Typelist Meta-Algorithm Implementation Tricks

In the "Once, Weakly" of 9 September 2003 we looked at the concept of typelist meta-algorithms. These are algorithms for manipulating typelists at compile time in a manner reminiscent of STL generic algorithms. We also saw how to create meta-function objects (including meta-predicates and meta-comparitors), and meta-function adapters.

In this installment of "Once, Weakly" we'll examine a few more typelist meta-algorithms to motivate a few somewhat half-baked metaprogramming techniques used to implement them. (That's why I'm calling them "tricks" instead of something more pretentious, like "strategies.")

Straightforward Leveraging

Many new meta-algorithms can be implemented in a straightforward way from existing meta-algorithms. An obvious example is the implementation of `Unique` in terms of `UniqueEquiv`.

```
template <class TList, template <class,class> class BPred>
struct UniqueEquiv;
```

`UniqueEquiv` removes duplicate adjacent types in a typelist that compare equal according to the binary predicate `BPred`. `Unique` does the same thing but uses the type equality by default. It is trivially implemented by invoking `UniqueEquiv` with the appropriate predicate.

```
template <class TList>
struct Unique {
  typedef typename UniqueEquiv<TList,IsSame>::R R;
};
```

In the same way, we can implement `Transform` in terms of `TransformIf` through use of a very agreeable predicate:

```
template <typename>
struct IsTrue { enum { r = true }; };
...
template <class TList, template <typename> class Op>
struct Transform {
  typedef typename TransformIf<TList,IsTrue,Op>::R R;
};
```

Another example is the implementation of `Find` in terms of `FindIf`. However, `Find` is searching for a particular type, whereas `FindIf` requires a predicate. We simply generate the appropriate predicate using an adapter to bind one argument of the `IsSame`[1] binary predicate to produce a unary predicate:

```
template <class TList, typename T>
```

[1] `IsSame` is from Andrei Alexandrescu's *Modern C++ Design*.

```
struct Find {
  enum {r = FindIf<TList, Bind2nd<IsSame,T>::template Adapted>::r};
};
```

So `Find` is implemented in terms of `FindIf`, where the predicate asks if a type is the same as `T`.

Ad Hoc Meta-Functions

Consider implementing an `EraseIf` algorithm:

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

We'd like to apply a predicate to each element of the typelist, and produce a typelist that contains only the elements that did not satisfy the predicate. We could implement this functionality from scratch, but we have an existing `TransformIf` algorithm and an existing `EraseAll` algorithm.[2] We can leverage these two if we can map the elements to be removed to a particular type:

```
struct ToErase {};
```

The `TransformIf` algorithm requires a meta-function to apply to its typelist. A very simple ad hoc meta-function will do:

```
template <typename>
struct MakeToErase {
  typedef ToErase R;
};
```

This function maps any type to `ToErase`. Now the implementation of `EraseIf` is trivial; we use `TransformIf` to convert any element that satisfies the predicate into `ToErase`, then use `EraseAll` to remove all the `ToErase`s.

```
template <class TList, template <typename> class Pred>
struct EraseIf {
  typedef typename TransformIf<TList,Pred,MakeToErase>::R Marked;
  typedef typename EraseAll<Marked,ToErase>::R R;
};
```

Ad Hoc Adapters

Consider the problem of implementing a set union of two typelists, where a "less-than" comparator is supplied explicitly:

```
template <class TList1, class TList2,
              template <typename,typename> class Comp>
struct SetUnionEquiv;
```

[2] `EraseAll` is also Andrei's, and `TransformIf` was described in the Once, Weakly of 9 September 2003. Any unattributed meta-algorithms may be found there, or in the source code that accompanies this installment of "Once, Weakly." That source code may be found at http://www.semantics.org/code.html.

This would seem like a fairly easy task, if we're not too concerned about compile time efficiency.[3] Using existing meta-algorithms from our toolkit,[4] we can just paste the two typelists together, sort the result, and get rid of adjacent duplicates.[5]

```
template <class TList1, class TList2,
            template <typename,typename> class Comp>
struct SetUnionEquiv {
  typedef typename Append<TList1,TList2>::R RR;
  typedef typename Sort<RR,Comp>::R SRR;
  typedef typename UniqueEquiv<SRR,???>::R R;
};
```

The problem is that we've been supplied with a comparator, but we need an equivalence operation to instantiate `UniqueEquiv`. (Speaking somewhat inaccurately, we need an `operator ==` of some sort, and all we have is an `operator <`.) Our existing meta-object adapters don't quite do what we need, although we can leverage them with a little ad hoc trickery. First we create an adapter that exchanges the order of arguments of a binary predicate:

```
template <template <class,class> class BPred>
struct ExchangeArgs {
  template <typename A, typename B>
  struct Adapted {
        enum { r = BPred<B,A>::r };
  };
};
```

Now we can produce an equivalence operation from a "less-than" operation as `A equiv B == !(A<B) && !(B<A)`. That is, `A` and `B` are equivalent if neither is less than the other.

```
...
typedef typename
  UniqueEquiv<SRR,And2<
        Not2<Comp>::template Adapted,
        Not2<
              ExchangeArgs<Comp>::template Adapted
        >::template Adapted
  >::template Adapted>::R R;
```

---

[3] Usually we aren't. However, if we were to deal with very long typelists, the asymptotic complexity of our meta-algorithms can be important. Also, see Scouting Out Optimizations. *C/C++ Users Journal Experts Forum*, 21, 4 (April 2003) for a number of techniques that can be used to improve compile time performance of metaprograms without changing their asymptotic complexity.

[4] `Append` is Andrei's too.

[5] Yes, this differs from the behavior of the STL `set_union`, and yes, I meant it to, and no, I am not going to change it to mimic the STL `set_union`. My way is better.

```
...
```

It's not hard to understand how this sort of code can be irritating to maintainers. It's probably better to create a simpler ad hoc adapter that does the same thing, but more clearly:

```
template <template <typename,typename> class Comp>
struct GenEquivalence {
  template <typename A, typename B>
  struct Equivalence {
       enum { r = !Comp<A,B>::r && !Comp<B,A>::r };
  };
};
```

This special-purpose adapter takes a comparator and produces an equivalence operation. It's instantiated with a comparator and produces a conformant equivalence operation as a nested template. Invoking the nested template involves the usual syntactic contortions to inform the compiler that the nested name `Equivalence` is a template name:

```
...
typedef typename
  UniqueEquiv<SRR,GenEquivalence<Comp>::template Equivalence>::R R;
...
```

Marking, Extracting, and Purging

Before we look at another user-level meta-algorithm, let's consider the implementation of some behind-the-scenes functionality.

Many meta-algorithms have a logical structure equivalent to selecting some subset of the elements of a typelist, and then doing something with that subset. We can reify that selection process with a marking algorithm:

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

`MarkList` "marks" the elements of `TList` that satisfy `Pred` by constructing a parallel Boolean typelist that indicates which elements are "marked." Rather than come up with a compile-time Boolean list construct, however, we can employ a typelist of known structure. This implementation uses a typelist that contains types of the form `char (*)[n]`, where n is in the range from 1 to some platform-specific upper bound. By convention, we'll interpret `char(*)[1]` as false, and other bounds as true.[6]

```
template <typename Head, class Tail, template <typename> class Pred>
struct MarkList<typelist<Head,Tail>,Pred> {
  typedef typelist<char(*)[Pred<Head>::r+1],
                   typename MarkList<Tail,Pred>::R> R;
};
```

[6] This encoding can also serve as a compile-time list of positive integers through the use of `sizeof` on the dereferenced pointer.

```
template <template <typename> class Pred>
struct MarkList<null_typelist,Pred> {
  typedef null_typelist R;
};
```

Once we have a means of identifying the subset of interest, we can implement other operations. For instance, we can extract the marked items into a typelist:

```
template <class TList, class Marks>
struct ExtractList;

template <typename Head, class Tail, int bound, class MTail>
struct ExtractList< typelist<Head,Tail>,
                    typelist<char(*)[bound],MTail> > {
  typedef typename ExtractList<Tail,MTail>::R ETail;
  typedef typename Select<
        !!(bound-1),
        typelist<Head,ETail>,
        ETail
  >::R R;
};

template <>
struct ExtractList<null_typelist,null_typelist> {
  typedef null_typelist R;
};
```

…or purge the items from the typelist:

```
template <class TList, class Marks>
struct PurgeList;

template <typename Head, class Tail, int bound, class MTail>
struct PurgeList< typelist<Head,Tail>,typelist<char(*)[bound],MTail>
> {
  typedef typename PurgeList<Tail,MTail>::R PTail;
  typedef typename Select<
        !(bound-1),
        typelist<Head,PTail>,
        PTail
  >::R R;
};

template <>
struct PurgeList<null_typelist,null_typelist> {
```

```
    typedef null_typelist R;
};
```

…or apply a meta-function to the marked items, or whatever.

Set Union Redux

Earlier, we examined the implementation of a set union algorithm, `SetUnionEquiv`, that employed a user-supplied comparator. However, that implementation of set cannot (easily) be used to union two typelists based on the traditional notion of set union; that is, that the resultant set would have no duplicate types, but also that no unique type would be omitted from the union. However, because the set of C++ types is not ordered (implicitly, at compile time) it is not (easily) possible to construct an appropriate comparator for `SetUnionEquiv`. Instead, let's write a special-purpose version of set union. Recall the implementation of `SetUnionEquiv`:

```
template <class TList1, class TList2,
            template <typename,typename> class Comp>
struct SetUnionEquiv {
    typedef typename Append<TList1,TList2>::R RR;
    typedef typename Sort<RR,Comp>::R SRR;
    UniqueEquiv<SRR,GenEquivalence<Comp>::template Equivalence>::R R;
};
```

The implementation of `SetUnion` should be similar:

```
template <class TList1, class TList2>
struct SetUnion {
    typedef typename Append<TList1,TList2>::R RR;
    typedef typename Sort<RR,???>::R SRR;
    typedef typename Unique<SRR>::R R;
};
```

However, we've run into a problem with `Sort` to which we alluded above. There is no well-defined ordering on C++ types, so we have to find some other mechanism to bring equivalent types into adjacency so that they can be eliminated with `Unique`.

One approach might be to abandon the notion of sorting the typelist, instead "clumping together" equivalent types based on an equivalence relation:

```
template <class TList, template <typename,typename> class Eq>
struct Clump;
```

We can implement the clumping functionality in a straightforward fashion by marking sets of equivalent types, extracting them from the typelist, purging them from the typelist, and continuing until there are no types left to mark.

```
template <typename Head, class Tail,
            template <typename,typename> class Eq>
struct Clump<typelist<Head,Tail>,Eq> {
    typedef typelist<Head,Tail> Orig;
    typedef typename MarkList<Orig,
```

```
        Bind1st<Eq,Head>::template Adapted>::R HeadMarks;
    typedef typename ExtractList<Orig,HeadMarks>::R HeadList;
    typedef typename PurgeList<Orig,HeadMarks>::R TailPurged;
    typedef typename Clump<TailPurged,Eq>::R TailList;
    typedef typename Append<HeadList,TailList>::R R;
};

template <template <typename,typename> class Eq>
struct Clump<null_typelist,Eq> {
    typedef null_typelist R;
};
```

Now that we have an implementation of `Clump`, we can rid ourselves of `Sort` in the implementation of `SetUnion`:

```
template <class TList1, class TList2>
struct SetUnion {
    typedef typename Append<TList1,TList2>::R RR;
    typedef typename Clump<RR,IsSame>::R SRR;
    typedef typename Unique<SRR>::R R;
};
```

Other Algorithms

Similar techniques are used to implement other meta-algorithms, and the implementations for the following are available at present on Semantics' code page.[7] (They'll eventually find their way into the Tyr library.[8])

```
template <class TList, typename T>
struct Find;
template <class TList, template <class> class Pred>
struct FindIf;
template <class Tlist, typename T>
struct Count;
template <class Tlist, template <class> class Pred>
struct CountIf;
template <class TList>
struct Unique;
template <class TList, template <class,class> class BPred>
struct UniqueEquiv;
template <class TList, template <typename> class Op>
struct Transform;
template <class TList1, class TList2,
```

[7] http://www.semantics.org/code.html

[8] http://www.semantics.org/tyr.html

```
        template <typename,typename> class Op>
struct Transform2;
template <class TList, template <typename> class Pred,
        template <typename> class Op>
struct TransformIf;
template <class TList, template <typename> class Pred>
struct EraseIf;
template <class TList, template <class,class> class Comp>
class Sort;
template <class TList>
struct Rotate;
template <class TList, int n>
struct RotateN;
template <class TList, template <typename,typename> class Comp>
struct MinElement;
template <class TList, template <typename,typename> class Comp>
struct MaxElement;
template <class TList1, class TList2,
        template <typename,typename> class Comp>
struct EqualIf;
template <class TList1, class TList2>
struct EqualSame;
template <int n, typename T>
struct FillN;
template <class TList, typename S, typename T>
struct Replace;
template <class TList, template <typename> class Pred, typename T>
struct ReplaceIf;
template <class TList1, class TList2,
        template <typename,typename> class Comp>
struct Merge;
template <class TList1, class TList2,
        template <typename,typename> class Comp>
struct SetUnionEquiv;
template <class TList1, class TList2>
struct SetUnion;
template <class TList1, class TList2>
struct SetIntersection;
template <class TList1, class TList2>
struct SetDifference;
template <class TList1, class TList2>
struct SetSymmetricDifference;
```