Document number: N4452
Date: 2015-04-11
Project: Programming Language C++, SG7, Reflection
Reply-to: Matúš Chochlík(chochlik@gmail.com)

A case for strong static reflection

Matúš Chochlík

Abstract In N3996 [1] and N4111 [2] we proposed the design and some hints on possible implementation of a compile-time reflection facility for standard C++. N3996 contained list of possible use-cases and a discussion about the usefulness of reflection. During the presentation of N4111 concerns were expressed about the level-of-detail and scope of the presented proposal and possible dangers of giving non-expert language users a *too powerful* tool to use and dangers for the implementers of this proposal. Examples and more detailed description of use cases were called for. This paper aims to address these issues.

Contents

1. Introduction

There is a wide range of computer programming scenarios in which the programmer has to (or had to) manualy implement some generic boilerplate code doing the same thing on multiple different types (or other language constructs, like namespaces, constructors, etc.). For example accessing member variables or calling free or member functions and operators in an uniform manner, converting data between the language's intrinsic representation and external formats, implementing delegation or some of the other established design patterns, etc.

With the right tools most of these tasks could be properly algorithmized, encapsulated into some form of parametric *'callable'* and then invoked with appropriate arguments.

The C/C++ preprocessor and C++'s templates were devised to help dealing with some of such cases and later gave rise to particular forms of metaprogramming, which have since become relatively popular. However, while preprocessor and template-based metaprogramming is being put to good use by more and more libraries and applications written in C++, we are starting to hit the limits of these tools.

The limits stem from the fact, that while we can now use the C++'s type system as a (meta)programming language of its own and a C++ compiler as its interpreter and write complicated metaprograms automatically generating a lot of code which we previously had to write manually, the programmer still needlessly has to do a lot of things manually. Just to mention a few; if we want portable type names, we usually have to hardcode them, if we want to create an identifier programmatically we need preprocessor token pasting, we cannot programmatically enumerate all member variables or base classes of a class, all operators working on a particular type, etc.

A C++ compiler processing a translation unit has access to a large amount of very useful metadata, but unfortunately shares only a tiny amount of it with the programmer. Not so long ago the *type-traits* – introspection primitives, many of which rely on some sort of compiler 'magic' (a.k.a. reflection) were introduced. Before that the only 'reflection-facility' in C++ was the `typeid` operator.

The aim of this paper is to show that we can and should be able to get much more information from the compiler and why that would be useful. Since this paper is a follow-up to N3996 and N4111, it also indicates how the metaobjects proposed in N4111 would be usable in the individual use-cases.

2. Use cases

This section describes several common use-cases of reflection ranging from trivial to fairly advanced. These use-cases mostly require only a limited reflection support as proposed in N4451. Appendix A contains several additional use-cases which are less common or require advanced reflection features.

2.1. Portable (type) names

One of the notorious problems of `std::type_info` is that the string returned by its `name` member function is not standardized and is not even guaranteed to return any meaningful, unique human-readable string, at least not without demangling, which is platform specific. Furthermore the returned string is not `constexpr` and cannot be reasoned about at compile-time and is applicable only to types. One other problem with `typeid` that it is not aware of `typedefs`. In some cases we would like to obtain the typedef name, instead of the 'real' name of the type or a class member or function parameter.

The ability to uniquely map any type used in a program to a human-readable, portable, compile-time string has several use-cases described in this paper.

The ***MetaNamed*** concept from N4111 reflects named language constructs and provides the `base_name` and `full_name` metafunctions, returning their basic name without any qualifiers or decorations and a fully-qualified portable type name.

2.2. Logging

When logging the execution of functions (especialy templated ones) it is sometimes desirable to also include the names of the parameter types or even the names of the parameters and other variables.

The best we can do with just the `std::type_info` is the following:

```
#if __PLATFORM_ABC__
std::string demangled_type_name(const char*) { /* implementation 1 */ }
#else if __PLATFORM_MNO__
std::string demangled_type_name(const char*) { /* implementation 2 */ }
#else if __PLATFORM_XYZ__
std::string demangled_type_name(const char*) { /* implementation N */ }
#else
std::string demangled_type_name(const char* mangled_name)
{
    // don't know how to demangle this; let's try our luck
    return mangled_name;
```

```
}
#endif

template <typename T>
T min(const T& a, const T& b)
{
    log()   << "min<"
            << demangled_type_name(typeid(T).name())
            << ">(" << a << ", " << b << ") = ";

    T result = a<b?a:b;

    log()   << result << std::endl;

    return result;
}
```

Which may or may not work, depending on the platform.

With the help of reflection as proposed in N4111 we could do:

```
template <typename T>
T min(const T& a, const T& b)
{
    log()   << "min<"
            << full_name<mirrored(T)>()
            << ">(" << a << ", " << b << ") = ";

    T result = a<b?a:b;

    log()   << result << std::endl;

    return result;
}
```

The `__PRETTY_FUNCTION__` macro generated by the compiler could be also used in this case, but the format of the string which this macro expands into is not customizable (which may be necessary for logs formatted in XML, JSON, etc.

A more elaborated output containing also the parameter names could be achieved by using reflection:

```
template <typename T>
T min(const T& a, const T& b)
{
    log()   << "function: min<"
            << full_name<mirrored(T)>()
```

```
            << ">"
        << std::endl
        << base_name<mirrored(a)>() << ": "
        << a << std::endl
        << base_name<mirrored(b)>() << ": "
        << b << std::endl;

    T result = a<b?a:b;

    log()   << base_name<mirrored(result)>() << ": "
        << b << std::endl;

    return result;
}
```

It is true that the lines:

```
        << base_name<mirrored(a)>() << ": "
        << base_name<mirrored(b)>() << ": "
```

could be replaced by preprocessor stringization or just hard coded strings, like

```
        << BOOST_PP_STRINGIZE(a) << ": "
        << BOOST_PP_STRINGIZE(b) << ": "
```

or

```
        << "a: "
        << "b: "
```

but the compiler would not force the programmer to change the macro parameter or the content of the string the if the parameters `a` and `b` were renamed for example to `first` and `second`. If would enforce the change if reflection was used.

Furthermore, if the ***MetaFunction*** concept was implemented and if it was possible to reflect the *'current function'* (i.e. to get a ***MetaFunction*** from inside of a function body via some invocation of the reflection operator), then even more would be possible; The function name and even the parameter names could be obtained from reflection and encapsulated into a function.

```
template <typename MetaFunction, typename ... P>
void log_function_exec(MetaFunction, const std::tuple<P&...>& params)
{
    log()   << "function: "
        << base_name<MetaFunction>()
        << std::endl;

    // obtain the MetaParameter(s) from the MetaFunction
```

```
    // and print them pairwise with the values from params.
    for_each<parameters<MetaFunction>>(
        [&params](auto meta_param)
        {
            typedef decltype(meta_param) MP;
            log()  << base_name<MP>() << ": "
                   << std::get<position<MP>::value>(params)
                   << std::endl;
        }
    );
}

template <typename T>
T min(T a, T b)
{
    log_function_exec(mirrored(this::function), std::tie(a, b));
    /* ... */
}

template <typename T>
T max(T a, T b)
{
    log_function_exec(mirrored(this::function), std::tie(a, b));
    /* ... */
}

template <typename T>
T avg(T a, T b)
{
    log_function_exec(mirrored(this::function), std::tie(a, b));
    /* ... */
}
```

This example used the following features:

- function reflection,
- function parameter reflection,
- context-dependent reflection[1],
- use of metaobject sequences,
- use of the reflection operator,
- base names and the ***MetaNamed*** concept.

[1] See appendix B.6.

2.3. Simple serialization

We need to serialize the instances of selected classes into a structured external format like XML, JSON, XDR or even into a format like Graphviz dot for the purpose of creating a visualization of a static class or dynamic object hierarchy or graph.

Reflection makes this task trivial[2]:

```
template <typename T>
void to_xml(const T& instance, std::true_type atomic)
{
    typedef mirrored(T) MetaType;
    std::cout << "<" << base_name<MetaType>() << ">";
    std::cout << instance;
    std::cout << "</" << base_name<MetaType>() << ">";
}

template <typename T>
void to_xml(const T& instance)
{
    to_xml(instance, std::is_fundamental<T>());
}

template <typename T>
void to_xml(const T& instance, std::false_type atomic)
{
    typedef mirrored(T) MetaType;
    std::cout << "<" << base_name<MetaType>() << ">";

    for_each<base_classes<MetaType>>(
        [](auto meta_inheritance)
        {
            typedef decltype(meta_inhertance) MetaInh;
            typedef original_type<base_class<MetaInh>>::type BT;

            to_xml(const BT&(instance));
        }
    );

    for_each<members<MetaType>>(
        [](auto meta_cls_mem)
        {
            typedef decltype(meta_cls_mem) MetaClsMem;
```

[2] Admittedly this is not the most clever XML schema ever devised, but let's stick to the basics.

```
            typedef original_type<type<MetaClsMem>>::type MT;

            if(std::is_base_of<
                meta_variable_tag,
                metaobject_category<MetaClsMem>
            >())
            {
                auto mvp = pointer<MetaClsMem>::get();
                std::cout << "<" << base_name<MetaClsMem> << ">";
                to_xml(instance.*mvp);
                std::cout << "</" << base_name<MetaClsMem> << ">";
            }
        }
    );

    std::cout << "</" << base_name<MetaType>() << ">";
}
```

Where necessary explicit specializations or function overloads can override the generic implementation:

```
template <typename Bool>
void to_xml(const std::string& instance, Bool)
{
    std::cout << "<string>";
    std::cout << instance;
    std::cout << "</string>";
}
```

This use-case shows the following:

- class member reflection,
- inheritance reflection,
- class member variable reflection,
- use of metaobject sequences,
- use of the interface of various metaobjects,
- use of the reflection operator,
- metaobject categorization,
- base names and the ***MetaNamed*** concept.

2.4. Cross-cutting aspects

We need to execute the same action (or a set of actions) at the entry of or at the exit from the body of a function (from a set of multiple functions meeting some conditions) each time it is called.

The action may be related to logging, debugging, profiling, but also access control, etc. The condition which selects the functions for which the action is invoked might be something like:

- each member function of a particular class,
- each function defined in some namespace,
- each function returning values of a particular type or having a particular set of parameters,
- each function whose name matches a search expression,
- each function declared in a particular source file,
- etc. and various combinations of the above.

It may not be possible to tell in advance the relations between the aspects and the individual functions or these relations may vary for different builds or build configurations. Furthermore we want to be able to quickly change the assignment of actions to functions in one place instead of going through the whole project source which may consists of dozens or even hundreds of files.

We want for example temporarily enable logging of the entry and exit of each member function of class `foo`, or we need to count the number of invocations of functions defined in the `bar` namespace with names not starting with an underscore, or we want to throw the `not_logged_in` exception at the entry of each member function of class `secure` if the global `user_logged_in` function returns `false`.

Without reflection something like this could be implemented in the following way:

```
class logging_aspect
{
public:
    template <typename ... P>
    logging_aspect(const char* func_name, P&&...)
    {
        // write to clog
    }
};

class profiling_aspect
{
```

```
    /* ... */
};

class authorization_aspect
{
public:
    template <typename ... P>
    authorization_aspect(const char* func_name, P&&...)
    {
        if(contains(func_name, "secure"))
        {
            if(!::is_user_logged_in())
            {
                throw not_authorized(func_name);
            }
        }
    }
};

template <typename RV, typename ... P>
class func_aspects
 : logging_aspect
 , profiling_aspect
 , authorization_aspect
/* ... etc. ... */
{
public:
    func_aspects(
        const char* name,
        const char* file,
        unsigned line,
        P&&... args
    ): logging_aspect(name, file, line, args...)
     , profiling_aspect(name, file, line, args...)
     , authorization_aspect(name, file, line, argc...)
    /* ... etc. ... */
    { }
};

template <typename RV, typename ... P>
func_aspects<RV, P...>
make_func_aspects(
    const char* name,
    const char* file,
```

```
    unsigned line,
    P&&...args
);


void func1(int a, int b)
{
    auto _fa = make_func_aspects<void>(
        __func__,
        __FILE__,
        __LINE__,
        a, b
    );
    /* function body */
}

double func2(double a, float b, long c)
{
    auto _fa = make_func_aspects<double>(
        __func__,
        __FILE__,
        __LINE__,
        a, b, c
    );
    /* function body */
}

namespace foo {

long func3(int x)
{
    auto _fa = make_func_aspects<long>(
        __func__,
        __FILE__,
        __LINE__,
        x
    );
    /* function body */
}

} // namespace foo
```

Obviously this is very repetitive and it can get quite tedious and error-prone to supply all this information to the aspects in each function manually. Also if the signature or the

name of the function changes the construction of the `func_aspects` instance must be updated accordingly. With the help of reflection things can be simplified considerably:

```
template <typename MetaFunction, typename Enabled>
class logging_aspect_impl;

template <typename MetaFunction>
class logging_aspect_impl<MetaFunction, false_type>
{ };

template <typename MetaFunction>
class logging_aspect_impl<MetaFunction, true_type>
{
public:
    logging_aspect_impl(void)
    {
        clog
            << base_name<MetaFunction>()
            << "("
            /* ... */
            << ")"
            << endl;
    }
};

template <typename MetaFunction>
struct logging_enabled
 : integral_constant<
    bool,
    is_base_of<mirrored(std), scope<MetaFunction>>() &&
    is_same<std::string, original_type<result<MetaFunction>>::type> &&
    /* ... etc. ... */
>
{ };

template <typename MetaFunction>
using logging_aspect =
    logging_aspect_impl<
        MetaFunction,
        typename logging_enabled<MetaFunction>::type
    >;

template <typename MetaFunction, typename Enabled>
class authorization_aspect_impl;
```

```
template <typename MetaFunction>
class authiorization_aspect_impl<MetaFunction, false_type>
{ };

template <typename MetaFunction>
class authorization_aspect_impl<MetaFunction, true_type>
{
public:
    authorization_aspect_impl(void)
    {
        if(!::is_user_logged_in())
        {
            throw not_authorized(
                full_name<MetaFunction>()
            );
        }
    }
};

template <typename MetaFunction>
struct autorization_enabled
 : integral_constant<
    bool,
    is_base_of<mirrored(foo::bar), scope<MetaFunction>>() &&
    constexpr_starts_with(base_name<MetaFunction>(), "secure_") &&
    /* ... etc. ... */
>
{ };

template <typename MetaFunction>
using authorization_aspect =
    authorization_aspect_impl<
        MetaFunction,
        typename authorization_enabled<MetaFunction>::type
    >;

template <typename MetaFunction>
class func_aspects
 : logging_aspect<MetaFunction>
 , profiling_aspect<MetaFunction>
 , authorization_aspect<MetaFunction>
/* ... etc. ... */
{
```

```
public:
};

void func1(int a, int b)
{
    func_aspects<mirrored(this::function)> _fa;
    /* function body */
}

double func2(double a, float b, long c)
{
    func_aspects<mirrored(this::function)> _fa;
    /* function body */
}

namespace foo {

long func3(int x)
{
    func_aspects<mirrored(this::function)> _fa;
    /* function body */
}

} // namespace foo
```

In this case the same expression is used in all functions regardless of their name and signature and the aspects get all the information they require from the metaobject reflecting the function. All the data obtained from the metaobjects is available at compile-time so various specializations of the aspect classes can be implemented as required.

This same technique could also be used with instances of classes:

```
template <typename MetaClass>
class class_aspects
 : logging_aspects<MetaClass>
/* ... etc. ... */
{
public:
    class_aspects(typename original_type<MetaClass>::type* that);
};

class cls1
{
private:
    int member1;
```

```
    /* ... other members ... */
    class_aspect<mirrored(this::class)> _ca;
public:
    cls1(void)
     : member1(...)
     , _ca(this)
    { }
};
```

Class aspects like these could also be used for logging, monitoring of object instantation, leak detection, etc.

This use-case shows the following:

- current function reflection,
- current class reflection.

2.5. Implementing the factory pattern

The purpose of the *factory* pattern is to separate its caller, who requires a new instance of a *Product* type, from the the details of this instance's construction. The caller only supplies the input data to the factory and waits for the new instance. There are several aspects that need to be considered when designing and implementing a factory.

The input data for the construction of an instance of the *product* can be stored in an external representation (an XML fragment, a RDBS database dataset, a JSON document, etc.) or even entered by the user through a GUI or on the command-line and so on, and would need to be converted into a native C++ representation. The new instance also might be constructed as a copy of another already existing *prototype* instance of the same type.

The product may be polymorphic and the exact type may not even be known to the user. It may have one or several constructors, each of which may require a different set of arguments. It may or may not have constructors with a specific signature, for example a default constructor.

A default constructor does not make sense for many types and requiring it just because the type will be used with a factory is problematic[3]. Consider for example what a "default" instance of `person` or `address` would look like – it would not have any meaning at all. Thus well-designed factories should not depend on the presence of constructors with specific signatures.

Furthermore it might be desirable, that the constructor used to construct a particular instance is picked based on the available input data which is known only at run-time,

[3] or even impossible with third-party code

but not when the factory is designed and implemented.

Let's consider the implementation of a factory for a rather simple `point` class, representing a point in 3-dimensional space:

```
struct point
{
    double _x, _y, _z;

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

    point(double w): _x(w), _y(w), _z(w) { }

    point(void): _x(0.0), _y(0.0), _z(0.0) { }

    point(const point&) = default;

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

A naive hand-coded implementation, of a factory constructing `point`s from some `Data` type (for example an XML node) might look like this:

```
class point_factory
{
private:
    unsigned pick_constructor(Data data)
    {
        // somehow examine the data and pick
        // the most suitable constructor of the point class
    }

    double extract(Data data, string param)
    {
        // somehow extract and convert the value
        // of a named parameter from the data
    }
public:
    point create(Data data)
    {
        switch(pick_constructor(data))
        {
            case 0: return point();

            case 1: return point(extract(data, "w"));
```

```
            case 2: return point(
                        extract(data, "x"),
                        extract(data, "y"),
                        extract(data, "z")
                    );

            default: throw exception(...);
        }
    }
};
```

Now suppose that there is some pool of existing `point` objects and let's extend the factory to use this pool and return copies if applicable:

```
extern pool_of<point>& point_pool;

class point_factory
{
private:
    unsigned pick_constructor(Data data)
    {
        // same as before but also allow the copy
        // constructor to be picked if the data says so
    }

    double extract(Data data, string param);
public:
    point create(Data data)
    {
        switch(pick_constructor(data))
        {
            // same as before, but add a new case
            // returning copies from the pool

            case 3: return point_pool.get(data);

            default: throw exception(...);
        }
    }
};
```

When looking at the hand-coded factories above, it is obvious that implementing and maintaining[4] factories for several dozens of classes in a larger application is a highly repetitive, tedious and possibly error-prone process and at least partial automation is

[4] as the constructed types evolve and change

desirable.

Factory classes must generally handle several tasks which fit into two distinct and nearly orthogonal categories:

- *Product type-related*
  - *Constructor description* – providing the metadata desribing the individual constructors, their parameters, etc.
  - *Constructor dispatching* – calling the selected constructor. with the supplied arguments which results in a new instance of the product type.
- *Input data representation-related*
  - *Input data validation* – checking if the input data match the available constructors.
  - *Constructor selection* – examining the input data, comparing it to the metadata describing product's constructors and determining which constructor should be called.
  - *Getting the argument values* – determining where the argument values should come from and getting them:
    - *Conversion from the external representation* – this usually applies to intrinsic C++ types, but complex types could be converted directly too.
    - *Recursive construction by using another factory* – this usually requires some form of cooperation between the parent and its child factories and it means that all the tasks discussed here must be repeated also for the recursivelly constructed parameter(s).
    - *Copying an existing instance* – for example from an object pool.

Parts from each category can be combined with parts from the other to create new factories which promotes code reusability. Factories constructing instances of a single product from various data representations share the product-related components and factories constructing instances of various product types from a single input data representation share the input-data-related parts. This approach has several advantages like better maintainability or the ability to develop the components separatelly and combine them later via metaprogramming.

If the input data for a metaprogram generating the factory class, i.e. the metadata describing the Product type[5] can be obtained by using compile-time reflection then new factory classes can be generated automatically for nearly arbitrary type privided that the input data type-related parts are implemented.

[5] specifically the constructors of Product

The Mirror library [3] implements two distinct frameworks for generating such factories; one working at compile-time and the other at run-time.

The scope of this proposal does not allow to fully explain the implementation of the factory generators. Please see [4] for further details.

3 References

[1] Chochlík M., N3996 - Static reflection, 2014, https://isocpp.org/files/papers/n3996.pdf.

[2] Chochlík M., N4111 - Static reflection (rev. 2), 2015, https://isocpp.org/files/papers/n4111.pdf.

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

[4] Chochlík M., Implementing the Factory pattern with the help of reflection, (pre-print), http://www.researchgate.net/publication/269668943_IMPLEMENTING_THE_FACTORY_PATTERN_WITH_THE_HELP_OF_REFLECTION.

A. Additional use cases

This sections describes further, less common, use-cases for reflection or use-cases requiring advanced reflection features.

A.1. SQL schema generation

We need to create an SQL/DDL (data definition language) script for creating a schema with tables which will be storing the values of all structures in namespace C++ `foo` having names starting with `persistent_`:

```
const char* translate_to_sql(const std::string& type_name)
{
    if(type_name == "int")
        return "INTEGER";
    /* .. etc. */
}

template <typename MetaMemVar>
void create_table_column_from(MetaMemVar)
{
    if(!std::is_base_of<
```

```
        variable_tag,
        metaobject_category<MetaMemVar>
    >()) return;

    std::cout << base_name<MetaMemVar>() << " ";

    std::cout << translate_to_sql(base_name<type<MetaMemVar>());

    if(starts_with(base_name<MetaMemVar>(), "id_"))
    {
        std::cout << " PRIMARY KEY";
    }
    std::cout << std::endl;
}

template <typename MetaClass>
void create_table_from(MetaClass)
{
    if(!std::is_base_of<
        class_tag,
        metaobject_category<MetaClass>
    >()) return;

    if(!starts_with(
        "persistent_",
        base_name<MetaClass>()
    )) return;

    std::cout << "CREATE TABLE "
              << strip_prefix("persistent_", base_name<MetaClass>())
              << "(" << std::endl;

    for_each<members<MetaClass>>(create_table_column_from);

    std::cout << ");"
}

template <typename MetaNamespace>
void create_schema_from(MetaNamespace)
{
    std::cout << "CREATE SCHEMA "
              << base_name<MetaNamespace>()
              << ";" << std::endl;
```

```
    for_each<members<MetaNamespace>>(create_table_from);
}

int main(void)
{
    create_schema_from(mirrored(foo));
    return 0;
}
```

This example shows the following features from N4111:

- namespace reflection,
- namespace member reflection,
- class member reflection,
- use of metaobject sequence,
- metaobject categorization,
- base names and the ***MetaNamed*** concept.

Furthermore reflection could be used to implement actual object-relational mapping, together with a library like `SOCI`, `ODBC`, `libpq`, etc. See for example [3].

A.2. Source text-based metaprogramming

In scripting languages metaprogramming often takes the form of dynamically creating a new script purely through text operations followed by the execution of that script.

While using this approach is more complicated with compiled languages it is not unheard of. A C++ source can be created by a program in C++, compiled by a compiler (invoked from that same or from a different program) into a shared library and then dynamically loaded and executed.

In some cases this approach could be used to generate source code on a local machine, which is then compiled and executed on a remote machine with a different architecture.

```
class foo64bit
{
/* ... */
};

class foo32bit
{
/* ... */
};
```

```
#if __THIS_IS_64BIT_ARCH
typedef foo64bit default_foo;
#else
typedef foo32bit default_foo;
#endif

struct plugin
{
    virtual void process_foo(default_foo&) = 0;
    /* ... */
};
```

We want to programatically create a new logging[6] implementation of `plugin` and we don't want to rewrite this program every time the interface is updated.

Furthermore it is possible that we will be generating the code on a 32-bit machine and then compiling and executing it on a 64-bit machine or vice-versa.

```
template <typename MetaFunction>
void print_func_impl(MetaFunction)
{
    using std::cout;
    using std::endl;

    // the result can have a typedef-ined type
    // and we want to print here the typedef name
    cout << base_name<result_type<MetaFunction>>() << " ";
    cout << base_name<MetaFunction>() << "(";

    for_each<parameters<MetaFunction>>(
        [](auto meta_param)
        {
            typedef decltype(meta_param) MetaParam;
            if(position<MetaParam>() > 0)
            {
                cout << ", ";
            }
            // the parameter can have a typedef-ined type
            // and we want to print here the typedef name
            cout << base_name<type<MetaParam>>() << " ";
            cout << base_name<MetaParam>();
        }
    );
```

[6] We're again trying to stick to the basics, here. Much more complicated examples could be devised.

```
    cout << base_name<MetaFunction>() << ")";
    cout << " override" << endl;
    cout << "{" << endl;
    cout << "  _do_log_call<MetaFunction>(" << endl;

    for_each<parameters<MetaFunction>>(
        /* Print out a parameter list for the call */
    );

    cout << "  );" << endl;

    /* Print out the rest of the implementation */

    cout << "}" << endl;
}

void main(void)
{
    using std::cout;
    using std::endl;

    cout << "#include <foo/plugin.hpp>" << endl;
    /* etc. */

    cout << "class logging_plugin" << endl;
    cout << " : virtual public plugin" << endl;
    cout << "{" << endl;
    cout << "private:" << endl;
    cout << "  template <typename MetaFunction, typename ... P>" << endl;
    cout << "  void _do_log_call(const P&...);" << endl;
    cout << "public:" << endl;

    for_each<members<mirrored(plugin)>>(print_func_impl);

    cout << "};" << endl;
    return 0;
}
```

The example above could be semi-automated using the preprocessor and (demangled) `type_info::name`. The problem is that `type_info` is not aware of the fact that the `default_foo` parameter type is a typedef and it would instead return either `"foo64bit"` or `"foo32bit"` based on the architecture on which the script was generated.

In N4111 (and subsequent papers) it is proposed that reflection is aware of `typedefs`

and distinguishes between `typedef`s and their *underlying types*.

This example shows the following features:

- typedef reflection,
- class member reflection,
- use of the reflection operator,
- use of the `base_name`, `type` and `original_type` templates.

A.3. Implementing delegation or decorators

We need to create a decorator class, which wraps an instance of another class, implements similar interface as the original class, writes info about each member function call into a log and then delegates the call to the private member object:

```
class foo
{
public:
    void f1(void);

    int f2(int a, int b);

    double f3(float a, long b, double c, const std::string& d);
};

class logging_foo
{
private:
    foo _obj;
    loglib::log_sink _log;

    template <typename MetaFunction, typename ... P>
    void _do_log_call(const P&...);
public:
    void f1(void)
    {
        _do_log_call<mirrored(this::function)>();
        _obj.f1();
    }

    int f2(int a, int b);
    {
        _do_log_call<mirrored(this::function)>();
```

```
        return _obj.f2(a, b);
    }

    double f3(float a, long b, double c, const std::string& d);
    {
        _do_log_call<mirrored(this::function)>();
        return _obj.f3(a, b, c, d);
    }
};
```

Obviously the definition of `logging_foo` is very repetitive and if this pattern is recurring in the code it may lead to subtle, hard to track bugs, so we may wish to automate the implementation.

Reflection to the rescue!

```
template <typename Wrapped>
class logging_base
{
protected:
    Wrapped _obj;
    loglib::log_sink _log;

    template <typename MetaFunction, typename ... P>
    void _do_log_call(const P&...);
};
```

`logging_base` is a common virtual base class holding the wrapped object and the log sink.

```
template <typename Wrapped, typename MetaFunction>
class logging_helper
 : virtual public logging_base<Wrapped>
{
public:
    template <typename ... P>
    auto identifier(base_name<MetaFunction>::value)(P&& ... p)
    {
        this->_do_log_call<MetaFunction>(std::forward<P>(p)...);

        auto mfp = pointer<MetaFunction>::get();

        return (this->_obj.*mfp)(std::forward<P>(p)...);
    }
};
```

`logging_helper` is a unit implementing the delegation of a single function call from the interface of the `Wrapped` class.

The `identifier` operator is used here to define the name of the member function to be the same as the name of the wrapped function.

If the idea of the `identifier` operator is scrapped, it would still be doable in terms of the `named_mem_var` template as defined in N4111, or some variation on that theme.

```
template <typename Wrapped, typename ... MetaFunctions>
class logging_helpers
 : public logging_helper<Wrapped, MetaFunctions>...
{ };
```

`logging_helpers` inherits from multiple `logging_helper` units each having a single ***MetaFunction*** reflecting respective member functions of the `Wrapper` class.

```
template <typename Wrapped, typename MetaFunctionSeq, typename IdxSeq>
class logging_impl;

template <typename Wrapped, typename MetaFunctionSeq, std::size_t ... I>
class logging_impl<Wrapped, MetaFunctionSeq, std::index_sequence<I...>>
 : public logging_helpers<Wrapped, at<MetaFunctionSeq, I>...>
{ };
```

`logging_impl` uses a standard `index_sequence` to extract the individual ***MetaFunctions*** from the metafunction sequence and passes them to `logging_helpers` as a parameter pack.

```
template <typename Wrapped>
class logging
 : public logging_impl<
    Wrapped,
    members<mirrored(Wrapped)>,
    std::make_index_sequence<size<members<mirrored(Wrapped)>>::value>
>
{ };

typedef logging<foo> logging_foo;
```

The `logging` template makes the use of `logging_impl` convenient.

Note that the metaobject sequence 'returned' by `members<...>` should be filtered to contain only ***MetaFunction***s.

This programming pattern of creating a new class with the same or similar interface than another class is quite frequent and includes not just typical decorators or delegation but

also adapters, type-erasures, mock classes used for unit testing, etc.[7].

The following features are shown in this use-case:

- class member reflection,
- class member function reflection,
- use of the reflection operator,
- use of the `identifier` operator or the `named_mem_var` templates.

A.4. Structure data member transformations

We need to create a new structure, which has data members with the same names as an original structure, but we need to change some of the properties of the data members (for example their types).

For example we need to transform:

```
struct foo
{
    bool b;
    char c;
    double d;
    float f;
    std::string s;
};
```

into

```
struct rdbs_table_placeholder_foo
{
    column_placeholder<bool>::type b;
    column_placeholder<char>::type c;
    column_placeholder<double>::type d;
    column_placeholder<float>::type f;
    column_placeholder<std::string>::type s;
};
```

By using the proposed `identifier` operator[8], class member reflection and multiple inheritance we can create a new structure that is nearly equivalent to `rdbs_table_foo` via metaprogramming:

```
template <typename MetaVariable>
struct rdbs_table_placeholder_helper
```

[7] See also the use-case described in A.4

[8] See appendix B.7.

```
{
    typename column_placeholder<
        typename original_type<type<MetaVariable>>::type
    >::type identifier(base_name<MetaVariable>::value);
};

template <typename ... MetaVariables>
struct rdbs_table_placeholder_helpers
 : rdbs_table_placeholder_helper<MetaVariables>...
{ };

template <typename MetaVariableSeq, typename IdxSeq>
class rdbs_table_impl;

template <typename MetaVariableSeq, std::size_t ... I>
class rdbs_table_impl<MetaFunctionSeq, std::index_sequence<I...>>
 : public rdbs_table_placeholder_helpers<at<MetaFunctionSeq, I>...>
{ };

typedef rdbs_table_impl<
    members<mirrored(foo)>,
    std::make_index_sequence<size<members<mirrored(foo)>>::value>
> rdbs_table_placeholder_foo;
```

This examples uses the following features.

- class member reflection,
- use of the reflection operator,
- use of the `identifier` operator or the `named_mem_var` templates.

B. Additions to N4111

The examples described in this paper use several features not described in N4111 (these will be added to the next revision of that paper – N4451):

B.1. Source file and line

Template class `source_file` should be defined for ***Metaobject***s and should "return" a compile-time string containing the path to the source file where the base-level construct reflected by the metaobject was declared.

```
template <>
struct source_file<MetaObject>
 : String
{ };
```

Template class `source_line` should be defined for ***Metaobject***s and should inherit from `integral_constant<unsigned, Line>` where `Line` is the line number in the source file where the base-level construct reflected by the metaobject was declared.

```
template <>
struct source_file<MetaObject>
 : String
{ };
```

B.2. For each element in Metaobject sequence

Template function `for_each`, should be defined for every metaobject sequence and should call the specified unary functor taking values of types conforming to the same metaobject concept as the elements of the metaobject sequence as arguments.

```
template <typename MetaobjectSequence, typename UnaryFunc>
void for_each(UnaryFunc func)
{
    /* call func on each element in the sequence */
}
```

The interface of ***MetaobjectSequence*** as defined in N4111 should be enough to define a single generic implementation of this function without the need to write specialization for every type modelling this concept.

B.3. The MetaPositional concept

The ***MetaPositional*** concept defines the interface for metaobjects reflecting base-level constructs having a fixed position, like function or template parameters, class inheritance clauses, etc.

The `has_position` template class can be used to distinguish metaobjects modelling this concept. Is should inherit from `true_type` for ***MetaPositional***s and from `false_type` otherwise.

```
template <typename X>
struct has_position
 : false_type
{ };
```

```
template <>
struct has_position<MetaPositional>
 : true_type
{ };
```

The `position` template class inheriting from `integral_constant<size_t, I>` type (where `I` is a zero-based position of the reflected base-level language construct) can be used to obtain the value of the index.

```
template <typename T>
struct position;

template <>
struct position<MetaPositional>
 : integral_constant<size_t, I>
{ };
```

B.4. Position of a base in the list of base classes

Every model of ***MetaInheritance*** should also conform to the ***MetaPositional*** concept described above.

B.5. Pointers to reflected variables and functions

For models of ***MetaVariable*** and ***MetaFunction*** the `pointer` template class should be defined as:

```
template <>
struct pointer<MetaVariable>
{
    typedef _unspecified_ type;

    static type get(void);
};

template <>
struct pointer<MetaFunction>
{
    typedef _unspecified_ type;

    static type get(void);
};
```

For ***MetaVariable***s or ***MetaFunction***s reflecting namespace-level variables or functions the `get` function should return a pointer to that variable or function respectively.

If the ***MetaVariable*** or ***MetaFunction*** reflect class members then the `get` function should return a pointer to data member or pointer to member function respectively.

B.6. Context-dependent reflection

Special expressions in the form of `this::{namespace,class,function}` should be added as valid arguments for the reflection operator and should return metaobjects depending on the context where such invocation of the reflection operator was used.

- `mirrored(this::namespace)` reflects the namespace inside of which the reflection operator was invoked.
- `mirrored(this::class)` reflects the class inside of which the reflection operator was invoked. This should also work inside of member functions, constructors and operators of that class.
- `mirrored(this::function)` reflects the function inside of which the reflection operator was invoked.

B.7. Turning compile-time strings into identifiers

Inspired by the idea of *name literals* as mentioned on the WG mailing list, we suggest to consider adding a new functionality to the core language, allowing to specify identifiers as compile-time constant C-string literal expression, i.e. expressions evaluating into values of `constexpr const char [N]`.

This could be implemented either by using a new operator (or recycling an old one), or maybe by using generalized attributes. In the use-cases described in this paper the `identifier` operator is used, but we do not have any strong preference for the name of this operator.

For example:

```
identifier("int") identifier("main")(
    int idenitifier("argc"),
    const identifier("char")* identifier("argv")
)
{
    using namespace identifier(base_name<mirrored(std)>::value);
    for(int i=0; i<argc; ++i)
    {
        cout << argv[i] << endl;
    }
```

```
    return 0;
}
```

would be equivalent to

```
int main(int argc, const char* argv)
{
    using namespace std;
    /* ... */
}
```

The content of the constexpr string passed as the argument to `identifier` should be encoded in the source character set and subject to the same restrictions which are placed on identifiers.

The idea is to replace preprocessor token pasting with much more flexible constexpr C++ expressions. Adding this feature would also allow to remove the `named_mem_var` and `named_typedef` metafunctions which were in N4111 defined as part of the interface of ***MetaNamed***.

This addition has the potential to complicate the processing of a translation unit by the compiler and would logically fit somewhere between phases 6 and 8 as described in the standard. If the use of regular templates for the purpose of creating the constexpr identifier strings would be too complicated to implement, phase 6 could be extended to allow simple compile-time text manipulation (comparison, concatenation, substrings, etc.) by a set of dedicated functions.