Type Structures

This installment discusses the concept of “type-structures” as a compile-time analog of runtime data structures, and demonstrates how to construct and manipulate simple list structures at compile time. Whereas runtime data structures contain and manipulate runtime objects, these structures contain compile-time entities such as integer constants, types, and templates.

Data Structures

Consider a simple, singly-linked list data structure:

```
struct List {
    int elem_;
    List *next_;
};
```

For clarity of exposition we’re not employing proper data abstraction in the implementation of this list-of-integers type. Ordinarily we would protect the representation of the list, and make the list operations members of the list itself. Here, they’re non-members:

```
List *insert( List *head, int val ) {
    List *newHead = new List;
    newHead->elem_ = val;
    newHead->next_ = head;
    return newHead;
}

int length( List *head ) {
    if( head )
        return 1 + length( head->next_ );
    else
        return 0;
}

int index( List *lst, int i ) { // get int at index i
    if( i )
        return index( lst->next_, i-1 );
    else
        return lst->elem_;
}
```

Note that we've employed a recursive approach to the list `length` and `index` operations, even though an iterative approach would have been more efficient. However, the implementation is effective (that is, it works) even though it leaves a lot to be desired in terms of abstraction and efficiency.

Recursive Type-Structures

Let's look at a compile-time analog to the above list data structure:

```
template <int n, class N>
struct List_i {
    enum { elem = n };
    typedef N Next;
};

typedef struct {} NullList_i;
```

The `List_i` template is a type-structure analog to the `List` data structure. The element value is part of the list node's type (and is recorded as an enumerator). The analog of the data structure's `next_` pointer is a nested type. The nested type is another instantiation of `List_i` with a different type for `Next`, and (probably) a different value for `elem`. The unique type `NullList_i` serves to terminate the list in the same way that a null `next_` pointer terminates the data structure version of the list. Therefore, an empty list implemented as a data structure is a simple null pointer, whereas an empty list implemented as a type-structure is the type `NullList_i`. A data structure list containing a single integer consists of a single `List` object with a null `next_` pointer. A type-structure list containing a single integer consists of an instantiation of `List_i` with `NullList_i` as the nested `Next` typename.

```
typedef NullList_i L0; // an empty list
typedef List_i<12,L0> L1; // a one-element list
typedef List_i<72,L1> L2; // a two-element list
typedef List_i< 72,List_i<12,NullList_i> > L2; // the same list
```

We can also implement compile-time analogs of the three list operations above. Inserting (prepending) an integer to an existing list is straightforward:

```
template <int val, class L>
struct Insert_i {
    typedef List_i<val,L> R;
};
```

This "insert" template simply prepends its integer argument onto an existing list.

```
typedef Insert_i<144,L2>::R L3; // a three-element list
```

Calculating the length of a list requires recursion:

```
template <typename L> struct Length_i;

template <int n, class L>
struct Length_i< List_i<n,L> > {
```

```
    enum { r = 1 + Length_i<L>::r }; // recurse for length of tail
};

template <>
struct Length_i<NullList_i> {
    enum { r = 0 };
};
```

As with the analogous runtime recursion employed in the `length` function, the `Length_i` template recursively calculates the length of the tail of the list, and adds one to the result. The recursion terminates with the complete specialization for a null list, just as the runtime recursion terminates when it encounters a null pointer. Note that the result of the calculation is an integer constant-expression.

```
double someArray[ Length_i<L3>::r ]; // array of 3 doubles
```

Extracting the integer value of a node at a given index within the list is only slightly more involved.

```
template <class List, int index>
struct Index_i;

template <int elem, class T, int i>
struct Index_i< List_i<elem,T>,i > {
    enum { r = Index_i<T,i-1>::r };
};

template <int elem, class T>
struct Index_i< List_i<elem,T>,0 > {
    enum { r = elem };
};
```

As with the runtime-recursive version, as shown in the `index` function, the compile-time-recursive version implemented in `Index_i` does a “countdown” of the requested index as it traverses the list. When the index has been counted down to zero, the integer at head of the list is the required element.

```
switch( anInt ) {
case Index_i<L3,0>::r:
    stmt1; break;
case Index_i<L3,1>::r:
    stmt2; break;
case Index_i<L3,2>::r:
    stmt3; break;
}
```

Note that, again, the result of the `Index_i` operation is an integer constant-expression, and (in this case) the code above results in a switch over the integer values 144, 72, and 12.

Type Lists

In point of fact, it's relatively rare to manipulate compile-time data structures of integers or other basic types (although this is occasionally useful). Templates are limited in what kings of arguments may be used for their instantiation: integers (as we've seen for `List_i`, above), the addresses of objects and functions with external linkage, and pointers to class members.

```
template <void (*f)(), class U>
struct FList { // list of pointers to external functions
  static void exec() { f(); }
  typedef U Tail;
};

typedef struct {} NullFList;

template <typename> struct FLength;

template <void(*f)(), class U>
struct FLength< FList<f,U> > {
  enum { value = 1 + FLength<U>::value };
};

template <>
struct FLength<NullFList> {
  enum { value = 0 };
};

// etc…
```

However, it *is* fairly common to manipulate collections of types at compile time.[1] A few minor changes to our implementation of `List_i` produce an equivalent type-structure whose elements are types, rather than integers.[2]

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

typedef struct {} NullTList;
```

[1] The original and best treatment of this subject may be found in Andrei Alexandrescu's *Modern C++ Design*, Addison-Wesley, 2001. The discussion here owes a lot to his presentation there.

[2] See the addendum for an example of a type-structure whose elements are templates, rather than types.

```
template <typename> struct TLength;

template <typename T, class U>
struct TLength< TList<T,U> > {
  enum { value = 1 + TLength<U>::value };
};

template <>
struct TLength<NullTList> {
  enum { value = 0 };
};

template <class,int> struct TIndex;

template <class H, class T, int i>
struct TIndex< TList<H,T>,i > {
  typedef typename TIndex<T,i-1>::R R;
};
```

A few preprocessor macros can be used to ease the creation of type lists:

```
#define TList1(T1) TList<T1,NullTList>
#define TList2(T1,T2) TList<T1,TList1(T2) >
#define TList3(T1,T2,T3) TList<T1,TList2(T2,T3) >
```

Now we can use our type-structures for compile-time manipulation of collections of types:[3]

```
typedef TList3(char,int,double) X;
cout << TLength<X>::value << endl;
TIndex<X,2>::R anObject; // an object of type double
```

Template Lists

It's instructive (and fun) to extend our notion of a type list to that of a template list. Here we have a list of class templates that (like the standard sequence containers) must be instantiated with two type arguments:[4]

```
template <template <typename,typename> class T, class U>
struct MList {
  template<typename E, typename A>
  struct Template {
        typedef T<E,A> C;
  };
```

[3] See Chapter 3 of *Modern C++ Design* to see why these typelists are so useful.

[4] These standard containers have a first parameter of the element type, and a second parameter of the allocator type, which has a default.

```
  typedef U Tail;
};
typedef struct {} NullMList;
```

(Note the need to wrap a template around the template template parameter in order to get access to it.)

We can manipulate these type-structures of templates just as we manipulate type structures of integers and types.

```
template <typename> struct MLength;

template <template <typename,typename> class T, class U>
struct MLength< MList<T,U> > {
  enum { value = 1 + MLength<U>::value };
};

template <>
struct MLength<NullMList> {
  enum { value = 0 };
};

template <class,int> struct MIndex;

template <template <typename,typename> class H, class T, int i>
struct MIndex< MList<H,T>,i > {
  typedef typename MIndex<T,i-1>::R R;
};

template <template <typename,typename> class H, class T>
struct MIndex< MList<H,T>,0 > {
  typedef MList<H,T> R;
};
```

Use of template lists can parallel that of type lists:

```
typedef MList3(list,vector,deque) A;
cout << MLength<A>::value << endl;
MIndex<A,1>::R::Template<char *,std::allocator<char *> >::C
  aContainer; // a vector of char *
```

We noted above the need to wrap a template around the template template parameter in order to access it as a nested type name of the list element. It would be preferable to access the name as an uninstantiated template name rather than a type name. For example, it would be convenient if we could simply use a nested “templatedef” similar to a typedef, or if the meaning of typedef could be extended to include templates. But we can’t…

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

```
struct MList {
  templatedef T Template; // illegal, templatedef doesn't exist
  typedef T Template; // illegal, T isn't a type
};
```

The Future

In future "Once, Weakly" installments, we'll look more deeply into the uses of type lists and template lists, and examine the possibilities of type-structures more complex than simple linear lists.