THE BOOST C++ METAPROGRAMMING LIBRARY

Aleksey Gurtovoy
MetaCommunications
agurtovoy@meta-comm.com

David Abrahams
Boost Consulting
david.abrahams@rcn.com

Abstract

This paper describes the Boost C++ template metaprogramming library (MPL), an extensible compile-time framework of algorithms, sequences and metafunction classes. The library brings together important abstractions from the generic and functional programming worlds to build a powerful and easy-to-use toolset which makes template metaprogramming practical enough for the real-world environments. The MPL is heavily influenced by its run-time equivalent — the Standard Template Library (STL), a part of the C++ standard library [STL94], [ISO98]. Like the STL, it defines an open conceptual and implementation framework which can serve as a foundation for future contributions in the domain. The library's fundamental concepts and idioms enable the user to focus on solutions without navigating the universe of possible ad-hoc approaches to a given metaprogramming problem, even if no actual MPL code is used. The library also provides a compile-time lambda expression facility enabling arbitrary currying and composition of class templates, a feature whose runtime counterpart is often cited as missing from the STL. This paper explains the motivation, usage, design, and implementation of the MPL with examples of its real-life applications, and offers some lessons learned about C++ template metaprogramming.

Table of Contents

1. Introduction

Metaprogramming is usually defined as the creation of programs which generate other programs. Parser generators such as YACC [Joh79] are examples of one kind of program-generating program. The input language to YACC is a context-free grammar in Extended Backus-Naur Form [EBNF], and its output is a program which parses that grammar. Note that in this case the metaprogram (YACC) is written in a language (*C*) which does not directly support the description of generated programs. These specifications, which we'll call *metadata*, are not written in *C*, but in a *meta-language*. Because the the rest of the user's program typically requires a general-purpose programming system and must interact with the generated parser, the metadata is translated into *C*, which is then compiled and linked together with the rest of the system. The metadata thus undergoes two translation steps, and the user is always very conscious of the boundary between her metadata and the rest of her program.

1.1. Native language metaprogramming

A more interesting form of metaprogramming is available in languages such as Scheme [SS75], where the generated program specification is given in the same language as the metaprogram itself. The metaprogrammer defines her meta-language as a subset of the expressible forms of the underlying language, and program generation can take place in the same translation step used to process the rest of the user's program. This allows users to switch transparently between ordinary programming, generated program specification, and metaprogramming, often without being aware of the transition.

1.2. Metaprogramming in C++

In C++, it was discovered almost by accident [Unr], [Vel95a] that the template mechanism provides a rich facility for computation at compile-time. In this section, we'll explore the basic mechanisms and some common idioms used for metaprogramming in C++.

1.2.1. Numeric computations

The availability of *non-type template parameters* makes it possible to perform integer computations at compile-time. For example, the following template computes the factorial of its argument:

```
template< unsigned n >
struct factorial
{
    static const unsigned value = n * factorial<n-1>::value;
};

template<>
struct factorial<0>
{
    static const unsigned value = 1;
};
```

The program fragment above is called a *metafunction*, and it is easy to see its relationship to a function designed to be evaluated at runtime: the "metafunction argument" is passed as a template parameter, and its "return value" is defined as a nested static constant. Because of the hard line between the expression of compile-time and runtime computation in C++, metaprograms look different from their runtime counterparts. Thus, although as in Scheme the C++ metaprogrammer writes her code in the same language as the ordinary program, only a subset of the full C++ language is available to her: those expressions which can be evaluated at compile-time. Compare the above with a straightforward runtime definition of the factorial function:

```
unsigned factorial(unsigned N)
{
```

```
    return N == 0 ? 1 : N * factorial(N - 1);
}
```

While it is easy to see the analogy between the two recursive definitions, recursion is in general more important to C++ metaprograms than it is to runtime C++. In contrast to languages such as Lisp where recursion is idiomatic, C++ programmers will typically avoid recursion when possible. This is done not only for efficiency reasons, but also because of "cultural momentum": recursive programs are simply harder (for C++ programmers) to think about. Like pure Lisp, though, the C++ template mechanism is a *functional* programming language: as such it rules out the use of data mutation required to maintain loop variables.

A key difference between the runtime and compile-time factorial functions is the expression of the termination condition: our meta-factorial uses template specialization as a kind of *pattern-matching* mechanism to describe the behavior when `N` is zero. The syntactic analogue in the runtime world would require two separate definitions of the same function. In this case the impact of the second definition is minimal, but in large metaprograms the cost of maintaining and understanding the terminating definitions can become significant.

Note also that a C++ metafunction's return value must be *named*. The name chosen here, `value`, is the same one used for all numeric returns in the MPL. As we'll see, establishing a consistent naming convention for metafunction returns is crucial to the power of the library.

1.2.2. Type computations

How could we apply our `factorial` metafunction? We might, for example, produce an array type of an appropriate size to hold all permutations of instances of another type:

```
// permutation_holder<T>::type is an array type which can contain
// all permutations of a given T.

// unspecialized template for scalars
template< typename T >
struct permutation_holder
{
    typedef T type[1][1];
};

// specialization for array types
template< typename T, unsigned N >
struct permutation_holder<T[N]>
{
    typedef T type[factorial<N>::value][N];
};
```

Here we have introduced the notion of a *type computation*. Like `factorial` above, `permutation_holder` template is a metafunction. However, where `factorial` manipulates unsigned integer values, `permutation_holder` accepts and "returns" a type (as the nested typedef `type`). Because the C++ type system provides a much richer set of expressions than anything we can use as a nontype template argument (e.g. the integers), C++ metaprograms tend to be composed mostly of type computations.

1.2.3. Type sequences

The ability to programmatically manipulate collections of types is a central tool of most interesting C++ metaprograms. Because this capability is so well-supported by the MPL, we'll provide just a brief introduction to the basics here. Later on, we'll revisit the example below to show how it can be implemented using MPL.

First, we'd need a way to represent the collection. One idea might be to store the types in a structure:

```
struct types
{
    int t1;
```

```
    long t2;
    std::vector<double> t3;
};
```

Unfortunately, this arrangement is not susceptible to the compile-time type introspection power that C++ gives us: there's no way to find out what the names of the members are, and even if we assume that they're named according to some convention as above, there's no way to know how many members there are. The key to solving this problem is to increase the uniformity of the representation. If we have a consistent way to get the first type of any sequence and the rest of the sequence, we can easily access all members:

```
template< typename First, typename Rest >
struct cons
{
    typedef First first;
    typedef Rest rest;
};

struct nil {};

typedef
      cons<int
    , cons<long
    , cons<std::vector<double>
    , nil
    > > > my_types;
```

The structure described by `types` above is the compile-time analogue of a singly-linked list; it has been first introduced by Czarnecki and Eisenecker in [CE98]. Now that we've adjusted the structure so that the C++ template machinery can "peel it apart", let's examine a simple metafunction which does so. Suppose a user wished to find the largest of an arbitrary collection of types. We can apply the recursive metafunction formula which should by now be familiar:

```
// choose the larger of two types
template<
      typename T1
    , typename T2
    , bool choose1 = (sizeof(T1) > sizeof(T2)) // hands off!
    >
struct choose_larger
{
    typedef T1 type;
};

// specialization for the case where sizeof(T2) >= sizeof(T1)
template< typename T1, typename T2 >
struct choose_larger< T1,T2,false >
{
    typedef T2 type;
};

// get the largest of a cons-list
template< typename T > struct largest;

// specialization to peel apart the cons list
template< typename First, typename Rest >
struct largest< cons<First,Rest> >
    : choose_larger< First, typename largest<Rest>::type >
{
    // type inherited from base
};

// specialization for loop termination
template< typename First >
struct largest< cons<First,nil> >
{
```

```
    typedef First type;
};

int main()
{
    // print the name of the largest of my_types
    std::cout
        << typeid(largest<my_types>::type).name()
        << std::endl
        ;
}
```

There are several things worth noticing about this code:

- It uses a few ad-hoc, esoteric techniques, or "hacks". The default template argument `choose1` (labeled "hands off!") is one example. Without it, we would have needed yet another template to provide the implementation of `choose_larger`, or we would have had to provide the computation explicitly as a parameter to the template — perhaps not bad for this example, but it would make `choose_larger` much less useful and more error-prone. The other hack is the derivation of a specialization of `largest` from `choose_larger`. This is a code-saving device which allows the programmer to avoid writing "`typedef typename ...::type type`" in the template body.
- Even this simple metaprogram uses three separate partial specializations. The `largest` metafunction uses *two* specializations. One might expect that this indicates there are two termination conditions, but there are not: one specialization is needed simply to deal with access to the sequence elements. These specializations make the code difficult to read by spreading the definition of a single metafunction over several C++ template definitions. Also, because they are *partial* specializations, they make the code unusable for a large community of C++ programmers whose compilers don't support that feature.

While these techniques are, of course, a valuable part of the arsenal of any good C++ metaprogrammer, their use tends to make programs written in what is already an unusual style harder-to-read and harder-to-write. By encapsulating commonly-used structures and dealing with loop terminations internally, the MPL reduces the need for both tricky hacks and for template specializations.

1.3. Why metaprogramming?

It's worth asking why anyone would want to do this. After all, even a simple toy example like the factorial metafunction is somewhat esoteric. To show how the type computation can be put to work, let's examine a simple example. The following code produces an array containing all possible permutations of another array:

```
// can't return an array in C++, so we need this wrapper
template< typename T >
struct wrapper
{
    T x;
};

// return an array of the N! permutations of 'in'
template< typename T >
wrapper< typename permutation_holder<T>::type >
all_permutations(T const& in)
{
    wrapper<typename permutation_holder<T>::type> result;

    // copy the unpermutated array to the first result element
    unsigned const N = sizeof(T) / sizeof(**result.x);
    std::copy(&*in, &*in + N, result.x[0]);

    // enumerate the permutations
    unsigned const result_size = sizeof(result.x) / sizeof(T);
```

```
    for (T* dst = result.x + 1; dst != result.x + result_size; ++dst)
    {
        T* src = dst - 1;
        std::copy(*src, *src + N, *dst);
        std::next_permutation(*dst, *dst + N);
    }
    return result;
}
```

The runtime definition of `factorial` would be useless in `all_permutations` above, since in C++ the sizes of array members must be computed at compile-time. However, there are alternative approaches; how could we avoid metaprogramming, and what would the consequences be?

- We could write programs to interpret the metadata directly. In our factorial example, the array size could have been a runtime quantity; then we'd have been able to use the straightforward factorial function. However, that would imply the use of dynamic allocation, which is often expensive.
- To carry this further, YACC might be rewritten to accept a pointer-to-function returning tokens from the stream to be parsed, and a string containing the grammar description. This approach, however, would impose unacceptable runtime costs for most applications: either the parser would have to treat the grammar nondeterministically, exploring the grammar for each parse, or it would have to begin by replicating at runtime the substantial table-generation and optimization work of the existing YACC for each input grammar.
- We could replace the compile-time computation with our own analysis. After all, the size of arrays passed to `all_permutations` are always known at compile-time, and thus can be known to its user. We could ask the user to supply the result type explicitly:

  ```
  template< typename Result, typename T >
  Result all_permutations(T const& input);
  ```

  The costs to this approach are obvious: we give up expressivity (by requiring the user to explicitly specify implementation details), and correctness (by allowing the user to specify them incorrectly). Anyone who has had to write parser tables by hand will tell you that the impracticality of this approach is the very reason of YACC's existence.

  In a language such as C++, where the metadata can be expressed in the same language as the rest of the user's program, expressivity is further enhanced: the user can invoke metaprograms directly, without learning a foreign syntax or interrupting the flow of her code.

So, the motivation for metaprogramming comes down to the combination of three factors: efficiency, expressivity, and correctness. While in classical programming there is always a tension between expressivity and correctness on one hand and efficiency on the other, in the metaprogramming world we wield new power: we can move the computation required for expressivity from runtime to compile-time.

1.4. Why a metaprogramming *library*?

One might just as well ask why we need any generic library:

- Quality. Code that is appropriate for a general-purpose library is usually incidental to the purpose of its users. To a library developer, it is the central mission. On average, the containers and algorithms provided by any given C++ standard library implementation are more-flexible and better-implemented than the project-specific implementations which abound, because library development was treated as an end in itself rather than a task incidental to the development of some other application. With a centralized implementation for any given function, optimizations and improvements are more likely to have been applied.
- Re-use. More important even than the re-use of code which all libraries provide, a well-designed generic library

establishes a *framework of concepts and idioms* which establishes a reusable mental model for approaching problems. Just as the C++ Standard Template Library gave us iterator concepts and a function object protocol, the Boost Metaprogramming Library provides type-iterators and metafunction class protocol. A well-considered framework of idioms saves the metaprogrammer from considering irrelevant implementation details and allows her to concentrate on the problem at hand.

- Portability. A good library can smooth over the ugly realities of platform differences. While in theory a metaprogramming library is fully generic and shouldn't be concerned with these issues, in practice support for templates remains inconsistent even four years after standardization. This should perhaps not be surprising: C++ templates are the language's furthest-reaching and most complicated feature, which largely accounts for the power of metaprogramming in C++.

- Fun. Repeating the same idioms over and over is *tedious*. It makes programmers tired and reduces productivity. Furthermore, when programmers get bored they get sloppy, and buggy code is even more costly than slowly-written code. Often the most useful libraries are simply patterns that have been "plucked" by an astute programmer from a sea of repetition. The MPL helps to reduce boredom by eliminating the need for the most commonly-repeated boilerplate coding patterns.

As one can see, the MPL's development is motivated primarily by the same practical, real-world considerations that justify the development of any other library. Perhaps this is an indication that template metaprogramming is finally ready to leave the realm of the esoteric and enter the lingua franca of every day programmers.

2. Basic usage

2.1. Conditional type selection

Conditional type selection is the simplest basic construct of C++ template metaprogramming. Veldhuizen [Vel95a] was the first to show how to implement it, and Czarnecki and Eisenecker [CE00] first presented it as a standalone library primitive. The MPL defines the corresponding facility as follows:

```
template<
      typename Condition
    , typename T1
    , typename T2
    >
struct if_
{
    typedef unspecified type;
};
```

Note that the first template parameter of the template is a type. The primitive's semantics intuitively matches its name:

```
typedef mpl::if_<mpl::true_,char,long>::type t1;
typedef mpl::if_<mpl::false_,char,long>::type t2;

BOOST_MPL_ASSERT(( is_same< t1, char > ));
BOOST_MPL_ASSERT(( is_same< t2, long > ));
```

The construct is important because template metaprograms often contain a lot of decision-making code, and, as we will show, spelling it manually every time via (partial) class template specialization quickly becomes impractical. The template is also important from the point of encapsulating the compiler workarounds.

2.1.1. Delayed evaluation

The way the C++ template instantiation mechanism works imposes some subtle limitations on applicability of the type selection primitive (`if_`), compared to a manually implemented equivalent of the selection code. For example, suppose we are implementing a `pointed_type` traits template such that `pointed_type<T>::type` instantiated for a `T` that is either a plain pointer (`U*`), `std::auto_ptr<U>`, or any of the Boost smart pointers [SPL], e.g. `boost::scoped_ptr<U>`, will give us the pointed type (`U`):

```
BOOST_MPL_ASSERT(( is_same< pointed_type<my*>::type, my > ));
BOOST_MPL_ASSERT(( is_same< pointed_type< std::auto_ptr<my> >::type, my > ));
BOOST_MPL_ASSERT(( is_same< pointed_type< boost::scoped_ptr<my> >::type, my> ));
```

Unfortunately, the straightforward application of `if_` to this problem does not work:[1]

```
template< typename T >
struct pointed_type
    : mpl::if_<
          boost::is_pointer<T>
        , typename boost::remove_pointer<T>::type
        , typename T::element_type // #1
        >
{
};

// the following code causes compilation error in line #1:
// name followed by "::" must be a class or namespace name
typedef pointed_type<char*>::type result;
```

Clearly, the expression `typename T::element_type` is not valid in the case of `T == char*`, and that's what the compiler is complaining about. Implementing the selection code manually solves the problem:

```
namespace aux {

// general case
template< typename T, bool is_pointer = false >
struct select_pointed_type
{
    typedef typename T::element_type type;
};

// specialization for plain pointers
template< typename  T >
struct select_pointed_type<T,true>
{
    typedef typename boost::remove_pointer<T>::type type;
};

}

template< typename T >
struct pointed_type
    : aux::select_pointed_type<
          T, boost::is_pointer<T>::value
        >
{
};
```

But this quickly becomes awkward if needs to be done repeatedly, and this awkwardness is compounded when partial specialization is not available. We can try to work around the problem as follows:

```
namespace aux {
template< typename T >
```

[1]Although it would be easy to implement `pointed_type` using partial specialization to distinguish the case where `T` is a pointer, `if_` is likely to be the right tool for dealing with more complex conditionals. For the purposes of exposition, please suspend disbelief!

```
struct element_type
{
    typedef typename T::element_type type;
};
}

template< typename T >
struct pointed_type
{
    typedef typename mpl::if_<
          boost::is_pointer<T>
        , typename boost::remove_pointer<T>::type
        , typename aux::element_type<T>::type
        >::type type;
};
```

but this doesn't work either — the access to the `aux::element_type<T>`'s nested `type` member still forces the compiler to instantiate `element_type<T>` with `T == char*`, and that instantiation is, of course, invalid. Also, although in our case this does not lead to a compile error, the `boost::remove_pointer<T>` template always gets instantiated as well, and for the same reason (because we are accessing its nested `type` member). Unnecessary instantiation that is not fatal may or may be not a problem, depending on the "weight" of the template (how much the instantiation taxes the compiler), but a general rule of thumb would be to avoid such code.

Returning to our error, to make the above code compile, we need to factor the act of "asking" `aux::element_type<T>` for its nested `type` out of the `if_` invocation. The fact that both the `boost::remove_pointer<T>` trait template and `aux::element_type<T>` use the same naming convention for their result types makes the refactoring easier:

```
template< typename T >
struct pointed_type
{
private:
    typedef typename mpl::if_<
          boost::is_pointer<T>
        , boost::remove_pointer<T>
        , aux::element_type<T>
        >::type func_;

public:
    typedef typename func_::type type;
};
```

Now the compiler is guaranteed not to instantiate both `boost::remove_pointer<T>` and `aux::element_type<T>`, even although they are used as actual parameters to the `if_` template, so we are allowed to get away with `aux::element_type<char*>` so long as it won't end up being selected as `func_`.

The above technique is so common in template metaprograms, that it even makes sense to facilitate the selection of a nested `type` member by introducing a high-level equivalent to `if_` — the one that will do the `func_::type` operation (that is called [nullary] metafunction class application) as a part of its invocation. The MPL provides such template — it's called `eval_if`. Using it, we can re-write the above code as simple as:

```
template< typename T >
struct pointed_type
{
    typedef typename mpl::eval_if<
          boost::is_pointer<T>
        , boost::remove_pointer<T>
        , aux::element_type<T>
        >::type type;
};
```

To make our techniques review complete, let's consider a slightly different example — suppose we want to define a high-level wrapper around `boost::remove_pointer` traits template [TTL], which will strip the pointer qualification conditionally. We will call it `remove_pointer_if`:

```
template<
      typename Condition
    , typename T
    >
struct remove_pointer_if
{
    typedef typename mpl::if_<
          Condition
        , typename boost::remove_pointer<T>::type
        , T
        >::type type;
};
```

Now the above works the first time, but it suffers from the problem we mentioned earlier — `boost::remove_pointer<T>` gets instantiated even if its result is never used. In the metaprogramming world compilation time is an important resource [Abr01], and it is wasted by unnecessary template instantiations. We've just seen how to deal with the problem when both arguments to `if_` are the results of nullary metafunction class applications, but in this example one of the arguments (`T`) is just a simple type, so the refactoring just doesn't seem possible.

The easiest way out of this situation would be to pass to `if_` a real nullary metafunction instead of `T` — the one that returns `T` on its invocation. The MPL provides a simple way to do it — we just substitute `identity<T>` and `eval_if` for `T` and `if_`:

```
template<
      typename Condition
    , typename T
    >
struct remove_pointer_if
{
    typedef typename mpl::eval_if<
          Condition
        , boost::remove_pointer<T>
        , mpl::identity<T>
        >::type type;
};
```

which gives us exactly what we wanted.

2.2. Metafunctions

2.2.1. The simple form

In C++, the basic underlying language construct which allows parameterized compile-time computation is the *class template* ([ISO98], section 14.5.1 [temp.class]). A bare class template is the simplest possible model we could choose for metafunctions: it can take types and/or non-type arguments as actual template parameters, and instantiation "returns" a new type. For example, the following produces a type derived from its arguments:

```
template< typename T1, typename T2 >
struct derive : T1, T2
{
};
```

However, this model is far too limiting: it restricts the metafunction result not only to class types, but to instantiations of a given class template, to say nothing of the fact that every metafunction invocation introduces an additional level of template nesting. While that might be acceptable for this particular metafunction, any model which prevented us from "returning", say, `int` is obviously not general enough. To meet this basic requirement, we must rely on a nested type to provide our return value:

```
template< typename T1, typename T2 >
struct derive
{
    struct type : T1, T2 {};
};

// silly specialization, but demonstrates "returning" int
template<>
struct derive<void,void>
{
    typedef int type;
};
```

Veldhuizen [Vel95a] was first to talk about class templates of this form as "compile-time functions", and Czarnecki and Eisenecker [CE00] have introduced "template metafunction" as an equivalent term (they also use the simpler term "metafunction", as do we). Czarnecki and Eisenecker have also recognized the limitations of the simple metafunction representation and suggested the form that we discuss in the *Metafunction classes* section.

2.2.2. Higher-order metafunctions

While syntactically simple, the simple template metafunction form does not always interact optimally with the rest of C++. In particular, the simple metafunction form makes it unnecessarily awkward and tedious to define and work with higher-order metafunctions (metafunctions that operate on other metafunctions). In order to pass a simple metafunction to another template, we need to use *template template parameters*:

```
// returns F(T1,F(T2,T3))
template<
      template<typename,typename> class F
    , typename T1
    , typename T2
    , typename T3
    >
struct apply_twice
{
    typedef typename F<
          T1
        , typename F<T2,T3>::type
        >::type type;
};

// a new metafunction returning a type derived from T1, T2, and T3
template<
    typename T1
    , typename T2
    , typename T3
    >
struct derive3
    : apply_twice<derive,T1,T2,T3>
{
};
```

This looks different, but it seems to work.[2]

However, things begin to break down noticeably when we want to "return" a metafunction from our metafunction:

```
// returns G s.t. G(T1,T2,T3) == F(T1,F(T2,T3))
template< template<typename,typename> class F >
```

[2]In fact it's already broken: `apply_twice` doesn't even fit the metafunction concept since it requires a template (rather than a type) as its first parameter, which breaks the metafunction protocol.

```
struct compose_self
{
    template<
          typename T1
        , typename T2
        , typename T3
        >
    struct type
        : apply_twice<F,T1,T2,T3>
    {
    };
};
```

The first and most obvious problem is that the result of applying `compose_self` is not itself a type, but a template, so it can't be passed in the usual ways to other metafunctions. A more subtle issue, however, is that the metafunction "returned" is not exactly what we intended. Although it acts just like `apply_twice`, it differs in one important respect: its identity. In the C++ type system, `compose_self<F>::template type<T,U,V>` is not a synonym for `apply_twice<F,T,U,V>`, and any metaprogram which compared metafunctions would discover that fact.

Because C++ makes a strict distinction between type and class template template parameters, reliance on simple metafunctions creates a "wall" between metafunctions and metadata, relegating metafunctions to the status of second-class citizens. For example, recalling our introduction to type sequences, there's no way to make a `cons` list of metafunctions:

```
typedef cons<derive, cons<derive3, nil> > derive_functions; // error!
```

We might consider redefining our `cons` cell so we can pass `derive` as the head element:

```
template <
      template< template<typename T, typename U> class F
    , typename Tail
    >
struct cons;
```

However, now we have another problem: C++ templates are polymorphic with respect to their type arguments, but not with respect to template template parameters. The arity (number of parameters) of any template template parameter is strictly enforced, so we *still* can't embed `derive3` in a `cons` list. Moreover, polymorphism *between* types and metafunctions is not supported (the compiler expects one or the other), and as we've seen, the syntax and semantics of "returned" metafunctions is different from that of returned types. Trying to accomplish everything with the simple template metafunction form would seriously limit the applicability of higher-order metafunctions and would have an overall negative effect on the both conceptual and implementation clarity, simplicity and size of the library.

2.2.3. Metafunction classes

Fortunately, the truism that "there is no problem in software which can't be solved by adding yet another level of indirection" applies here. To elevate metafunctions to the status of first-class objects, the MPL introduces the concept of a "metafunction class":

```
// metafunction class form of derive
struct derive
{
    template< typename N1, typename N2 >
    struct apply
    {
        struct type : N1, N2 {};
    };
};
```

This form should look familiar to anyone acquainted with function objects in STL, with the nested `apply` template taking the same role as the runtime function-call operator. In fact, compile-time metafunction classes have the same relationship to metafunctions that runtime function objects have to functions:

```
// function form of add
template< typename T > T add(T x, T y) { return x + y; }

// function object form of add
struct add
{
    template< typename T >
    T operator()(T x, T y) { return x + y; }
};
```

2.2.4. One size fits all?

The metafunction class form solves all the problems with ordinary template metafunction mentioned earlier: since it is a regular class, it can be placed in compile-time metadata sequences and manipulated by other metafunctions using the same protocols as for any other metadata. We thereby avoid the code-duplication needed to provide versions of each library component to operate on ordinary metadata and on metafunctions with each distinct supported arity.

On the other hand, it seems that accepting metafunction classes as *the* representation for compile-time function entities imposes code duplication danger as well: if the library's own primitives, algorithms, etc. are represented as class templates, that means that one either cannot reuse these algorithms in the context of higher-order functions, or she have to duplicate all algorithms in the second form, so, for instance, there would be two versions of `find`:

```
// user-friendly form
template<
      typename Sequence
    , typename T
    >
struct find
{
    typedef /* ... */ type;
};

// "metafunction class" form
struct find_func
{
    template< typename Sequence, typename T >
    struct apply
    {
        typedef /* ... */ type;
    };
};
```

Of course, the third option is to eliminate "user-friendly form" completely so one would always have to write

```
typedef mpl::find::apply<list,long>::type iter;
```

or even

```
typedef mpl::apply< mpl::find,list,long >::type iter;
```

instead of

```
typedef mpl::find<list,long>::type iter;
```

That too would hurt usability, considering that the direct invocations of library's algorithms are far more often-used than passing algorithms as arguments to other algorithms/metafunctions.

2.2.5. From metafunction to metafunction class

The MPL's answer to this dilemma is *lambda expressions*. Lambda is the mechanism that enables the library to curry metafunctions and convert them into metafunction classes, so when one wants to pass the `find` algorithm as an argument to a higher-order metafunction, she just write:

```
using namespace mpl::placeholders;
typedef mpl::apply< my_f, mpl::find<_1,_2> >::type result;
```

where `_1` and `_2` are *placeholders* for the first and second arguments to the resulting metafunction class. This preserves the intuitive syntax below for when the user wants to use `find` directly in her code:

```
typedef mpl::find<list,long>::type iter;
```

This functionality is described in more details in the *Lambda facility* section.

2.3. Sequences, algorithms, and iterators

2.3.1. Introduction

Compile-time iteration over a sequence (of types) is one of the basic concepts of template metaprogramming. Differences in types of objects being manipulated is the most common point of variability of similar but not identical code/design, and such designs are the direct target for some metaprogramming. Templates were originally designed to solve this exact problem (e.g. `std::vector`). However, without predefined abstractions/constructs for manipulating/iterating over *sequences* of types (as opposed to standalone types), and without known techniques for emulating these constructs using the current language facilities, their effect on helping high-level metaprogramming happen has been limited.

Czarnecki and Eisenecker [CE98], [CE00] were the first to introduce compile-time sequences of types and some simple algorithms on them, although the idea of representing common data structures like trees, lists, etc. at compile time, using class template composition has been around for a while (e.g. most of the expression template libraries build such trees as a part of their expression "parsing" process [Vel95b]). Alexandrescu [Ale01] used lists of types and some algorithms on them to implement several design patterns; the accompanying code is known as the Loki library [Loki].

2.3.2. Algorithms and sequences

Most of the algorithms in the Boost Metaprogramming Library operate on sequences. For example, searching for a type in a list looks like this:

```
typedef mpl::list<char,short,int,long,float,double> types;
typedef mpl::find<types,long>::type iter;
```

Here, `find` accepts two parameters — a sequence to search (`types`) and the type to search for (`long`) — and returns an iterator `iter` pointing to the first element of the sequence such that `iter::type` is identical to `long`. If no such element exists, `iter` is identical to `end<types>::type`. Basically, this is how one would search for a value in a `std::list` or `std::vector`, except that `mpl::find` accepts the sequence as a single parameter, while `std::find` takes two iterators. Everything else is pretty much the same — the names are the same, the semantics are very close, there are iterators, and one can search not only by type, but also by using a predicate:

```
typedef mpl::find_if< types,boost::is_float<_> >::type iter;
```

This conceptual/syntactical similarity with the STL is not coincidental. Reusing the conceptual framework of the STL in the compile-time world allows us to apply familiar and sound approaches for dealing with sequential data structures. The algorithms and idioms which programmers already know from the STL can be applied again at compile-time. We consider this to be one of MPL's greatest strengths, distinguishing it from earlier attempts to build a template metaprogramming library.

2.3.3. Sequence concepts

In the `find` example above, we searched for the type in a sequence built using the `mpl::list` template; but `list` is not the only sequence that the library provides. Neither is `mpl::find` or any other algorithm hard-coded to work only with `list` sequences. `list` is just one model of MPL's *Forward Sequence* concept, and `find` works with anything that satisfies this concept's requirements. The hierarchy of sequence concepts in MPL is quite simple — a *Forward Sequence* is any compile-time entity for which `begin<>` and `end<>` produce iterators to the range of its elements; a *Bidirectional Sequence* is a *Forward Sequence* whose iterators satisfy *Bidirectional Iterator* requirements; finally, a *Random Access Sequence* is a *Bidirectional Sequence* whose iterators satisfy *Random Access Iterator* requirements.[3]

Decoupling algorithms from particular sequence implementations (through iterators) allows a metaprogrammer to create her own sequence types and to retain the rest of the library at her disposal. For example, one can define a `tiny_list` for dealing with sequences of three types as follows:

```
template< typename TinyList, long Pos >
struct tiny_list_item;

template< typename TinyList, long Pos >
struct tiny_list_iterator
{
    typedef typename tiny_list_item<TinyList,Pos>::type type;
    typedef tiny_list_iterator<TinyList, Pos-1> prior;
    typedef tiny_list_iterator<TinyList, Pos+1> next;
};

template< typename T0, typename T1, typename T2 >
struct tiny_list
{
    typedef tiny_list_iterator<tiny_list, 0> begin;
    typedef tiny_list_iterator<tiny_list, 3> end;
    typedef T0 type0;
    typedef T1 type1;
    typedef T2 type2;
};

template< typename TinyList >
struct tiny_list_item<TinyList,0>
{
    typedef typename TinyList::type0 type;
};

template< typename TinyList >
struct tiny_list_item<TinyList,1>
{
    typedef typename TinyList::type1 type;
};

template< typename TinyList >
struct tiny_list_item<TinyList,2>
{
    typedef typename TinyList::type2 type;
};
```

and then use it with any of the library algorithms as if it were `mpl::list`:

[3] A more precise definition of these concepts can be found in the library reference documentation [MPLR].

```
typedef tiny_list< char,short,int > types;
typedef mpl::transform<
      types
    , boost::add_pointer<_1>
    >::type pointers;
```

As written, `tiny_list` is a model of *Bidirectional Sequence*; to turn it into a *Random Access Sequence*, we need to promote `tiny_list_iterator` into a *Random Access Iterator* by specializing `mpl::advance` and `mpl::distance` metafunctions:

```
namespace boost { namespace mpl {

template< typename TinyList, long Pos, typename N >
struct advance< tiny_list_iterator<TinyList,Pos>, N >
{
    typedef tiny_list_iterator<
          TinyList
        , Pos + N::value
        > type;
};

template< typename TinyList, long Pos1, long Pos2 >
struct distance<
      tiny_list_iterator<TinyList,Pos1>
    , tiny_list_iterator<TinyList,Pos2>
    >
{
    typedef mpl::integral_c<long, Pos2 - Pos1> type;
};

}}
```

While the `tiny_list` itself might be not that interesting (after all, it can hold only three elements), if the technique above could be automated so we would be able to define not-so-tiny sequences (with five, ten, twenty, etc. elements), it would be very valuable.[4]

External code generation is an option, but there exists a solution within the language. However, it is not a template metaprogramming, but rather *preprocessor metaprogramming*. In fact, MPL's `vector` — a fixed-size type sequence that provides random-access iterators — is implemented very much like the above `tiny_list` — using the Boost Preprocessor library [PRE].

2.3.4. Ad hoc example revisited

So, the library provides its users with almost complete compile-time equivalent of the STL framework. Does it help them to solve their metaprogramming tasks? Let's return to our earlier `largest` example to see if we can rewrite it in a better way with what MPL has to offer. Well, actually, there is not much to look at, because the MPL implementation is a one-liner (we'll spread it out here for readability) [5] :

```
template< typename Sequence >
struct largest
{
    typedef typename mpl::max_element<
          Sequence
        , mpl::less<
              mpl::sizeof_<_1>
            , mpl::sizeof_<_2>
            >
        >::type iter;
```

[4]Random access is almost as important at compile-time as it is at run-time. For example, searching for an item in a sorted random-access sequence using `lower_bound` can be much faster than performing the same operation on a forward-access-only `list`.

```
    typedef typename deref<iter>::type type;
};
```

There are no more termination conditions with tricky pattern matching, no more partial specializations; and even more importantly, it's *obvious* what the above code does — even although it's all templates — something that one could not say about the original version.

2.3.5. iter_fold as the main iteration algorithm

For the purpose of examining a little bit more of the library's internal structure, let's look at how `max_element` from the above example is implemented. One might expect that *now* we will again see all these awkward partial specializations, esoteric pattern matching, etc. Well, let's see:

```
template<
      typename Sequence
    , typename Predicate
    >
struct max_element
{
    typedef typename mpl::iter_fold<
          Sequence
        , typename mpl::begin<Sequence>::type
        , if_< less< deref<_1>,deref<_2> >, _2, _1 >
        >::type type;
};
```

The first thing to notice here is that this algorithm is implemented in terms of another one: `iter_fold`. In fact, this is probably the most important point of the example, because nearly all other generic sequence algorithms in the library are implemented in terms of `iter_fold`. If a user should ever need to implement her own sequence algorithm, she'll almost certainly be able to do so using this primitive, which means she won't have to resort to implementing hand-crafted iteration, pattern matching of special cases for loop termination, or workarounds for lack of partial specialization. It also means that her algorithm will automatically benefit from any optimizations the library has implemented, (e.g. recursion unrolling), and that it will work with any sequence that is a model of ForwardSequence, because `iter_fold` does not require anything more of its sequence argument.

`iter_fold` algorithm is basically a compile-time equivalent of the `fold` or `reduce` functions that comprise the basic and well-known primitives of many functional programming languages. An analogy more familiar to a C++ programmer would be the `std::accumulate` algorithm from the C++ standard library ([ISO98], section 26.4.1 [lib.accumulate]). However, `iter_fold` is designed to take advantage of the natural characteristics of recursive traversal: it accepts *two* metafunction class arguments, the first of which is applied to the state "on the way in" and the second of which is applied "on the way out".

The interface to `iter_fold` is defined in MPL as follows:[6]

[5] Here is another, even more elegant implementation:

```
template< typename Sequence >
struct largest
{
    typedef typename mpl::max_element<
          mpl::transform_view<
              Sequence
            , mpl::sizeof_<_>
            >
        >::type iter;

    typedef typename deref<iter>::type type;
};
```

[6] The `iter_fold`'s interface in the current version of the library is slightly different from the one presented here. Please refer to the MPL reference manual for the up-to-date information.

```
template<
      typename Sequence
    , typename InitialState
    , typename ForwardOp
    , typename BackwardOp = _1
    >
struct iter_fold
{
    typedef unspecified type;
};
```

The algorithm "returns" the result of two-way successive applications of binary `ForwardOp` and `BackwardOp` operations to iterators in range [`begin<Sequence>::type`, `end<Sequence>::type`) and previous result of an operation; the `InitialState` is logically placed before the sequence and included in the forward traversal. The result `type` is identical to `InitialState` if the sequence is empty.

The library also provides `reverse_iter_fold`, `fold`, and `reverse_fold` algorithms which wrap `iter_fold` to accommodate its most common usage patterns.

2.3.6. Sequences of numbers

What we've seen so far were sequences (and algorithms on sequences) of types. It is both possible and easy to manipulate compile-time *values* using the library as well. The only thing to remember is that in C++, class template non-type template parameters give us one more example of non-polymorphic behavior. In other words, if one declared a metafunction to take a non-type template parameter (e.g. `long`) it's not possible to pass anything besides compile-time integral constants to it:

```
template< long N1, long N2 >
struct equal_to
{
    static bool const value = (N1 == N2);
};

equal_to<5,5>::value; // OK
equal_to<int,int>::value; // error!
```

And of course this doesn't work the other way around either:

```
typedef mpl::list<1,2,3,4,5> numbers; // error!
```

While this may be an obvious limitation, it imposes yet another dilemma on the library design: on the one hand, we don't want to restrict users to type manipulations only, and on the other hand, full support for integral manipulations would require at least duplication of most of the library facilities[7] — the same situation as we would have if we had chosen to represent metafunctions as ordinary class templates. The solution for this issue is the same as well: we represent integral values by wrapping them in types.[8] For example, to create a list of numbers one can write:

```
typedef mpl::list<
      mpl::int_<1>
    , mpl::int_<2>
    , mpl::int_<3>
    , mpl::int_<4>
    , mpl::int_<5>
    > numbers;
```

Wrapping integral constants into types to make them first-class citizens is important well inside metaprograms, where one often doesn't know (and doesn't care) if the metafunctions she is using operate on types, integral values, other metafunctions, or something else, like compile-time fixed-point or rational numbers.

[7]Ideally, if going this route, all the templates should be re-implemented for every integral type — `char`, `int`, `short`, `long`, etc.
[8]The same technique was suggested by Czarnecki and Eisenecker in [CE00].

But, from the user's perspective, the above example is much more verbose than the shorter, incorrect one. Thus, for the purpose of convenience, the library does provide users with a template that takes non-type template parameters, but offers a more compact notation:

```
typedef mpl::list_c<long,1,2,3,4,5> numbers;
```

There is a similar `vector` counterpart as well:

```
typedef mpl::vector_c<long,1,2,3,4,5> numbers;
```

2.3.7. A variety of sequences

Previous efforts to provide generalized metaprogramming facilities for C++ have always concentrated on -`cons`style type lists and a few core algorithms like `size` and `at`, which are tied to the specific sequence implementation. Such systems have an elegant simplicity reminiscent of the analogous functionality in pure functional Lisp. It is much more time-consuming to implement even a basic set of the sequence algorithms provided by equivalent runtime libraries (the STL in particular), but if we have learned anything from the STL, it is that tying those algorithms' implementations to a specific sequence implementation is a misguided effort!

The truth is that there is no single "best" type sequence implementation for the same reasons that there will never be a single "best" runtime sequence implementation. Furthermore, there are *already* quite a number of type list implementations in use today; and just as the STL algorithms can operate on sequences which don't come from STL containers, so the MPL algorithms are designed to work with foreign type sequences.

It may be an eye-opening fact for some that type lists are not the only useful compile-time sequence. Again, the need for a variety of compile-time containers arises for the same reasons that we have lists, vectors, deques, and sets in the C++ standard library — different containers have different functional and performance characteristics which determine not only applicability and efficiency of particular algorithms, but also the expressiveness or verbosity of the code that uses them. While runtime performance is not an issue for C++ metaprograms, compilation speed is often a significant bottleneck to advanced C++ software development [Abr01].

The MPL provides five built-in sequences: `list`, `list_c` (really just a `list` of value wrappers), `vector`, a randomly-accessible sequence of fixed maximum size, `vector_c`, and `range_c`, a randomly-accessible sequence of consecutive integral values. More important, however, is its ability to adapt to arbitrary sequence types. The only core operations that a sequence is required to provide in order to be used with the library algorithms are `begin<>` and `end<>` metafunctions which "return" iterators into the sequence. As with the STL, it is the iterators which are used to implement most of the general-purpose sequence algorithms the library provides. Also, as with the STL, algorithm specialization is used to take advantage of implementation knowledge about particular sequences: many of the "basic" sequence operations such as `back<>`, `front<>`, `size<>`, and `at<>` are specialized on sequence type to provide a more efficient implementation than the fully generic version.

2.3.8. Loop/recursion unrolling

Almost coincidentally, loop unrolling can be as important to compile-time iterative algorithms as it is to runtime algorithms. To see why, one must first remember that all "loops" in C++ metaprograms, are in fact, implemented with recursion, and that the template instantiation depth can be a valuable resource in a compiler implementation. In fact, Annex B of the C++ standard ([ISO98], annex B [limits]) *recommends* a minimum depth of 17 recursively nested template instantiations; but this is far too low for many serious metaprograms, some of which easily exceed the hard-coded instantiation limits of some otherwise excellent compilers. To see how this works in action, let's examine a straightforward implementation of the `fold` metafunction, which combines some algorithm state with each element of a sequence:

```
namespace aux {

// unspecialized version combines the initial state and first element
```

```
// and recurses to process the rest
template<
      typename Start
    , typename Finish
    , typename State
    , typename BinaryFunction
    >
struct fold_impl
    : fold_impl<
          typename next<Start>::type
        , Finish
        , typename apply<
                BinaryFunction
              , State
              , typename deref<Start>::type
              >::type
        , BinaryFunction
        >
{
};

// specialization for loop termination
template<
      typename Finish
    , typename State
    , typename BinaryFunction
    >
struct fold_impl<Finish,Finish,State,BinaryFunction>
{
    typedef State type;
};

} // namespace aux

// public interface
template<
      typename Sequence
    , typename State
    , typename ForwardOp
    >
struct fold
    : aux::fold_impl<
        , typename begin<Sequence>::type
        , typename end<Sequence>::type
        , State
        , ForwardOp
        >
{
};
```

Although simple and elegant, this implementation will always incur at least as many levels of recursive template instantiation as there are elements in the input sequence.[9] The library addresses this problem by explicitly "unrolling" the recursion. To apply the technique to our `fold` example, we begin by factoring out a single step of the algorithm. Our `fold_impl_step` metafunction has two results: `type` (the next state), and `iterator` (the next sequence position).

```
template<
      typename BinaryFunction
    , typename State
    , typename Start
    , typename Finish
    >
struct fold_impl_step
{
    typedef typename apply<
          BinaryFunction
```

[9]It could be much more, depending on the complexity of the `apply<...>` expression, whose depth is added to the overall recursion depth.

```
          , State
          , typename deref<Start>::type
          >::type type;

      typedef typename next<Start>::type iterator;
  };
```

As with our main algorithm implementation, we specialize for the loop termination condition so that the step becomes a no-op:

```
  template<
        typename BinaryFunction
      , typename State
      , typename Finish
      >
  struct fold_impl_step<BinaryFunction,State,Finish,Finish>
  {
      typedef State type;
      typedef Finish iterator;
  };
```

Now we can now reduce `fold`'s instantiation depth by any constant factor N simply by inserting N invocations of `fold_impl_step`. Here we've chosen a factor of 4:

```
  template<
        typename Start
      , typename Finish
      , typename State
      , typename BinaryFunction
      >
  struct fold_impl
  {
  private:
      typedef fold_impl_step<
            BinaryFunction
          , State
          , Start
          , Finish
          > next1;

      typedef fold_impl_step<
            BinaryFunction
          , typename next1::type
          , typename next1::iterator
          , Finish
          > next2;

      typedef fold_impl_step<
            BinaryFunction
          , typename next2::type
          , typename next2::iterator
          , Finish
          > next3;

      typedef fold_impl_step<
            BinaryFunction
          , typename next3::type
          , typename next3::iterator
          , Finish
          > next4;

      typedef fold_impl_step<
            typename next4::iterator
          , Finish
          , typename next4::type
          , BinaryFunction
```

```
        > recursion;

public:
    typedef typename recursion::type type;
};
```

The MPL applies this unrolling technique across all algorithms with an unrolling factor tuned according to the demands of the C++ implementation in use, and with an option for the user to override the value.[10]

This fact enables users to push beyond the metaprogramming limits they would usually encounter with more naive algorithm implementations. Experiments also show a small (up to 10%) increase in metaprogram instantiation speed on some compilers when loop unrolling is used.

3. Lambda facility

The MPL's lambda facility allows the *inline composition* of class templates into "lambda expressions", which are classes and can therefore be passed around as ordinary metafunction classes, or transformed into metafunction classes before application using the expression:

```
typedef mpl::lambda<expr>::type func;
```

For example, `boost::remove_const` traits template from Boost `type_traits` library [TTL] is a class template, or a *metafunction* in MPL terminology. The simplest example of an "inline composition" of it would be something like:

```
typedef boost::remove_const<_1> expr;
```

This forms a so called "lambda expression", which is neither a metafunction class, nor a metafunction, yet can be passed around everywhere because it's an ordinary C++ class, because all MPL facilities are polymorphic with respect to their arguments. Now, that lambda expression can be *transformed* into a metafunction class using the MPL's `lambda` facility:

```
typedef boost::remove_const<_1> expr;
typedef mpl::lambda<expr>::type func;
```

The `func` is a unary metafunction class and can be used as such. In particular, it can be pass around or invoked (applied):

```
typedef mpl::apply<func,int const>::type res;
BOOST_MPL_ASSERT(( is_same<res, int> ));
```

or even

```
typedef func::apply<int const>::type res;
BOOST_MPL_ASSERT(( is_same<res, int >));
```

Inline composition is very appealing syntactically when one deals with metafunctions, because it makes the expression obvious:

```
typedef mpl::or_<
      mpl::less< mpl::sizeof_<_1>, mpl::int_<16> >
    , boost::is_same<_1,_2>
    > expr;
```

[10]This implementation detail is made relatively painless through heavy reliance on the Boost Preprocessor Library [PRE], so only one copy of the code needs to be maintained.

```
typedef mpl::lambda<expr>::type func;
```

In fact, that last bit (`typedef lambda<expr>::type func`) is unnecessary, because all MPL algorithms perform this transformation to all of their metafunction class operands internally (a `lambda<T>::type` expression applied to a metafunction class gives back the same metafunction class, so it's safe to apply the expression unconditionally).

The alternative way to write an equivalent of the above metafunction class would be:

```
typedef mpl::bind<
      mpl::quote2<mpl::or_>
    , mpl::bind< mpl::quote2<mpl::less>
        , mpl::bind< mpl::quote1<mpl::sizeof_>,_1 >
        , mpl::int_<16>
        >
    , mpl::bind< mpl::quote2<boost::is_same>,_1,_2 >
    > func;
```

Here, we use `mpl::quote`*n* templates to convert metafunctions into metafunction classes and then combine them using `mpl::bind` primitive. The transformation from this form to the above inline lambda expression and vice-versa is mechanical, and that is essentially what happens under the hood when we write `typedef mpl::lambda<expr>::type`.

For its own metafunctions (algorithms, primitives, etc.), MPL enables us to write the above in a less cumbersome way:

```
typedef mpl::bind<
      mpl::or_<>
    , mpl::bind< mpl::less<>, mpl::bind<mpl::sizeof_<>,_1>, mpl::int_<16> >
    , mpl::bind< mpl::quote2<boost::is_same>, _1,_2 >
    > func;
```

Note that because `is_same` is not an MPL primitive, we still have to wrap it using `quote2`.

4. Code generation facilities

There are cases, especially in the domain of numeric computation, when one wants to perform some part of the calculations at compile-time, and then pass the results to a run-time part of the program for further processing. For example, suppose one has implemented a complex compile-time algorithm that works with fixed-point arithmetic:

```
// fixed-point algorithm input
typedef mpl::vector<
      fixed_c<-1,2345678>
    , fixed_c<9,0001>
    // ..
    , fixed_c<3,14159>
    > input_data;

// complex compile-time algorithm
// ...
typedef /*...*/ result_data;
```

Suppose the `result_data` here is a sequence of `fixed_c` types that keeps the results of the algorithm, and now one wishes to feed that result to the run-time part of the algorithm. With MPL she can do this:

```
double my_algorithm()
{
    // passing the results to the run-time part of the program
```

```
    std::vector<double> results;
    results.reserve(mpl::size<result_data>::value);
    mpl::for_each<numbers,_>(
          boost::bind(&std::vector<double>::push_back, &results, _1)
        );
    // ...
}
```

The `for_each<numbers,_>(...)` call is what actually transfers the compile-time `result_data` into run-time `results`. `for_each` is a function template declared as:

```
template<
      typename Seq
    , typename TransformOp
    , typename F
    >
void for_each(F f)
{
    // ...
}
```

To call the function, one is required to explicitly provide two actual template parameters, a compile-time sequence `Seq` and a unary transformation metafunction `TransformOp`, plus a run-time function argument `f` (in our example, `numbers`, `_`, and `boost::bind(...)` correspondingly). `f` is a function object which `operator()` is called for every element in the `Seq` tranfromed by `TransformOp`.

Applying this to our example, the

```
mpl::for_each<numbers,_>(
    boost::bind(&std::vector<double>::push_back, &results, _1)
    );
```

call is roughly equivalent to this:

```
f(mpl::apply< _,mpl::at_c<result_data,0>::type >::type());
f(mpl::apply< _,mpl::at_c<result_data,1>::type >::type());
// ...
f(mpl::apply< _,mpl::at_c<result_data,n>::type >::type());
```

where `n == mpl::size<result_data>::value`.

5. Example: a compile-time FSM generator

Finite state machines (FSMs) are an important tool for describing and implementing program behavior [HU79], [Mar98]. They also are a good example of a domain in which metaprogramming can be applied to reduce the amount of repetitive and boilerplate operations one must perform in order to implement these simple mathematical models in code. Below we present a simple state machine generator that has been implemented using Boost Metaprogramming Library facilities. The generator takes a compile-time automata description, and converts it into C++ code that implements the FSM at run-time.

The FSM description is basically a combination of states and events plus a state transition table (STT), which ties them all together. The generator walks through the table and generates the state machine's `process_event` method that is the essence of an FSM.

Suppose we want to implement a simple music player using a finite state machine model. The state transition table for the FSM is shown in the table below. The STT format reflects the way one usually describes the behavior of an FSM in plain English. For example, the first line of the table can be read as follows: "If the model is in the `stopped` state and the `play_event` is received, then the `do_play` transition function is called, and the model transitions to the `playing` state.

State	Event	Next state	Transition function
stopped	play_event	playing	do_play
playing	stop_event	stopped	do_stop
playing	pause_event	paused	do_pause
paused	play_event	playing	do_resume
paused	stop_event	stopped	do_stop

Table 1. Player's state transition table with actions

The transition table provides us with a complete formal definition of the target FSM, and there are several ways to transform that definition into code. For instance, if we define states as members of an enumeration type, and events as classes derived from some base `event` class,[11] like so:

```
class player
{
public:
    // event declarations
    struct event;
    struct play_event;
    struct stop_event;
    struct pause_event;

    // "input" function
    void process_event(event const&); // throws

private:
    // states
    enum state_t { stopped, playing, paused };

    // transition functions
    void do_play(play_event const&);
    void do_stop(stop_event const&);
    void do_pause(pause_event const&);
    void do_resume(play_event const&);

private:
    state_t m_state;
};
```

then the most straightforward way to derive the FSM implementation from the above table would be something like this:

```
void player::process_event(event const& e)
{
    if (m_state == stopped)
    {
        if (typeid(e) == typeid(play_event))
        {
            do_play(static_cast<play_event const&>(e));
            m_state = playing;
            return;
        }
    }
    else if (m_state == playing)
    {
        if (typeid(e) == typeid(stop_event))
        {
            do_stop(static_cast<stop_event const&>(e));
```

[11] The events need to be passed to action functions, as they may contain some event-specific information for an action.

```
            m_state = stopped;
            return;
        }

        if (typeid(e) == typeid(pause_event))
        {
            do_pause(static_cast<pause_event const&>(e));
            m_state = paused;
            return;
        }
    }
    else if (m_state == paused)
    {
        if (typeid(e) == typeid(stop_event))
        {
            do_stop(static_cast<stop_event const&>(e));
            m_state = stopped;
            return;
        }

        if (typeid(e) == typeid(play_event))
        {
            do_play(static_cast<play_event const&>(e));
            m_state = playing;
            return;
        }
    }
    else
    {
        throw logic_error(
            boost::format("unknown state: %d")
                % static_cast<int>(m_state)
            );
    }

    throw std::logic_error(
          "unexpected event: " + typeid(e).name()
        );
}
```

Although there is nothing particularly wrong with implementing an FSM's structure using nested `if` (or `switch-case`) statements, the obvious weakness of this approach is that most of the above code is boilerplate. What one tends to do with boilerplate code is to copy and paste it, then change names etc. to adjust it to its new location; and that's where the errors are most likely to creep in. Since all the lines of event processing look alike (structurally), it's very easy to overlook or forget something that needs to be changed, and many such errors won't appear until the run-time.

The transition table of our FSM is just five lines long; ideally, we would like the skeleton implementation of the automata's controlling logic to be equally short (or, at least, to look equally short, i.e. to be encapsulated in some form so we never worry about it).

5.1. Implementation

To represent the STT in a C++ program, we define a `transition` class template that represents a single line of the table. Then the table itself can be represented as a sequence of such lines:

```
typedef mpl::list<
      transition<stopped, play_event,  playing, &player::do_play>
    , transition<playing, stop_event,  stopped, &player::do_stop>
    , transition<playing, pause_event, paused,  &player::do_pause>
    , transition<paused,  play_event,  playing, &player::do_resume>
    , transition<paused,  stop_event,  stopped, &player::do_stop>
    >::type transition_table;
```

Now, the complete FSM will look like this:

```
class player
    : state_machine<player>
{
private:
    typedef player self_t;

    // state invariants
    void stopped_state_invariant();
    void playing_state_invariant();
    void paused_state_invariant();

    // states (invariants are passed as non-type template arguments,
    // and are called then the FSM enters the corresponding state)
    typedef state<0, &self_t::stopped_state_invariant> stopped;
    typedef state<1, &self_t::playing_state_invariant> playing;
    typedef state<2, &self_t::paused_state_invariant> paused;

private:
    // event declarations; events are represented as types,
    // and can carry a specific data for each event;
    // but it's not needed for generator, so we define them later
    struct play_event;
    struct stop_event;
    struct pause_event;

    // transition functions
    void do_play(play_event const&);
    void do_stop(stop_event const&);
    void do_pause(pause_event const&);
    void do_resume(play_event const&);

    // STT
    friend class state_machine<player>;
    typedef mpl::list<
          transition<stopped, play_event,  playing, &player::do_play>
        , transition<playing, stop_event,  stopped, &player::do_stop>
        , transition<playing, pause_event, paused,  &player::do_pause>
        , transition<paused,  play_event,  playing, &player::do_resume>
        , transition<paused,  stop_event,  stopped, &player::do_stop>
        >::type transition_table;
};
```

That's all — the above will generate a complete FSM implementation according to our specification. The only thing we need before using it is the definition of the event types (that were just forward declared before):

```
// event definitions
struct player::play_event
    : player::event
{
};

// ...
```

The usage is simple as well:

```
int main()
{
    // usage example
    player p;
    p.process_event(player::play_event());
    p.process_event(player::pause_event());
    p.process_event(player::play_event());
```

```
    p.process_event(player::stop_event());
    return 0;
}
```

5.2. Related work

A notable prior work in the field of automation of general-purpose state machine implementation in C++ is the Robert Martin's *State Machine Compiler* [SMC]. The SMC takes an ASCII description of the machine's state transition table and produces C++ code that implements the FSM using a variation of State design pattern [Hun91], [GHJ95]. Lafreniere [Laf00] presents another approach, where no external tools are used, and the FSMs are table driven.

6. Acknowledgements

Peter Dimov contributed the `bind` functionality without which compile-time lambda expressions wouldn't have been possible. The MPL implementation would have been much more difficult without Vesa Karvonen's wonderful Boost Preprocessor Metaprogramming Library. Authors are also greatly indebted to David B. Held who kindly volunteered to thoroughly edit this document. Of course, any remaining errors are exclusively ours.

7. References

[Abr01] David Abrahams and Carlos Pinto Coelho, *Effects of Metaprogramming Style on Compilation Time*, 2001

[Ale01] Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, ISBN 0-201-70431-5, 2001

[CE98] Krzysztof Czarnecki and Ulrich Eisenecker, *Metalisp*

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, ISBN 0-201-30977-7, 2000

[EBNF] *ISO/IEC 14977:1996(E), Information technology — Syntactic metalanguage — Extended BNF*, ISO/IEC, 1996

[GHJ95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, ISBN 0-201-63361-2, 1995

[HU79] Hopcroft and Ullman, *Introduction to automata theory, languages and computations*, Addison-Wesley, 1979

[Hud89] Paul Hudak, *Conception, Evolution, and Application of Functional Programming Languages*, ACM Computing Surveys, Association for Computing Machinery (ACM), ISSN 0360-0300, Vol. 21, Issue 3, pp. 359-411, September, 1989

[Hun91] Immo Huneke, *Finite State Machines: A Model of Behavior for C++*, C++ Report, SIGS Publications Inc., ISSN 1040-6042, 1991

[ISO98] *ISO/IEC 14882:1998(E), Programming languages — C++*, ISO/IEC, 1998

[Joh79] Stephen C. Johnson, *Yacc: Yet Another Compiler Compiler*, UNIX Programmer's Manual, Vol. 2b, pp. 353-387, 1979

[Laf00] David Lafreniere, *State Machine Design in C++*, C/C++ User Journal, CMP Media LCC, ISSN 1075-2838, Vol. 18, Issue 5, May 1998

[Loki] *The Loki library*

[Mar98] Robert C. Martin, *UML Tutorial: Finite State Machines*, C++ Report, SIGS Publications Inc., ISSN 1040-6042, June 1998

[MPLR] *Boost MPL Library Reference Documentation*

[PRE] Vesa Karvonen, *Boost Preprocessor Metaprogramming library*

[SMC] Robert C. Martin, *SMC - Finite State Machine Compiler (C++)*

[STL94] A. A. Stepanov and M. Lee, *The Standard Template Library, Hewlett-Packard Laboratories*, 1994

[SPL] *Boost Smart Pointer library*

[SS75] Gerald J. Sussman and Guy L. Steele Jr., *Scheme: An interpreter for extended lambda calculus*, MIT AI Memo 349, Massachusetts Institute of Technology, May 1975

[TTL] *Boost Type Traits library*

[Vel95a] Todd Veldhuizen, *Using C++ template metaprograms*, C++ Report, SIGS Publications Inc., ISSN 1040-6042, Vol. 7, Issue 4, pp. 36-43, May 1995

[Vel95b] Todd Veldhuizen, *Expression templates*, C++ Report, SIGS Publications Inc., ISSN 1040-6042, Vol. 7, Issue 5, pp. 26-31, Jun 1995

[Unr] Erwin Unruh, *Prime number computation*, ANSI X3J16-94-0075/ISO WG21-462