The following is an excerpt from Scott Meyers' new book, *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs.*

Item 47: Use traits classes for information about types.

The STL is primarily made up of templates for containers, iterators, and algorithms, but it also has a few utility templates. One of these is called advance. advance moves a specified iterator a specified distance:

```
template<typename IterT, typename DistT>        // move iter d units
void advance(IterT& iter, DistT d);             // forward; if d < 0,
                                                // move iter backward
```

Conceptually, advance just does iter += d, but advance can't be implemented that way, because only random access iterators support the += operation. Less powerful iterator types have to implement advance by iteratively applying ++ or -- d times.

Um, you don't remember your STL iterator categories? No problem, we'll do a mini-review. There are five categories of iterators, corresponding to the operations they support. *Input iterators* can move only forward, can move only one step at a time, can only read what they point to, and can read what they're pointing to only once. They're modeled on the read pointer into an input file; the C++ library's istream_iterators are representative of this category. *Output iterators* are analogous, but for output: they move only forward, move only one step at a time, can only write what they point to, and can write it only once. They're modeled on the write pointer into an output file; ostream_iterators epitomize this category. These are the two least powerful iterator categories. Because input and output iterators can move only forward and can read or write what they point to at most once, they are suitable only for one-pass algorithms.

A more powerful iterator category consists of *forward iterators*. Such iterators can do everything input and output iterators can do, plus they can read or write what they point to more than once. This makes them viable for multi-pass algorithms. The STL offers no singly linked list, but some libraries offer one (usually called slist), and iterators into such containers are forward iterators. Iterators into TR1's hashed containers (see Item 54) may also be in the forward category.

Bidirectional iterators add to forward iterators the ability to move backward as well as forward. Iterators for the STL's list are in this category, as are iterators for set, multiset, map, and multimap.

The most powerful iterator category is that of *random access iterators*. These kinds of iterators add to bidirectional iterators the ability to perform "iterator arithmetic," i.e., to jump forward or backward an arbitrary distance in constant time. Such arithmetic is analogous to pointer arithmetic, which is not surprising, because random access iterators are modeled on built-in pointers, and built-in pointers can act as random access iterators. Iterators for vector, deque, and string are random access iterators.

For each of the five iterator categories, C++ has a "tag struct" in the standard library that serves to identify it:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag: public input_iterator_tag {};
struct bidirectional_iterator_tag: public forward_iterator_tag {};
struct random_access_iterator_tag: public bidirectional_iterator_tag {};
```

The inheritance relationships among these structs are valid is-a relationships (see Item 32): it's true that all forward iterators are also input iterators, etc. We'll see the utility of this inheritance shortly.

But back to advance. Given the different iterator capabilities, one way to implement advance would be to use the lowest-common-denominator strategy of a loop that iteratively increments or decrements the iterator. However, that approach would take linear time. Random access iterators support constant-time iterator arithmetic, and we'd like to take advantage of that ability when it's present.

What we really want to do is implement advance essentially like this:

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (iter is a random access iterator) {
    iter += d;                              // use iterator arithmetic
  }                                         // for random access iters
  else {
    if (d >= 0) { while (d--) ++iter; }     // use iterative calls to
    else { while (d++) --iter; }            // ++ or -- for other
  }                                         // iterator categories
}
```

This requires being able to determine whether iter is a random access iterator, which in turn requires knowing whether its type, IterT, is a random access iterator type. In other words, we need to get some information about a type. That's what *traits* let you do: they allow you to get information about a type during compilation.

Traits aren't a keyword or a predefined construct in C++; they're a technique and a convention followed by C++ programmers. One of the demands made on the technique is that it has to work as well for built-in types as it does for user-defined types. For example, if advance is called with a pointer (like a const char*) and an int, advance has to work, but that means that the traits technique must apply to built-in types like pointers.

The fact that traits must work with built-in types means that things like nesting information inside types won't do, because there's no way to nest information inside pointers. The traits information for a type, then, must be external to the type. The standard technique is to put it into a template and one or more specializations of that template. For iterators, the template in the standard library is named iterator_traits:

```
template<typename IterT>           // template for information about
struct iterator_traits;            // iterator types
```

As you can see, iterator_traits is a struct. By convention, traits are always implemented as structs. Another convention is that the structs used to implement traits are known as — I am not making this up — traits *classes*.

The way iterator_traits works is that for each type IterT, a typedef named iterator_category is declared in the struct iterator_traits<IterT>. This typedef identifies the iterator category of IterT.

iterator_traits implements this in two parts. First, it imposes the requirement that any user-defined iterator type must contain a nested typedef named iterator_category that identifies the appropriate tag struct. deque's iterators are random access, for example, so a class for deque iterators would look something like this:

```
template < ... >                   // template params elided
class deque {
public:
  class iterator {
  public:
    typedef random_access_iterator_tag iterator_category;
    ...
  }:
  ...
};
```

list's iterators are bidirectional, however, so they'd do things this way:

```
template < ... >
class list {
public:
  class iterator {
  public:
    typedef bidirectional_iterator_tag iterator_category;
    ...
  }:
  ...
};
```

iterator_traits just parrots back the iterator class's nested typedef:

```
// the iterator_category for type IterT is whatever IterT says it is;
// see Item 42 for info on the use of "typedef typename"
template<typename IterT>
struct iterator_traits {
  typedef typename IterT::iterator_category iterator_category;
  ...
};
```

This works well for user-defined types, but it doesn't work at all for iterators that are pointers, because there's no such thing as a pointer with a nested typedef. The second part of the iterator_traits implementation handles iterators that are pointers.

To support such iterators, iterator_traits offers a *partial template specialization* for pointer types. Pointers act as random access iterators, so that's the category iterator_traits specifies for them:

```
template<typename IterT>                    // partial template specialization
struct iterator_traits<IterT*>              // for built-in pointer types
{
  typedef random_access_iterator_tag iterator_category;
  ...
};
```

At this point, you know how to design and implement a traits class:

- Identify some information about types you'd like to make available (e.g., for iterators, their iterator category).
- Choose a name to identify that information (e.g., iterator_category).
- Provide a template and set of specializations (e.g., iterator_traits) that contain the information for the types you want to support.

Given iterator_traits — actually std::iterator_traits, since it's part of C++'s standard library — we can refine our pseudocode for advance:

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (typeid(typename std::iterator_traits<IterT>::iterator_category) ==
    typeid(std::random_access_iterator_tag))
  ...
}
```

Although this looks promising, it's not what we want. For one thing, it will lead to compilation problems, but we'll explore that in Item 48; right now, there's a more fundamental issue to consider. IterT's type is known during compilation, so iterator_traits<IterT>::iterator_category can also be determined during compilation. Yet the if statement is evaluated at runtime. Why do something at runtime that we can do during compilation? It wastes time (literally), and it bloats our executable.

What we really want is a conditional construct (i.e., an if...else statement) for types that is evaluated during compilation. As it happens, C++ already has a way to get that behavior. It's called overloading.

When you overload some function f, you specify different parameter types for the different overloads. When you call f, compilers pick the best overload, based on the arguments you're passing. Compilers essentially say, "If this overload is the best match for what's being passed, call this f; if this other overload is the best match, call it; if this third one is best, call it," etc. See? A compile-time conditional construct for types. To get advance to behave the way we want, all we have to do is create two versions of an overloaded function containing the "guts" of advance, declaring each to take a different type of iterator_category object. I use the name doAdvance for these functions:

```
template<typename IterT, typename DistT>          // use this impl for
void doAdvance(IterT& iter, DistT d,              // random access
               std::random_access_iterator_tag)   // iterators
{
  iter += d;
}

template<typename IterT, typename DistT>          // use this impl for
void doAdvance(IterT& iter, DistT d,              // bidirectional
               std::bidirectional_iterator_tag)   // iterators
{
  if (d >= 0) { while (d--) ++iter; }
  else { while (d++) --iter; }
}

template<typename IterT, typename DistT>          // use this impl for
void doAdvance(IterT& iter, DistT d,              // input iterators
               std::input_iterator_tag)
{
  if (d < 0 ) {
    throw std::out_of_range("Negative distance");  // see below
  }
  while (d--) ++iter;
}
```

Because forward_iterator_tag inherits from input_iterator_tag, the version of doAdvance for input_iterator_tag will also handle forward iterators. That's the motivation for inheritance among the various iterator_tag structs. (In fact, it's part of the motivation for *all* public inheritance: to be able to write code for base class types that also works for derived class types.)

The specification for advance allows both positive and negative distances for random access and bidirectional iterators, but behavior is undefined if you try to move a forward or input iterator a negative distance. The implementations I checked simply assumed that d was non-negative, thus entering a *very* long loop counting "down" to zero if a negative distance was passed in. In the code above, I've shown an exception being thrown instead. Both implementations are valid. That's the curse of undefined behavior: you *can't predict* what will happen.

Given the various overloads for doAdvance, all advance needs to do is call them, passing an extra object of the appropriate iterator category type so that the compiler will use overloading resolution to call the proper implementation:

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  doAdvance(                                              // call the version
    iter, d,                                              // of doAdvance
    typename                                              // that is
      std::iterator_traits<IterT>::iterator_category()    // appropriate for
  );                                                      // iter's iterator
}                                                         // category
```

We can now summarize how to use a traits class:

- Create a set of overloaded "worker" functions or function templates (e.g., doAdvance) that differ in a traits parameter. Implement each function in accord with the traits information passed.
- Create a "master" function or function template (e.g., advance) that calls the workers, passing information provided by a traits class.

Traits are widely used in the standard library. There's iterator_traits, of course, which, in addition to iterator_category, offers four other pieces of information about iterators (the most useful of which is value_type — Item 42 shows an example of its use). There's also char_traits, which holds information about character types, and numeric_limits, which serves up information about numeric types, e.g., their minimum and maximum representable values, etc. (The name numeric_limits is a bit of a surprise, because the more common convention is for traits classes to end with "traits," but numeric_limits is what it's called, so numeric_limits is the name we use.)

TR1 (see Item 54) introduces a slew of new traits classes that give information about types, including is_fundamental<T> (whether T is a built-in type), is_array<T> (whether T is an array type), and is_base_of<T1, T2> (whether T1 is the same as or is a base class of T2). All told, TR1 adds over 50 traits classes to standard C++.

Things to Remember

✦ Traits classes make information about types available during compilation. They're implemented using templates and template specializations.

✦ In conjunction with overloading, traits classes make it possible to perform compile-time if...else tests on types.