Document number: Dnnnn=12-mmmm
Date: 2012-09-25
Project: Programming Language C++, Library Working Group
Reply-to: Matúš Chochlík (Matus.Chochlik@fri.uniza.sk)

Contents

1. Introduction

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 lacks a standardized reflection facility.

In this paper we propose the addition of native support for compile-time reflection to C++ and a library built on top of the metadata provided by the compiler.

The basic static metadata provided by compile-time reflection should be as complete as possible to be applicable in a wide range of scenarios and allow to implement custom higher-level static and dynamic reflection libraries and reflection-based utilities.

The term *reflection* refers to 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. Reflective programming is a particular kind of *metaprogramming*.

Reflection should follow the principle of *Ontological correspondence*, i.e. should *reflect* the base-level program constructs as closely as possible to a reasonable level. Reflection should not omit existing language features not invent new ones that do not exist at the base-level.

What reflection "looks like" is thefore very language-specific. Reflection for C++ is necessary different from reflection in Smalltalk since these are two quite different languages.

The "reasonability" applies to the level-of-detail of the metadata provided by reflection. It is a tradeoff between the complexity of the reflection system and its usefulness. The "metadata" provided by the currently standard `typeid` operator are rather simple (which may be good), but their usefulness is very limited (which is bad). On the other hand a fictional reflection facility that would allow to inspect the individual instructions of a function could be useful for some specific applications, but this system would also be very complex to implement and use. The proposed reflection system tries to walk a "middle ground" and be usable in many situations without unmanagable complexity.

The advantage of using reflection is in the fact 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.

The solution proposed in this paper is based on the expirience with *Mirror* reflection utilities [1] and with reflection-based metaprogramming.

2. Motivation and Scope

2.1. Usefullness of reflection

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

- serialization or storing of persistent data in a custom binary format or in XML, JSON, XDR, etc.,
- (re-)construction of class instances from external data representations (like those listed above), from the data stored in a relational database, from data entered by a user through a user interface or queried through a web service API,

- automatic generation of a relational schema from the application object model and object-relational mapping (ORM),
- support for scripting
- support remote procedure calls (RPC) / remote method invocation (RMI),
- inspection and manipulation of existing objects via a (graphic) user interface or a web service,
- visualization of objects or data and the relations between objects or relations in the data,
- automatic or semi-automatic implementation of certain software design patterns,
- etc.

There are several aproaches to the implementation of such functionality. The most straightforward and also usually the most error-prone is manual implementation. Many of the tasks listed above are inherently repetitive and basically require to process programming language constructs (types, structures, containers, functions, constructors, class member variables, enumerated values, etc.) in a very uniform way that could be easily transformed into a meta-algorithm.

While it is acceptable (even if not very advantageous) for example, for a design pattern implementation to be made by a human, writing RPC/RMI-related code is a task much better suited for a computer.

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

This approach has several problems. First, it requires the external tools which may not fit well into the build system or may not be portable between platforms or be free; second, such tools are task-specific and many of them allow only a limited, if any, customization of the output.

Another way to automate these tasks is to use reflection, reflective programming, metaprogramming and generic programming as explained below.

2.2. Motivational examples

This section describes some of the many possible uses of reflection and reflective programming on concrete real-world examples.

2.2.1. Factory generator

As already said above, it is possible (at least partially) to automate the implementation of several established software design patterns. This example shows how to implement a variant of the *Factory* pattern.

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 (in the programming language) nor supplies the required arguments directly in the C++ intrinsic data representation.

So instead of direct construction of a Product type,

```
// get the values of arguments from the user
int arg1 = get_from_user<int>("Product arg1");
double arg2 = get_from_user<double>("Product arg2");
std::string arg3 = get_from_user<std::string>("Product arg3");
//
// call a constructor with these arguments
Product* pp = new Product(arg1, arg2, arg3);
// default construct a Product
Product p;
// copy construct a Product
Product cp = p;
```

which involves selection of a specific constructor, getting the values of the required arguments and possibly converting them from an external representation and calling the selected constructor with the arguments, factories pick or let the application user pick the Product's most appropriate constructor, they gather the necessary parameters in a generic way and use the selected constructor to create an instance of the Product:

```
// get data necessary for construction in xml
XMLNode xml_node_1 = get_xml_node(...);
XMLNode xml_node_2 = get_xml_node(...);

// make a factory for the product type
Factory<Product, XMLWalker> xml_factory;

// use the factory to create instances of Product
// from the external representation
Product p = xml_factory(xml_node_1);
Product* pp = xml_factory.new_(xml_node_2);
```

One of the interesting features of these factories is, that they separate the caller (who just needs to get an instance of the specified type) from the actual method of creation.

By using a factory, the constructor to be called can be automatically picked depending

on the data available only at run-time and not be chosen by the programmer (at least not directly as in the code above). Factory can match the constructor to best fit the data available in the external representation (XML or JSON fragment, dataset resulting from a RDBS query, etc.)

Even more interesting is, that such factories can be implemented semi-automatically with the help of reflection.

Every factory is a composition of two distinct (and nearly orthogonal) parts:

- Product-type-dependent: includes the enumeration of Product's constructors, enumeration of their parameters, information about the context in which a constructor is called, etc. This part is based on reflection and independent on the representation of the input data.
- Data representation-dependent: includes the scanning of the available input data, conversion into C++ intrinsic data representation, and the selection of the best constructor. This part is user-defined and specifies how the input data is gathered and converted into the C++ representation.

These two parts are then tied together into the factory class. Based on the input-data related components, the factory can include a script parser or XML document tree walker or code dynamically generating a GUI for the input of the necessary values and the selection of the preferred constructor. Figure 1 shows such a GUI created by factory automatically generated by the Mirror's *factory generator* utility for a tetrahedron class with the following definition:

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

        vector(double _x, double _y, double _z)
         : x(_x), y(_y), z(_z)
        { }

        vector(double _w)
         : x(_w), y(_w), z(_w)
        { }

        vector(void)
         : x(0.0), y(0.0), z(0.0)
        { }

        /* other members */
};

struct triangle
```

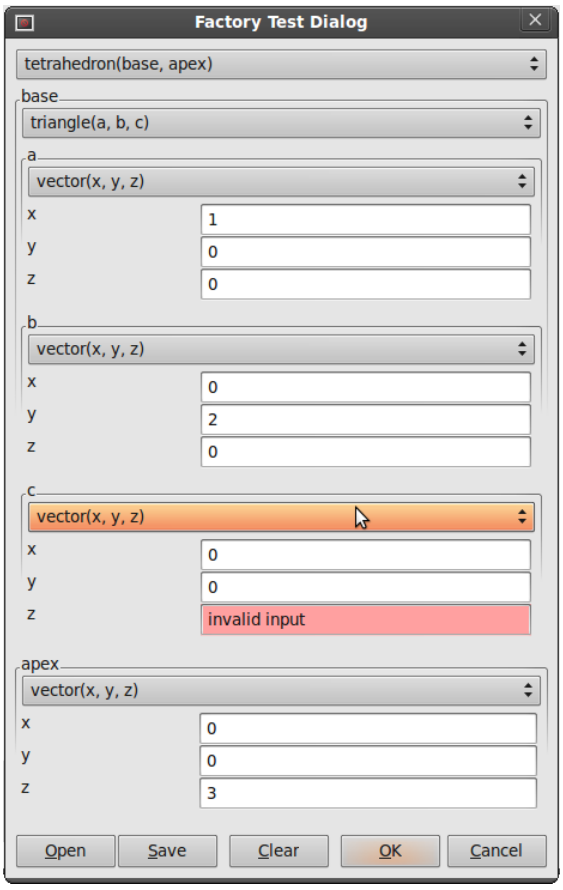

Figure 1: Example of a GUI created by a factory generated by the Mirror's factory generator.

```
{
        vector a, b, c;

        triangle(
                const vector& _a,
                const vector& _b,
                const vector& _c
        ): a(_a), b(_b), c(_c)
        { }

        triangle(void){ }

        /* other members */
};
```

```
struct tetrahedron
{
        triangle base;
        vector apex;

        tetrahedron(const triangle& _base, const vector& _apex)
         : base(_base), apex(_apex)
        { }

        tetrahedron(
                const vector& a,
                const vector& b,
                const vector& c,
                const vector& d
        ): base(a, b, c), apex(d)
        { }

        /* other members */
};
```

3. Design Decisions

3.1. Desired features

The proposed reflection facility is designed with the following goals in mind:

- *Reusability*: The provided metadata should be reusable in many situations and for many different purposes, not only the obvious ones. This is closely related to *completeness* (below).
- *Flexibility*: The basic reflection and the libraries built on top of it should be designed in a way that they are eventually usable during both compile-time and run-time and under various paradigms (object-oriented, functional, etc.), depending on the application needs.
- *Encapsulation*: The metadata should be accessible through conceptually well-defined interfaces.
- *Stratification*: Reflection should be non-intrusive, and the meta-level should be separated from the base-level language constructs it reflects. Also, reflection should not be implemented in a all-or-nothing manner. Things that are not needed, should not generally be compiled-into the final application.
- *Ontological correspondence*: The meta-level facilities should correspond to the ontology of the base-level C++ language constructs which they reflect. This basically

means that all existing language features should be reflected and new ones should not be invented. This rule may have some important exceptions, like the reflection of containers.

- *Completeness*: The proposed reflection facility should provide as much useful metadata as possible, including various specifiers, (like constness, storage-class, access, etc.), namespace members, enumerated types, iteration of namespace members and much more.
- *Ease of use*: Although reflection-based metaprogramming allows to implement very complicated things, simple things should be kept simple.
- *Cooperation with other librares*: Reflection should be usable with the existing introspection facilites (like `type_traits`) already provided by the standard library and with other libraries.

3.2. Layered approach and extensibility

The purpose of this section is to show that a *static* → *dynamic* and *basic* → *complex* approach in designing reflection can accomodate a wide variety of programming styles and is arguably the "best" one. We do not propose to add all layers described below into the standard library. They are mentioned here only to show that a well designed compile-time reflection is a good foundation for many (if not all) other reflection facilities.

The Mirror reflection utilities [1] on which this proposal is based, implements several distinct components which are stacked on top of each other. From the low-level metadata, through a functional-style compile-time interface to a completely dynamic object-oriented run-time layer (all described in greater detail below).

3.2.1. Basic metaobjects

The very basic metadata, which are in Mirror provided (registered) by the user (or an automated command-line tool) via a set of preprocessor macros. This approach is both inconvenient and error-prone in many situations, but also has its advantages.

We propose that a standard compiler should make these metadata available to the programmer through the static basic metaobject interfaces described below. These should serve as the basis for other (standard and non-standard) higher-level reflection libraries and utilities.

In the Mirror utilities the basic metadata is not used directly by applications.

3.2.2. Mirror

Mirror is a compile-time functional-style reflective programming library, which is based directly on the basic metadata and is suitable for generic programming, similar to the standard `type_traits` library.

Mirror is the original library from which the Mirror reflection utilities started.

It provides a more user-friendly and rich interface than the basic-metaobjects. and a set of metaprogramming utilities which allow to write compile-time meta-programs, which can generate efficient and optimized program code using only those metadata that are required.

The following text contains several (rather simple) examples of usage and the functional style of the algorithms based on metadata provided by Mirror.

The first example prints some information about the members of selected namespaces to `std::cout`.

```
struct info_printer
{
    template <typename MetaObject>
    void operator()(MetaObject mo) const
    {
        MIRRORED_META_OBJECT(MetaObject) mmo;
        std::cout
            << mmo.construct_name()
            << ": "
            << mo.full_name()
            << std::endl;
    }
};

int main(void)
{
    using namespace mirror;

    // print the info about each of the members
    // of the global scope
    mirror::mp::for_each<
        members<

            // this should be in standard C++
            // be replaced by a specialstandard library
            // function or operator
            MIRRORED_GLOBAL_SCOPE()
```

```
        >
    >(info_printer());

    // print the info about each of the members
    // of the std namespace
    mp::for_each<
        members<

            // this should be in standard C++
            // be replaced by a special standard
            // library function or operator
            MIRRORED_NAMESPACE(std)
        >
    >(info_printer());
    //
    return 0;
}
```

This program produces the following output:

```
namespace: std
namespace: boost
type: void
type: bool
type: char
type: unsigned char
type: wchar_t
type: short int
type: int
type: long int
type: unsigned short int
type: unsigned int
type: unsigned long int
type: float
type: double
type: long double
class: std::string
class: std::wstring
class: std::tm
template: std::pair
template: std::tuple
template: std::allocator
template: std::equal_to
template: std::not_equal_to
template: std::less
```

```
   template: std::greater
   template: std::less_equal
   template: std::greater_equal
   template: std::vector
   template: std::list
   template: std::deque
   template: std::map
   template: std::set
```

The next example gets all types in the global scope, applies some `type_traits` modifiers like `std::add_pointer` `std::add_const` and for each of such modified types calls a functor that prints the names of the individual types to the standard output:

```
struct name_printer
{
    template <typename MetaNamedObject>
    void operator()(MetaNamedObject mo) const
    {
        std::cout << mo.base_name() << std::endl;
    }
};

int main(void)
{
  using namespace mirror;

  // this function calls the name_printer functor passed
  // as the function argument on each element in the
  // range that is passed as the template argument
  mp::for_each<

    // this template transforms the elements in the range
    // passed as the first argument by the unary template
    // passed as the second argument
    mp::transform<

      // this template filters out only those metaobjects
      // that satisfy the predicate passed as the second
      // argument from the range of metaobjects passed
      // as the first argument
      mp::only_if<

        // this template "returns" a range of metaobjects
        // reflecting the members of the namespace
        // (or other scope) that is passed as argument
```

```
        members<

          // this macro expands into a class
          // conforming to the Mirror's MetaNamespace
          // concept and provides metadata describing
          // the global scope namespace.
          // in the proposed solution for standard C++
          // this should be relaced by a special stdlib
          // function or by an operator.
          MIRRORED_GLOBAL_SCOPE()
        >,

        // this is a lambda function testing if its first
        // argument falls to the MetaType category
        mp::is_a<
          mp::arg<1>,
          meta_type_tag
        >
      >,

      // this is a unary lambda function that modifies
      // the type passed as its argument by
      // the add_pointer and add_const type traits
      apply_modifier<
        mp::arg<1>,
        mp::protect<
          std::add_pointer<
            std::add_const<
              mp::arg<1>
            >
          >
        >
      >
    >
  >(name_printer());
  std::cout << std::endl;
  return 0;
}
```

This short program produces the following output:

```
void const *
bool const *
char const *
unsigned char const *
```

```
wchar_t const *
short int const *
int const *
long int const *
unsigned short int const *
unsigned int const *
unsigned long int const *
float const *
double const *
long double const *
```

The printing of names is definitely not the only usage of reflection. The scope of this proposal does not allow to include and fully explain the more elaborated applications. For some other examples of usage see [2].

3.2.3. Puddle

Puddle is a OOP-style (mostly) compile-time interface built on top of Mirror. It copies the metaobject concept hierarchy of Mirror, but provides a more "object-ish" interface as shown below:

Instead of Mirror's:

```
static_assert(
  is_public<
    access_type<
      at_c<
        member_variables<
          reflected<person>
        >,
        0
      >
    >
  >::value,
  "Shoot, persons first mem. variable is not public!"
)
```

Puddle allows to do the following:

```
assert(
  reflected_type<person>()
    member_variables().
      at_c<0>().
        access_type().
          is_public()
```

```
);
```

or a more complex use-case, in which a reflection-based algorithm traverses the global scope namespace and its nested scopes and prints information about their members:

```
struct object_printer
{
  std::ostream& out;
  int indent_level;

  std::ostream& indented_output(void)
  {
    for(int i=0;i!=indent_level;++i)
      out << "  ";
    return out;
  }

  template <class MetaObject>
  void print_details(MetaObject obj, mirror::meta_object_tag)
  {
  }

  template <class MetaObject>
  void print_details(MetaObject obj, mirror::meta_scope_tag)
  {
    out << ": ";
    if(obj.members().empty())
    {
      out << "{ }";
    }
    else
    {
      out << "{" << std::endl;
      object_printer print_members = {out, indent_level+1};
      obj.members().for_each(print_members);
      indented_output() << "}";
    }
  }

  template <class MetaObject>
  void print(MetaObject obj, bool last)
  {
    indented_output()
      << obj.self().construct_name()
      << " "
```

```
      << obj.base_name();
    print_details(obj, obj.category());
    if(!last) out << ",";
    out << std::endl;
  }
  template <class MetaObject>
  void operator()(MetaObject obj, bool first, bool last)
  {
    print(obj, last);
  }

  template <class MetaObject>
  void operator()(MetaObject obj)
  {
    print(obj, true);
  }


int main(void)
{
  object_printer print = {std::cout, 0};
  print(puddle::adapt<MIRRORED_GLOBAL_SCOPE()>());
  return 0;
}
```

which prints the following on the standard output:

```
 global scope : {
   namespace std: {
     class string: { },
     class wstring: { },
     template pair,
     template tuple,
     template initializer_list,
     template allocator,
     template equal_to,
     template not_equal_to,
     template less,
     template greater,
     template less_equal,
     template greater_equal,
     template deque,
     class tm: {
       member variable tm_sec,
       member variable tm_min,
```

```
        member variable tm_hour,
        member variable tm_mday,
        member variable tm_mon,
        member variable tm_year,
        member variable tm_wday,
        member variable tm_yday,
        member variable tm_isdst
      },
      template vector,
      template list,
      template set,
      template map
    },
    namespace boost: {
      template optional
    },
    namespace mirror: { },
    type void,
    type bool,
    type char,
    type unsigned char,
    type wchar_t,
    type short int,
    type int,
    type long int,
    type unsigned short int,
    type unsigned int,
    type unsigned long int,
    type float,
    type double,
    type long double
  }
```

For more examples of usage see [3].

3.2.4. Rubber

Rubber is a OOP-style run-time type erasure utility built on top of Mirror and Puddle. It again follows the metaobject concept hierarchy of Mirror and Puddle. Rubber allows to access and store metaobjects of the same category in a single type, so in contrast to Mirror and Puddle where a meta-type reflecting the `int` type and a meta-type reflecting the `double` type have different types in Rubber they can both be stored in a variable of the same type. Rubber does not use virtual functions but rather pointers to existing

functions implemented by Mirror to achieve run-time polymorphism.

The first example shows the usage of type-erased metaobjects with a C++11 lambda function which could not be used with Mirror's or Puddle's meda-objects (because lambdas are not templated):

```
#include <mirror/mirror.hpp>
#include <rubber/rubber.hpp>
#include <iostream>

int main(void)
{
    // use the Mirror's for_each function, but erase
    // the types of the iterated compile-time metaobjects
    // before passing them as arguments to the lambda function.
    mirror::mp::for_each<
        mirror::members<
            MIRRORED_GLOBAL_SCOPE()
        >
    >(
        // the rubber::meta_named_scoped_object type is
        // constructible from a Mirror MetaNamedScopedObject
        [](const rubber::meta_named_scoped_object& member)
        {
            std::cout <<
                member.self().construct_name() <<
                " " <<
                member.base_name() <<
                std::endl;
        }
    );
    return 0;
}
```

This simple application prints the following on the standard output:

```
namespace std
namespace boost
namespace mirror
type void
type bool
type char
type unsigned char
type wchar_t
type short int
type int
```

```
type long int
type unsigned short int
type unsigned int
type unsigned long int
type float
type double
type long double
```

The next example prints different information for different categories of metaobjects:

```
#include <mirror/mirror.hpp>
#include <rubber/rubber.hpp>
#include <iostream>
#include <vector>

int main(void)
{
    using namespace rubber;
    mirror::mp::for_each<
        mirror::members<
            MIRRORED_GLOBAL_SCOPE()
        >
    >(
        eraser<meta_scope, meta_type, meta_named_object>(
            [](const meta_scope& scope)
            {
                std::cout <<
                    scope.self().construct_name() <<
                    " '" <<
                    scope.base_name() <<
                    "', number of members = " <<
                    scope.members().size() <<
                    std::endl;
            },
            [](const meta_type& type)
            {
                std::cout <<
                    type.self().construct_name() <<
                    " '" <<
                    type.base_name() <<
                    "', size in bytes = " <<
                    type.sizeof_() <<
                    std::endl;
            },
            [](const meta_named_object& named)
```

```
            {
                std::cout <<
                    named.self().construct_name() <<
                    " '" <<
                    named.base_name() <<
                    "'" <<
                    std::endl;
            }
        )
    );
    return 0;
}
```

It has the following output:

```
namespace 'std', number of members = 20
namespace 'boost', number of members = 0
namespace 'mirror', number of members = 0
type 'void', size in bytes = 0
type 'bool', size in bytes = 1
type 'char', size in bytes = 1
type 'unsigned char', size in bytes = 1
type 'wchar_t', size in bytes = 4
type 'short int', size in bytes = 2
type 'int', size in bytes = 4
type 'long int', size in bytes = 8
type 'unsigned short int', size in bytes = 2
type 'unsigned int', size in bytes = 4
type 'unsigned long int', size in bytes = 8
type 'float', size in bytes = 4
type 'double', size in bytes = 8
type 'long double', size in bytes = 16
```

For more examples of usage see [4].

3.2.5. Lagoon

Lagoon defines run-time polymorphic interfaces and classes implementing these interfaces and wrapping the compile-time metaobjects from Mirror and Puddle. While Rubber is more suitable for simple decoupling of reflection-based algorithms from the real types of the metaobjects that the algorithms operate on, Lagoon is full-blown run-time reflection utility that can be even decoupled from the application using it and loaded dynamically on-demand.

This example queries the meta-types reflecting types in the global scope, orders them

by the value of `sizeof` and prints their names:

```
#include <mirror/mirror.hpp>
#include <lagoon/lagoon.hpp>
#include <lagoon/range/extract.hpp>
#include <lagoon/range/sort.hpp>
#include <lagoon/range/for_each.hpp>
#include <iostream>

int main(void)
{
    using namespace lagoon;
    typedef shared<meta_named_scoped_object> shared_mnso;
    typedef shared<meta_type> shared_mt;
    //
    // traverses the range of meta-objects passed as
    // the first argument and on each of them executes
    // the functor passed as the second argument
    for_each(

        // sorts the range passed as the first argument
        // using the functor passed as the second argument
        // for comparison
        sort(

            // extracts only those having the meta_type
            // interface
            extract<meta_type>(

                // gets all members of the global scope
                reflected_global_scope()->members()
            ),

            // compares two meta-types on the value
            // of sizeof(reflected-type)
            [](const shared_mt& a, const shared_mt& b)
            {
                return a->size_of() < b->size_of();
            }
        ),

        // prints the full name of a type
        [](const shared_mt& member)
        {
```

```
            std::cout << member->full_name() << std::endl;
        }
    );
    return 0;
}
```

This application prints the following on the standard output:

```
void
bool
char
unsigned char
short int
unsigned short int
wchar_t
int
long int
unsigned int
unsigned long int
float
double
long double
```

The following example is more complex and shows the usage of Lagoon's object factories, in this case a factory using a text-script similar to C++ uniform initializers to provide input data from which a set of instances is constructed:

```
#include <mirror/mirror_base.hpp>
#include <mirror/pre_registered/basic.hpp>
#include <mirror/pre_registered/class/std/vector.hpp>
#include <mirror/pre_registered/class/std/map.hpp>
#include <mirror/utils/quick_reg.hpp>
#include <lagoon/lagoon.hpp>
#include <lagoon/utils/script_factory.hpp>
#include <iostream>

namespace morse {

// declares an enumerated class for morse code symbols
enum class signal : char { dash = '-', dot = '.' };

// declares a type for a sequence of morse code symbols
typedef std::vector<signal> sequence;

// declares a type for storing morse code entries
typedef std::map<char, sequence> code;
```

```
} // namespace morse

MIRROR_REG_BEGIN

// registers the morse namespace
MIRROR_QREG_GLOBAL_SCOPE_NAMESPACE(morse)
// registers the signal enumeration
MIRROR_QREG_ENUM(morse, signal, (dash)(dot))

MIRROR_REG_END

int main(void)
{
    try
    {
        using namespace lagoon;

        // a factory builder class provided by Lagoon
        // that can be used together with a meta-type
        // to build a factory
        c_str_script_factory_builder builder;

        // a class storing the input data for the factory
        // built by the builder
        c_str_script_factory_input in;

        // the input data for the factory
        auto data = in.data();

        // polymorphic meta-type reflecting the morse::code type
        auto meta_morse_code = reflected_class<morse::code>();

        // a polymorphic factory that can be used to construct
        // instances of the morse::code type, that is built by
        // the builder and the meta-type reflecting morse::code.
        auto morse_code_factory = meta_morse_code->make_factory(
            builder,
            raw_ptr(&data)
        );

        // the input string for this factory
        const char input[] = "{" \
            "{'A', {dot, dash}}," \
```

```
    "{'B', {dash, dot, dot, dot}}," \
    "{'C', {dash, dot, dash, dot}}," \
    "{'D', {dash, dot, dot}}," \
    "{'E', {dot}}," \
    "{'F', {dot, dot, dash, dot}}," \
    "{'G', {dash, dash, dot}}," \
    "{'H', {dot, dot, dot, dot}}," \
    "{'I', {dot, dot}}," \
    "{'J', {dot, dash, dash, dash}}," \
    "{'K', {dash, dot, dash}}," \
    "{'L', {dot, dash, dot, dot}}," \
    "{'M', {dash, dash}}," \
    "{'N', {dash, dot}}," \
    "{'O', {dash, dash, dash}}," \
    "{'P', {dot, dash, dash, dot}}," \
    "{'Q', {dash, dash, dot, dash}}," \
    "{'R', {dot, dash, dot}}," \
    "{'S', {dot, dot, dot}}," \
    "{'T', {dash}}," \
    "{'U', {dot, dot, dash}}," \
    "{'V', {dot, dot, dot, dash}}," \
    "{'W', {dot, dash, dash}}," \
    "{'X', {dash, dot, dot, dash}}," \
    "{'Y', {dash, dot, dash, dash}}," \
    "{'Z', {dash, dash, dot, dot}}," \
    "{'1', {dot, dash, dash, dash, dash}}," \
    "{'2', {dot, dot, dash, dash, dash}}," \
    "{'3', {dot, dot, dot, dash, dash}}," \
    "{'4', {dot, dot, dot, dot, dash}}," \
    "{'5', {dot, dot, dot, dot, dot}}," \
    "{'6', {dash, dot, dot, dot, dot}}," \
    "{'7', {dash, dash, dot, dot, dot}}," \
    "{'8', {dash, dash, dash, dot, dot}}," \
    "{'9', {dash, dash, dash, dash, dot}}," \
    "{'0', {dash, dash, dash, dash, dash}}" \
"}";

// passes the input data to the factory
in.set(input, input+sizeof(input));

// use the factory built above to create
// a new instance of the morse::code type
raw_ptr pmc = morse_code_factory->new_();
```

```
        // cast of the raw pointer returned by the factory
        // to the concrete type (morse::code)
        morse::code& mc = *raw_cast<morse::code*>(pmc);

        // the morse::code type is just a map of char to
        // a vector of morse signals, this prints them
        // to cout in a standard way
        for(auto i = mc.begin(), e = mc.end(); i != e; ++i)
        {
            std::cout << "Morse code for '" << i->first << "': ";
            auto j = i->second.begin(), f = i->second.end();
            while(j != f)
            {
                std::cout << char(*j);
                ++j;
            }
            std::cout << std::endl;
        }

        // uses the meta-type reflecting morse::code to delete
        // the instance constructed by the factory
        meta_morse_code->delete_(pmc);
    }
    catch(std::exception const& error)
    {
        std::cerr << "Error: " << error.what() << std::endl;
    }
    //
    return 0;
}
```

This application has the following output:

```
Morse code for '0': -----
Morse code for '1': .----
Morse code for '2': ..---
Morse code for '3': ...--
Morse code for '4': ....-
Morse code for '5': .....
Morse code for '6': -....
Morse code for '7': --...
Morse code for '8': ---..
Morse code for '9': ----.
Morse code for 'A': .-
Morse code for 'B': -...
```

```
Morse code for 'C': -.-.
Morse code for 'D': -..
Morse code for 'E': .
Morse code for 'F': ..-.
Morse code for 'G': --.
Morse code for 'H': ....
Morse code for 'I': ..
Morse code for 'J': .---
Morse code for 'K': -.-
Morse code for 'L': .-..
Morse code for 'M': --
Morse code for 'N': -.
Morse code for 'O': ---
Morse code for 'P': .--.
Morse code for 'Q': --.-
Morse code for 'R': .-.
Morse code for 'S': ...
Morse code for 'T': -
Morse code for 'U': ..-
Morse code for 'V': ...-
Morse code for 'W': .--
Morse code for 'X': -..-
Morse code for 'Y': -.--
Morse code for 'Z': --..
```

For more examples of usage see [5].

3.3. Class generators

There are situations where the following transformation of scopes (classes, enumerations, etc.) and their members would be very useful. Consider a simple user-defined `structs` `address` and `person`,

```
struct address
{
        std::string street;
        std::string number;
        std::string postal_code;
        std::string city;
        std::string country;
};

struct person
{
```

```
        std::string name;
        std::string surname;

        address residence;

        std::tm birth_date;
};
```

and an object-relational mapping (ORM) library, that would allow automatic generation of SQL queries from strongly typed expressions in a DSEL in C++. It would be advantageous to have some counterparts for all "ORM-aware" classes having members with the same names as the original class, but with different types, like:

```
template <class T>
struct orm_table;

template <>
struct orm_table<address>
 : public base_table
{
        orm_column<std::string> street;
        orm_column<std::string> number;
        orm_column<std::string> postal_code;
        orm_column<std::string> city;
        orm_column<std::string> country;

        orm_table(orm_param& param)
         : base_table(param)
         , street(this, param)
         , number(this, param)
         , postal_code(this, param)
         , city(this, param)
         , country(this, param)
        { }
};

template <>
struct orm_table<person>
 : public base_table
{
        orm_column<std::string> name;
        orm_column<std::string> surname;
        orm_column<address> residence;
        orm_column<std::tm> birth_date;
```

```
        orm_table(orm_param& param)
         : base_table(param)
         , name(this, param)
         , surname(this, param)
         , residence(this, param)
         , birth_date(this, param)
        { }
};
```

Generating such or similar classes can also be achieved with reflection. The Mirror library implements the `by_name` metafunction template and the `class_generator` utility for this purpose.

The *Puddle* layer, described above, uses this functionality and allows access to metadata reflecting member variables of a class or free variables of a namespace through the overloaded `operator ->` of a meta-class or meta-namespace:

```
auto meta_person = puddle::reflected_type<person>();
// access the metavariable reflecting
// the birth_date member of person
assert(meta_person->birth_date().access_type().is_public());
// access the metadata for person::name
// and person::surname by their names
assert(
        meta_person->name() ==
        meta_person.member_variables().at_c<0>()
);
assert(
        meta_person->surname() !=
        meta_person.member_variables().at_c<0>()
);
```

This functionality could be extended to any scope member and the mechanism is described below.

3.4. Compile-time vs. Run-time reflection

Run-time, dynamic reflection facilities may seem more readily usable, but with the increasing popularity of compile-time metaprogramming, the need for compile-time introspection (already taken care of by `type_traits`) and reflection also increases.

Also, if compile-time reflection is well supported it is relatively easy to implement run-time or even dynamically loadable reflection on top of it. The oposite is not true: One cannot use run-time metaobjects or the value returned by their member functions as template parameters or compile-time constants.

From the performance point of view, algorithms based on static meta-data offer much more possibilities for the compiler to do optimizations.

Thus, taking shortcuts directly to run-time reflection, without compile-time support has obvious drawbacks.

3.5. Annotations and relations

Strict adhering to the principle of *Ontological correspondence* can pose a problem and decrease the usefullness of reflection in certain cases. Probably the most important case in C++ is the reflection of containers which are not implemented as first-class citizens but rather by libraries.

The principle of ontological correspondence says that the meta-level facilities should not invent metaobjects that do not correspond to base-level language features. In C++ where containers are basically regular classes internally implementing a data structure that is not standardized and is platform-specific and having only a public interface defined by the standard, automated reflection-based implementation of operations like serialization, de-serialization, and others may become complicated.

The purpose of serialization (for example as a part of RPC) may be to convert an instance of a container class into an external representation that can be used to transfer the instance (for example via network) to a machine running the receiving application, but having a different OS, compiler, using a different implementation of the standard library and thus a different implementation of the container class.

If the serialization algorithm would use the description of the internal (non-standard vendor-specific) structure of a class, then the receiving application would not be able to restore the instance, because the internal structure of the class would be different.

Because of this a high-level mechanism is required, that would allow reflection-based metaalgorithms to handle such cases.

One possible option is to break the principle of correspondence and add new concepts for metaobjects allowing for example "high-level" traversal or insertion of elements in an arbitrary container. This approach has its advantages and could be implemented for such special classes as containers, but is unsystematic.

Another option is to allow the base-level constructs to be annotated and let the metaobjects to provide these annotations to the metaalgorithms.

These annotations should take the form of *tags*: identifiers assigned to base-level constructs, like types, variables, class members, parameters, etc. and binary directional *relations* between two language constructs, for example a relation between a class member and a constructor parameter initializing the class member, a relation between (a pair of) container element traversal functions (like begin and end in std. containers) and functions or constructors doing element insertion into the same container class.

4. Technical Specifications

We propose that the basic metadata describing a program written in C++ should be made available through a set of *anonymous* classes defined by the compiler. These classes should describe various program constructs like, namespaces, types, typedefs, classes, their member variables (member data), member functions, inheritance, templates, template parameters, enumerated values, etc.

The compiler should generate metadata for the program constructs defined in the currently processed translation unit. Indexed sets of metaobjects, like scope members, parameters of a function, etc. should be listed in the order of appearance in the processed source code.

Since we want the metadata to be available at compile-time, different base-level constructs should be reflected by *"statically" different* metaobjects and thus by *different* types. For example a metaobject reflecting the global scope namespace should be a different *type* than a metaobject reflecting the `std` namespace, a metaobject reflecting the `int` type should have a different type then a metaobject reflecting the `double` type, a metaobject reflecting `::foo(int)` function should have a different type than a metaobject reflecting `::foo(double)`, function, etc.

In a manner of speaking these special types (metaobjects) should become "instances" of the meta-level concepts (static interfaces which should not exist as concrete types, but rather only at the "specification-level" similar for example to the iterator concepts). This section describes a set of metaobject concepts, their interfaces, tag types for metaobject classification and functions (or operators) providing access to the metaobjects.

4.1. Metaobject Concepts

This section describes the requirements that various metaobjects need to satisfy in order to be considered models of the individual concepts.

4.1.1. Categorization and Traits

In order to provide means for distinguishing between regular types and metaobjects the `is_metaobject` trait should be added and should "return" `true_type` for metaobjects (types defined by the compiler providing metadata) and `false_type` for non-metaobjects (native or user defined types).

The `metaobject_traits` structure should be defined to provide categorization and additional information about the interface of metaobjects.

```
template <typename Metaobject>
struct metaobject_traits
```

```
{
        typedef typename Metaobject::category category;

        typedef integral_constant<bool, ...> has_name;

        typedef integral_constant<bool, ...> has_scope;

        typedef integral_constant<bool, ...> is_scope;

        typedef integral_constant<bool, ...> is_class_member;

        typedef integral_constant<bool, ...> has_template;

        typedef integral_constant<bool, ...> is_template;
};
```

The meaning of the individual trait typedefs is following:

- `category` Is one of the following types and specifies the category of the metaobject:
  - `specifier_tag` indicates a *Specifier*.
  - `namespace_tag` indicates a *Namespace*.
  - `global_scope_tag` indicates the *GlobalScope*.
  - `type_tag` indicates a *Type*.
  - `typedef_tag` indicates a *Typedef*.
  - `class_tag` indicates a *Class* or a *Template* class.
  - `function_tag` indicates a *Function* or a *Template* function.
  - `constructor_tag` indicates a *Constructor*.
  - `operator_tag` indicates an *Operator*.
  - `overloaded_function_tag` indicates an *OverloadedFunction*.
  - `enum_tag` indicates an *Enum*.
  - `inheritance_tag` indicates an *Inheritance*.
  - `constant_tag` indicates an *NamedConstant*.
  - `variable_tag` indicates a *Variable*.
  - `parameter_tag` indicates a *Parameter*.
- `has_name` indicates that the reflected object is *Named*. By default it is defined as `false_type` unless specified otherwise in the concept description below.

- `has_scope` indicates that the reflected object is *Scoped*. By default it is defined as `false_type` unless specified otherwise in the concept description below.
- `is_scope` indicates that the reflected object is a *Scope*. By default it is defined as `false_type` unless specified otherwise in the concept description below.
- `is_class_member` indicates that the reflected object is a *ClassMember*. By default it is defined as `false_type` unless specified otherwise in the concept description below.
- `has_template` indicates that the reflected function or class is a template *Instantiation*. By default it is defined as `false_type` unless specified otherwise in the concept description below.
- `is_template` indicates that the reflected object is function or class *Template*. By default it is defined as `false_type` unless specified otherwise in the concept description below.

4.1.2. String

String is a stateless (or monostate) `class` that represents a compile-time character string constant storing for example a name of a type, function, namespace, etc. or the keyword of a specifier. It allows compile-time metaprograms to examine and make decisions based on the value of such strings. If necessary, the stored string can be returned as a regular C-string. See for example the Mirror's compile-time strings [6].

One of the use-cases for these string is the filtering of scope members based on their names if a good naming policy is consistently applied. For example: filter out all scope members whose name starts with an underscore, or process only classes with names starting with `DB`, etc.

Types conforming to this concept must implement the following:

- `static const char* c_str(void);` static member function returning the static string as a regular null-terminated C-string.
- `static integral_constant<int,` *number-of-characters*`> size(void);` static member function returning the number of characters in the string (obviously without counting any terminating character).
- `static integral_constant<char,` *i-th-character*`> at(integral_constant<int,` i `>);` overloaded member function defined for for $i \in \{0, 1, \ldots, n-1\}$;$n$ = *number-string-characters*, each overload returning the respective character in the string.

4.1.3. Metaobject

Metaobject is a stateless (or monostate) anonymous `struct` that provides metadata reflecting certain program features and has the following properties:

- For every *Metaobject* the `is_metaobject` trait returns `true_type`.
- For every *Metaobject* the `metaobject_traits` structure is defined.
- For every *Metaobject* the `typedef Metaobject::category` is defined and has the same meaning as `metaobject_category<Metaobject>::category`.

The exact type of a specific *Metaobject* reflecting a specific program feature is not defined by the standard, instances of metaobjects should be always declared through the `auto` type specifier.

All instances (in the classical sense) of a concrete *Metaobject* (i.e all instances of the concrete anonymous type satysfying the requirements of the *Type* concept reflecting for example the `int` type) should be equal to the programmer.

Instances (in the classical sense) of two different metaobjects (like an instance of the concrete anonymous type satysfying the requirements of the *Type* concept reflecting the `int` type and an instance of the concrete anonymous type satysfying the requirements of the *Type* concept reflecting the `double` type) of course can (and will) be different.

```
// for all purposes these two instances of (Meta)Type
// should be equal and interchangable without any change
// to the behavior of the program
auto meta_int_1 = reflected<int>();
auto meta_int_2 = reflected<int>();
```

4.1.4. Specifier

Specifier is a *Metaobject*, which reflects specifiers like `const`, `volatile`, `private`, `protected`, `public`, `virtual`, etc. and has the following requirements:

- `static` *String* `keyword(void);` returns the keyword of the reflected specifier. If `category` is `spec_none_tag` then `keyword` returns "" (an empty c-string).
- `typedef Category category;` is defined as one of the following types:
  - `spec_none_tag` a category for missing specifiers, for example a non-const member function would have a `spec_none_tag` constness specifier or a variable with automatic storage class would have a `spec_none_tag` storage class specifier, etc.
  - `spec_extern_tag` indicates `extern` storage class / linkage.
  - `spec_static_tag` indicates `static` storage class / linkage.

- `spec_mutable_tag` indicates `mutable` storage class / linkage.
- `spec_register_tag` indicates `register` storage class / linkage.
- `spec_thread_local_tag` indicates `thread_local` storage class / linkage.
- `spec_const_tag` indicates `const` member functions.
- `spec_virtual_tag` indicates `virtual` inheritance or function linkage.
- `spec_private_tag` indicates `private` member access.
- `spec_protected_tag` indicates `protected` member access.
- `spec_public_tag` indicates `public` member access.
- `spec_class_tag` indicates the `class` elaborated type specifier.
- `spec_struct_tag` indicates the `struct` elaborated type specifier.
- `spec_union_tag` indicates the `union` elaborated type specifier.
- `spec_enum_tag` indicates the `enum` elaborated type specifier.

4.1.5. Named

Named is a *Metaobject* reflecting program constructs, which have a name, like namespaces, types, functions, variables, etc. and has the following requirements:

- `static` *String* `base_name(void);` member function that returns the base name of the reflected construct, without the nested name specifier. For namespace `std` this function should return "std", for namespace `foo::bar::baz` this function should return "baz", for the global scope this function should return "" (an empty c-string literal).
  For `std::vector<int>::iterator` it should return "iterator". For derived and qualified types like
  `volatile std::vector<const foo::bar::fubar*> * const *` it should return "volatile vector<const fubar*> * const *", etc. The string returned by this function is owned by the function and should not be freed by the caller. Alternatively this member function could be called `identifier`.
- `metaobject_traits<Named>::has_name` is defined as `true_type`.

4.1.6. Scoped

Scoped is a *Metaobject* reflecting program constructs, which are defined inside a scope (global scope, namespace, class, etc.). *Scoped* metaobjects have the following requirements:

- `static Scope scope(void);` static member function returning a *Scope* metaobject reflecting the scope of the scoped object. In concrete metaobjects the result can be a *Namespace*, *Class*, etc.
  The `metaobject_traits<decltype(Scoped::scope())>::category` typedef can be used to query the kind of the scope.

- `metaobject_traits<Scoped>::has_scope` is defined as `true_type`.

4.1.7. Named and Scoped

Many of the concepts described below are specializations of both the *Scoped* and *Named* concepts. Metaobjects conforming to these concepts have the following additional requirements:

- `static` *String* `full_name(void);` member function that returns the full name of the reflected construct, with the nested name specifier. For namespace `std` this function should return "std", for namespace `foo::bar::baz` this function should return "foo::bar::baz", for the global scope this function should return "" (an empty c-string literal).
  For `std::vector<int>::iterator` it should return "std::vector<int>::iterator". For derived and qualified types like
  `volatile std::vector<const foo::bar::fubar*> * const *` it should return "volatile std::vector<const foo::bar::fubar*> * const *", etc. For some metaobjects this function may return the same value as the `base_name` function. The string returned by this function is owned by the function and should not be freed by the caller.

- `named_typedef` nested member template struct defined as in the following pseudocode:

```
template <typename X>
struct named_typedef
{
        typedef X <NAME>;
};
```

  The `<NAME>` expression above would be replaced by the name of the reflected named scoped object. This structure could be used to generate new classes with member typedefs having the same names as the members of the scope of the named object reflected by this *Named*, *Scoped* metaobject. One way to combinine the `<NAME>` typedefs from various *Named* and *Scoped* scope members into a single class would be to let the class inherit from multiple `named_typedef`s from the metaobjects obtained by reflection.

- `named_mem_var` nested member template struct defined as in the following pseudocode:

```
template <typename X>
struct named_mem_var
{
        X <NAME>;

        named_mem_var(void) = default;

        template <class Parent, class Param>
        named_mem_var(Parent& parent, Param param)
         : <NAME>(parent, param)
        { }
};
```

The `<NAME>` expression above would be replaced by the name of the reflected named scoped object. This structure could be used to generate new classes with member variables having the same names as the members of the scope of the named object reflected by this *Named*, *Scoped* metaobject. The member variable `<NAME>` could be default constructible or constructible from two parameters; a reference to the generated class to which the member variable will belong and an application specific parameter. One way to combinine the `<NAME>` member variables from various *Named* and *Scoped* scope members into a single class would be to let the class inherit from multiple `named_mem_var`s from the metaobjects obtained by reflection.

4.1.8. Scope

Scope is a *Named* and *Scoped* metaobject, which reflects scopes like namespaces, classes, enums, etc. *Scope* has the following requirements:

- `static integral_constant<int,` *number-of-scope-members* `> member_count(void);` static member function returning the total number of various members like types, namespaces, functions, variables, etc. defined inside the scope reflected by a *Scope*.
- `static` *Scoped* `member(integral_constant<int,` i`>);` overloaded member function defined for $i \in \{0, 1, \ldots, n-1\}$;$n =$ *number-of-scope-members*, each overload returns a different *Scoped* metaobject reflecting the i-th member defined inside the scope reflected by a *Scope*. In concrete metaobjects reflecting various kinds of scopes the `member(...)` function can return metaobjects like *Namespace*, (*ClassMember*) *Variable*, (*ClassMember*) *OverloadedFunction*, *Typedef*, *Enum*, etc.
- `metaobject_traits<Scope>::is_scope` is defined as `true_type`.

4.1.9. Namespace

Namespace is a *Scope* with the following requirements:

- `metaobject_traits<Namespace>::category` is defined as `namespace_tag`.

4.1.10. GlobalScope

GlobalScope is a *Namespace* reflecting the global scope and requires the following:

- `metaobject_traits<GlobalScope>::category` is defined as `global_scope_tag`.

4.1.11. Type

Type is a *Named* and *Scoped* metaobject which has the following requirements:

- `typedef` *original-type* `original_type;` member typedef defined as the original type reflected by the *Type*.
- `metaobject_traits<Type>::category` is defined as `type_tag`.

The `is_template` typedef in `metaobject_traits` changes the requirements in the concepts derived from *Type*.

4.1.12. Typedef

Typedef is a *Type* metaobject that reflects typedefs, i.e. types that were defined as alternate names for another types. *Typedef* has the following requirements:

- `static` *Type* `type(void);` static member function returning a *Type* reflecting the "source" type of the typedef. In concrete *Typedef* metaobjects `type` can return a *Type*, *Class*, *Enum* or *Typedef*.
- `metaobject_traits<Typedef>::category` is defined as `typedef_tag`.

4.1.13. Class

Class is a *Type* and a *Scope* that reflects an elaborated type (class, struct, union) or a class template. *Class* has the following requirements, but the `is_template` typedef in the `metaobject_traits` changes the requirements inherited from the *Type* concept as described below.

- `static` *Specifier* `elaborated_type(void);` static member function returning a *Specifier* reflecting the elaborated type specifier used to define the class (`class`, `struct`, `union`).

- `static integral_constant<int,` *number-of-base-classes* `> base_class_count(void);` static member function returning the total number of base classes that the class reflected by *Class* inherits from.
- `static` *Inheritance* `base_class(integral_constant<int,` i`>);` overloaded member function defined for $i \in \{0, 1, \ldots, n-1\}$; n = *number-of-base-classes*, each overload returns a different *Inheritance* metaobject reflecting the inheritance of the i-th base class of the class reflected by *Class*.
- `metaobject_traits<Class>::category` is defined as `class_tag`.

If `metaobject_traits<Class>::is_template` is `true_type` it indicates that the reflected program feature is not a regular class, but a class template. In such case the `original_type` typedef inherited from *Type* is not defined.

4.1.14. Function

Function is a *Scope* metaobject that reflects a function or a function template and requires the following (the requirements are influenced by the `metaobject_traits<Function>::is_template` typedef as described below):

- `static` *Specifier* `linkage(void);` static member function returning a *Specifier* reflecting the linkage specifier of the function.
- `static` *Type* `result_type(void);` static member function returning a *Type* reflecting the result type of the function.
- `static integral_constant<int,` *number-of-parameters* `> parameter_count(void);` static member function returning the total number of parameters of the function reflected by *Function*.
- `static` *Parameter* `parameter(integral_constant<int,` i`>);` overloaded member function defined for $i \in \{0, 1, \ldots, n-1\}$; n = *number-of-parameters*, each overload returns a different *Parameter* metaobject reflecting the i-th parameter of the function reflected by *Function*.
- `static integral_constant<bool,` *true-or-false*`> throw_limited(void);` static member function indicating whether the reflected function has only a limited list of exceptions that it can throw. If the reflected function was declared with the `throw` exception specification (either empty or with a list of exception types) then `throw_limited` returns `true_type`, otherwise if the function can throw any exception, `throw_limited` returns `false_type`. If `true_type` is returned, then the `exception_count` and `exception` functions can be used to traverse the list of exception types that the reflected function is allowed to throw.
- `static integral_constant<int,` *number-of-exceptions* `> exception_count(void);` static member function returning either the total num-

ber of exceptions that the function reflected by *Function* can throw or returning `integral_constant<int, -1>`. The result type of this function depends on the result of the `throw_limited` member function. If `throw_limited` returns `true_type` then the actual number of allowed exception types is returned. If `throw_limited` returns `false_type` then `integral_constant<int, -1>` is returned to indicate that the reflected function can throw anything.

- `static` *Type* `exception(integral_constant<int,` i`>);` overloaded member function defined for $i \in \{0, 1, \ldots, n-1\}$; n = *number-of-exceptions*, each overload returns a different *Type* metaobject reflecting the i-th exception from the explicit exception specification of the function reflected by *Function*.
- `metaobject_traits<Function>::category` is defined as `function_tag`.

If `metaobject_traits<Function>::is_template` is defined as `false_type` i.e. the reflected feature is not a template but a regular function then the following is also required:

- `static inline ResultType::original_type call(` *parameters...*`);` static inline member function with the same return value type and the same number and type of parameters as the original function reflected by *Function*. Calls to this function should be replaced with the call of the reflected function with the arguments passed to `call`. Additionaly if the reflected function is

  a member function, then the first of the *parameters* of `call` should be a reference to the class where the member function is defined and should be used as the `this` argument when calling the member function. If the member function is declared as `const` then the reference to the class should also be `const`.

If `metaobject_traits<Function>::is_class_member` is defined as `true_type` i.e. the reflected is a member function and not a free function or lambda function, then the following is also required:

- `static` *Specifier* `constness(void);` static member function returning the constness *Specifier* reflecting the constness of the member functions.
- `static integral_constant<bool,` *true-or-false*`> is_pure(void);` static inline member function that returns `true_type` or `false_type` indicating whether the reflected member function is a *pure* virtual function. For non-virtual functions it always returns `false_type`.

Function metaobjects are *not* direct members of scopes. Instead, all functions with the same name (even those that are not overloaded) in a specific scope are grouped into a *OverloadedFunction*. Individual overloaded *Function*s in the group can be obtained through the interface of *OverloadedFunction*. The same should also apply to *Constructor*s and *Operator*s.

The idea is that (direct) scope members (i.e. metaobjects accessible through `Scope::member(...)`) should have unique names.

The *Scope* returned by the `scope` member function of every single *Function* in a *OverloadedFunction* is the same as the `scope` of that *OverloadedFunction*, i.e. the `scope` of a *Function* can be a *Namespace* or a *Class* but *not* a *OverloadedFunction*.

4.1.15. ClassMember

ClassMember is a *Named* and *Scoped* metaobject that reflects a member of a class. It has the following requirements:

- `static` *Specifier* `access_type(void);` static member function returning a *Specifier* reflecting the access type specifier of the class member (`private`, `protected` or `public`).
- `metaobject_traits<ClassMember::scope>::is_class_member` is `true_type`.

Concrete metaobjects that are models of this concept can also be some of the following:

- *Typedef*
- *Class*
- *Enum*
- *OverloadedFunction*

4.1.16. Constructor

Constructor is a *ClassMember* and a *Function* that reflects a constructor and requires the following:

- `metaobject_traits<Constructor>::category` is defined as `constructor_tag`.
- The result of `Constructor::result_type()` is the same as the result of `Constructor::scope()`.

4.1.17. Operator

Operator is a *Function* and possibly a *ClassMember* that reflects an operator and requires the following:

- `metaobject_traits<Operator>::category` is defined as `operator_tag`.

4.1.18. OverloadedFunction

OverloadedFunction is a *Named* and *Scoped* metaobject and possibly a *ClassMember* that reflects a set of overloaded functions, i.e. functions with the same name. *OverloadedFunction* has the following requirements:

- `static integral_constant<int,` *number-of-overloads* `>`
  `overload_count(void);` static member function returning the total number of overloads of the function reflected by *OverloadedFunction*.
- `static` *Function* `overload(integral_constant<int,` i `>);` overloaded member function defined for $i \in \{0, 1, \ldots, n-1\}$; n = *number-of-overloads*, each overload returns a different *Function* metaobject reflecting the i-th overload in the set of functions reflected by *OverloadedFunction*.
- `metaobject_traits<OverloadedFunction>::category` is defined as `overloaded_function_tag`.

4.1.19. Template

Template is a *Function* or a *Class* metaobject that reflects a function or class template. It has the following requirements:

- `static integral_constant<int,` *number-of-template-parameters* `>`
  `template_parameter_count(void);` static member function returning the total number of parameters of the template reflected by *Template*.
- `static` *TemplateParameter*
  `template_parameter(integral_constant<int,` i `>);` overloaded member function defined for $i \in \{0, 1, \ldots, n-1\}$; n = *number-of-parameters*, each overload returns a different *Parameter* metaobject reflecting the i-th parameter of the template reflected by *Template*.
- `template <` *template-parameters...* `>`
  `static Instantiation instantiation(void);` static member template function returning an *Instantiation* reflecting the instantiation of the template with the specified parameters. The *template-parameters* passed to this function must be valid template parameters for the reflected template.
- `metaobject_traits<Template>::is_template` is defined as `true_type`.

4.1.20. TemplateParameter

TemplateParameter is a *Typedef* or a *NamedConstant* that reflects a template parameter. In class templates the types of member variables, typedefs and the return value type and parameters of member functions may be *TemplateParameter* metaobjects.

- `static integral_constant<int,` *position-of-parameter*`> postion(void);` static member function returning the postion of the template parameter.
- The `scope` member function inherited from *Scoped* returns a *Template* reflecting the template which defined this template parameter.
- `metaobject_traits<TemplateParameter>::is_template` is defined as `true_type`.

The `metaobject_traits<TemplateParameter>::category` typedef should be used to distinguish between type and non-type template parameters.

4.1.21. Instantiation

Instantiation is a *Function* or *Class* metaobject that reflects a templated function or class for which the following is required:

- `static` *Template* `model(void);` static member function returning a *Template* reflecting the template that the class or function (reflected by an *Instantiation*) is an instantiation of.
- `metaobject_traits<Instantiation>::has_template` is defined as `true_type`. This trait is used to distinguish classes and functions which are instantiations of a template from non-templated classes and functions.

4.1.22. Enum

Enum is a *Type* and a *Scope* that reflects an enumerated type with the following requirements:

- `metaobject_traits<Enum>::category` is defined as `enum_tag`.
- The `members` of *Enum* are only *NamedConstant* metaobjects.

4.1.23. Inheritance

Inheritance is a *Metaobject* that is reflecting class inheritance and has the following requirements:

- `static` *Specifier*`access_type(void);` static member function returning a access-type *Specifier* that reflects the inheritance access type (private, protected or public).
- `static` *Specifier*`inheritance_type(void);` static member function returning an inheritance-type *Specifier* that reflects the inheritance access type (virtual or non-virtual).
- `static` *Class* `base_class(void);` static member function returning a *Class* reflecting the base class in the inheritance.
- `static` *Class* `derived_class(void);` static member function returning a *Class* reflecting the derived class in the inheritance.
- `metaobject_traits<Inheritance>::category` is defined as `inheritance_tag`.

4.1.24. Variable

Variable is a *Named* and *Scoped* metaobject and possibly a *ClassMember*, that reflects some kind of variable defined in a namespace, class, function, etc. and has the following requirements:

- `static` *Specifier* `storage_class();` static member function returning a storage-class *Specifier* reflecting the storage class of the variable.
- `static` *Type* `type();` static member function returning a *Type* reflecting the type of the variable.
- `metaobject_traits<Variable>::category` is defined as `variable_tag`.

4.1.25. Parameter

Parameter is a *Variable* that reflects a parameter of a function. The following is required for metaobjects reflecting parameters:

- `static integral_constant<int,` *position-of-parameter*`> position(void);` static member function returning the position of the parameter in the function parameter list declaration.
- `metaobject_traits<Parameter>::category` is defined as `parameter_tag`.
- The `scope` member function inherited from *Scoped* returns the *Function* that the parameter belongs to.

4.1.26. NamedConstant

NamedConstant is a *Named* and possibly *Scoped* metaobject reflecting named compile-time constant values like the non-type template parameters and enumeration values.

- `static integral_type<`*value-type*`,` *constant-value*`> value(void);` static member function returning the reflected value wrapped in `integral_constant`.
- `metaobject_traits<NamedConstant>::category` is defined as `constant_tag`.

4.2. Reflection

The metaobjects can be provided either via a set of overloaded functions defined in the `std` namespace or by a new operator. Both of these approaches have advantages and disadvantages.

4.2.1. Reflection functions

In this approach at least two functions should be defined in the `std` namespace:

- *unspecified-type* `reflected_global_scope(void);` (or alternatively `mirrored_global_scope()`) This function should return a type conforming to the *GlobalScope* concept, reflecting the global scope. The real type of the result is not defined by the standard, i.e. it is an implementation detail. If the caller needs to store the result of this function the `auto` type specifier should always be used.
- `template <typename Type>`
  unspecified-type `reflected(void);` (or alternativelly `mirrored<Type>()`) This function should return a type conforming to the *Type* concept, reflecting the `Type` passed as template argument to this function. The real type of the result is not defined by the standard, i.e. it is an implementation detail. If the caller needs to store the result of this function the `auto` type specifier should always be used.

Several other similar functions could be added to the list above for reflection of templates, enumerated values, etc. without defining new rules for what regular function and template parameters can be. The advantages of using reflection functions are following:

- No need to add a new keyword to the language.
- Reduced chance of breaking existing code. The `reflected_global_scope()` and `reflected<Type>()` (nor `mirrored_global_scope()` and `mirrored<Type>()`) functions are currently not defined in the `std` namespace and therefore should not clash with existing user code.

This approach has the following disadvantages:

- Less direct reflection. Using this approach it is not possible (at least without adding new rules for possible values of template and function parameters) to reflect constructors, overloaded functions and some other things.

4.2.2. Reflection operator

In this approach a new operator (we suggest the name) `mirrored(`*param*`)` (or `reflected(`*param*`)`) should be added (for additional alternatives see below). Depending on *param*, which could be a type name, namespace name, template name, overloaded function name, enumerated value name, etc. the operator should return a *Named* metaobject reflecting the specified feature. If the parameter is ommited a type conforming to the *GlobalScope* metaobject concept should be returned. The exact types returned by the operators should be implementation details and if the result needs to be stored in a variable the `auto` type specifier should always be used. For example:

```
//
typedef integral_constant<int, 0> _0;
```

```
typedef integral_constant<int, 1> _1;
typedef integral_constant<int, 2> _2;
typedef integral_constant<int, 3> _3;
//
// reflect the global scope
// meta_gs conforms to the GlobalScope concept
auto meta_gs = mirrored();

static_assert(
        decltype(meta_gs.member_count())::value > 0,
        "The global scope has no members!"
);

static_assert(
        decltype(meta_gs.base_name().size())::value == 0,
        "Name of the global scope is not an empty string!"
);


//
// reflect the std namespace
// meta_std conforms to the Namespace concept
auto meta_std = mirrored(std);

static_assert(
        is_same<
                decltype(meta_gs),
                decltype(meta_std.scope())
        >::value,
        "Namespace std is not in the global scope!"
);

static_assert(
        decltype(meta_std.base_name().size())::value == 3,
        "Name of the std namespace does not have 3 characters!"
);

static_assert(
        decltype(meta_std.base_name().at(_0))::value == 's',
        "Name of the std namespace does not start with 's'!"
);

assert(strcmp(meta_std.base_name().c_str(), "std") == 0);
```

```
//
// reflect the errno variable
// meta_errno conforms to the Variable concept
auto meta_errno = mirrored(errno);

//
// reflect the int type
// meta_int conforms to the Type concept
auto meta_int = mirrored(int);

//
// reflect the std::string typedef
// meta_std_string conforms to the Typedef concept
auto meta_std_string = mirrored(std::string);

//
// reflect the std::map template
// meta_std_map conforms to the Template
// and Class concepts
auto meta_std_map = mirrored(std::map);

//
// reflect the std::map<int, std::string> type
// meta_std_map_int_std_string conforms to Class
// and Instantiation concepts
auto meta_std_map_int_std_string =
        mirrored(std::map<int, std::string>);

//
// reflect the std::string's (overloaded) constructors
// meta_std_string_string conforms to
// the OverloadedFunction concept and the individual
// overloads that it allows to traverse conform
// to the Constructor concept
auto meta_std_string_string =
        mirrored(std::string::string);

//
// reflect ths std::string's copy constructor
// meta_std_string_string_copy conforms to
// the Constructor concept
auto meta_std_string_string_copy =
        mirrored(std::string::string(const std::string&));
```

```
//
// reflect the std::swap overloaded free function
// meta_std_swap conforms to OverloadedFunction
auto meta_std_swap = mirrored(std::swap);

// reflect the (local) variable i
// meta_i conforms to Variable
int i = 42;
auto meta_i = mirrored(i);
```

Using a new operator has the following advantages:

- More direct reflection. Even features that could not be reflected by using a (templated) function could be reflected with an operator.
- More consistent reflection. Everything is reflected with a single operator.

and these disadvantages:

- Requires a new keyword or the usage of an existing keyword in a new context or the usage of a character sequence that is currently invalid.
- Increased risk of breaking existing code. Could be resolved by using an existing operator like `%`, `|`, etc., or the use of a currently invalid character or character sequence like `@`, `$` or the usage of a new set of quotations like ‘ (backtick character). For example:

```
// instead of:
auto meta_std_string = mirrored(std::string);
// use
auto meta_std_string = %std::string;
// or
auto meta_std_string = |std::string;
// or
auto meta_std_string = @std::string;
// or
auto meta_std_string = ‘std::string‘;
```

  The problem with these may be the reflection of the global scope, which when using some of the above would result in awkward expressions like:

```
// instead of
auto meta_gs = mirrored();
// use
auto meta_gs = %;
auto meta_gs = |;
auto meta_gs = @;
```

```
// or
auto meta_gs = ``;
```

5. Impact On the Standard

The impact on the standard and the existing applications depends mainly on the method of reflection (functions vs. operators). Reflection functions pose a very small risk of breaking existing standard-conforming code. The `mirrored` operator on the other hand has a considerable potential of breaking existing applications. This can be alleviated by using existing keywords like `%` as suggested above.

Since compilers already have all the metadata required to generate the proposed metaobjects, making them available to programmers should not pose a big problem to the compiler vendors.

TODO: to be revised/completed

6. Implementation hints

6.1. Generation of metaobjects

The metaobjects should be generated / instantiated by the compiler only when explicitly requested. This also applies to members of the metaobjects. For example when a *Namespace* reflecting the `std` namespace is generated the individual `member(...)` functions (and the resulting metaobjects) should *not* be generated automatically unless the `Scope::member(...)` function is called or its type queried (by `decltype` or otherwise).

This should probably improve the compilation times and avoid reflection-related overhead when reflection is not used.

7. Unresolved Issues

- *Normalization of names returned by* `Named::base_name()` *and* `Named::full_name()`*:* The strings returned by the `base_name` and `full_name` functions should be implementation-independent and the same on every platform/compiler.
- *The syntax of annotation of base-level program constructs with tags and relations.*
- *Explicit specification of what should be reflected.* It might be useful to have the ability to explicitly specify either what to reflect or what to hide from reflection. For example the "whitelisting" (explicitly specifying of what should be reflected) of

namespace or class members could simplify reflective meta-algorithms so that they would not have to implement complicated filters when traversing scope members, to hide implementation details and to improve compilation times. It is important that this functionality is decoupled from the scope member declarations, since it would allow applications to cherry-pick what should be reflected even in third-party libraries. This might be a separate feature, but it also could be merged with the tagging functionality. Applications could tag program constructs in which they are interested and apply a simple filter based on the tag(s) when traversing scope members.

- *The use of "generalized attributes" for annotations.* Would it be possible (and wise) to use the generalized attributes (from N2761) to annotate program features as suggested in subsection 3.5?

8. Acknowledgements

9 References

[1] Mirror C++ reflection utilities (C++11 version), `http://kifri.fri.uniza.sk/~chochlik/mirror-lib/html/`.

[2] Mirror - Examples of usage, `http://kifri.fri.uniza.sk/~chochlik/mirror-lib/html/doxygen/mirror/html/examples.html`.

[3] Mirror - The Puddle layer - examples of usage, `http://kifri.fri.uniza.sk/~chochlik/mirror-lib/html/doxygen/puddle/html/examples.html`.

[4] Mirror - The Rubber layer - examples of usage, `http://kifri.fri.uniza.sk/~chochlik/mirror-lib/html/doxygen/rubber/html/examples.html`.

[5] Mirror - The Lagoon layer - examples of usage, `http://kifri.fri.uniza.sk/~chochlik/mirror-lib/html/doxygen/lagoon/html/examples.html`.

[6] Mirror - Compile-time strings `http://kifri.fri.uniza.sk/~chochlik/mirror-lib/html/doxygen/mirror/html/d5/d1d/group__ct__string.html`.

A. Examples of metaobjects

This appendix contains several examples of how the generated metaobjects should look like for various base-level program constructs.

```
auto meta_int = mirrored(int);
```