aims to solve reflection related tasks with a completely different aspect. N3883 [19] tries to answer this question: How to solve enumeration of members without template recursion? It introduces "static if" and "static for" like language constructs. Therefore template metaprogramming could be avoided in case of reflection tasks. Also the goal of this proposal is to expose an AST like interface into the language with which all the meta data can be queried. Though this proposal has trivial advantages, it is not well elaborated and there are lot of opened questions. In the far future similar solutions might appear in the language, but currently it looks like there is a consensus in SG7 to strive for a lower level and simpler compile-time reflection first.

3.4. **Compile-time Strings.** Compile-time strings are playing an important role as being the carrier of a reflected identifier's name. N3815 and N3951 proposes to use char arrays as a carrier for names, this is because currently there is no better alternative in C++. However it might be possible that in the future a `basic_string_literal` will be introduced as it is stated in D3933 [20]. When that happens, reflection related proposals might be discussed again to reflect the names into `basic_string_literals`.

3.5. **Code Generators.** For the sake of completeness, code generators like Qt's Meta Object Compiler (MOC) and OpenC++ Meta Object Protocol (MOP) must be referenced [4, 5]. The idea behind these approaches is to extend the base C++ language with some reflection and meta object creation capabilities. In both cases a pre-compile phase needs to be added to the compilation process. Before the C++ compiler is called, the meta compiler must be invoked to translate the extended C++ into standardized C++. We can see the obvious disadvantages:

(1) One additional compilation step is needed along with a new parsing and semantic analysis.

(2) Lack of standardization.

The goal of SG7 is to provide a powerful native reflection, with which such precompilation is not needed. According to Olivier Goffart, Qt's MOC might be replaced with an extended version of N3951 [28].

4. Static Reflection for Proxy and Mock Objects

In the following we describe our reflection approach which could help to create a generic proxy or mock object. First we describe proxy and mock objects, then we display the new expressions.

4.1. **Mock and Proxy Objects (and Classes).** Mock objects are used in unit tests to substitute real dependencies of a unit. (A unit is typically a class (struct) or a free function.) The programmer can formulate expectations towards a mock object, e.g. how many times a member function is called with a certain value? Proxy objects are those objects which are having the exact same interface as the original object, but the implementation of each member function could be different. Therefore mock objects are special kind of proxy objects. Proxy objects seemed to be so useful that Java introduced the *Dynamic Proxy* concept to ease the creation of proxies [21]. Mock objects are instances of mock classes, proxy objects are instances of proxy classes.

A simple aggregate class is a C++ struct with publicly available fields and without methods. The definition of a simple aggregate *proxy* class is a recursive definition: A simple aggregate class is a proxy class, if all of its field have a proxy class type. The built in types like `int`, `double`, `float` are considered as proxy types.

4.2. **Proposed Approach - Defining New Expressions.** To successfully solve the problem of creating proxy and mock classes we need to have two new expressions.

(1) `variable_decl` for declaring and defining variables based on reflected types and names.
(2) `function_decl` for declaring and defining functions based on reflected types and names.

These expressions ideally would be mapped under std::reflect namespace. This mapping is needed in order to hide the compiler specific implementation details. This is the exact case with some already existent type traits as well, e.g. std::is_pod.

4.3. **Declare a New Variable.** Let's assume we have the following simple struct:

```
struct A {
    int m_a;
    float m_b;
};
```

The usage of `variable_decl` is shown through the below example:

```
// B has exactly the same field as A.
// Note: Only m_a is replicated.
struct B {
    reflect::variable_decl<
        reflect::record_member_field_type<A, 0>,
        reflect::record_member_field_identifier<A, 0> >;
};
```

If we had written this, the above code would have been equivalent to

```
// B has exactly the same field as A
struct B {
    int m_a;
};
```

In this example `variable_decl` has two subexpressions

(1) A *type-specifier*, which refers to the newly declared variable's type.
(2) A *compile-time string*, which is equal to the **name** of the newly declared variable.

The type-specifier is an expression whose value is a type, which can be evaluated during the compilation process. For instance this can be a result of any kind of meta function or can be a result of any kind of reflection expression. Expression `variable_decl` is not bound to any concrete reflection query implementation, it just requires the first parameter to be a type. The compile-time string can be either the C++14's compile time string which is a simple char array; or this can be a *basic_string_literal* as described in D3933 proposal [20].

Despite of the independence of reflection implementations, here in this paper `record_member_field_identifier` is used as it is implemented in the N3815 proposal related implementation [22]. Here we use the expression `record_member_field_type<T,N>` which is equal to the type of `T`'s `N`-th field type. Currently such intrinsic is not implemented.

The expression `variable_decl` shall be handled as a normal variable declaration/definition, therefore if the variable is needed to be initialized, then the following code should be written:

```
struct B {
    reflect::variable_decl<
        reflect::record_member_field_type<A, 0>,
```

```
        // initialize
        reflect::record_member_field_identifier<A, 0> > = 0;
};
```

Once `variable_decl` is implemented then, an aggregate proxy class can be created recursively for `struct A`. Note, the start of the recursion is missing, that will be elaborated later.

```
template <unsigned int Index> // C has the same field names as A,
struct C : C<Index-1> {       // but all fields are proxied.
    reflect::variable_decl<
        Proxy<reflect::record_member_field_type<A, Index>>,
        reflect::record_member_field_identifier<A, Index> >;
};
template <>
struct C<0> {
    reflect::variable_decl<
        Proxy<reflect::record_member_field_type<A, 0>>,
        reflect::record_member_field_identifier<A, 0> >;
};
```

Here it is assumed that such a `Proxy` class is existent, which can do the proxying for all field types of `struct A`.

If the proxy task is mocking (being able to create expectations), then it is assumed that it shall be possible to create a `Proxy` class for each primary types (integral type, floating point type, pointer type, etc) and POD types.

Making `struct A` to be a template parameter we get the generic proxy aggregate struct:

```
template <typename A, unsigned int Index>
struct D : D<A, Index-1> {
    reflect::variable_decl<
        Proxy<reflect::record_member_field_type<A, Index>>,
        reflect::record_member_field_identifier<A, Index> >;
};
template <typename A>
struct D<A, 0> {
    reflect::variable_decl<
        Proxy<reflect::record_member_field_type<A, 0>>,
        reflect::record_member_field_identifier<A, 0> >;
};
```

The template recursion must be started with the number of fields in type `A`.

```
template <typename A>
struct GenericAggregateProxy :
```

```
    D<A, reflect::record_member_field_count<A>> {};
```

`record_member_field_count` is implemented in N3815's reference implementation [22]. Expression `variable_decl` should be implemented similarly as type_traits expressions. In case of `clang` this means

- A new expression should be introduced in TokenKinds.def.
- Parsing actions should be created in `clang::Parser`.
- Semantic analysis should be added to `clang::Sema`.
- A new AST node should be introduced for `variable_decl`.
- Template instantiation rules for this AST should be given.[24]

The template instantiation rules should include the template transformation rules, which finally should result a modified AST for this new expression. After the instantiation the specific AST node should look like if it had been manually written. The template transformation rules are delegated back to the original `clang::FieldDecl` AST transformations. Similarly, intermediate code generation could be delegated as well. N3815's reference implementation could be a good example to follow: there too a new expression – `ReflectionTypeTraits` – is introduced in a similar way as it is described above [22].

4.4. **Declare a New Method.** On the way to provide a generic mock class the next step is to be able to define functions based on reflected information. That is the exact purpose of introducing the `function_decl` expression. The idea is really similar to the one in case of `variable_decl`. The following recursively built `struct C` will have exactly the same functions declared as `struct A`. In this case the start of the recursion will use the number of member functions in `struct A`. Note, all member functions in `struct A` have only one parameter.

```
struct A {
    int m_func1(int);
    float m_func2(float);
};
template <unsigned int Index> // C has the same functions as A
struct C : C<Index-1> {       // but they all have one parameter
    reflect::function_decl<
        reflect::record_member_function_result_type<A, Index>,
        reflect::record_member_function_identifier<A, Index>,
        reflect::record_member_function_param<A, Index, 0> >;
};
template <>
struct C<0> {
    reflect::function_decl<
        reflect::record_member_function_result_type<A, 0>,
```

```
        reflect::record_member_function_identifier<A, 0>,
        reflect::record_member_function_param<A, 0, 0> >;
};
```

Here `function_decl` has three subexpressions:
(1) A *type-specifier*, which refers to the newly declared function's return type.
(2) A *compile-time string*, equal to the **name** of the newly declared variable.
(3) parameter type of the function. In this case the declared function can have only one parameter.

`reflect::record_member_function_param` should be exposed as a type list in case of functions with more parameters:

```
// C has exactly the same functions as A
template <unsigned int Index>
struct C : C<Index-1> {
    reflect::function_decl<
        reflect::record_member_function_result_type<A, Index>,
        reflect::record_member_function_identifier<A, Index>,
        // list of types !
        reflect::record_member_function_params<A, Index> >;
};
template <>
struct C<0> {
    reflect::function_decl<
        reflect::record_member_function_result_type<A, 0>,
        reflect::record_member_function_identifier<A, 0>,
        reflect::record_member_function_params<A, 0> >;
};
```

Defining functions based on reflected information is more complex:

```
template <unsigned int Index>
struct C : C<Index-1> {
    reflect::function_decl<
        reflect::record_member_function_result_type<A, Index>,
        reflect::record_member_function_identifier<A, Index>,
        reflect::record_member_function_params<A, Index> >
    {
        struct Handler {
            template <typename... Ts>
            auto operator()(std::tuple<Ts...>& args)
            {
                // ...
            }
        };
```

```
        Handler{}(reflect::function_decl_params);
    }
};
template <>
struct C<0> { /* ... similar as before */ };
```

Here `reflect::function_decl_params` is again a necessary new expression, which would be exposed as an `std::tuple` object. Each n-th type of the tuple should be the type of the n-th function parameter, and each n-th value shall be a reference to the n-th function parameter. Note that, it might be more feasible to use a function parameter pack instead of `std::tuple`, but for the ease of explanation, tuple had been used.

When the C++ compiler reaches the parsing of the function's body, at that point the function parameters are already parsed and the corresponding semantic actions had been taken. Therefore it is assumed when the compiler parses `reflect::function_decl_params` the parameters can be gathered and tied into a tuple object.

For instance, the `clang` compiler provides the `getParamDecl()` function in the AST class `FunctionDecl`, with which the parameters can be queried. As the following back trace extraction illustrates, in case of the `clang` compiler version 3.4 the `ActOnStartOfFunctionDef` semantic action is executed, during the parsing of a function declaration:

```
#6  0x... in clang::Sema::ActOnStartOfFunctionDef (...)
#7  0x... in clang::Parser::ParseFunctionDefinition (...)
#8  0x... in clang::Parser::ParseDeclGroup (...)
#9  0x... in clang::Parser::ParseDeclOrFunctionDefInternal (...)
#10 0x... in clang::Parser::ParseDeclarationOrFunctionDefinition
#11 0x... in clang::Parser::ParseExternalDeclaration (...)
#12 0x... in clang::Parser::ParseTopLevelDecl (...)
```

If we take a look into this function, it can be seen that the function parameters are indeed used for registering them into the function's body scope [25].

```
Decl *Sema::ActOnStartOfFunctionDef(Scope *FnBodyScope, Decl *D)
{
  ...
  FunctionDecl *FD = 0;
  if (FunctionTemplateDecl *FunTmpl =
         dyn_cast<FunctionTemplateDecl>(D))
    FD = FunTmpl->getTemplatedDecl();
  else
    FD = cast<FunctionDecl>(D);
  ...
  // Introduce our parameters into the function scope
```

```
  for (unsigned p = 0, NumParams = FD->getNumParams();
      p < NumParams; ++p) {
    ParmVarDecl *Param = FD->getParamDecl(p);
    Param->setOwningFunction(FD);

    // If this has an identifier, add it to the scope stack.
    if (Param->getIdentifier() && FnBodyScope) {
      CheckShadow(FnBodyScope, Param);
      PushOnScopeChains(Param, FnBodyScope);
    }
  }
  ...
}
```

This means our assumption about the parsed function parameters is correct, at least in case of this specific `clang` compiler version. The assumption could be proved similarly for other vendor's compilers as well.

4.5. **Overloading the Dot Operator.** By overloading the member access operator, the simplest proxy cases could be solved. However more complex cases cannot be implemented with it. For instance if we want to have a class B with the exact same fields as class A, but fields whose name is starting with a specific prefix are not needed.

Overloading the dot operator similarly as it is done with the `->` operator, had been proposed by Jim Adcock in 1990 [26], but was rejected for various reasons [27, 2]. In 2013 Sebastian Redl had a presentation about overloadable template operator dot, with a compile-time string template parameter [3]. This approach looks promising, but it has its own difficulties and drawbacks.

5. Open Questions and Further Researches

So far, a new approach had been introduced and the basic idea had been illustrated, but only the simplest cases had been covered. These simple cases are based on the current reflection proposals, which are providing a quite clear reflection interface for the most primitive cases, But we have to notice, there are lots of open questions about this approach. Regarding the variables:

(1) How to declare static variables?
(2) How to handle C++14's templated variables?

In respect of the functions:

(1) How to handle template functions?
(2) How to handle constructors?
(3) How to handle ellipsis function parameters?
(4) How to handle exception specifications?

These questions cannot be answered at the time of writing this document. This is because first it must be decided how to reflect ellipsis, template functions, exception specifications, etc. Also a proof-of-concept implementation for the simple cases would be needed to demonstrate that the approach is viable.

6. Conclusion

Reflection in C++ is a hot research area and it is a subject of frequent changes. Many application areas require it, but the approaches to define a firm interface are different. Lot of people want it, but the approaches are different. Reflection itself is a large topic, it is not even clear what meta information could be queried in future C++. For example there is a debate whether the contents of a namespace should be query able or not. Despite of these uncertainties it is sure that the most general reflection queries like getting the fields of a class will be part of some future C++ standard. Generally speaking, querying meta information is one layer of reflection, though there is higher layer when this meta data is used to create new program elements like types, variables, etc. Sometimes it is called intercession.

In this article an approach has been presented, with which declaring or defining new variables and functions based on reflected meta information is possible. The main advantage of this approach is that the existing C++ metaprogramming practices can be reused. It is possible to create generic classes which could behave as generic proxy, mock or serialization classes. The disadvantages are that new expressions will be introduced, but generally with introducing reflection this cannot be avoided. The basic idea has been shown here, but there are lots of opened questions, therefore additional researches and reference implementations need to be done.

References

[1] ISO International Standard, ISO/IEC 14882:2011(E) – Programming Language C++
[2] **Bjarne Stroustrup,** Design and Evolution of C++, *ISBN 0-201-54330-3*
[3] **Sebastian Redl,** Overloading the Member Access Operator, *C++Now 2013*
[4] **Shigeru Chiba,** A Metaobject Protocol for C++, *In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), page 285-299, October 1995.*
[5] **Shigeru Chiba,** OpenC++ Programmer's Guide for Version 2, *Technical Report SPL-96-024, Xerox PARC, 1996.*
[6] Oracle, Trail: The Reflection API, *http://docs.oracle.com/javase/tutorial/reflect/*
[7] Microsoft Developer Network, Reflection (C# and Visual Basic), *http://msdn.microsoft.com/en-us/library/ms173183.aspx*
[8] Scala Documentation, REFLECTION, *http://docs.scala-lang.org/overviews/reflection/overview.html*
[9] D Programming Language, Traits, *http://dlang.org/traits.html*

[10] C++ Reference, *http://en.cppreference.com*

[11] **Robert Ramey,** Boost Serialization Library, *http://www.boost.org/doc/libs/1_55_0/libs/serialization/doc/index.html*

[12] Google, googlemock - Google C++ Mocking Framework, *https://code.google.com/p/googlemock/*

[13] **Joel de Guzman, Dan Marsden, Tobias Schwinger,** Boost Fusion Library, *http://www.boost.org/doc/libs/1_55_0/libs/fusion/doc/html/*

[14] **Jeff Snyder, Chandler Carruth,** Call for Compile-Time Reflection Proposals (N3814), *http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3814.html*

[15] **Andrew Tomazos, Christian Käser,** Enumerator List Property Queries (N3815), *http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3815.html*

[16] **Jonathan Wakely,** Compile-time integer sequences (N3658), *http://www.open-std.org/JTC1/sc22/WG21/docs/papers/2013/n3658.html*

[17] SG 7 - Reflection Forum, *https://groups.google.com/a/isocpp.org/forum/#!forum/reflection*

[18] **Cleiton Santoia Silva, Daniel Auresco,** C++ type reflection via variadic template expansion (N3951), *https://github.com/cleitonsantoia/reflection/blob/master/N3951_reflection_cpp.pdf*

[19] **Péter Németh,** Code checkers & generators (N3883), *http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3883.html*

[20] **Michael Price,** Compile-time string draft (D3933)

[21] Java Platform Standard Ed. 7, Class Proxy, *http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html*

[22] **Christian Kaeser,** Clang Reflection, *https://github.com/ChristianKaeser/clang-reflection*

[23] **Gábor Márton,** Clang Reflection, *https://github.com/martong/clang-reflection/tree/member_function*

[24] Clang 3.5 documentation, CLANG CFE INTERNALS MANUAL - How to add an expression or statement, *http://clang.llvm.org/docs/InternalsManual.html#how-to-add-an-expression-or-statement*

[25] llvm-project Revision 207549, *http://llvm.org/svn/llvm-project/cfe/tags/RELEASE_34/final/*

[26] **Jim Adcock,** Request for Consideration: Overloadable Unary operator.(), *https://groups.google.com/forum/#!msg/comp.lang.c++/GauQas227YM/XRZVkkyYT3UJ*

[27] **Bjarne Stroustrup,** C++ Style and Technique FAQ, *http://www.stroustrup.com/bs_faq2.html#overload-dot*

[28] **Olivier Goffart,** Can Qt's moc be replaced by C++ reflection? *http://woboq.com/blog/reflection-in-cpp-and-qt-moc.html*

Dept. of Programming Languages and Compilers, Eötvös Loránd University
E-mail address: `martongabesz@gmail.com, gsd@elte.hu`