Introduction to Traits

Knowing the type of an object allows you to find out a lot about the object. For example, you can determine if an object has a particular data or function member, the size of the object, and so on. However, sometimes it's not enough to know just an object's type. Often, there is information *related to* the object's type that is essential to working with the object.

Conventional Personal Information

For example, for a container type, we may want to know the type of the elements that the container holds:

```
template <typename T>
class Seq {
    //…
};
```

At first, this may not seem to be a problem. The element type of `Seq<std::string>` is `std::string`, right? Not necessarily. There's nothing to prevent the implementation of our (non-standard) sequence container from making the element type `const T`, `T *`, or `CountedPtr<T>`. (A particularly weird container could simply ignore the template parameter and always set the element type to `void *`!) But vagary of implementation is not the only reason we may not be able to determine the element type of our container. We often write generic code in which that information is simply not available.

```
template <class Container>
Elem process( Container &c, int size ) {
    Temp temp = Elem();
    for( int i = 0; i < size; ++i )
        temp += c[i];
    return temp;
}
```

In the `process` generic algorithm above, we need to know the element type of `Container`, as well as a type that could serve to declare a temporary for holding objects of the element type, but that information is not available until the `process` function template is instantiated with a specific container.

A common way to handle this situation is to have a type provide "personal" information about itself. This information is often embedded in the type itself, rather like embedding a microchip in a person that can be queried for the person's name, identifying number, blood type, and so on.[1] We not interested in our sequence container's blood type, but we do want to know its element's type.

[1] This is an analogy, not a sign of approval for general employment of such a procedure. Good object-oriented designers employ analogy and metaphor to clarify their designs, but good human beings (no matter how they feel about embedded microchips) know the difference between a computer program and reality.

```
template <class T>
class Seq {
  public:
    typedef T Elem; // element type
    typedef T Temp; // temporary type
    typedef T *Ptr; // a pointer to an element
    size_t size() const;
    //…
};
```

This information can be queried at compile time.

```
typedef Seq<std::string> Strings;
//…
Strings::Elem aString;
Strings::Ptr pString = &aString;
```

More importantly, this approach allows us to write generic code that makes the *assumption* that the required information is present.

```
template <class Container>
typename Container::Elem process( Container &c, int size ) {
    typename Container::Temp temp = typename Container::Elem( );
    for( int i = 0; i < size; ++i )
        temp += c[i];
    return temp;
}
```

The `process` algorithm queries the `Container` type for its personal information, and makes the assumption that `Container` defines the nested type names `Elem` and `Temp`.[2]

```
Strings strings;
aString = process( strings, strings.size() );  // OK
std::vector<std::string> strings2;
aString = process( strings2, strings2.size() ); // error!
extern double readings[RSIZ];
double r = process( readings, RSIZ ); // error!
```

The `process` algorithm works well with our `Seq` container, but fails with a standard `vector` container, because `vector` does not define the nested type names that `process` assumes are present.

However, we are able to process any container that follows our convention.

```
template <typename T>
```

[2] Note that, due to the paucity of information about the `Container` type, we had to use the `typename` keyword to tell the compiler explicitly that the nested names were type names and not some other sort of nested name. This subject is covered in detail in the “Once, Weakly” of 11 March 2003.

```
class ReadonlySeq {
  public:
    typedef const T Elem;
    typedef T Temp;
    typedef AccessPtr<Elem> Ptr; // a smart pointer
    //…
};
```

We can `process` a `ReadonlySeq` container because it validates our assumptions, but we may also want to `process` containers that do not follow our rather parochial convention, and we may want to `process` container-like things that are not even classes.

Traits

A traits class is a collection of information about a type. Unlike our nested container information, however, the traits class is independent of the type it describes.

```
template <typename Cont>
struct ContainerTraits;
```

One common use of a traits class is to put a conventional layer between our generic algorithms and types that don't follow the algorithms' expected convention. We write the algorithm in terms of the type's traits. The general case will often assume some sort of convention. In this case, our `ContainerTraits` will assume the convention used by our `Seq` and `ReadonlySeq` containers.

```
template <typename Cont>
struct ContainerTraits {
    typedef typename Cont::Elem Elem;
    typedef typename Cont::Temp Temp;
    typedef typename Cont::Ptr Ptr;
};
```

With the addition of this traits class template, we have the choice of referring to the nested `Elem` type of one of our container types either through the container type or through the traits type instantiated with the container type.

```
typedef Seq<int> Cont;
Cont::Elem e1;
ContainerTraits<Cont>::Elem e2; // same type as e1
```

We can rewrite our generic algorithm to employ traits in place of direct access to the container's nested type names.

```
template <typename Container>
typename ContainerTraits<Container>::Elem
process( Container &c, int size ) {
    typename ContainerTraits<Container>::Temp temp
        = typename ContainerTraits<Container>::Elem();
    for( int i = 0; i < size; ++i )
```

```
            temp += c[i];
        return temp;
    }
```

It may seem that all we've managed to do is to make the syntax of the generic `process` algorithm even more impenetrable! Previously, to get the type of the container's element, we wrote `typename Container::Elem`. Put in plain language, we said, "Get `Container`'s nested name `Elem`. By the way, it's a type name." Now, we have to write `typename ContainerTraits<Container>::Elem`. Essentially, we say, "Instantiate the `ContainerTraits` class that corresponds to this container, and get its nested name `Elem`. By the way, it's a type name." We've essentially taken a step back from getting the information directly from the container type itself, and are going through the intermediary of the traits class. If accessing nested type information is like reading information about a person from an embedded microchip, using a traits class is like looking up someone's information in a database, using the person's name as a key. You'll get the same information, but the database lookup approach is certainly less invasive and more flexible.

For example, you can't get information from someone's microchip if he doesn't have one. Perhaps the person comes from a region where embedded microchips are not *de rigeur*. However, you can always create a new entry in a database for such a person without the necessity of even informing the individual concerned.[3] Similarly, we can specialize the traits template to provide information about a particular non-conforming container without affecting the container itself:

```
    class ForeignContainer {
        // no nested type information…
    };
    //…
    template <>
    struct ContainerTraits<ForeignContainer> {
        typedef int Elem;
        typedef Elem Temp;
        typedef Elem *Ptr;
    };
```

With this specialization of `ContainerTraits` available, we can `process` a `ForeignContainer` as effectively as one that is written to our convention. The original implementation of `process` would have failed on a `ForeignContainer` because it would have attempted to access nested information that did not exist.

```
    ForeignContainer::Elem x; // illegal!
    ContainerTraits<ForeignContainer>::Elem y; // OK
```

[3] Remember, these are analogies, not suggested procedures.

It's helpful to think of a traits template as a collection of information that is indexed by a type, much as an associative container is indexed by a key. But the "indexing" of traits happens at compile time, through template instantiation.

Extending Utility

Another advantage of accessing information about a type through a traits class is that the technique can be used to provide information about types that are not classes, and therefore can have no nested information. Even though traits classes are classes, the types whose traits they encapsulate don't have to be. For example, an array is a kind of degenerate container that we might like to manipulate as a container.

```
template <>
struct ContainerTraits<const char *> {
    typedef const char Elem;
    typedef char Temp;
    typedef const char *Ptr;
};
```

With this specialization in place for the "container" type `const char *`, we can `process` an array of characters.[4]

```
const char *name = "Arsene Lupin";
const char *r = process( name, strlen(name) );
```

We can continue in this fashion for other types of arrays, producing specializations for `int *`, `const double *`, and so on. However, it would be more convenient to specify a single case for any type of pointer, since they all will have similar properties. For this purpose, we employ partial specialization of the traits template for pointers.

```
template <typename T>
struct ContainerTraits<T *> {
    typedef T Elem;
    typedef T Temp;
    typedef T *Ptr;
};
```

Instantiating `ContainerTraits` with any pointer type, whether it be `int *` or `const float *(*const*)(int)` will result in instantiation of this partial specialization.

```
extern double readings[RSIZ];
double r = process( readings, RSIZ ); // works!
```

We're not quite there yet, however. Notice that using the partial specialization for a pointer to constant will not result in the correct type for use as a "temporary." That is, constant temporary values are not of much use because they cannot be assigned to, so what we'd like is to have the non-constant analog of the element type as the type of a

[4] Note that we did not say `process( "Arsene Lupin", sizeof("Arsene Lupin")-1 )`. See Vandevoorde & Jossutis, *C++ Templates*, p. 57.

temporary. In the case of `const char *`, for instance, `ContainerTraits<const char *>::Temp` should be `char`, not `const char`. We can handle this case with an additional partial specialization:

```
template <typename T>
struct ContainerTraits<const T *> {
    typedef const T Elem;
    typedef T Temp; // note: non-const analog of Elem
    typedef const T *Ptr;
};
```

This more specific partial specialization will be selected in preference to the previous one in those cases where the template argument is a pointer to constant, rather than a pointer to non-constant.

Partial specialization can also help us to extend our traits mechanism to convert a "foreign" convention to be in line with a local convention. For example, the STL is very heavy on convention, and the standard containers have concepts similar to those encapsulated in our `ContainerTraits`, but are expressed differently. For example, we earlier attempted to call the `process` algorithm with a standard `vector`, but failed. Let's fix that.

```
template <class T>
struct ContainerTraits< std::vector<T> > {
    typedef typename std::vector<T>::value_type Elem;
    typedef typename
        std::iterator_traits<typename std::vector<T>::iterator>
        ::value_type Temp;
    typedef typename
        std::iterator_traits<typename std::vector<T>::iterator>
        ::pointer Ptr;
};
```

It's not the most readable implementation one can imagine, but it's hidden, and our users can now invoke our generic algorithm with a container generated from a standard `vector`.

```
std::vector<std::string> strings2;
aString = process( strings2, strings2.size() ); // works!
```

More Traits

This discussion of traits has been heavy in its description of mechanism. In a future "Once, Weakly," we'll consider other aspects of implementing and using traits, including extended trait functionality and the use of traits in compile-time algorithm selection.