Eric Niebler

Judge me by my C++, not my WordPress

Tiny Metaprogramming Library

Posted on **November 13, 2014** by **Eric Niebler** —

(Difficult-to-grok metaprogramming below. Not for the faint of heart.)

At the recent Urbana-Champaign meeting of the C++ Standardization Committee, Bill Seymour presented his paper N4115: Searching for Types in Parameter Packs which, as its name suggests, describes a library facility for, uh, searching for a type in a parameter pack, among other things. It suggests a template called `packer` to hold a parameter pack:

```
1 // A class template that just holds a parameter pack:
2 template <class... T> struct packer { };
```

Many of you are probably already familiar with such a facility, but under a different name:

```
1 // A class template that is just a list of types:
2 template <class... T> struct typelist { };
```

It became clear in the discussion about N4115 that C++ needs a standard `typelist` template and some utilities for manipulating them. But what utilities, exactly?

Metaprogramming in the Wild

When it comes to metaprogramming in C++, there is no shortage of prior art. Andrei Alexandrescu started the craze with his Loki library. Boost got in on the act with Boost.MPL, Boost.Fusion, and (currently under development) Hana. All of these libraries are feature-rich and elaborate with their own philosophy, especially Boost.MPL, which takes inspiration from the STL's containers, iterators, and algorithms.

It wasn't until recently that I came to doubt MPL's slavish aping of the STL's design. The abstractions of the STL were condensed from real algorithms processing real data structures on real computer hardware. But metaprograms don't run on hardware; they run on compilers. The algorithms and data structures for our metaprograms should be tailored to their peculiar problem domain and execution environment. If we did that exercise, who is to say what abstractions would fall out? Compile-time iterators? Or something else entirely?

Dumb Typelists

If we were to standardize some metaprogramming facilities, what should they look like? It's an interesting question. N4115 gets one thing right: parameter packs are the compile-time data structure of choice. As of C++11, C++ has *language support* for lists of types. We would be foolish to work with anything else. IMO, if a standard metaprogramming facility did *nothing* but manipulate parameter packs — dumb typelists — it would cover 95% of the problem space.

But parameter packs themselves are not first-class citizens of the language. You can't pass a parameter pack to a function without expanding it, for instance. Wrapping the parameter pack in a variadic `typelist` template is a no-brainer.

So, like N4115 suggests, this is a sensible starting point:

```
1 // A class template that just a list of types:
2 template <class... T> struct typelist { };
```

It's a rather inauspicious start, though; clearly we need more. But what? In order to answer that, we need to look at examples of real-world metaprogramming. With concrete examples, we can answer the question, What he heck is this stuff *good* for, anyway? And for examples, we have to look no farther than the standard library itself.

Tuple_cat

Stephan T. Lavavej drew my attention to the `tuple_cat` function in the standard library, a function that takes N `tuples` and glues them together

into one. It sounds easy, but it's tricky to code efficiently, and it turns out to be a great motivating example for metaprogramming facilities. Let's code it up, and posit a few typelist algorithms to make our job easier. (All the code described here can be found in my range-v3 library on GitHub.)

First, I'm going to present the final solution so you have an idea of what we're working toward. Hopefully, by the time you make it to the end of this post, this will make some sort of sense.

```
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)...));
}
```

That's only 43 lines of code. The implementation in stdlib++ is 3x longer, no easier to understand (IMHO), *and* less efficient. There's real value in this stuff. Really.

Let's look first at the return type:

```
// What return type???
template< typename ...Tuples >
???? tuple_cat( Tuples &&... tpls );
```

You can think of a tuple as a list of types *and* a list of values. To compute the return type, we only need the list of types. So a template that turns a tuple into a typelist would be useful. Let's call it `as_typelist`. It takes a tuple and does the obvious. (Another possibility would be to make tuples usable as typelists, but let's go with this for now.)

If we convert all the tuples into typelists, we end up with a list of typelists. Now, we want to concatenate them. Ah! We need an algorithm for that. Let's call it `typelist_cat` in honor of `tuple_cat`. (Functional programmers: typelist_cat is join in the List Monad. Shhh!! Pass it on.) Here's what we have so far:

```
// Concatenate a typelist of typelists
template< typename ...Tuples >
typelist_cat_t<
    typelist< as_typelist_t< Tuples >... >
>
tuple_cat( Tuples &&... tpls );
```

Here, I'm following the convention in C++14 that `some_trait_t<X>` is a template alias for `typename some_trait<X>::type`.

The above signature isn't right yet — `tuple_cat` needs to return a `tuple`, not a `typelist`. We need a way to convert a typelist back to a tuple. It turns out that expanding a typelist into a variadic template is a useful operation, so let's create an algorithm for it. What should it be called? Expanding a typelist into a template is a lot like expanding a tuple into a function call. There's a tuple algorithm for that in the Library Fundamentals TS called `apply`. So let's call our metafunction `typelist_apply`. It's implementation is short and interesting, so I'll show it here:

```
template<template<typename...> class C, typename List>
struct typelist_apply;
```

```
3
4  template<template<typename...> class C, typename...List>
5  struct typelist_apply<C, typelist<List...>>
6  {
7      using type = C<List...>;
8  };
```

The first parameter is a rarely-seen template template parameter. We'll tweak this interface before we're done, but this is good enough for now.

We can now write the signature of `tuple_cat` as:

```
1  template<typename...Tuples>
2  typelist_apply_t<
3      std::tuple,
4      typelist_cat_t<typelist<as_typelist_t<Tuples>...> > >
5  tuple_cat(Tuples &&... tpls);
```

Not bad, and we have discovered three typelist algorithms already.

Tuple_cat Implementation

It's time to implement `tuple_cat`, and here's where things get weird. It's possible to implement it by peeling off the first tuple and exploding it into the tail of a recursive call. Once you've recursed over all the tuples in the argument list, you have exploded all the tuple elements into function arguments. From there, you bundle them into a final tuple and you're done.

That's a lot of parameter passing.

Stephan T. Lavavej tipped me off to a better way: Take all the tuples and bundle them up into a tuple-of-tuples with `std::forward_as_tuple`. Since tuples are random-access, a tuple of tuples is like a jagged 2-dimensional array of elements. We can index into this 2-dimensional array with *(i,j)* coordinates, and if we have the right list of *(i,j)* pairs, then we can fetch each element in turn and build the resulting tuple in one shot, without all the explosions.

To make this more concrete, imaging the following call to `tuple_cat`:

```
1  std::tuple<int, short, long> t1;
2  std::tuple<> t2;
3  std::tuple<float, double, long double> t3;
4  std::tuple<void*, char*> t4;
5
6  auto res = tuple_cat(t1,t2,t3,t4);
```

We want the result to be a monster tuple of type:

```
1  std::tuple<int, short, long, float, double,
2             long double, void*, char*>
```

This call to `tuple_cat` corresponds to the following list of *(i,j)* coordinates:

```
1  [(0,0),(0,1),(0,2),(2,0),(2,1),(2,2),(3,0),(3,1)]
```

Below is a `tuple_cat_` helper function that takes the *i*'s, *j*'s, and tuple of tuples, and builds the resulting tuple:

```
1  template<typename Ret, typename...Is, typename ...Js,
2      typename Tuples>
3  Ret tuple_cat_(typelist<Is...>, typelist<Js...>,
4      Tuples tpls)
5  {
6      return Ret{std::get<Js::value>(
7          std::get<Is::value>(tpls))...};
8  }
```

Here, the `Is` and `Js` are instances of `std::integral_constant`. `Is` contains the sequence [0,0,0,2,2,2,3,3] and `Js` contains [0,1,2,0,1,2,0,1].

Well and good, but how to compute `Is` and `Js`? Hang on tight, because Kansas is going bye bye.

Higher-Order Metaprogramming, Take 1

Let's first consider the sequence of `Js` since that's a little easier. Our job is to turn a list of typelists [[int,short,long],[],[float,double,long double],[void*,char*]] into a list of integers [0,1,2,0,1,2,0,1]. We can do it in four stages:

1. Transform the lists of typelist into a list of typelist *sizes*: [3,0,3,2],
2. Transform that to a list of index sequences [[0,1,2],[],[0,1,2],[0,1]] using `std::make_index_sequence`,

3. Transform the `std::index_sequence` into a typelist of `std::integral_constants` with `as_typelist`, and
4. Flatten that into the final list using `typelist_cat`.

By now it's obvious that we've discovered our fourth typelist algorithm: `typelist_transform`. Like `std::transform`, `typelist_transform` takes a sequence and a function, and returns a new sequence where each element has been transformed by the function. (Functional programmers: it's fmap in the List Functor). Here's one possible implementation:

```
template<typename List, template<class> class Fun>
struct typelist_transform;

template<typename ...List, template<class> class Fun>
struct typelist_transform<typelist<List...>, Fun>
{
    using type = typelist<Fun<List>...>;
};
```

Simple enough.

Metafunction Composition

Above, we suggested three consecutive passes with `typelist_transform`. We can do this all in one pass if we compose the three metafunctions into one. Metafunction composition seems like a very important utility, and it's not specific to typelist manipulation. So far, we've been using template template parameters to pass metafunctions to other metafunctions. What does metafunction composition look like in that world? Below is a higher-order metafunction called `meta_compose` that composes two other metafunctions:

```
template<template<class> class F0,
         template<class> class F1>
struct meta_compose
{
    template<class T>
    using apply = F0<F1<T>>;
};
```

Composing two metafunction has to result in a new metafunction. We have to use an idiom to "return" a template by defining a nested template alias `apply` which does the composition.

Seems simple enough, but in practice, this quickly becomes unwieldy. If you want to compose three metafunctions, the code looks like:

```
meta_compose<F0, meta_compose<F1, F2>::template apply>
    ::template apply
```

Gross. What's worse, it's not very general. We want to compose `std::make_index_sequence`, and that metafunction doesn't take a type; it takes an integer. We can't pass it to to a `meta_compose`. Let's back up.

Higher-Order Metaprogramming, Take 2

What if, instead of passing `meta_compose<X,Y>::template apply` to a higher-order function like `typelist_transform`, we just passed `meta_compose<X,Y>` and let `typelist_transform` call the nested `apply`? Now, higher-order functions like `typelist_transform` take ordinary types instead of template template parameters. `typelist_transform` would now look like:

```
template<typename ...List, typename Fun>
struct typelist_transform<typelist<List...>, Fun>
{
    using type =
        typelist<typename Fun::template apply<List>...>;
};
```

That complicates the implementation of `typelist_transform`, but makes the interface much nicer to deal with. The concept of a class type that behaves like a metafunction comes from Boost.MPL, which calls it a *Metafunction Class*.

We can make Metafunction Classes easier to deal with with a little helper that applies the nested metafunction to a set of arguments:

```
template<typename F, typename...As>
using meta_apply = typename F::template apply<As...>;
```

With `meta_apply`, we can rewrite `typelist_transform` as:

```
template<typename ...List, typename Fun>
struct typelist_transform<typelist<List...>, Fun>
{
    using type = typelist<meta_apply<Fun, List>...>;
};
```

That's not bad at all. Now we can change `meta_compose` to also operate on Metafunction Classes:

```
template<typename F1, typename F2>
struct meta_compose
{
    template<class T>
    using apply = meta_apply<F1, meta_apply<F2, T>>;
};
```

With a little more work, we could even make it accept an arbitrary number of Metafunction Classes and compose them all. It's a fun exercise; give it a shot.

Lastly, now that we have Metafunction Classes, we should change `typelist_apply` to take a Metafunction Class instead of a template template parameter:

```
template<typename C, typename...List>
struct typelist_apply<C, typelist<List...> >
{
    using type = meta_apply<C, List...>;
};
```

Metafunctions to Metafunction Classes

Recall the four steps we're trying to evaluate:

1. Transform the lists of typelist into a list of typelist *sizes*: [3,0,3,2],
2. Transform that to a list of index sequences [[0,1,2],[],[0,1,2],[0,1]] using `std::make_index_sequence`,
3. Transform the `std::index_sequence` into a typelist of `std::integral_constants` with `as_typelist`, and
4. Flatten that into the final list using `typelist_cat`.

In step (1) we get the typelist sizes, so we need another typelist algorithm called `typelist_size` that fetches the size of type typelist:

```
template<typename...List>
struct typelist_size<typelist<List...> >
  : std::integral_constant<std::size_t, sizeof...(List)>
{};
```

We're going to want to pass this to `meta_compose`, but `typelist_size` is a template, and `meta_compose` is expecting a Metafunction Class. We can write a wrapper:

```
struct typelist_size_wrapper
{
    template<typename List>
    using apply = typelist_size<List>;
};
```

Writing these wrappers is quickly going to get tedious. But we don't have to. Below is a simple utility for turning a boring old metafunction into a Metafunction Class:

```
template<template<class...> class F>
struct meta_quote
{
    template<typename...Ts>
    using apply = F<Ts...>;
};
```

The name `quote` comes from LISP via Boost.MPL. With `meta_quote` we can turn the `typelist_size` template into a Metafunction Class with `meta_quote<typelist_size>`. Now we can pass it to either `meta_compose` or `typelist_transform`.

Our steps call for composing three metafunctions. It will look something like this:

```
meta_compose<
    meta_quote<as_typelist_t>,            // Step 3
    meta_quote<std::make_index_sequence>, // Step 2
    meta_quote<typelist_size_t> >         // Step 1
```

As I already mentioned, `std::make_index_sequence` takes an integer not a type, so it can't be passed to `meta_quote`. This is a bummer. We can work around the problem with a variant of `meta_quote` that handles those kinds of templates. Let's call it `meta_quote_i`:

```
template<typename Int, template<Int...> class F>
struct meta_quote_i
{
    template<typename...Ts>
    using apply = F<Ts::value...>;
};
```

With `meta_quote_i`, we can compose the three functions with:

```
1 meta_compose<
2     meta_quote<as_typelist_t>,              // Step 3
3     meta_quote_i<std::size_t,
4                  std::make_index_sequence>, // Step 2
5     meta_quote<typelist_size_t> >           // Step 1
```

Now we can pass the composed function to `typelist_transform`:

```
1 typelist_transform_t<
2     typelist<as_typelist_t<Tuples>...>,
3     meta_compose<
4         meta_quote<as_typelist_t>,
5         meta_quote_i<std::size_t, make_index_sequence>,
6         meta_quote<typelist_size_t> > > >;
```

Voila! We have turned our lists of tuples into the list of lists: [[0,1,2],[],[0,1,2],[1,2]]. To get the final result, we smoosh this into one list using `typelist_cat`:

```
1 // E.g. [0,1,2,0,1,2,0,1]
2 typelist_cat_t<
3     typelist_transform_t<
4         typelist<as_typelist_t<Tuples>...>,
5         meta_compose<
6             meta_quote<as_typelist_t>,
7             meta_quote_i<std::size_t, make_index_sequence>,
8             meta_quote<typelist_size_t> > > >;
```

The result is the K indices that we pass to the `tuple_cat_` helper. And to repeat from above, the `I` indices are computed with:

```
1 // E.g. [0,0,0,2,2,2,3,3]
2 typelist_cat_t<
3     typelist_transform_t<
4         typelist<as_typelist_t<Tuples>...>,
5         typelist_transform_t<
6             as_typelist_t<make_index_sequence<N> >,
7             meta_quote<meta_always> >,
8         meta_quote<typelist_transform_t> > >;
```

I won't step through it, but I'll draw your attention to two things: on line (7) we make use of a strange type called `meta_always` (described below), and on line (8) we pass `typelist_transform` as the function argument to another call of `typelist_transform`. Talk about composability!

So what is `meta_always`? Simply, it's a Metafunction Class that always evaluates to the same type. Its implementation couldn't be siimpler:

```
1 template<typename T>
2 struct meta_always
3 {
4     template<typename...>
5     using apply = T;
6 };
```

I'll leave you guys to puzzle out why the above code works.

Summary

I set out trying to find a minimal useful set of primitives for manipulating lists of types that would be fit for standardization. I'm happy with the result. What I've found is that in addition to the `typelist` template, we need a small set of algorithms like the ones needed to implement `tuple_cat`:

- `typelist_apply`
- `typelist_size`
- `typelist_transform`
- `typelist_cat`
- `as_typelist`

Some other typelist algorithms come up in other metaprogramming tasks:

- `make_typelist` (from a count and type)
- `typelist_push_front`
- `typelist_push_back`
- `typelist_element` (indexing into a typelist)
- `typelist_find` and `typelist_find_if`
- `typelist_foldl` (aka, accumulate) and `typelist_foldr`
- etc.

In addition, for the sake of higher-order metafunctions like `typelist_transform` and `typelist_find_if`, it's helpful to have a notion of a Metafunction Class: an ordinary class type that can be used as a metafunction. A small set of utilities for creating and manipulating Metafunction Classes is essential for the typelist algorithms to be usable:

- `meta_apply`
- `meta_quote`
- `meta_quote_i`
- `meta_compose`
- `meta_always`

For other problems, the ability to partially apply (aka bind) Metafunction Classes comes in very handy:

- `meta_bind_front`
- `meta_bind_back`

And that's it, really. In my opinion, those utilities would meet the needs of 95% of all metaprograms. They are simple, orthogonal, and compose in powerful ways. Since we restricted ourselves to the `typelist` data structure, we ended up with a design that is *vastly* simpler than Boost.MPL. No iterators needed here, which makes sense since iterators are a pretty stateful, iterative abstraction, and metaprogramming is purely functional.

One Last Thing…

Below is one more metafunction to tickle your noodle. It's an N-way variant of `transform`: it takes a list of typelists and a Metafunction Class, and builds a new typelist by mapping over all of them. I'm not suggesting this is important or useful enough to be in the standard. I'm only showing it because it demonstrates how well these primitive operations compose to build richer functionality.

```
1  // ([[a,b,c],[x,y,z]], F) -> [F(a,x),F(b,y),F(c,z)]
2  template<typename ListOfLists, typename Fun>
3  struct typelist_transform_nary :
4    typelist_transform<
5      typelist_foldl_t<
6        ListOfLists,
7        make_typelist<
8          typelist_front_t<ListOfLists>::size(),
9          Fun>,
10       meta_bind_back<
11         meta_quote<typelist_transform_t>,
12         meta_quote<meta_bind_front> > >,
13     meta_quote<meta_apply> >
14 {};
```

Enjoy!

Update: This comment by tkamin helped me realize that the above `typelist_transform_nary` is really just the `zipWith` algorithm from the functional programming world. I've renamed it in my latest code, and provided a `typelist_zip` metafunction that dispatches to `typelist_zip_with` with `meta_quote<typelist>` as the function argument. Very nice!

42

This entry was posted in **library-design**, **std** by **Eric Niebler**. Bookmark the **permalink [http://ericniebler.com/2014/11/13/tiny-metaprogramming-library/]** .

56 thoughts on "TINY METAPROGRAMMING LIBRARY"

amohr

on **November 13, 2014 at 4:37 pm** said:

I'd love to see a comparison to a Hana implementation of tuple_cat. Mr. Dionne had this on his cppcon slides. It does two tuples — you cat more by recursion/fold. His approach of lifting types to values and metafunctions to functions is fascinating.

```
template
/* constexpr */ auto tuple_cat(Xs xs, Ys ys) {
 return xs.unpack_into([=](auto ...x) {
```

```
  return ys.unpack_into([=](auto ...y) {
   return make_tuple(x..., y...);
  });
 });
}
```


Eric Niebler

on **November 13, 2014 at 9:27 pm** said:

Above, I describe an inefficient way to implement `tuple_cat` by exploding the tuples into function arguments. This is exactly this inefficient implementation. I would be curious to see the Hana implementation of the algorithm I show above.

Louis Dionne

on **January 13, 2015 at 1:40 pm** said:

I agree that this is inefficient in its current form because it copies the parameters. But if we were able to use perfect forwarding (which is not possible because of the lambda captures), I don't see the problem in doing a lot of parameter passing. Shouldn't all the extra parameter passing required to implement `tuple_cat` be elided? Also, for the record, Hana now uses an hybrid implementation based on Thomas Petit's comment. Lambdas are not used anymore because of the limitations of constexpr and compiler bugs.

Eric Niebler

on **January 13, 2015 at 3:34 pm** said:

Not all types are efficiently movable. See `std::array`. Perfect forwarding isn't a perfect solution.

Nicola Gigante

on **November 13, 2014 at 11:33 pm** said:

This is *amazing*! It's both the description of a great idea for a proposal and the best medium-to-advanced introduction to metaprogramming that I've ever read!
Just one thing: meta_always should really be called meta_id or meta_identity. That's a pretty old convention… And maybe meta_transform should be called meta_map. I understand that "transform" is coherent with the STL, but that one is modifying its elements while this one is purely functional, so maybe it makes sense to call it by its right name.

Again, wonderful post!
Bye,
Nicola

Eric Niebler

on **November 14, 2014 at 9:05 am** said:

Thanks! And actually, no, `meta_always` is not like `id`. It is like Haskell's `const` function. `id` is basically:

```
id x = x
```

But `const` is:

```
const x = (\y -> x)
```

See the difference? And the word “const” in C++ already means something else, so I went with “always”.

And yeah, I *could* rename `meta_transform` to `meta_map`, but the term “map” in C++ refers to a data structure, not an algorithm. I’m not bothered by it. It just is.

Eric Niebler
on **November 14, 2014 at 12:41 pm** said:

By the way, I got a chuckle out of this:

> *best medium-to-advanced introduction to metaprogramming that I’ve ever read!*

Should an “introduction” be advanced? 😀

tkamin
on **November 14, 2014 at 6:30 am** said:

I that the second example that generates indicies [0,0,0,2,2,2,3,3] is using binary version of typelist_transform_t that is not presented in the article. Am I correct?

Eric Niebler
on **November 14, 2014 at 7:33 am** said:

You’re right!

tkamin
on **November 14, 2014 at 10:17 am** said:

The tuple_transform_t may be extended not only to binary type function, but the one with arbitrary number of arguments. The example implemenation of such functor may be found as: http://goo.gl/pQFHqF. The trick relies of tuple_zip function that transorm typelist<A1, A2> and typelist<B1, B2> into two level structure: typelist<typelist<A1, B1>, typelist<A2, B2>>.

I have used very similiar trick in my implementation of group placehoderls for std::bind: https://github.com/tomaszkam/proposals/blob/cpp14-implementation/bind/bind.hpp

Eric Niebler
on **November 14, 2014 at 10:35 am** said:

Zip! Yes, that’s a clever way to implement the n-way transform — which, by the way, is exactly what my `typelist_transform_nary` metafunction does. But it does it a different way.

As for your interface to `typelist_transform`, I considered that. But it moves the function to the front, which is different than `std::transform`. Also, I have found that variadic metafunctions are unwieldy to work with for the same reason that template template parameters are awkward. Parameter packs are not first-class citizens of C++.

Notice how `typelist_cat` takes a `typelist` of `typelists`. And look at how it's used to compute the `Is` and `Js` in my `tuple_cat`. Had it just taken a variadic list of `typelists`, I would have needed an extra step to unpack the `typelist` of `typelists` into the call to `typelist_cat`.

Basically, metafunctions take and return *types*. If you want your metaprograms to easily compose, they should also take and return types. That way, the computed result of one algorithm can be easily used as the input to another.

Does that make sense?

tkamin

on **November 14, 2014 at 11:05 am** said:

Yes, but sometimes you need to just cat a two or three typelist and writting: `typelist_cat_t<typelist<L1, L2>>` is not so convieniet. In your library you have two version of `tuple_transform`, while only only taking list of list would be necessary. I think we can achieve best of both world by defining additional "_ts" alias that will accept varidatic number of arguments, example:

```
template<typename ListOfLists>
struct typelist_cat;

template<typename ListOfLists>
using typelist_cat_t = typename typelist_cat<ListOfLists>::type;

template<typename... Lists>
using typelist_cat_ts = typename typelist_cat<typelist<Lists...>>::type;
```

Eric Niebler

on **November 14, 2014 at 12:00 pm** said:

By the way, you helped me realize that my `typelist_transform_nary` is really just `typelist_zip_with`. I'm renaming it, and providing a `typelist_zip` that dispatches to `typelist_zip_with` with `meta_quote<typelist>` as the function argument. Much better. You should take a look at my `zip_with` (aka `transform_nary`). It's shorter and more general than your `typelist_zip`.

Eric Niebler

on **November 14, 2014 at 12:07 pm** said:

Also, regarding your recommendation to provide *two* variants of `typelist_cat`, one that takes a typelist of typelists, and one that takes a variadic list of typelists — I certainly considered that. My feeling is that there's value in having fewer named algorithms. Instead, I'll note that a function that takes one typelist can be converted into a function that takes N separate arguments. In functional circles, this operation is called `curry`. In my library, `meta_curry` would look like this:

```
template<typename F>
struct meta_curry
{
    template<typename...Ts>
    using apply = meta_apply<F, typelist<Ts...>>;
};
```

Could be useful, I don't know.

tkamin

on **November 14, 2014 at 1:11 pm** said:

I do not think that there is difference in generality, between my implemenation that implements zip_with in terms of zip and your that implements zip in terms of zip_with. They provide same functionality.

Altought I must agree that your implemenation are much more shorter. I did not came up with idea of use of fold expression to zip tuples:

```
template<typename ListOfLists>
struct typelist_zip
{
  using type = typelist_foldl_t<
                ListOfLists,
                make_typelist<typelist_front_t<ListOfLists>::size(), ty
                meta_bind_back<meta_quote<typelist_transform_t>, meta_q
};
```

Did you consider idea to remove typelist_ prefix from algorithms names and putting them into same namespace instead? That will open a way to use namespace alias of using directive to avoid typing, at least for experiments.

Eric Niebler

on **November 14, 2014 at 1:20 pm** said:

> *Did you consider idea to remove typelist_ prefix from algorithms names and putting them into same namespace instead?*

I thought about it. The utilities in `std::` having to do with tuples are prefixed with `tuple_` (e.g. `tuple_element`, `tuple_size`), so I followed suit. But it gets tedious. Maybe these are all meta-facilities, and they belong in a `meta` namespace with all the utilities for manipulating Metafunction Classes. But then how to distinguish `meta_apply` from `typelist_apply`? `typelist_apply` is similar in flavor to `uncurry`. Maybe there's something deeper going on.

Brett Hall
on **November 14, 2014 at 10:19 am** said:

In your “Some other typelist algorithms come up in other metaprogramming tasks” list you’ve got the folds, I’m wondering if unfold has ever proven useful in this context (if you’re using catamorphisms maybe anamorphisms would come in handy too). I’m half tempted to try and implement it for kicks (if I could find the time).

Eric Niebler
on **November 14, 2014 at 10:25 am** said:

You’ve reached the limits of my Functional Programming-foo. What I know about catamorphisms and anamorphisms would fit in a thimble *>this big<*. If you show me how, and give me a convincing example of where it’s useful, I’d be very interested.

Brett Hall
on **November 14, 2014 at 2:04 pm** said:

The only case that I’ve been able to come up with would be generating a typelist of a given length of a given type, e.g. typelist<int, int, int>. Something like haskell’s “replicate” function, only for types. But that’s probably not enough motivation to implement unfold, replicate is easy enough to implement with what you’ve already got laid out. If I get some time I still might take a crack at implementing it for kicks (that’s a big “if” though).

Louis Dionne
on **January 12, 2015 at 3:46 pm** said:

For the record, unfolds are implemented in Hana here:https://github.com/ldionne/hana/blob/master/include/boost/hana/list.hpp#L364. The implementation is quite inefficient and it is unclear how it could be much better, but it might be possible. Quite frankly, I have not found any use for it so far (let’s say it is provided for completeness).

Roland Bock
on **November 15, 2014 at 3:24 am** said:

This is a really nice collection of utilities!

There is just one group of utilities missing that I use all over the place in sqlpp11: set operations, starting with

all<P, T…>: all T meet predicate P
any<P, T…>: at least one T meets predicate P
none<T, T…>: no T meets predicate P
unique<T…>: no two T’s are identical

I am using these to briefly check the arguments of variadic template functions.

Also, I am using type sets to do some more elaborate checks for SQL statements, e.g. “does the FROM clause contain all the tables used by all other clauses of the statement?” For these I use

type_set
union_set<S…>

intersect_set<S…>
difference_set<S1, S2>
is_element_of<T, S>
is_subset_of<S1, S2>
is_disjunct_from<S1, S2>

I am convinced that the first block is well within the 95%. Admittedly, I am not so sure about type sets.

Cheers,

Roland

Eric Niebler

on **November 15, 2014 at 8:51 am** said:

`all`, `any`, `none`: most certainly. Regarding `unique`, how do you keep it from being O(N^2) without the ability to sort on types? I would love to provide it (I've needed it too!), but I worry about people calling it on larger typelists and having their compile times go through the roof.

And the same question about the set algorithms.

Roland Bock

on **November 15, 2014 at 1:17 pm** said:

My version of unique is based on type_set and constructing the type_set is O(N^2).

Using the recursion-free any, the compile times aren't bad for quite a few arguments, but yes, they are bound to go through the roof eventually with large numbers of arguments.

Roland Bock

on **November 15, 2014 at 1:47 pm** said:

Just did a quick measurement:

args | seconds

10: 0.3
50: 0.4
100: 0.8
200: 2.4
300: 6.1
400: 12.0
500: 22

Eric Niebler

on **November 15, 2014 at 3:38 pm** said:

That's exponential. That's bad.

Roland Bock

on **November 15, 2014 at 4:20 pm** said:

Hmm. Maybe a problem with the experimental g++ I was using. clang is quadratic:

10 0.1
100 0.2
200 0.6
300 1.4
400 2.1
500 3.4
600 4.5
700 6.6
800 8.6
900 11
1000 13.5

Sorry for the noise…

Eric Niebler

on **November 15, 2014 at 4:39 pm** said:

Quadratic is still bad. I *think* I have an implementation of `typelist_unique` in mind that is linear, but I need to investigate.

Anders Sjögren

on **November 19, 2014 at 6:17 am** said:

Would it be possible to propose static_typeid(), with corresponding constexpr hash_code() before() and name() for a future C++, perhaps as part of the reflection work? That would then make sets and the like possible.

Eric Niebler

on **November 19, 2014 at 8:09 am** said:

I've spoken with Daveed Vandevorde, and he thinks compile-time type ordering wouldn't be hard to add. I suppose it could be even simpler: `typeid(T).before(typeid(U))` could be constexpr. That would enable a lot.

Bruno Dutra

on **August 30, 2015 at 5:22 pm** said:

Did you actually find a O(N) way to implement unique for both GCC and clang?

I've managed to make it quasi O(N) on GCC 5.2, but it still strongly O(N^2) on clang 3.8 (trunk). With some extra effort I also managed to get it working on MSVC 14, but I didn't actually try measuring how it performs there. One interesting thing I noticed is that memory consumption tends to be very low and constant in time both on GCC and clang, I could even get it

working for 100K arguments on my laptop with less than 4GB RAM compromised.

The trick was to indirectly inherit from all types and use SFINAE to select between two function overloads called for N pointers of the derived type. One of these overloads expects a pointer to each base and is only selected in case the conversion is unambiguous for all pointers.

You can find it here: https://github.com/brunocodutra/metal/blob/develop/include/metal/list/distinct.hpp

The numbers on GCC 5.2 are

1000: 0.3
2000: 0.6
3000: 1.1
4000: 1.7
5000: 2.5
6000: 3.4
7000: 4.5
8000: 6.2
9000: 7.6
10000: 10

Bruno Dutra
on **August 30, 2015 at 7:40 pm** said:

for the record
100k: 1356

Not so “quasi linear” once one goes past 10k elements, but still, it takes some effort for the user to see compilation times going through the roof.
Is is even mathematically possible to find an algorithm better than O(n^2) for unorderable sets?

Thomas Petit
on **November 16, 2014 at 3:55 pm** said:

I guess it’s because of my feeble imperative mind, but I found this functional style really hard to grok. It’s just too dense actually, If you don’t have a precise understanding of what does transform_t, meta_quote, meta_always, meta_quote_i, meta_compose, meta_meta etc. you have no chance of even guessing what’s going on.

Since C++14 I like to back down on the TMP stuff and use constexpr function more often. It’s a bit more verbose, but it’s look more like regular code.

Here is a version of tuple_cat where the jagged 2-dimensional array of indices is computed inside a constepxr function with plain old for loop.
It compiles an run with clang trunk and std=c++1z (it can be made to run with c++14 compiler)

http://pastebin.com/bzJFNZNv

Some drawback :
1) For some weird reason, std::array operator[] is not constexpr so for doing computation inside constexpr function we have to use plain C array then convert to std::array to return it.

2) We are forced to use a lot of little helper function just to get sets of indices into template parameter pack form, then immediately unpack them.

It would be really nice if we could create a std::index_sequence and unpack it immediately in the same function, something like :

```
std::tuple</some types/> stuff that_you_want_to_rearrange
constexpr std::array<int, N> indices_rearranged = compute_indices();
constepxr std::make_sequence_index<N> Idx; // 1..N indices
foo(std::get<indices_rearranged[Idx]...>(stuff that_you_want_to_rearrange));
```

Eric Niebler

on **November 16, 2014 at 9:19 pm** said:

To a degree, I agree with you. `constexpr` is a good thing, and the C++14 extensions are wonderful. And this particular example actually lends itself to `constexpr` handling because it's a computation of integers. Not all metaprogramming problems are like that, though. For computing types rather than integers, `constexpr` is less help.

Your larger point though seems to be that functional programming is hard and that it's impenetrable unless you know the meaning of `transform`, `always`, `apply`, `quote`, etc. Functional programming is a muscle that you develop over time. It brings its own rewards, like thread-safety. And as for needing the know the meaning of a few cryptic primitives, the same is true for *any* piece of reusable code. Until you know how to use a library, it's going to seem easier to code it by hand. But with the right abstractions, you can go much farther if you invest a little effort.

The code in this article took me a long time to write. I'm embarrassed to say that it took me about an hour to condense `typelist_transform_nary` (aka `typelist_zip_with`) down to what I show in the article. It's not surprising that it seems cryptic. But once you get it, it gives you a whole other way to approach problems — another tool for your toolbox.

Anders Sjögren

on **November 19, 2014 at 6:09 am** said:

Thanks for yet another excellent post!
Regarding naming: I guess tuple_cat is one of few c++ functions where the type is part of the name. Generally, that is avoided to enable overloading/templatization/specialization, right? I mean std::transform is not called std::vector_transform. I guess that should be the case here as well?
Currently, typelist might seem like the one and only data structure for meta-programming, but I wouldn't be surprised if others pop up, and then it'd be neat to be able to use the same names for meta-function-classes for those as well (by use of specialization?).

meta_transform or meta::transform might thus be better choices than typelist_transform? Or maybe std::Transform, where upper_case might signify meta-functions. After all, they are not functions, but types.

Eric Niebler

on **November 19, 2014 at 8:06 am** said:

Good point. I should probably rename these.

Robert Klarer

on **November 24, 2014 at 8:49 am** said:

This is a very useful approach, and a good candidate for standardization, IMO. I have been using a very similar approach in my work. Even the names I used were similar to yours. One major difference, though, is that I didn't bother with a typelist template. I think std::tuple is sufficient. The main distinction between the two is that std::tuple occupies space at execution time, but since typelist is not meant to be an execution time entity,

that is really a distinction without a difference. Furthermore, I think the typelists will often be used to generate tuple types in actual practice anyway, so I view typelists as an unnecessary middleman (at least in my use cases).

The tuple operations that I have found most useful are a tuple for_each, a tuple transform that is pretty much the same as your typelist_transform, and a “find by type” metafunction.

If all C++ classes in a system are written as tuples, you can do some pretty amazing things with just this small toolset.

> 
>
> **Eric Niebler**
> on **November 24, 2014 at 9:13 am** said:
>
> I * almost* agree with you. A few things, though. The definition of `typelist` is one line long. To use `tuple`, you need to `#include <tuple>`, which is prtty big.
>
> But the real killer is that `tuple` can’t be instantiated on incomplete types, `void`, function types, array types, and abstract bases. So, `typelist<void, int()>` is fine, but with `tuple<void, int()>` you’re playing with fire.
>
> Finally, I’ll note that it is *occasionally* useful to actually instantiate a `typelist` at runtime and pass it to a function. That lets you pass several different parameter packs into a function, having the packs deduced, while keeping them separate. If you try that with a `tuple<void, void, void>`, you’ll be unhappy with the result.
>
> > 
> >
> > Robert Klarer
> > on **November 24, 2014 at 2:16 pm** said:
> >
> > Good points. I hadn’t considered incomplete types and the rest (wasn’t a requirement in my use cases). Thanks!

Charlie Dyson
on **November 25, 2014 at 12:15 am** said:

Very much enjoyed reading this. I’ve had a go at manipulating containers of types myself, taking a slightly different approach in that I consider pretty much any template to be a container of types.

E.g.
using T1 = std::tuple<int, double>;
using T2 = std::tuple;
using J = Join<T1, T2>; // = std::tuple<int, double, char>

Code here: https://github.com/cdyson37/rebind

Would be good to have some kind of boost::mpl-like-thing in the standard!

> 
>
> Charlie Dyson
> on **November 25, 2014 at 12:15 am** said:
>
> Not sure how I’ve managed to misformat the above, but I’m sure you can guess what T2 was supposed to be!

Achuthan Krishna

on **December 9, 2014 at 10:40 pm** said:

Eric,

Excellent post as always!! Enjoy reading every one of your posts.

Any specific reason why you used typelist instead of std::tuple? I was able to implement tuple_cat using std::tuple.

I dont know how to measure compile time complexity or number of instantiations etc.

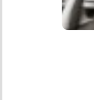
Anyways, here you go:

```
template&amp;amp;lt;int Row, int Col&amp;amp;gt; struct ij_pair {
    static const int I = Row;
    static const int J = Col;
};</p>

<p>template&amp;amp;lt;typename Result, int Row, int Col,
         typename Tuple&amp;amp;gt;
struct col_indices_impl;</p>

<p>template&amp;amp;lt;typename Result, int Row,
         typename ...Tuples&amp;amp;gt;
struct rowcol_indices_impl;</p>

<p>template&amp;amp;lt;typename ...Result, int Row, int Col,
         typename T, typename ...Rest&amp;amp;gt;
struct col_indices_impl&amp;amp;lt;std::tuple&amp;amp;lt;Result...&amp;amp;gt;,
                        Row, Col,
                        std::tuple&amp;amp;lt;T, Rest...&amp;amp;gt;&amp;amp;gt; {
    using type = typename
      col_indices_impl&amp;amp;lt;std::tuple&amp;amp;lt;Result...,
                  ij_pair&amp;amp;lt;Row, Col&amp;amp;gt;&amp;amp;gt;, Row, Col+1,
                  std::tuple&amp;amp;lt;Rest...&amp;amp;gt;&amp;amp;gt;::type;
};</p>

<p>template&amp;amp;lt;typename ...Result, int Row, int Col&amp;amp;gt;
struct col_indices_impl&amp;amp;lt;std::tuple&amp;amp;lt;Result...&amp;amp;gt;,
                        Row, Col, std::tuple&amp;amp;lt;&amp;amp;gt;&amp;amp;gt; {
    using type = std::tuple&amp;amp;lt;Result...&amp;amp;gt;;
};</p>

<p>template&amp;amp;lt;typename ...IJ, int Row, typename T,
         typename ...Rest&amp;amp;gt;
struct rowcol_indices_impl&amp;amp;lt;std::tuple&amp;amp;lt;IJ...&amp;amp;gt;, Row,
                           T, Rest...&amp;amp;gt; {
    using type = typename
    rowcol_indices_impl&amp;amp;lt;typename
      col_indices_impl&amp;amp;lt;std::tuple&amp;amp;lt;IJ...&amp;amp;gt;, Row, 0,
                       T&amp;amp;gt;::type, Row+1, Rest...
    &amp;amp;gt;::type;
};</p>

<p>template&amp;amp;lt;typename ...Result, int Row&amp;amp;gt;
struct rowcol_indices_impl&amp;amp;lt;std::tuple&amp;amp;lt;Result...&amp;amp;gt;,
                           Row&amp;amp;gt; {
    using type = std::tuple&amp;amp;lt;Result...&amp;amp;gt;;
};</p>

<p>template&amp;amp;lt;typename ...Tuples&amp;amp;gt;
using rowcol_indices = typename
  rowcol_indices_impl&amp;amp;lt;std::tuple&amp;amp;lt;&amp;amp;gt;, 0, Tuples...&amp;amp;gt;::type;</

<p>template&amp;amp;lt;typename Tuples, typename ...Indices&amp;amp;gt;
auto tuple_cat_impl(Tuples&amp;amp;amp;&amp;amp;amp; ts,
                    std::tuple&amp;amp;lt;Indices...&amp;amp;gt;) {
    return std::make_tuple(std::get&amp;amp;lt;Indices::J&amp;amp;gt;
           (std::get&amp;amp;lt;Indices::I&amp;amp;gt;
             (std::forward&amp;amp;lt;Tuples&amp;amp;gt;(ts)))...);
}</p>

<p>template&amp;amp;lt;typename... Tuples&amp;amp;gt;
auto tuplecat(Tuples&amp;amp;amp;&amp;amp;amp;... ts) {
    using Indices =
    rowcol_indices&amp;amp;lt;std::decay_t&amp;amp;lt;Tuples&amp;amp;gt;...&amp;amp;gt;;
    return tuple_cat_impl(std::forward_as_tuple
         (std::forward&amp;amp;lt;Tuples&amp;amp;gt;(ts)...), Indices());
}
```

Eric Niebler

on **December 10, 2014 at 9:54 am** said:

> `std::tuple` doesn't make a very good typelist when the types in the list are things like `void`, function types, array types, and abstract types. That's not the case for this algorithm, but it most certainly will be for others.



Scott Santucci
on **January 9, 2015 at 10:05 am** said:

You can add me to the list of people hoping you propose(d) to the relevant people or institutions that "packer" be renamed "typelist". When I hear the name "packer" I think it *makes* a parameter pack; but if I'm following the discussion, that's misleading since what it actually does it *take* a parameter pack and turn it into a metaprogramming container of types. (At least, I think it's just types — parameter packs and by extension the "packer"/"typelist" can't include constants/constant expressions, can they? If they could, then we'd need a name like "type_or_constant_list"…) Of course, "typelist" sounds like it means "a metaprogramming container of types", so it's a much more descriptive name for just that.

Tangentially, is there a primer anywhere for functional programming paradigms and their relationship to metaprogramming/type programming, or even a good exaplanation of the differences between a template, a metafunction and a metafunction class? I find this fascinating, but it's pretty difficult to follow (or think of applications for) given a background in stateful procedural and object oriented programming, even with a little knowledge of metaprogramming (mostly SFINAE sorts of generic things). It'd be nice if I could just have laid out for me what each of the techniques for building things out of templates is in terms of "think of this like a function, here are the inputs, here is the 'output' in effect…" and have multiple examples of each to back it up in and of itself (more "here are more examples of how this part works" and less "here is how these parts can fit together", although the latter obviously is important sooner or later). I like what you said in a comment-reply about functional programming being a muscle (and, I'd add, "functional C++ metaprogramming" is probably a related but not quite the same one, given that how you would implement some basic things or sorts of things in it is different from how you'd implement the same with, say, function pointers); I just can't help wondering where the gym is, I guess. ;^)

Eric Niebler
on **January 10, 2015 at 10:52 am** said:

> *is there a primer anywhere for functional programming paradigms and their relationship to metaprogramming/type programming*

For the intersection between functional programming and C++ TMP, you could do far worse than Bartosz Milewski's blog, this post in particular.

> *or even a good exaplanation of the differences between a template, a metafunction and a metafunction class?*

For the full picture, I refer you to the TMP book by Abrahams and Gurtovoy. But to answer your question in a nutshell:

Template
A way of stamping out copies of code, parameterized on a type. std::vector is a class template. std::for_each is a function template.

Metafunction
A class template with a nested ::type typedef. It's used like the compile-time equivalent of a runtime function. The template parameters are like the function arguments, and the nested ::type typedef is like the return value.

Metafunction class
An ordinary class type (not a template) that has a nested metafunction (see above) called apply. It fills the same role as a metafunction except that, being a class, it's a first-order metaprogramming entity; that is, it can be used as a parameter to another metafunction [class], whereas an ordinary metafunction cannot, since it's a template and not a type.

Hope that helps.

Scott Santucci
on **January 23, 2015 at 9:39 pm** said:

[For the record, I read the reply and the post by Milewski a couple weeks ago and then was too busy to compose another comment… You know how some people say that reading code is harder than writing code? Well for me blogs are the other way around!]

Ah, I think I get it. A metafunction has to take its template parameters to be resolved from a class template to an actual class, so it can only be passed to other metafunctions as a template template parameter for them to provide the parameters to resolve the class. A metafunction class wraps a metafunction as a member template using-alias, so the resolution of the metafunction by providing template parameters isn't required in order to make the metafunction class itself an actual class in the first place (even if, as in the case of meta_quote, the metafunction class is also templated, it's given parameters and resolved before being given as a parameter to another metafunction), so it can be passed as a regular type parameter, inherited from (if there are any cases where that's worth doing), and… Are there any other things you can do with metafunction classes that you can't do with metafunctions? I actually kinda like that "raw" metafunctions have to be passed as template template parameters since that way it's more obvious from just the "interface" that that parameter is supposed to be a metafunction, but maybe there are bigger advantages to a uniform interface?

I'll definitely check out the book (literally if the local library has it, otherwise I guess it would probably be worth some of my leftover Christmas money).

That post from Milewski's blog is an excellent starter. Not only do I feel like I "get" the basic mindset of functional programming since reading it, but I am pretty sure I can read Haskell statements — I've wondered for years what those unusual little syntaxes meant, now I know about list comprehension. I'm still not sure it's any easier to read than C++ templates (if only because it's so compact and has so few visual cues — "a b c = b c" doesn't strike me as more obvious than "a(b, c) { b(c) }") but at least I *can* read it — and I can think in terms of what it's meant to achieve.

Here's a couple quick questions — would you say that in functional languages recursion is the equivalent of conditional (while/do-while) loops, and/or that list comprehension is the equivalent of iterative (for-each-in) loops?

Another thing I'm wondering is whether an if-statement-like metafunction would be useful — something along the lines of…

```
template &lt;bool Condition, typename IfTrue, typename IfFalse&gt;
struct meta_if&lt;Condition, IfTrue, IfFalse&gt;;

template &lt;typename IfTrue, typename IfFalse&gt;
struct meta_if&lt;true, IfTrue, IfFalse&gt; : public IfTrue {};

template &lt;typename IfTrue, typename IfFalse&gt;
struct meta_if&lt;false, IfTrue, IfFalse&gt; : public IfFalse {};
```

I'm not sure off the top of my head if that would actually compile — but more importantly, I'm not sure if there are good use cases for it or for something along the lines of it.

And if you don't mind, one last question… C++14 has, if I understand correctly, allowed compile-time computation of *values* to break somewhat out of the functional mold by relaxing a lot of the constraints on constexpr functions (obviously no side-effects allowed, so still rather functional-like in that respect, but the stuff that can be done purely within the constexpr function now looks rather procedural); do you think there would be anything to gain by having a more "traditional" (er… imperative?) syntax for type functions? I tried to come up with some examples of what it could look like only to realize that everything I could think of was some combination of A) not any more stateful than functional programming anyway, B) atrociously contrived and/or C) probably more straightforwardly expressed by something like list comprehension and/or a list building/filtering syntax/construct (in functional programming I suppose there'd be a single function that takes a list and a predicate to filter on and returns a list of the entries in the parameter list that met the predicate?):

```
// Effectively stateless and substantially the same as a functional implementation
type pointerize_if_nonpointer(type original) {
    if (is_pointer(original)) {
        return original;
    } else {
        return original*;
    }
}

// Why would anyone do this...
type last_matching_in_list(typelist options, bool(type) predicate, type default = null
    type result = default;
    for (type option : options) {
        if (predicate(option)) {
            result = option;
        }
    }
    return result; // Presumably returning nulltype would be a compiler error (unless
}

// Clunky substitute for list comprehension or some such functional operation/construc
typelist filter(typelist originals, bool(type) predicate) {
    typelist result;
```

```
    for (type candidate : originals) {
        if (predicate(candidate)) {
            result.push_back(candidate);
        }
    }
    return result;
}
```

If I haven't said it already, thanks for all the great info!

Scott Santucci

on **January 28, 2015 at 9:06 pm** said:

Ok, I think I am getting the hang of this — recursion with head:tail is more like for each in range/container loops, list comprehension is more sophisticated than raw loops… And my notion of "meta_if" is something like std::conditional except std::conditional uses a member type alias instead of inheritance (which, I suppose, takes a little more to write when using, but is more consistent with other metafunctions and there's probably an advantage to not having it technically be a derived type of whatever). Well, I'll be around trying to learn more in any case.

Aditya

on **January 18, 2015 at 2:24 am** said:

The ideas in this blog post along with the `meta.hpp` header helped me drastically cut down the amount of time spent and TMP code written for some of my projects. Thanks for sharing your ideas!

I implemented `std::make_integer_sequence` without using template recursion, but the implementation is clumsy (I have no experience with FP). Some of the other TMP code I've written uses similar patterns, so I was wondering if you knew a better way to go about it. The `combine` function takes *n* unary functions *f_1, …, f_n* and uses them to create an nary function *f : (T_1, … T_n) -> list<f_1(T_1), …, f_n(T_n)>.* I also made `compose` left-associative instead of right-associative, because I find that the resulting code is more legible. Here's the implementation:

```
template &amp;lt;std::uintmax_t N, class Init&amp;gt;
using iota_c = foldl&amp;lt;repeat_nc&amp;lt;N - 1&amp;gt;, list&amp;lt;Init&amp;gt;,
compose&amp;lt;
    quote&amp;lt;list&amp;gt;,
    quote&amp;lt;erase_back&amp;gt;,
    quote&amp;lt;front&amp;gt;,
    bind_front&amp;lt;quote&amp;lt;repeat_n&amp;gt;, int_&amp;lt;2&amp;gt;&amp;gt;,
    uncurry&amp;lt;combine&amp;lt;
        compose&amp;lt;
            quote&amp;lt;back&amp;gt;,
            bind_front&amp;lt;quote&amp;lt;plus&amp;gt;, int_&amp;lt;1&amp;gt;&amp;gt;
        &amp;gt;,
        quote_trait&amp;lt;id&amp;gt;
    &amp;gt;&amp;gt;,
    uncurry&amp;lt;quote&amp;lt;append&amp;gt;&amp;gt;
&amp;gt;&amp;gt;;
```

The reason I think this is clumsy is because I have to pack the arguments into a list and perform some awkward manipulations in order to avoid creating helper structs. Is there a cleaner way to do this? Thanks for your help!

Aditya

on **January 18, 2015 at 4:11 am** said:

I'd like to make a small update. Firstly, the `erase_back` is redundant; I accidentally left that in. Secondly, I wrote a metafunction class called `broadcast_c<std::uintmax_t... Indices>` which creates a new list from an existing one by calling collecting the elements corresponding to `Indices`. The implementation is now slightly less clumsy:

```
template &amp;lt;std::uintmax_t N, class Init&amp;gt;
using iota_c = foldl&amp;lt;repeat_nc&amp;lt;N - 1&amp;gt;, list&amp;lt;Init&amp;gt;,
compose&amp;lt;
    quote&amp;lt;list&amp;gt;,
```

```
    broadcast_c&amp;lt;0, 0&amp;gt;,
    uncurry&amp;lt;combine&amp;lt;
        compose&amp;lt;
            quote&amp;lt;back&amp;gt;,
            bind_front&amp;lt;quote&amp;lt;plus&amp;gt;, int_&amp;lt;1&amp;gt;&amp;gt;
        &amp;gt;,
        quote_trait&amp;lt;id&amp;gt;
    &amp;gt;&amp;gt;,
    uncurry&amp;lt;quote&amp;lt;append&amp;gt;&amp;gt;
&amp;gt;&amp;gt;;
```


Thort

on **February 5, 2015 at 2:53 pm** said:

Hi Eric. Awesome post! I also played around with creating a modernized meta library and came to some similar conclusions. Just curious, why constrain the helpers to accept only typelist, when its possible to accept any class template? I found the typelist concept to be useful as a default, but also found it useful to allow my metafunctions, such as transform, to work with any class template. Possible use case: transforming the element types of a given tuple. If transform accepts any class template, the conversion to/from typelist can be skipped.

Eric Niebler

on **February 5, 2015 at 3:57 pm** said:

It’s certainly possible to generalize these metafunctions the way you describe. The implementation gets more complex, though. I was primarily interested in making a very small, very powerful metaprogramming library.

In my current lib, the metafunctions `meta::curry` and `meta::uncurry` (used to implement `meta::as_list`) handle arbitrary templates like you want. With them, you can turn any template instance into a typelist and back without needing to specialize anything. Not as concise, but it makes the library vastly simpler.

Thort

on **February 6, 2015 at 1:19 pm** said:

Definitely agree with that, these meta helpers can get complicated very quickly, and there are some special rules you have to remember or additional questions to ask if they accept arbitrary templates (should concat be able to concat different templates? Would the result be an instantiation of the first template argument?.. etc). Thanks!

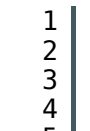
Chris Kohlhepp

on **March 30, 2015 at 6:09 pm** said:

Type lists, lists, meta, in compiler execution, quoting cat (cons), functional programming … you are converging on Lisp, slowly. The connections was made in the article already, but it’s really much more comprehensive and all encompasing than what is suggested here. The real gripe I have with C++ going down the meta-road is that the execution of programs becomes intracktable by design. The C++ compiler, unlike Lisp, gives you no access to the process or output of meta constructs. This is fine as long as they are trivial. As they become complex & layered, the architecture and design of an C++ MPL application becomes ever more intracktable. This is different for Lisp where you have direct access to each layer of the onion as the compiler works through meta constructs. We’re in IT. Process and output of a function, any function should be tractable. Perhaps the answer is not in stretching C++, but in a combination of Lisp and C++. With the LLVM this becomes very much possible. https://github.com/drmeister/clasp

Eric Niebler

> on **March 30, 2015 at 7:51 pm** said:
>
> I couldn't agree more. Metaprogramming in C++ is unpleasant because of the lack of tool support. There is a clang-based utility that allows for debugging metaprogramming (see this talk), but I haven't used it so I can't speak for it.
>
> > *Perhaps the answer is not in stretching C++, but in a combination of Lisp and C++. With the LLVM this becomes very much possible.* *https://github.com/drmeister/clasp*
>
> Looks like an interesting approach.



pongba

on **August 19, 2015 at 4:30 pm** said:

Nice article. Would've been really nice if future standard could support the heterogeneous pack e.g.

```
template <auto... args>
```

Scott Santucci

on **October 1, 2015 at 8:09 pm** said:

Hey, Eric,

Off the top of your head, do you know if there are ways to specialize on parts of typelists with the same sort of "best-match" (or most-specific match) requirements that you'd get in regular template specialization? For instance, I'm under the impression that you can say this with regular template specialization:

```
template <typename... Types>
struct some_result {
// general form here
};
template <typename... Types>
struct some_result<char, Types...> {
// type starting with char here
};
template <typename... Types>
struct some_result<char, int, Types...> {
// type starting with char then int here
};
```

…And, of course, the general case will be used for things that don't start in char, the second for things that start in char but not char then int, and the last only for things that start in char then int; all valid, nifty pattern matching.

As far as I know, you can't do that if you convert Types to a typelist — you can specialize on an exact match for the whole list, or SFINAE on properties like "starts with char" or "starts with char then int", but my attempts to SFINAE on "starts with char" and "starts with char then int" get complaints of ambiguity because both SFINAE'd-in specializations are equally valid and therefore ambiguous. (Let me know if posting code for this would be helpful — I'm trying to keep this question/example from getting too out of hand.) Yet on the other hand, you can't use more than one typename… parameter pack in the same template, which limits some interesting use cases (for instance, a tree of types: you'd want one pack of ancestry and another of preceding siblings [assuming there weren't some reasonable way to handle each ancestor's preceding siblings too — a multi-dimensional parameter pack? multi-dimensional typelists or typelist trees would be trivial, of course]); you can, on the other hand, use multiple typelists in as parameters to the same template, I'm pretty sure. Furthermore, I'm not sure whether you could partially specialize on the pack *ending* in specific types instead of beginning with them (at least, Clang and GCC don't accept it, GCC saying the pack must be at the end of the specialization list and Clang saying something [Types, I think] can't be deduced so the partial specialization won't be used) — obviously, any SFINAE trait on the typelist could check for types at the beginning, the end or anywhere in between. So is there some way to get the best of both worlds — a most-specific/best match behavior on arbitrary criteria about a typelist? Because I have some pretty awesome potential use cases for something like that…

(Well, that came out… a bit less jointed than I'd hoped. Disjointed is the opposite of jointed, right? Whatever. If I keep trying to make explanations like this perfect I'll never get around to posting it, sort of like most of the code I write outside of my day job…)

(P.S. Just so you're aware, the preview button doesn't seem to be working.)

Mortine

on **December 15, 2015 at 9:04 am** said:

if everything was a placeholder would be more simple, or storing extended typelist containers instead of the other way around + back and forth ; somehow a smart placeholder.