

Unfinished Business

Abstract

All good things must come to an end and so, coincidentally, must this column. In this last installment of Common Knowledge we'll look at combining some template metaprogramming odds and ends that have been kicking around in my "to think about sometime" folder to see if we can come up with a reasonable framework for ad hoc type manipulation. We'll fail, but it'll be fun to make the attempt. Along the way we'll consider the creative use of rocks in automotive maintenance, type translation to facilitate compile-time manipulation, un-instantiation of class templates, giving the STL allocator "rebind" mechanism the boot, typelist meta-generic algorithms, meta-predicate adaptation, and the importance of having fun.

Part 1: Type Dismantling and Regeneration

In past installments of Common Knowledge, we've developed techniques that perform compile time translation of a type into a large integer,¹ and a way of representing integers as types so as to perform compile time arithmetic on very large integers.² Now we'll look at a more general compile time type-to-type translation.

Hammers and Rocks

In template metaprogramming, we often have to make "adjustments" to a type before we can use it. For example, we may have to adjust the type of container element to produce the correct type for use as a temporary:

```
template <class Cont>
void doItToCont( Cont &c ) {
    typename DeConst<typename Cont::element_type>::R temp;
    //...
}
```

Here, we've recognized that `Cont` (which we're assuming to be some sort of container) may have constant elements. In that event, a temporary would have to be the non-constant analog of `element_type`. To ensure this, we perform a small bit of type manipulation (with the `DeConst` template) to remove a top-level `const` qualifier if it's present.³ I find this sort of operation analogous to working on an automobile engine.⁴ Use of the `DeConst` type manipulation operation is similar to smacking an engine part with a hammer or, in a pinch, a rock (I speak from experience here) in order to make it fit the purpose for which you intend it, no matter what its manufacturer intended.

Effective though that approach can be, it's a bit heavy-handed. Additionally, every different "mechanic" will find different ways to accomplish similar ends. Some prefer hammers, some prefer rocks. (I knew one mechanic who could make parts fit by sheer force of language.) Sometimes it's best to go back to the shop manual, dismantle the engine, change a few parts, and put it back together correctly with a minimum of physical violence or verbal persuasion.

Common Knowledge: Unfinished Business

Type Dismantling

The first step in this process is the dismantling of a complex structure (whether it is a type or an engine) into its constituent parts. However, the dismantling must be orderly so we don't lose parts or switch them accidentally. Since we're working with types, we'll employ the `typelist` compile time list structure popularized by Andrei Alexandrescu.⁵

```
template <typename T, class U>
struct typelist {
    typedef T Head;
    typedef U Tail;
};

typedef struct null_typelist{};
```

A `typelist` is just a sequence of types terminated with a `null_typelist`. The structure is recursively defined, so that a `typelist` is either a `null_typelist`, or a pair consisting of a type and a `typelist`.

To dismantle a type, we examine how it's constructed and string its constituent parts on a `typelist`. The `catchall` simply creates a one-element `typelist`:

```
template <typename T>
struct Dis {
    typedef typelist<T, null_typelist> R;
};
```

This primary `Dis` template is used to record the base type of a (potentially) complex type. We'll see later how we can customize this template to dismantle complex base types, or special case for base types with particular properties.

In order to record information about type modifiers in a `typelist`, we create types that represent modifiers.

```
struct Const {};
struct Volatile {};
struct Ref {};
struct Ptr {};
```

Once we have these representations available, we can partially-specialize `Dis` to recognize and dismantle modified types.

```
template <typename T>
struct Dis<const T> {
    typedef typelist<Const, typename Dis<T>::R> R;
};

template <typename T>
struct Dis<T *> {
    typedef typelist<Ptr, typename Dis<T>::R> R;
};

template <typename T>
struct Dis<T &> {
```

Common Knowledge: Unfinished Business

```
typedef typelist<Ref, typename Dis<T>::R> R;  
};
```

For example, `Dis<int * const>::R` will produce the three-element typelist `[Const, Ptr, int]`. Other modifiers are handled similarly, but are marginally more complex. Array and pointer-to-data-member modifiers must encode their bound and class type information, respectively:

```
template <int>  
struct Ary  
{  
};  
  
template <class>  
struct Pcm  
{  
};
```

Dismantling these types is similar to dismantling a pointer type, above.

```
template <typename T, int b>  
struct Dis<T[b]> {  
    typedef typelist< Ary<b>,typename Dis<T>::R > R;  
};  
  
template <class C, typename T>  
struct Dis<T C::*> {  
    typedef typelist<Pcm<C>,typename Dis<T>::R> R;  
};
```

Therefore, the code `Dis<const int X::* const>::R` results in the type list `[Const, Pcm<X>, Const, int]`. Although we do not use this information currently, it may also be a good idea to provide the user of our type-dismantling framework more information about the `Pcm` modifier:

```
template <class C>  
struct Pcm  
{ typedef typename Dis<C>::R Class; };
```

Note that, with this additional information recorded in the `Pcm` template, we now no longer have a simple linear type list, but a list of lists. The function and pointer-to-member-function modifiers are more complex, because they can take arguments:⁶

```
struct Fun0 {};  
  
template <typename A>  
struct Fun1  
{ typedef typename Dis<A>::R Arg; };  
  
template <typename Ret>  
struct Dis<Ret(void)> {  
    typedef typelist<Fun0,typename Dis<Ret>::R> R;
```


Common Knowledge: Unfinished Business

Now we have the ability to go back and forth between the normal and dismantled versions of a type.

```
template <typename T>
void aTemplateContext() {
    typedef typename Dis<T>::R ExplodedT;
    //...
    typedef typename Regen<ExplodedT>::R TheSameT;
    //...
```

So What's The Point?

In effect, we now have two equivalent, but structurally distinct, versions of the same type. The normal version is optimized for use in a traditional fashion: accessing its operations, causing code to be generated that will execute at runtime, etc. The dismantled version is optimized for compile time analysis and manipulation. Therefore, we now have the ability to move an arbitrary type between representations according to how we want to use it. The invertibility of the representations assures us that either representation will contain all the information present in the other.

Part 2: Extending Capability with Template Un-Instantiation

As we mentioned above, the primary `Dis` template is a catchall for any type not more specifically described by `Dis`'s partial specializations. The existing partial specializations cover all possible type modifiers,⁷ so the catchall dismantles only base types. However, we can be more selective. For instance, we could decide to handle a particular base type or set of base types differently:

```
struct Int
{
};

template <>
struct Dis<int> {
    typedef typename typelist<Int,null_typelist> R;
};

template <>
struct Regen< typename typelist<Int,null_typelist> > > {
    typedef int R;
};
```

These three new elements extend the framework to special case on `int`. Useless.

However, there are more useful extensions. For example, it's possible to dismantle a type generated from a class template into the original template and the arguments used in its instantiation. We then have the ability to re-instantiate the template with different arguments, or (as we'll see later in this article) produce an adapted template that can be instantiated with different numbers or types of arguments.⁸

```
template <template <typename> class Templ, typename T>
struct TemplT {
    template <typename S>
```

Common Knowledge: Unfinished Business

```
struct Subst // rebind with a different type name
    { typedef Templ<S> R; };
typedef T Arg;
typedef typename Dis<T>::R R;
};

template <template <typename> class Templ, typename T>
struct Dis < Templ<T> > {
    typedef typename list<TemplT<Templ,T>, null_typelist> R;
};

template <template <typename> class Templ, typename T>
struct Regen< typename list<TemplT<Templ,T>,null_typelist> > {
    typedef Templ<T> R;
};
```

This extension to the dismantling framework allows us to examine the fine structure of a base type generated from a class template with a single type parameter. In a similar fashion, we can extend the framework to handle types generated from templates with other parameter types. The following extension handles class templates with two type parameters:

```
template <template <typename,typename> class Templ,
    typename T1, typename T2>
struct TemplTT {
    template <typename S1, typename S2>
    struct Subst
        { typedef Templ<S1,S2> R; };
    typedef T1 Arg1;
    typedef T2 Arg2;
    typedef typename Dis<T1>::R R1;
    typedef typename Dis<T2>::R R2;
};

template <template <typename,typename> class Templ,
    typename T1, typename T2>
struct Dis < Templ<T1,T2> > {
    typedef typename list<TemplTT<Templ,T1,T2>, null_typelist> R;
};

template <template <typename,typename> class Templ,
    typename T1, typename T2>
struct Regen< typename list<TemplTT<Templ,T1,T2>,null_typelist> > {
    typedef Templ<T1,T2> R;
};
```

Common Knowledge: Unfinished Business

Similar extensions may be used for templates that take integral or pointer arguments, as well as type arguments.

Part 3: Typelist Meta-Algorithms

Typelists are a common, general mechanism for representing an ordered sequence of types at compile time. The analogy with an STL sequence defined by a pair of iterators is nearly irresistible, so I didn't resist. In this section, we develop a suite of typelist meta-algorithms in (somewhat broad) imitation of STL generic algorithms on sequences.

Type Predicates and Comparators

Many interesting generic algorithms on sequences require the ability to compare two elements of the sequence, or ask a yes/no question of a sequence element.

The STL generally uses function objects (though it may also use function pointers) to provide these capabilities. Typically, a predicate or comparator is implemented as a class template that is instantiated and used to generate a function object, which is then passed as an argument to a generic function.

We have to use a somewhat different approach to provide a compile-time “function object.” Instead of instantiating our generic meta-algorithms with the type of a function object, we'll instantiate them with the template of a meta-function object. Where a runtime generic algorithm uses an object generated from a class type as a function object, our compile time meta-algorithms will use a class type generated from a class template as a “meta-function object.”

For example, a simple meta-predicate that determines whether or not a particular type is `double` could be implemented as follows:

```
template <class T>
struct IsDouble { enum { r = false }; };
template <>
struct IsDouble<double> { enum { r = true }; };
```

By convention (OK, *my* convention), the result of applying a predicate or comparator is a nested compile time value named `r` that is convertible to `bool`. While an STL predicate provides an answer to a question about an object, a meta-predicate provides an answer to a question about a type.

As in the STL, we may also consider comparators; a subset of binary predicates that implement a strict weak ordering. As with meta-predicates, our meta-comparators implement an ordering on types, rather than objects:

```
template <typename A, typename B>
struct IsSmaller {
    enum { r = sizeof(A) < sizeof(B) };
};

template <class A, class B>
struct IsDerivedFrom {
    enum { r = SUPERSUBCLASS_STRICT(A,B) }; // from Modern C++ Design
```

Common Knowledge: Unfinished Business

```
};
```

Using these meta-comparators, we can compare pairs of types based on the relative sizes of their objects, or on their inheritance relationship. Note that the `IsDerivedFrom` comparator does not implement a total ordering on the set of types, since it's possible (or likely) that two types have no inheritance relationship.

Generic Typelist Algorithms

Meta-predicates and comparators are most useful when used to parameterize meta-generic algorithms. For instance, we can use a meta-predicate to implement a partition algorithm on typelists:

```
template <class TList, template <typename> class Pred>
struct Partition;

template <template <typename> class Pred>
struct Partition<null_typelist,Pred> {
    typedef null_typelist R;
    enum { r = 0 };
};

template <class Head, class Tail, template <typename> class Pred>
struct Partition<typelist<Head,Tail>,Pred> {
    typedef typename Select<
        Pred<Head>::r,
        typelist<Head,typename Partition<Tail,Pred>::R>,
        typename Append<typename Partition<Tail,Pred>::R,Head>::R
    >::R R;
    enum { r = Partition<Tail,Pred>::r + Pred<Head>::r };
};
```

The `Partition` meta-algorithm on typelists performs in a similar fashion to the STL `partition` algorithm on sequences. The nested type `R` is a copy of the original typelist, but reordered in such a way that all the types that satisfy the predicate occur before types that do not. The index of the first type that does not satisfy the predicate is available in the nested value `r`.

```
typedef Partition<ATypelist,IsDouble>::R Partitioned;
const int index = Partition<ATypelist,IsDouble>::r;
```

In the code snippet above, `Partitioned` is a reorganized version of `ATypelist` such that all the types that satisfy the predicate `IsDouble` (that is, all doubles) appear first, and `index` is the index of the first type in `Partitioned` that does not satisfy `IsDouble` (that is, is not double).

In a similar fashion, we can perform a compile time sort of a typelist with the help of a meta-comparator:

```
typedef Sort<ATypelist,IsSmaller>::R SortedSmaller;
```

Common Knowledge: Unfinished Business

```
typedef Sort<ATypeList,IsDerivedFrom>::R SortedHier;
```

The nested typename `R` is the reordered `ATypeList`, sorted according to the argument comparator. `SortedSmaller` is sorted by the `sizeof` of the type, and `SortedHier` gives a result based on the partial ordering of the typelist based on the is-a relationship.

Meta-Function Adapters

The STL includes the concept of function object adapters that can be used to modify and combine function objects to produce new function objects. We can do the same with our meta-predicates and comparators through the use of meta-function adapters. For example, we can negate the sense of a unary or binary meta-predicate:

```
template <template <typename> class X>
struct Not1 { // negate a unary predicate
    template <typename A>
    struct Adapted {
        enum { r = !X<A>::r };
    };
};
template <template <typename,typename> class X>
struct Not2 { // negate a binary predicate/comparator
    template <typename A, typename B>
    struct Adapted {
        enum { r = !X<A,B>::r };
    };
};
```

Our meta-predicates are implemented as class templates, so the adapter must accept a template template parameter. The result must be a template that can be instantiated with the same set of arguments as the original predicate. This template is available (again, by my convention only) as the nested template name `Adapted`.

```
typedef Sort<ATypeList,Not2<IsSmaller>::Adapted>::R NotSmallerFirst;
```

The typelist `NotSmallerFirst` contains the content of `ATypeList` sorted according to the `>=` operation on the type's sizes.⁹

As another example, it's sometimes useful to be able to change a binary predicate into a unary predicate by binding one of its arguments to a fixed value, or in the case of our meta-predicates, to a fixed type:

```
template <template <typename,typename> class X, typename A>
struct Bind1st { // bind the first type argument to A
    template <typename B>
    struct Adapted {
        enum { r = X<A,B>::r };
    };
};
```

Common Knowledge: Unfinished Business

```
template <template <typename,typename> class X, typename B>
struct Bind2nd { // bind the second type argument to B
    template <typename A>
    struct Adapted {
        enum { r = X<A,B>::r };
    };
};
```

For example, we can use one of our binders to reimplement `IsDouble`

```
template <class T>
struct IsDouble {
    enum { r = Bind2nd<IsSame,double>::template Adapted<T>::r };
};
```

assuming that we have available an implementation of `IsSame`:¹⁰

```
template <class A, class B>
struct IsSame { enum { r = false }; };
template <class A>
struct IsSame<A,A> { enum { r = true }; };
```

The availability of meta-adapters provides even more flexibility in composing complex predicates and comparators:

```
typedef Partition<
    ATypeList,
    Not1<Bind2nd<IsSmaller,int>::Adapted>::Adapted
>::R Partitioned2;
```

The typelist `Partitioned2` will be a reordering of `ATypeList` such that those types that are not smaller than an `int` will occur first. Similar adapters allow the composition of predicates, etc.

```
template <template <class> class X, template <class> class Y>
struct Or1 { // Or together two unary predicates
    template <class A>
    struct Adapted {
        enum { r = X<A>::r || Y<A>::r };
    };
};
```

The availability of STL-like algorithms on typelists appears to be occasionally useful. For example, if a typelist is going to be used to generate type-based conditional code, then it may be important that the typelist be ordered in a particular way. In an ad hoc Visitor¹¹ implementation over a hierarchy of shapes, it is important that derived shape classes appear before their base classes in the typelist if the generated conditional code is to be correct. This can be accomplished by a variety of means, one of which is to sort the typelist with the appropriate comparator:

```
typedef // typelist containing the types of Shape we know about
    typelist< Ellipse, // oops, out of order!
```

Common Knowledge: Unfinished Business

```
typelist< Circle,  
typelist< Square,  
typelist< Triangle,  
null_typelist> > > > Shapes;  
  
struct ShapeVisitor  
    : AdHocVisitor<Sort<Shapes,IsDerivedFrom>::R> { // OK, in order  
    //...  
};
```

This capability is particularly important if several disjoint pieces of code participate in construction of the typelist. The consumer of the typelist has the ability to enforce local constraints on its structure.

Other applications are easy to imagine. Consider a multi-conversion operation that performs a conversion of its argument to the first valid element of a sequence of types organized according to some set of criteria.¹²

```
template <class TList, typename T>  
struct ConversionRetType { // a helper for multiconvert  
    // remove any types for which there is no conversion  
    typedef typename  
        EraseIf<TList,  
            Not1<  
                Bind1st<CanConvert,T>::template Adapted  
            >::template Adapted  
        >::R Candidates;  
    // stick on a void so we know there's *something* there  
    typedef typename Append<Candidates,void>::R CandidatesV;  
    // take the first of what remains  
    typedef typename CandidatesV::head R;  
};  
  
template <class TList, typename T>  
typename ConversionRetType<TList,T>::R multiconvert( const T &t ) {  
    typedef typename ConversionRetType<TList,T>::R Ret;  
    return static_cast<Ret>(t); // cast to avoid warnings  
}
```

The availability of this operation will allow us to express a sequence of target types for a conversion:

```
Sort<ATypeList,IsSmaller>::R Sequence; // ordered targets  
//...  
T var; // a variable of some type  
cout << multiconvert<Sequence>( var ) << endl;
```

Common Knowledge: Unfinished Business

Of course, similar operations can be written for sets of conversions that can be obtained by `static_cast`, `const_cast`, and so on.

Part 4: Putting It All Together for Ad Hoc Type Manipulation

It would seem to be a generally useful capability to disassemble a type into its constituent parts, perform some analysis and transformation on its parts, then reassemble the modified sequence of parts into a new type.

```
template <typename T>
void aTemplateContext() {
    typedef typename Regen<
        typename TransformIf<typename Dis<T>::R, IsRef, MakePtr>::R
    >::R ModifiedT;
    //...
}
```

In the example above, we've decided to modify the template parameter `T` by replacing every reference modifier in the type with a pointer modifier through use of the `TransformIf` typelist meta-algorithm on the dismantled `T`. This is silly, but compelling. Unfortunately, it's not easy to find generally useful manipulations involving meta-algorithms on dismantled types, though there may be a few.

Oh well, not every technique is immediately—or even eventually—useful. At such times it helps me to reflect on two pieces of advice from Dr Seuss:

“If you never did you should. These things are fun, and fun is good.”¹³

“It's fun to have fun, but you have to know how.”¹⁴

Come to think of it, that's not a bad summary of the philosophy behind this series of columns. So long, keep learning and experimenting, and have fun.

Copyright © 2003 by Stephen C. Dewhurst

¹ “A Bit-Wise Typeof Operator,” Parts 1-3. *C/C++ Users Journal* 20, 8, 10,12 (August, October, and December 2002).

² “Typeints,” *C/C++ Users Journal Experts Forum*, 21, 2 (February 2003).

³ In fact, it would probably be better in this case to employ some sort of traits class to get the `element_type` of the container.

⁴ Particularly the big-block Ford models of the late '70s.

⁵ I've taken a few liberties here, but this is still the basic Alexandrescu type list. See his *Modern C++ Design*.

⁶ Obviously, a lot of the implementation has been omitted here for reasons of space and tedium. The full implementation (which will continue to evolve as I find time to work on it) is available on <http://www.semantics.org/code.html>.

Common Knowledge: Unfinished Business

⁷ To be perfectly precise, the current implementation handles only a bounded number of arguments for pointers-to-member-functions and functions, so any function or member function type that has more arguments than the implementation can handle will be recorded by the catchall.

⁸ This would also be an effective way to produce a compile time predicate to determine whether a particular type was generated from a template, or to circumvent the necessity of the “rebind” mechanism employed by the standard STL allocators, by moving the rebind capability into a facility separate from the allocator itself.

⁹ No, this is not a strict weak ordering, and may cause some implementations of `Sort` to fail. It just so happens that the `Sort` implemented here doesn't mind.

¹⁰ See *Modern C++ Design*, for instance. Note also the required use of the `template` keyword before the nested `Adapted` template name in the implementation of the `IsDouble` template.

¹¹ See Andrei Alexandrescu, “Typelists and Applications,” February 2002 installment of *Generic<Programming>*, <http://www.cuj.com/documents/s=7986/cujcexp2002alexandr/>.

¹² The `CanConvert` binary predicate is (again) from *Modern C++ Design*.

¹³ Dr. Seuss, *One Fish, Two Fish, Red Fish, Blue Fish*.

¹⁴ Dr. Seuss, *The Cat in the Hat*.