Simple C++11 metaprogramming

With variadic templates, parameter packs and template aliases

Peter Dimov, 26.05.2015

I was motivated to write this after I read Eric Niebler's thought-provoking Tiny Metaprogramming Library article. Thanks Eric.

C++11 changes the playing field

The wide acceptance of Boost.MPL made C++ metaprogramming seem a solved problem. Perhaps MPL wasn't ideal, but it was good enough to the point that there wasn't really a need to seek or produce alternatives.

C++11 changed the playing field. The addition of variadic templates with their associated parameter packs added a compile-time list of types structure directly into the language. Whereas before every metaprogramming library defined its own type list, and MPL defined several, in C++11, type lists are as easy as

```
// C++11
template<class... T> struct type_list {};
```

and there is hardly a reason to use anything else.

Template aliases are another game changer. Previously, "metafunctions", that is, templates that took one type and produced another, looked like

```
// C++03
template<class T> struct add_pointer { typedef T* type; };
```

and were used in the following manner:

```
// C++03
typedef typename add_pointer<X>::type Xp;
```

In C++11, metafunctions can be template aliases, instead of class templates:

```
// C++11
template<class T> using add_pointer = T*;
```

The above example use then becomes

```
// C++11
typedef add_pointer<X> Xp;
```

or, if you prefer to be seen as C++11-savvy,

```
// C++11
using Xp = add_pointer<X>;
```

This is a considerable improvement in more complex expressions:

```
// C++03
typedef
   typename add_reference<
      typename add_const<
         typename add_pointer<X>::type
      >::type
   >::type Xpcr;
```

```
// C++11
using Xpcr = add_reference<add_const<add_pointer<X>>>;
```

(The example also takes advantage of another C++11 feature - you can now use >> to close templates without it being interpreted as a right shift.)

In addition, template aliases can be passed to template template parameters:

```
// C++11
template<template<class... T> class F> struct X
{
};

X<add_pointer>; // works!
```

These language improvements allow for C++11 metaprogramming that is substantially different than its idomatic C++03 equivalent. Boost.MPL is no longer good enough, and *something must be done*. But what?

Type lists and mp_rename

Let's start with the basics. Our basic data structure will be the type list:

```
template<class... T> struct mp_list {};
```

Why the mp_ prefix? mp obviously stands for metaprogramming, but could we not have used a namespace?

Indeed we could have. Past experience with Boost.MPL however indicates that name conflicts between our metaprogramming primitives and standard identifiers (such as list) and keywords (such as if, int or true) will be common and will be a source of problems. With a prefix, we avoid all that trouble.

So we have our type list and can put things into it:

```
using list = mp_list<int, char, float, double, void>;
```

but can't do anything else with it yet. We'll need a library of primitives that operate on mp_lists. But before we get into that, let's consider another interesting question first.

Suppose we have our library of primitives that can do things with a mp_list, but some other code hands us a type list that is not an mp_list, such as for example an std::tuple<int, float, void*>, or std::packer<int, float, void*>.

Suppose we need to modify this external list of types in some manner (change the types into pointers, perhaps) and give back the transformed result in the form it was given to us, std::tuple<int*, float*, void**> in the first case and std::packer<int*, float*, void**> in the second.

To do that, we need to first convert std::tuple<int, float, void*> to mp_list<int, float, void*>, apply add_pointer to each element obtaining mp_list<int*, float*, void**>, then convert that back to std::tuple.

These conversion steps are a quite common occurence, and we'll write a primitive that helps us perform them, called mp_rename. We want

```
mp_rename<std::tuple<int, float, void*>, mp_list>
```

to give us

```
mp_list<int, float, void*>
```

and conversely,

```
mp_rename<mp_list<int, float, void*>, std::tuple>
```

to give us

```
std::tuple<int, float, void*>
```

Here is the implementation of mp_rename:

```
template<class A, template<class...> class B> struct mp_rename_impl;

template<template<class...> class A, class... T, template<class...> class B>
   struct mp_rename_impl<A<T...>, B>
{
   using type = B<T...>;
};

template<class A, template<class...> class B>
   using mp_rename = typename mp_rename_impl<A, B>::type;
```

(This pattern of a template alias forwarding to a class template doing the actual work is common; class templates can be specialized, whereas template aliases cannot.)

Note that mp_rename does not treat any list type as special, not even mp_list; it can rename any variadic class template into any other. You could use it to rename std::packer to std::tuple to std::variant (once there is such a thing) and it will happily oblige.

In fact, it can even rename non-variadic class templates, as in the following examples:

```
mp_rename<std::pair<int, float>, std::tuple>       // -> std::tuple<int, float>
mp_rename<mp_list<int, float>, std::pair>          // -> std::pair<int, float>
mp_rename<std::shared_ptr<int>, std::unique_ptr>   // -> std::unique_ptr<int>
```

There is a limit to the magic; unique_ptr can't be renamed to shared_ptr:

```
mp_rename<std::unique_ptr<int>, std::shared_ptr>   // error
```

because unique_ptr<int> is actually unique_ptr<int, std::default_delete<int>> and mp_rename renames it to shared_ptr<int, std::default_delete<int>>, which doesn't compile. But it still works in many more cases than one would naively expect at first.

With conversions no longer a problem, let's move on to primitives and define a simple one, mp_size, for practice. We want mp_size<mp_list<T...>> to give us the number of elements in the list, that is, the value of the expression sizeof...(T).

```
template<class L> struct mp_size_impl;

template<class... T> struct mp_size_impl<mp_list<T...>>
{
   using type = std::integral_constant<std::size_t, sizeof...(T)>;
};

template<class L> using mp_size = typename mp_size_impl<L>::type;
```

This is relatively straightforward, except for the std::integral_constant. What is it and why do we need it?

std::integral_constant is a standard C++11 type that wraps an integral constant (that is, a compile-time constant integer value) into a type.

Since metaprogramming operates on type lists, which can only hold types, it's convenient to represent compile-time constants as types. This allows us to treat lists of types and lists of values in a uniform manner. It is therefore idiomatic in metaprogramming to take and return types instead of values, and this is what we have done. If at some later point we want the actual value, we can use the expression mp_size<L>::value to retrieve it.

We now have our mp_size, but you may have noticed that there's an interesting difference between mp_size and mp_rename. Whereas I made a point of mp_rename not treating mp_list as a special case, mp_size very much does:

```
template<class... T> struct mp_size_impl<mp_list<T...>>
```

Is this really necessary? Can we not use the same technique in the implementation of mp_size as we did in mp_rename?

```
template<class L> struct mp_size_impl;

template<template<class...> class L, class... T> struct mp_size_impl<L<T...>>
{
   using type = std::integral_constant<std::size_t, sizeof...(T)>;
};

template<class L> using mp_size = typename mp_size_impl<L>::type;
```

Yes, we very much can, and this improvement allows us to use mp_size on any other type lists, such as std::tuple. It turns mp_size into a truly generic primitive.

This is nice. It is so nice that I'd argue that all our metaprogramming primitives ought to have this property. If someone hands us a type list in the form of an std::tuple, we should be able to operate on it directly, avoiding the conversions to and from mp_list.

So do we no longer have any need for mp_rename? Not quite. Apart from the fact that sometimes we really do need to rename type lists, there is another surprising task for which mp_rename is useful.

To illustrate it, let me introduce the primitive mp_length. It's similar to mp_size, but while mp_size takes a type list as an argument, mp_length takes a variadic parameter pack and returns its length; or, stated differently, it returns its number of arguments:

```
template<class... T> using mp_length = std::integral_constant<std::size_t, sizeof...(T)>;
```

How would we implement mp_size in terms of mp_length? One option is to just substitute the implementation of the latter into the former:

```
template<template<class...> class L, class... T> struct mp_size_impl<L<T...>>
{
   using type = mp_length<T...>;
};
```

but there is another way, much less mundane. Think about what mp_size does. It takes the argument

```
mp_list<int, void, float>
```

and returns

```
mp_length<int, void, float>
```

Do we already have a primitive that does a similar thing?

(Not much of a choice, is there?)

Indeed we have, and it's called mp_rename.

```
template<class L> using mp_size = mp_rename<L, mp_length>;
```

I don't know about you, but I find this technique fascinating. It exploits the structural similarity between a list, L<T...>, and a metafunction "call", F<T...>, and the fact that the language sees the things the same way and allows us to pass the template alias mp_length to mp_rename as if it were an ordinary class template such as mp_list.

(Other metaprogramming libraries provide a dedicated apply primitive for this job. apply<F, L> calls the metafunction F with the contents of the list L. We'll add an alias mp_apply<F, L> that calls mp_rename<L, F> for readability.)

```
template<template<class...> class F, class L> using mp_apply = mp_rename<L, F>;
```

mp_transform

Let's revisit the example I gave earlier - someone hands us std::tuple<X, Y, Z> and we need to compute std::tuple<X*, Y*, Z*>. We already have add_pointer:

```
template<class T> using add_pointer = T*;
```

so we just need to apply it to each element of the input tuple.

The algorithm that takes a function and a list and applies the function to each element is called transform in Boost.MPL and the STL and map in functional languages. We'll use transform, for consistency with the established C++ practice (map is a data structure in both the STL and Boost.MPL.)

We'll call our algorithm mp_transform, and mp_transform<F, L> will apply F to each element of L and return the result. Usually, the argument order is reversed and the function comes last. Our reasons to put it at the front will become evident later.

There are many ways to implement mp_transform; the one we'll pick will make use of another primitive, mp_push_front. mp_push_front<L, T>, as its name implies, adds T as a first element in L:

```
template<class L, class T> struct mp_push_front_impl;

template<template<class...> class L, class... U, class T>
   struct mp_push_front_impl<L<U...>, T>
{
   using type = L<T, U...>;
};

template<class L, class T>
   using mp_push_front = typename mp_push_front_impl<L, T>::type;
```

There is no reason to constrain mp_push_front to a single element though. In C++11, variadic templates should be our default choice, and the implementation of mp_push_front that can take an arbitrary number of elements is almost identical:

```
template<class L, class... T> struct mp_push_front_impl;

template<template<class...> class L, class... U, class... T>
   struct mp_push_front_impl<L<U...>, T...>
{
   using type = L<T..., U...>;
};
```

```
template<class L, class... T>
   using mp_push_front = typename mp_push_front_impl<L, T...>::type;
```

On to mp_transform:

```
template<template<class...> class F, class L> struct mp_transform_impl;

template<template<class...> class F, class L>
   using mp_transform = typename mp_transform_impl<F, L>::type;

template<template<class...> class F, template<class...> class L>
   struct mp_transform_impl<F, L<>>
{
   using type = L<>;
};

template<template<class...> class F, template<class...> class L, class T1, class... T>
   struct mp_transform_impl<F, L<T1, T...>>
{
   using _first = F<T1>;
   using _rest = mp_transform<F, L<T...>>;

   using type = mp_push_front<_rest, _first>;
};
```

This is a straightforward recursive implementation that should be familiar to people with functional programming background.

Can we do better? It turns out that in C++11, we can.

```
template<template<class...> class F, class L> struct mp_transform_impl;

template<template<class...> class F, class L>
   using mp_transform = typename mp_transform_impl<F, L>::type;

template<template<class...> class F, template<class...> class L, class... T>
   struct mp_transform_impl<F, L<T...>>
{
   using type = L<F<T>...>;
};
```

Here we take advantage of the fact that pack expansion is built into the language, so the F<T>... part does all the iteration work for us.

We can now solve our original challenge: given an std::tuple of types, return an std::tuple of pointers to these types:

```
using input = std::tuple<int, void, float>;
using expected = std::tuple<int*, void*, float*>;

using result = mp_transform<add_pointer, input>;

static_assert( std::is_same<result, expected>::value, "" );
```

mp_transform**, part two**

What if we had a pair of tuples as input, and had to produce the corresponding tuple of pairs? For example, given

```
using input = std::pair<std::tuple<X1, X2, X3>, std::tuple<Y1, Y2, Y3>>;
```

we had to produce

```
using expected = std::tuple<std::pair<X1, Y1>, std::pair<X2, Y2>, std::pair<X3, Y3>>;
```

We need to take the two lists, represented by tuples in the input, and combine them pairwise by using std::pair. If we think of std::pair as a function F, this task appears very similar to mp_transform, except we need to use a binary function and two lists.

Changing our unary transform algorithm into a binary one isn't hard:

```
template<template<class...> class F, class L1, class L2>
   struct mp_transform2_impl;
```

```
template<template<class...> class F, class L1, class L2>
   using mp_transform2 = typename mp_transform2_impl<F, L1, L2>::type;

template<template<class...> class F,
   template<class...> class L1, class... T1,
   template<class...> class L2, class... T2>
   struct mp_transform2_impl<F, L1<T1...>, L2<T2...>>
{
   static_assert( sizeof...(T1) == sizeof...(T2),
      "The arguments of mp_transform2 should be of the same size" );

   using type = L1<F<T1,T2>...>;
};
```

and we can now do

```
using input = std::pair<std::tuple<X1, X2, X3>, std::tuple<Y1, Y2, Y3>>;
using expected = std::tuple<std::pair<X1, Y1>, std::pair<X2, Y2>, std::pair<X3, Y3>>;

using result = mp_transform2<std::pair, input::first_type, input::second_type>;

static_assert( std::is_same<result, expected>::value, "" );
```

again exploiting the similarity between metafunctions and ordinary class templates such as std::pair, this time in the other direction; we pass std::pair where mp_transform2 expects a metafunction.

Do we *have* to use separate transform algorithms for each arity though? If we need a transform algorithm that takes a ternary function and three lists, should we name it mp_transform3? No, this is exactly why we put the function first. We just have to change mp_transform to be variadic:

```
template<template<class...> class F, class... L> struct mp_transform_impl;

template<template<class...> class F, class... L>
   using mp_transform = typename mp_transform_impl<F, L...>::type;
```

and then add the unary and binary specializations:

```
template<template<class...> class F, template<class...> class L, class... T>
   struct mp_transform_impl<F, L<T...>>
{
   using type = L<F<T>...>;
};

template<template<class...> class F,
   template<class...> class L1, class... T1,
   template<class...> class L2, class... T2>
   struct mp_transform_impl<F, L1<T1...>, L2<T2...>>
{
   static_assert( sizeof...(T1) == sizeof...(T2),
      "The arguments of mp_transform should be of the same size" );

   using type = L1<F<T1,T2>...>;
};
```

We can also add ternary and further specializations.

Is it possible to implement the truly variadic mp_transform, one that works with an arbitrary number of lists? It is in principle, and I'll show one possible abridged implementation here for completeness:

```
template<template<class...> class F, class E, class... L>
   struct mp_transform_impl;

template<template<class...> class F, class... L>
   using mp_transform = typename mp_transform_impl<F, mp_empty<L...>, L...>::type;

template<template<class...> class F, class L1, class... L>
   struct mp_transform_impl<F, mp_true, L1, L...>
{
   using type = mp_clear<L1>;
};

template<template<class...> class F, class... L>
   struct mp_transform_impl<F, mp_false, L...>
```

```
{
   using _first = F< typename mp_front_impl<L>::type... >;
   using _rest = mp_transform< F, typename mp_pop_front_impl<L>::type... >;

   using type = mp_push_front<_rest, _first>;
};
```

but will omit the primitives that it uses. These are

- mp_true — an alias for std::integral_constant<bool, true>.
- mp_false — an alias for std::integral_constant<bool, false>.
- mp_empty<L...> — returns mp_true if all lists are empty, mp_false otherwise.
- mp_clear<L> — returns an empty list of the same type as L.
- mp_front<L> — returns the first element of L.
- mp_pop_front<L> — returns L without its first element.

There is one interesting difference between the recursive mp_transform implementation and the language-based one. mp_transform<add_pointer, std::pair<int, float>> works with the F<T>... implementation and fails with the recursive one, because std::pair is not a real type list and can only hold exactly two types.

The infamous tuple_cat challenge

Eric Niebler, in his Tiny Metaprogramming Library article, gives the function std::tuple_cat as a kind of a metaprogramming challenge. tuple_cat is a variadic template function that takes a number of tuples and concatenates them into another std::tuple. This is Eric's solution:

```
namespace detail
{
   template<typename Ret, typename...Is, typename ...Ks,
      typename Tuples>
   Ret tuple_cat_(typelist<Is...>, typelist<Ks...>,
      Tuples tpls)
   {
      return Ret{std::get<Ks::value>(
         std::get<Is::value>(tpls))...};
   }
}

template<typename...Tuples,
   typename Res =
      typelist_apply_t<
         meta_quote<std::tuple>,
         typelist_cat_t<typelist<as_typelist_t<Tuples>...> > > >
Res tuple_cat(Tuples &&... tpls)
{
   static constexpr std::size_t N = sizeof...(Tuples);
   // E.g. [0,0,0,2,2,2,3,3]
   using inner =
      typelist_cat_t<
         typelist_transform_t<
            typelist<as_typelist_t<Tuples>...>,
            typelist_transform_t<
               as_typelist_t<make_index_sequence<N> >,
               meta_quote<meta_always> >,
            meta_quote<typelist_transform_t> > >;
   // E.g. [0,1,2,0,1,2,0,1]
   using outer =
      typelist_cat_t<
         typelist_transform_t<
            typelist<as_typelist_t<Tuples>...>,
            meta_compose<
               meta_quote<as_typelist_t>,
               meta_quote_i<std::size_t, make_index_sequence>,
               meta_quote<typelist_size_t> > > >;
   return detail::tuple_cat_<Res>(
      inner{},
      outer{},
      std::forward_as_tuple(std::forward<Tuples>(tpls)...));
}
```

All right, challenge accepted. Let's see what we can do.

As Eric explains, this implementation relies on the clever trick of packing the input tuples into a tuple, creating two arrays of indices, inner and outer, then indexing the outer tuple with the outer indices and the result, which is one of our input tuples, with the inner indices.

So, for example, if tuple_cat is invoked as

```
std::tuple<int, short, long> t1;
std::tuple<> t2;
std::tuple<float, double, long double> t3;
std::tuple<void*, char*> t4;

auto res = tuple_cat(t1, t2, t3, t4);
```

we'll create the tuple

```
std::tuple<std::tuple<int, short, long>, std::tuple<>,
   std::tuple<float, double, long double>, std::tuple<void*, char*>> t{t1, t2, t3, t4};
```

and then extract the elements of t via

```
   std::get<0>(std::get<0>(t)), // t1[0]
   std::get<1>(std::get<0>(t)), // t1[1]
   std::get<2>(std::get<0>(t)), // t1[2]
   std::get<0>(std::get<2>(t)), // t3[0]
   std::get<1>(std::get<2>(t)), // t3[1]
   std::get<2>(std::get<2>(t)), // t3[2]
   std::get<0>(std::get<3>(t)), // t4[0]
   std::get<1>(std::get<3>(t)), // t4[1]
```

(t2 is empty, so we take nothing from it.)

The first column of integers is the outer array, the second one - the inner array, and these are what we need to compute. But first, let's deal with the return type of tuple_cat.

The return type of tuple_cat is just the concatenation of the arguments, viewed as type lists. The metaprogramming algorithm that concatenates lists is called meta::concat in Eric Niebler's Meta library, but I'll call it mp_append, after its classic Lisp name.

(Lisp is today's equivalent of Latin. Educated people are supposed to have studied and forgotten it.)

```
template<class... L> struct mp_append_impl;

template<class... L> using mp_append = typename mp_append_impl<L...>::type;

template<> struct mp_append_impl<>
{
   using type = mp_list<>;
};

template<template<class...> class L, class... T> struct mp_append_impl<L<T...>>
{
   using type = L<T...>;
};

template<template<class...> class L1, class... T1,
   template<class...> class L2, class... T2, class... Lr>
   struct mp_append_impl<L1<T1...>, L2<T2...>, Lr...>
{
   using type = mp_append<L1<T1..., T2...>, Lr...>;
};
```

That was fairly easy. There are other ways to implement mp_append, but this one demonstrates how the language does most of the work for us via pack expansion. This is a common theme in C++11.

Note how mp_append returns the same list type as its first argument. Of course, in the case in which no arguments are given, there is no first argument from which to take the type, so I've arbitrarily chosen to return an empty mp_list.

We're now ready with the declaration of tuple_cat:

```
template<class... Tp,
   class R = mp_append<typename std::remove_reference<Tp>::type...>>
   R tuple_cat( Tp &&... tp );
```

The reason we need remove_reference is because of the rvalue reference parameters, used to implement perfect forwarding. If the argument is an lvalue, such as for example t1 above, its corresponding type will be a reference to a tuple — std::tuple<int, short, long>& in t1's case. Our primitives do not recognize references to tuples as type lists, so we need to strip them off.

There are two problems with our return type computation though. One, what if tuple_cat is called without any arguments? We return mp_list<> in that case, but the correct result is std::tuple<>.

Two, what if we call tuple_cat with a first argument that is a std::pair? We'll try to append more elements to std::pair, and it will fail.

We can solve both our problems by using an empty tuple as the first argument to mp_append:

```
template<class... Tp,
    class R = mp_append<std::tuple<>, typename std::remove_reference<Tp>::type...>>
    R tuple_cat( Tp &&... tp );
```

With the return type taken care of, let's now move on to computing inner. We have

```
[x1, x2, x3], [], [y1, y2, y3], [z1, z2]
```

as input and we need to output

```
[0, 0, 0, 2, 2, 2, 3, 3]
```

which is the concatenation of

```
[0, 0, 0], [], [2, 2, 2], [3, 3]
```

Here each tuple is the same size as the input, but is filled with a constant that represents its index in the argument list. The first tuple is filled with 0, the second with 1, the third with 2, and so on.

We can achieve this result if we first compute a list of indices, in our case [0, 1, 2, 3], then use binary mp_transform on the two lists

```
[[x1, x2, x3], [], [y1, y2, y3], [z1, z2]]
[0, 1, 2, 3]
```

and a function which takes a list and an integer (in the form of an std::integral_constant) and returns a list that is the same size as the original, but filled with the second argument.

We'll call this function mp_fill, after std::fill.

Functional programmers will immediately realize that mp_fill is mp_transform with a function that returns a constant, and here's an implementation along these lines:

```
template<class V> struct mp_constant
{
   template<class...> using apply = V;
};

template<class L, class V>
   using mp_fill = mp_transform<mp_constant<V>::template apply, L>;
```

Here's an alternate implementation:

```
template<class L, class V> struct mp_fill_impl;

template<template<class...> class L, class... T, class V>
   struct mp_fill_impl<L<T...>, V>
{
   template<class...> using _fv = V;
   using type = L<_fv<T>...>;
};

template<class L, class V> using mp_fill = typename mp_fill_impl<L, V>::type;
```

These demonstrate different styles and choosing one over the other is largely a matter of taste here. In the first case, we combine existing primitives; in the second case, we "inline" mp_const and even mp_transform in the body of mp_fill_impl.

Most C++11 programmers will probably find the second implementation easier to read.

We can now mp_fill, but we still need the [0, 1, 2, 3] index sequence. We could write an algorithm mp_iota for that (named after std::iota), but it so happens that C++14 already has a standard way of generating an index sequence, called std::make_index_sequence. Since Eric's original solution makes use of make_index_sequence, let's follow his lead.

Technically, this takes us outside of C++11, but make_index_sequence is not hard to implement (if efficiency is not a concern):

```
template<class T, T... Ints> struct integer_sequence
```

```
{
};

template<class S> struct next_integer_sequence;

template<class T, T... Ints> struct next_integer_sequence<integer_sequence<T, Ints...>>
{
   using type = integer_sequence<T, Ints..., sizeof...(Ints)>;
};

template<class T, T I, T N> struct make_int_seq_impl;

template<class T, T N>
   using make_integer_sequence = typename make_int_seq_impl<T, 0, N>::type;

template<class T, T I, T N> struct make_int_seq_impl
{
   using type = typename next_integer_sequence<
      typename make_int_seq_impl<T, I+1, N>::type>::type;
};

template<class T, T N> struct make_int_seq_impl<T, N, N>
{
   using type = integer_sequence<T>;
};

template<std::size_t... Ints>
   using index_sequence = integer_sequence<std::size_t, Ints...>;

template<std::size_t N>
   using make_index_sequence = make_integer_sequence<std::size_t, N>;
```

We can now obtain an index_sequence<0, 1, 2, 3>:

```
template<class... Tp,
   class R = mp_append<std::tuple<>, typename std::remove_reference<Tp>::type...>>
   R tuple_cat( Tp &&... tp )
{
   std::size_t const N = sizeof...(Tp);

   // inner

   using seq = make_index_sequence<N>;
}
```

but make_index_sequence<4> returns integer_sequence<std::size_t, 0, 1, 2, 3>, which is not a type list. In order to work with it, we need to convert it to a type list, so we'll introduce a function mp_from_sequence that does that.

```
template<class S> struct mp_from_sequence_impl;

template<template<class T, T... I> class S, class U, U... J>
   struct mp_from_sequence_impl<S<U, J...>>
{
   using type = mp_list<std::integral_constant<U, J>...>;
};

template<class S> using mp_from_sequence = typename mp_from_sequence_impl<S>::type;
```

We can now compute the two lists that we wanted to transform with mp_fill:

```
template<class... Tp,
   class R = mp_append<std::tuple<>, typename std::remove_reference<Tp>::type...>>
   R tuple_cat( Tp &&... tp )
{
   std::size_t const N = sizeof...(Tp);

   // inner

   using list1 = mp_list<typename std::remove_reference<Tp>::type...>;
   using list2 = mp_from_sequence<make_index_sequence<N>>;

   // list1: [[x1, x2, x3], [], [y1, y2, y3], [z1, z2]]
   // list2: [0, 1, 2, 3]
```

```
    return R{};
}
```

and finish the job of computing inner:

```
template<class... Tp,
    class R = mp_append<std::tuple<>, typename std::remove_reference<Tp>::type...>>
    R tuple_cat( Tp &&... tp )
{
    std::size_t const N = sizeof...(Tp);

    // inner

    using list1 = mp_list<typename std::remove_reference<Tp>::type...>;
    using list2 = mp_from_sequence<make_index_sequence<N>>;

    // list1: [[x1, x2, x3], [], [y1, y2, y3], [z1, z2]]
    // list2: [0, 1, 2, 3]

    using list3 = mp_transform<mp_fill, list1, list2>;

    // list3: [[0, 0, 0], [], [2, 2, 2], [3, 3]]

    using inner = mp_rename<list3, mp_append>; // or mp_apply<mp_append, list3>

    // inner: [0, 0, 0, 2, 2, 2, 3, 3]

    return R{};
}
```

For outer, we again have

```
[x1, x2, x3], [], [y1, y2, y3], [z1, z2]
```

as input and we need to output

```
[0, 1, 2, 0, 1, 2, 0, 1]
```

which is the concatenation of

```
[0, 1, 2], [], [0, 1, 2], [0, 1]
```

The difference here is that instead of filling the tuple with a constant value, we need to fill it with increasing values, starting from 0, that is, with the result of make_index_sequence<N>, where N is the number of elements.

The straightforward way to do that is to just define a metafunction F that does what we want, then use mp_transform to apply it to the input:

```
template<class N> using mp_iota = mp_from_sequence<make_index_sequence<N::value>>;

template<class L> using F = mp_iota<mp_size<L>>;

template<class... Tp,
    class R = mp_append<std::tuple<>, typename std::remove_reference<Tp>::type...>>
    R tuple_cat( Tp &&... tp )
{
    std::size_t const N = sizeof...(Tp);

    // outer

    using list1 = mp_list<typename std::remove_reference<Tp>::type...>;
    using list2 = mp_transform<F, list1>;

    // list2: [[0, 1, 2], [], [0, 1, 2], [0, 1]]

    using outer = mp_rename<list2, mp_append>;

    // outer: [0, 1, 2, 0, 1, 2, 0, 1]

    return R{};
}
```

Well that was easy. Surprisingly easy. The one small annoyance is that we can't define F inside tuple_cat - templates can't be defined in functions.

Let's put everything together.

```
template<class N> using mp_iota = mp_from_sequence<make_index_sequence<N::value>>;

template<class L> using F = mp_iota<mp_size<L>>;

template<class R, class...Is, class... Ks, class Tp>
R tuple_cat_( mp_list<Is...>, mp_list<Ks...>, Tp tp )
{
   return R{ std::get<Ks::value>(std::get<Is::value>(tp))... };
}

template<class... Tp,
   class R = mp_append<std::tuple<>, typename std::remove_reference<Tp>::type...>>
   R tuple_cat( Tp &&... tp )
{
   std::size_t const N = sizeof...(Tp);

   // inner

   using list1 = mp_list<typename std::remove_reference<Tp>::type...>;
   using list2 = mp_from_sequence<make_index_sequence<N>>;

   // list1: [[x1, x2, x3], [], [y1, y2, y3], [z1, z2]]
   // list2: [0, 1, 2, 3]

   using list3 = mp_transform<mp_fill, list1, list2>;

   // list3: [[0, 0, 0], [], [2, 2, 2], [3, 3]]

   using inner = mp_rename<list3, mp_append>; // or mp_apply<mp_append, list3>

   // inner: [0, 0, 0, 2, 2, 2, 3, 3]

   // outer

   using list4 = mp_transform<F, list1>;

   // list4: [[0, 1, 2], [], [0, 1, 2], [0, 1]]

   using outer = mp_rename<list4, mp_append>;

   // outer: [0, 1, 2, 0, 1, 2, 0, 1]

   return tuple_cat_<R>( inner(), outer(),
      std::forward_as_tuple( std::forward<Tp>(tp)... ) );
}
```

This almost compiles, except that our inner happens to be a std::tuple, but our helper function expects an mp_list. (outer is already an mp_list, by sheer luck.) We can fix that easily enough.

```
   return tuple_cat_<R>( mp_rename<inner, mp_list>(), outer(),
      std::forward_as_tuple( std::forward<Tp>(tp)... ) );
```

Let's define a print_tuple function and see if everything checks out.

```
template<int I, int N, class... T> struct print_tuple_
{
   void operator()( std::tuple<T...> const & tp ) const
   {
      using Tp = typename std::tuple_element<I, std::tuple<T...>>::type;

      print_type<Tp>( " ", ": " );

      std::cout << std::get<I>( tp ) << ";";

      print_tuple_< I+1, N, T... >()( tp );
   }
};

template<int N, class... T> struct print_tuple_<N, N, T...>
```

```
{
   void operator()( std::tuple<T...> const & ) const
   {
   }
};

template<class... T> void print_tuple( std::tuple<T...> const & tp )
{
   std::cout << "{";
   print_tuple_<0, sizeof...(T), T...>()( tp );
   std::cout << " }\n";
}

int main()
{
   std::tuple<int, long> t1{ 1, 2 };
   std::tuple<> t2;
   std::tuple<float, double, long double> t3{ 3, 4, 5 };
   std::pair<void const*, char const*> t4{ "pv", "test" };

   using expected = std::tuple<int, long, float, double, long double,
      void const*, char const*>;

   auto result = ::tuple_cat( t1, t2, t3, t4 );

   static_assert( std::is_same<decltype(result), expected>::value, "" );

   print_tuple( result );
}
```

Output:

```
{ int: 1; long: 2; float: 3; double: 4; long double: 5; void const*: 0x407086;
   char const*: test; }
```

Seems to work. But there's at least one error left. To see why, replace the first tuple

```
   std::tuple<int, long> t1{ 1, 2 };
```

with a pair:

```
   std::pair<int, long> t1{ 1, 2 };
```

We now get an error at

```
   using inner = mp_rename<list3, mp_append>;
```

because the first element of list3 is an std::pair, which mp_append tries and fails to use as its return type.

There are two ways to fix that. The first one is to apply the same trick we used for the return type, and insert an empty mp_list at the front of list3, which mp_append will use as a return type:

```
   using inner = mp_rename<mp_push_front<list3, mp_list<>>, mp_append>;
```

The second way is to just convert all inputs to mp_list:

```
   using list1 = mp_list<
      mp_rename<typename std::remove_reference<Tp>::type, mp_list>...>;
```

In both cases, inner will now be an mp_list, so we can omit the mp_rename in the call to tuple_cat_.

We're done. The results hopefully speak for themselves.

Higher order metaprogramming, or lack thereof

Perhaps by now you're wondering why this article is called "Simple C++11 metaprogramming", since what we covered so far wasn't particularly simple.

The *relative* simplicity of our approach stems from the fact that we've not been doing any higher order metaprogramming, that is, we haven't introduced any primitives that return metafunctions, such as compose, bind, or a lambda library.

I posit that such higher order metaprogramming is, in the majority of cases, not necessary in C++11. Consider, for example, Eric Niebler's solution given above:

```
using outer =
   typelist_cat_t<
      typelist_transform_t<
         typelist<as_typelist_t<Tuples>...>,
         meta_compose<
            meta_quote<as_typelist_t>,
            meta_quote_i<std::size_t, make_index_sequence>,
            meta_quote<typelist_size_t> > > >;
```

The meta_compose expression takes three other ("quoted") metafunctions and creates a new metafunction that applies them in order. Eric uses this example as motivation to introduce the concept of a "metafunction class" and then to supply various primitives that operate on metafunction classes.

But when we have metafunctions F, G and H, instead of using meta_compose, in C++11 we can just do

```
template<class... T> using Fgh = F<G<H<T...>>>;
```

and that's it. The language makes defining composite functions easy, and there is no need for library support. If the functions to be composed are as_typelist_t, std::make_index_sequence and typelist_size_t, we just define

```
template<class... T>
   using F = as_typelist_t<std::make_index_sequence<typelist_size_t<T...>::value>>;
```

Similarly, if we need a metafunction that will return sizeof(T) < sizeof(U), there is no need to enlist a metaprogramming lambda library as in

```
lambda<_a, _b, less<sizeof_<_a>, sizeof_<_b>>>>
```

We could just define it inline:

```
template<class T, class U> using sizeof_less = mp_bool<(sizeof(T) < sizeof(U))>;
```

One more thing

Finally, let me show the implementations of mp_count and mp_count_if, for no reason other than I find them interesting. mp_count<L, V> returns the number of occurences of the type V in the list L; mp_count_if<L, P> counts the number of types in L for which P<T> is true.

As a first step, I'll implement mp_plus. mp_plus is a variadic (not just binary) metafunction that returns the sum of its arguments.

```
template<class... T> struct mp_plus_impl;

template<class... T> using mp_plus = typename mp_plus_impl<T...>::type;

template<> struct mp_plus_impl<>
{
   using type = std::integral_constant<int, 0>;
};

template<class T1, class... T> struct mp_plus_impl<T1, T...>
{
   static constexpr auto _v = T1::value + mp_plus<T...>::value;

   using type = std::integral_constant<
      typename std::remove_const<decltype(_v)>::type, _v>;
};
```

Now that we have mp_plus, mp_count is just

```
template<class L, class V> struct mp_count_impl;

template<template<class...> class L, class... T, class V>
   struct mp_count_impl<L<T...>, V>
{
   using type = mp_plus<std::is_same<T, V>...>;
};

template<class L, class V> using mp_count = typename mp_count_impl<L, V>::type;
```

This is another illustration of the power of parameter pack expansion. It's a pity that we can't use pack expansion in mp_plus

as well, to obtain

```
T1::value + T2::value + T3::value + T4::value + ...
```

directly. It would have been nice for T::value + ... to have been supported, and it appears that in C++17, it will be.

mp_count_if is similarly straightforward:

```
template<class L, template<class...> class P> struct mp_count_if_impl;

template<template<class...> class L, class... T, template<class...> class P>
   struct mp_count_if_impl<L<T...>, P>
{
   using type = mp_plus<P<T>...>;
};

template<class L, template<class...> class P>
   using mp_count_if = typename mp_count_if_impl<L, P>::type;
```

at least if we require P to return bool. If not, we'll have to coerce P<T>::value to 0 or 1, or the count will not be correct.

```
template<bool v> using mp_bool = std::integral_constant<bool, v>;

template<class L, template<class...> class P> struct mp_count_if_impl;

template<template<class...> class L, class... T, template<class...> class P>
   struct mp_count_if_impl<L<T...>, P>
{
   using type = mp_plus<mp_bool<P<T>::value != 0>...>;
};

template<class L, template<class...> class P>
   using mp_count_if = typename mp_count_if_impl<L, P>::type;
```

The last primitive I'll show is mp_contains. mp_contains<L, V> returns whether the list L contains the type V:

```
template<class L, class V> using mp_contains = mp_bool<mp_count<L, V>::value != 0>;
```

At first sight, this implementation appears horribly naive and inefficient — why would we need to count all the occurences just to throw that away if we're only interested in a boolean result — but it's actually pretty competitive and perfectly usable. We just need to add one slight optimization to mp_plus, the engine behind mp_count and mp_contains:

```
template<class T1, class T2, class T3, class T4, class T5,
   class T6, class T7, class T8, class T9, class T10, class... T>
   struct mp_plus_impl<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T...>
{
   static constexpr auto _v = T1::value + T2::value + T3::value + T4::value +
      T5::value + T6::value + T7::value + T8::value + T9::value + T10::value +
      mp_plus<T...>::value;

   using type = std::integral_constant<
      typename std::remove_const<decltype(_v)>::type, _v>;
};
```

This cuts the number of template instantiations approximately tenfold.

Conclusion

I have outlined an approach to metaprogramming in C++11 that

- takes advantage of variadic templates, parameter pack expansion, and template aliases;
- operates on any variadic template L<T...>, treating it as its fundamental data structure, without mandating a specific type list representation;
- uses template aliases as its metafunctions, with the expression F<T...> serving as the equivalent of a function call;
- exploits the structural similarity between the data structure L<T...> and the metafunction call F<T...>;
- leverages parameter pack expansion as much as possible, instead of using the traditional recursive implementations;
- relies on inline definitions of template aliases for function composition, instead of providing library support for this task.

Further reading

Part 2 is now available, in which I show algorithms that allow us to treat type lists as sets, maps, and vectors, and demonstrate various C++11 implementation techniques in the process.
