You Dis’n Me?

In recent installments of this feature, we looked at the concepts of compile time data structures (6 December, “Type Structures”) and the prospect of dismantling to their constituent parts types generated from class templates (6 January, “Template Dismantling”). In this installment, we’ll combine these two concepts as a first step on the way to a unified type-manipulation framework for use in template metaprogramming.

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 ) {
  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.[1] I find this sort of operation analogous to working on an automobile engine.[2] 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.

Type Dismantling

The first step in this process is the dismantling of a complex structure (whether it be 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 type list compile time list structure popularized by Andrei Alexandrescu:[3]

---

[1] 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.

[2] Particularly the big-block Ford models of the late ‘70s.

[3] I’ve taken a few liberties here, but this is still the basic Alexandrescu type list. See *Modern C++ Design*.

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

typedef struct {} NullTList;
```

A type list is just a sequence of types terminated with a `NullTList`. The structure is recursively defined, so that a type list is either a `NullTList`, or a pair consisting of a type and a type list.[4]

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

```
template <typename T>
struct Dis {
  typedef TList<T,NullTList> 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 additionally dismantle complex base types, or special case for base types with particular properties.

In order to record information about type modifiers in a type list, 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 TList<Const, typename Dis<T>::R> R;
};
template <typename T>
struct Dis<T *> {
  typedef TList<Ptr, typename Dis<T>::R> R;
};
template <typename T>
struct Dis< T *const> {
  typedef TList<Const, typename Dis<T *>::R> R;
};
```

[4] See “Once, Weakly” for 6 December 2002.

For example, `Dis<int * const>::R` will produce the three-element type list `[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 TList< Ary<b>,typename Dis<T>::R > R;
};

template <class C, typename T>
struct Dis<T C::*> {
  typedef TList<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 right now, 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:[5]

```
struct Fun0 {};

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

[5] 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.

```
    typedef TList<Fun0,typename Dis<Ret>::R> R;
  };

  template <typename Ret, typename Arg>
  struct Dis<Ret(Arg)> {
    typedef TList<Fun1<Arg>,typename Dis<Ret>::R> R;
  };
```

With this implementation of `Dis` in place, we can now dismantle an arbitrary type into its constituent parts:

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

Type Regeneration

Of course, a collection of piece parts, no matter how well organized, is not as useful as a functioning engine. Therefore, we also have a facility to regenerate a "normal" type from a type list that contains a dismantled type.

```
  template <class T>
  struct Regen;
```

We have no need of the implementation of the primary template, so we omit it. The partial specializations of `Regen` handle the translation back to a normal type. The inversion of the catchall case described above simply pulls the type out of the single-element type list:

```
  template <typename T>
  struct Regen< TList<T,NullTList> > {
    typedef T R;
  };
```

The other cases are pretty much what one would expect. The regeneration process recursively regenerates the modified type, then attaches the appropriate modifier:

```
  template <class T>
  struct Regen< TList<Const,T> > {
    typedef const typename Regen<T>::R R;
  };
  template <class T>
  struct Regen< TList<Ptr,T> > {
    typedef typename Regen<T>::R *R;
  };
  template <class T, class C>
  struct Regen< TList<Pcm<C>,T> > {
    typedef typename Regen<T>::R C::*R;
  };
```

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.

Domain-Specific Extension

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,[6] 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 Tlist<Int,NullTList> R;
};
template <>
struct Regen< TList<Int,NullTList> >  {
  typedef int R;
};
```

These three new elements extend the framework to special case on `int`. Useless. However, there are more useful extensions. For example, we saw in an earlier “Once, Weakly” that it is possible to dismantle a type generated from a class template into the original template and the arguments used in its instantiation.[7]

[6] 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.

[7] “Template Dismantling,” 6 January 2003.

```
template <template <typename> class Templ, typename T>
struct TemplT {
  template <typename S>
  struct Subst
        { 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 TList<TemplT<Templ,T>, NullTList> R;
};

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

The above 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.[8]

Type Manipulation

What's the use of all this mechanism, if we're only going to dismantle a type to put it back together again? As with an engine, once we have a type totally dismantled we have the ability to substitute one part for another, remove parts, and modify parts before reassembly.

For example, we can write a simple substitution facility that allows us to replace, within a dismantled type, a particular part with some other (perhaps more complex) part:

```
template <class T, template <typename> class Pred, class S>
struct Subst {
  typedef typename SubstN<1,T,Pred,S>::R R;
};
```

The `Subst` template makes use of a compile time predicate function (similar to an STL runtime predicate) that is applied to each component of the dismantled representation of the type. If the predicate is true, then the component is replaced by `S`, where `S` is a dismantled type. (`Subst` is actually implemented as a special case of a more general facility, `SubstN`, which performs up to `N` substitutions on `T`. See the implementation for details.)

[8] Exercise for the reader: Extend the type-dismantling framework so that it can handle the type-dismantling framework. (Yes, this should cause one to revisit the results of Gödel's incompleteness theorem.)

A slightly more useful substitution operation would involve the component being replaced in the construction of what replaces it. For this, we can employ a compile time applicator that can be used to transform a component that satisfies the predicate:

```
template <class T, template <typename> class Pred,
                    template <typename> class Apply>
struct SubstA {
  typedef typename SubstAN<1,T,Pred,Apply>::R R;
};
```

`SubstA` is similar to `Subst`, but components that satisfy the predicate are replaced by their transformation by the applicator, rather than by a fixed, dismantled type. (Like `Subst`, `SubstA` is a specialization of a more general facility.)

These substitution mechanisms are fairly flexible. For instance, it's easy to write predicates and applicators to modify a function type's arguments to be references to const, or to substitute the use of one standard container for another, etc. However, it would be even better to have a more flexible set of facilities that would employ more general, regular-expression type matching and replacement, or one that would allow specification of a context-free grammar that would control the manipulation of the dismantled type. Stay tuned…