Template Dismantling

Dismantling Types

It's a fairly common practice in template metaprogramming to dismantle a type into its component parts in order to perform some analysis or manipulation on the type. For example, we can ask if a particular type is an array type. If it is, then we can dismantle it into its base type and array bound.

```
template <typename T>
struct IsAry {
  enum { r = false };
};

template <typename T, int n>
struct IsAry<T[n]> {
  enum {
        r = true,
        bound = n
  };
  typedef T Base;
};
```

Once we've reduced a type to its piece parts, we can modify or reassemble the pieces to suit our needs, in much the same way one might dismantle and rebuild a machine composed of many different parts.

```
template <typename Cont>
//…
// if container is an array, make an array type twice as big
typedef IsAry<Cont> C;
typedef typename Select<
                    C::r,
                    typename C::Base[C::bound*2],
                    Cont
               >::R NewCont;

// if container is an array, make type with same bound and char base
typedef typename Select<
                    C::r,
                    char[C::bound],
                    Cont
               >::R NewerCont;
```

Dismantling Templates

Oddly, I haven't seen any similar manipulations of types generated from templates (but then, I don't get around much). However, it's fairly straightforward (albeit syntactically challenging) to do so.

Suppose we'd like to know whether a particular type was generated from a template. We'll proceed as we did above, using class template partial specialization. Our primary template assumes the negative:

```
template <typename T>
struct IsTemplate {
  enum { r = false };
};
```

Our first partial specialization will catch cases in which a type is generated from a class template with a single type argument:

```
template <template <typename> class X, typename T>
struct IsTemplate< X<T> > {
  enum { r = true };
  typedef T Type;
};
```

Notice our use of a template template parameter. In this case, we've specified a template template argument that can be instantiated with a single type argument. We can now distinguish between types that are generated from a class template that accepts a single type argument and all other types:

```
template <typename T> struct Templ;
//…
cout << IsTemplate<char>::r << endl; // false
cout << IsTemplate< Templ<int> >::r << endl; // true
cout << IsTemplate< vector<int> >::r << endl; // false!
```

Note that this mechanism fails for the `std::vector` template. Recall that the standard sequence containers are defined with two type parameters, an element type and an allocator type. The second, allocator parameter has a default, but that does not help us here. We can improve this situation by providing partial specializations for templates that accept other sets of arguments.

```
template <template <typename,typename> class X,
  typename T1, typename T2>
struct IsTemplate< X<T1,T2> > {
  enum { r = true };
};
```

Now we can handle the standard sequence containers as well as any other templates that accept two type parameters.

```
cout << IsTemplate< vector<int> >::r << endl; // works…
```

We can continue in this fashion until we have a set of partial specializations that can handle all cases of interest.

```
template <template <int> class X, int n>
struct IsTemplate< X<n> > {
  enum { r = true };
};
```

Of course, our goal is not only to identify types that have been generated from templates, but also to dismantle them for later manipulation.

```
template <template <typename,typename> class X,
  typename T1, typename T2>
struct IsTemplate< X<T1,T2> > {
  enum { r = true };
  typedef T1 FirstType;
  typedef T2 SecondType;
  typedef X Templ; // error! illegal code!
};
```

It's easy enough to record the type and non-type parameters as nested values; for example, we've provided access to the two type parameters of the template above.

```
typedef IsTemplate< vector<int> >::SecondType AllocatorType;
```

However, it's less straightforward to provide access to the template template parameter. It would be nice if we could provide direct access to the name through a typedef, as we do with a type name, but the parameter `X` is the name of a template, not a type.

Substitutions

Even if we can't provide direct access to the template template parameter, we can still provide indirect access to it. For example, we can allow it to be used to generate new types through substitution.

```
template <template <typename,typename> class X,
  typename T1, typename T2>
struct IsTemplate< X<T1,T2> > {
  enum { r = true };
  typedef T1 FirstType;
  typedef T2 SecondType;
  template <typename S1, typename S2 = T2>
  struct SubstType {
        typedef X<S1,S2> R;
  };
  template <template <typename,typename> class Templ,
        typename S1 = T1, typename S2 = T2>
  struct SubstTempl {
        typedef Templ<S1,S2> R;
  };
};
```

Here we've provided a set of nested templates that provide the user of `IsTemplate` the ability to substitute either or both type arguments, or the template argument, to produce a new type. To continue the machine analogy, we can still manipulate the individual parts of the machine, but we cannot remove them from the machine's housing.

```
typedef IsTemplate< vector<int> >::SubstType<char>::R MyVec;
```

The type `MyVec` is `vector< char, allocator<int> >` since only the first type parameter to the `vector< int, allocator<int> >` was substituted.

```
typedef IsTemplate< vector<int> >::SubstTempl<list>::R MyList;
```

In the case above, we've defined the type `MyList` to be the `list` analog of the `vector<int>`.

Applications

I suspect this technique is going to prove to be fairly useful although, because it is new, there are no current uses of it. However, we can outline an example that demonstrates its potential.

Suppose we have a situation in which we have an STL-compliant container, but we require that the container have a random-access iterator. If it doesn't, we have to construct and populate a container that does have a random access iterator, but still has the same element and allocator types as the original container. (Now, in fact, this is fairly easy to do using existing STL facilities, but the point of this exercise is to indicate the possibility of doing similar things in a context where we cannot rely on convention.)

```
template <class Cont>
void process( Cont &c ) {
  typedef typename Select< // #1
    IsRand<typename Cont::iterator>::r, // #2
    Cont, // #3
    typename IsTemplate<Cont>::template SubstTempl<vector>::R // #4
  >::R MyCont;
  cout << "process: " << typeid(Cont).name()
       << "\n\tbecame " << typeid(MyCont).name() << endl;
  //…
}
```

The `typedef` starting on line #1 defines the type described above; if the type `Cont` has a random access iterator (line #2, see the definition of `IsRand` below) then the type is unchanged from that of the function's argument (line #3). Otherwise, the `IsTemplate` facility is used to generate a new container type (line #4). The type generated will be a standard `vector` with the same element and allocator types as `Cont`. (Note the required use of the `template` keyword on line #4 to allow the compiler to parse the nested template name `SubstTempl` correctly.)

Appendix: Miscellaneous Utilities

Here's the source code for some utilities that appeared above.

Select

`Select` is basically a compile time if-statement whose result type is either the second or third argument, based on the first argument.

```
template <bool, typename A, typename>
struct Select {
    typedef A R;
};

template <typename A, typename B>
struct Select<false,A,B> {
    typedef B R;
};
```

IsRand

`IsRand` assumes that its argument is an STL-compliant iterator, and determines whether the iterator is random access or not.

```
template <typename In>
struct IsRand {
    enum { r =
        IsSame<typename std::iterator_traits<In>::iterator_category,
                 std::random_access_iterator_tag>::r };
    typedef typename Select< r, std::random_access_iterator_tag,
        std::input_iterator_tag>::R R;
};
```

IsSame

`IsSame` determines whether two types are the same or not. Of course.

```
template <typename T1, typename T2>
struct IsSame {
  enum { r = false };
};

template <typename T>
struct IsSame<T,T> {
  enum { r = true };
};
```