Typelist Meta-Algorithms

The other day I was reading Andrei Alexandrescu's clever implementation of an ad hoc Visitor that I had unaccountably overlooked when it first appeared. (See "Typelists and Applications," Andrei's February 2002 installment of his Generic<Programming> online column for *The C/C++ Users Journal*.[1]) The implementation uses a compile time traversal of a typelist to avoid many—though far from all—of the problems associated with nested if-statements whose conditions are `dynamic_casts`. That is, code like the following

```
Shape *s = get_a_Shape_of_some_kind();
if( Circle *c = dynamic_cast<Circle *>(s) {
   do something…
}
else if( Ellipse *e = dynamic_cast<Ellipse *>(shape) ) {
   do something;
}
else if( Square *s = dynamic_cast<Square *>(shape) ) {
   do something…
}
else if( Triangle *t = dynamic_cast<Triangle *>(shape) ) {
   do something…
}
```

can be replaced by mechanism that effectively has the compiler write the nested if-statements automatically based on the ordering of types in a typelist:[2]

```
typedef // typelist containing the types of Shape we know about
   typelist< Ellipse,
   typelist< Circle,
   typelist< Square,
   typelist< Triangle,
   null_typelist> > > > Shapes;

struct ShapeVisitor : AdHocVisitor<Shapes> {
   void visit( Circle * );
   void visit( Ellipse * );
   void visit( Square * );
   void visit( Triangle * );
};
```

[1] http://www.cuj.com/documents/s=7986/cujcexp2002alexandr/

[2] I plan to discuss aspects of Andrei's ad hoc Visitor in a future Once, Weakly, so I'm offering no explanation here as to the details of the implementation.

```
ShapeVisitor v;
Shape *s = get_a_Shape_of_some_kind();
v.startVisit(s);
```

Since the sequence of conditions tested depends on the order of the types in the typelist, it's essential to make sure that the typelist (in the case above, `Shapes`) is properly ordered. In the case of our ad hoc `Shape` Visitor, it's essential that derived classes appear before their base classes in the typelist, so that the `Shape` object being processed will be associated with the most specific type on the typelist. In the hand-coded version of the nested if-statement, this is done properly. However, the `Shapes` typelist is not in the proper order, and a `Circle` object will be processed as an `Ellipse` rather than a more-specific `Circle` (assuming that `Circle` is derived from `Ellipse`).

Andrei addresses this particular problem in his *Modern C++ Design* (MCD, p. 62) with an implementation of a compile time algorithm, `DerivedToFront`, that organizes a typelist so that derived classes appear before their base classes. This can be used in the implementation of an ad hoc Visitor to ensure that its typelist is properly ordered:

```
struct ShapeVisitor
   : AdHocVisitor<DerivedToFront<Shapes>::Result> { // etc…
```

This mechanism works well in this specific case, but it seems fairly clear that in other contexts we might want to perform different or more complex orderings and other manipulations on typelists.

In fact Andrei (MCD, p. 62) mentions the possibility of using a sort of some kind on a typelist, but then dismisses the idea in favor of the more specific `DerivedToFront`. His decision was based on the observation that inheritance is not a total ordering relationship, and that there is no other equivalent for an `operator <` for types.

In this installment of Once, Weakly, we'll take up the subject of more general manipulations of typelists, based on the STL model of generic algorithms, but one step removed. Meta-generic algorithms.

Typelists in Five Words

They're structured pairs of types. Here's all you need to know:

```
template <class H, class T>
struct typelist {
   typedef H head;
   typedef T tail;
};
class null_typelist {};
```

A typelist is a list of types that is recursively defined as a type followed by a typelist. The end of a typelist is signaled by the appearance of a `null_typelist` rather than an instantiated `typelist`. Therefore the `head` of a typelist is any sort of type (possibly, but not typically, another typelist) and the `tail` of a typelist is either an instantiated `typelist` or `null_typelist`. An empty typelist is represented by `null_typelist`. Here are some example typelists:

```
typedef null_typelist TL1; // empty typelist
typedef typelist<int,null_typelist> TL2; // 1-element typelist
typedef typelist<string,
  typelist<int,null_typelist> > TL3; // 2-element typelist
typedef typelist<string,TL2> TL3; // same as previous
```

Obviously, the hand-coding approach to typelist creation is tedious in the extreme, and there are several more convenient ways to construct them. See *Modern C++ Design* and the "Typelists and Applications" article referenced earlier for several useful techniques.

Typelists may be manipulated in various ways at compile time. For example, we can calculate the length of a typelist with a simple "metaprogram":

```
template <class TList>
struct Length;

template <class T, class U>
struct Length< typelist<T,U> > {
  enum { r = 1 + Length<U>::r };
};

template <>
struct Length<null_typelist> {
  enum { r = 0 };
};
```

That is, the length of an empty typelist is 0, and the length of a non-empty typelist is recursively calculated as one plus the length of its tail. Many basic typelist algorithms are implemented in a similar way, and you may find algorithms to in index, search, append, etc. typelists. If you are unfamiliar with basic typelist manipulation, now would be a good time to read chapter 3 of *Modern C++ Design*, or have a look at the implementations of some of the typelist operations in the file `typelist.h` in the code that accompanies this installment of Once, Weakly.[3]

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. This is the case, for example, for the sort algorithm mentioned above.

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.

[3] The implementation of all the facilities discussed here (and more) may be found at http://www.semantics.org/code.html, under the "Typelist Meta-Algorithms" heading.

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 it 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 <class A, class B>
struct IsSmaller {
  enum { r = sizeof(A) < sizeof(B) };
};

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

Using these meta-comparators, we can compare pairs of types based on their 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 <class> class Pred>
struct Partition;

template <template <class> class Pred>
struct Partition<null_typelist,Pred> {
  typedef null_typelist R;
```

```
  enum { r = 0 };
};

template <class Head, class Tail, template <class> 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 `double`s) 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;
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` the type, and `SortedHier` gives a result similar to that of the `DerivedToFront` special-case algorithm mentioned above.[4]

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 <class> class X>
struct Not1 { // negate a unary predicate
```

[4] The code available on semantics.org gives implementations for `Partition`, `Sort`, and `TransformIf` algorithms.

```
  template <class A>
  struct Adapted {
        enum { r = !X<A>::r };
  };
};
template <template <class,class> class X>
struct Not2 { // negate a binary predicate/comparator
  template <class A, class 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) 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.[5]

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

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

This provides even more flexibility in composing complex predicates and comparators:

```
typedef Partition<
```

[5] 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.

```
    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.

Exercise for the Reader: TemplateLists

An earlier Once, Weakly ("Type Structures," 6 December 2002) discussed the design of templatelists, which are simply lists of templates analogous to the lists of types we've been discussing here. An interesting exercise might be to extend the approach of this Once, Weakly to encompass templatelists, and see what effect this expanded use of template lists would have on the implementation of policy-based designs. Just a suggestion.