Item 3: Predicates, Part 2: Matters of State

ITEM 3: PREDICATES, PART 2: MATTERS OF STATE **DIFFICULTY: 7**

Following up from the introduction given in Item 2, we now examine "stateful" predicates. What are they? When are they useful? How compatible are they with standard containers and algorithms?

1. What are predicates, and how are they used in STL? Give an example.
2. When would a "stateful" predicate be useful? Give examples.
3. What requirements on algorithms are necessary in order to make stateful predicates work correctly?

SOLUTION

Unary and Binary Predicates

1. What are predicates, and how are they used in STL?

A predicate is a pointer to a function, or a function object (an object that supplies the function call operator, `operator()()`), that gives a yes/no answer to a question about an object. Many algorithms use a predicate to make a decision about each element they operate on, so a predicate `pred` should work correctly when used as follows:

ITEM1_11new.fm Page 2 Tuesday, November 27, 2001 12:41 PM

```
// Example 3-1(a): Using a unary predicate
//
if( pred( *first ) )
{
  /* ... */
}
```

As you can see from this example, `pred` should return a value that can be tested as `true`. Note that a predicate is allowed to use `const` functions only through the dereferenced iterator.

Some predicates are binary—that is, they take two objects (often dereferenced iterators) as arguments. This means that a binary predicate `bpred` should work correctly when used as follows:

```
// Example 3-1(b): Using a binary predicate
//
if( bpred( *first1, *first2 ) )
{
  /* ... */
}
```

Give an example.

Consider the following implementation of the standard algorithm `find_if()`:

```
// Example 3-1(c): A sample find_if()
//
template<typename Iter, typename Pred> inline
Iter find_if( Iter first, Iter last, Pred pred )
{
  while( first != last && !pred(*first) )
  {
    ++first;
  }
  return first;
}
```

This implementation of the algorithm visits every element in the range `[first, last)` in order, applying the predicate function pointer, or object `pred`, to each element. If there is an element for which the predicate evaluates to `true`, `find_if()` returns an iterator pointing to the first such element. Otherwise, `find_if()` returns `last` to signal that an element satisfying the predicate was not found.

We can use `find_if()` with a function pointer predicate as follows:

```
// Example 3-1(d):
// Using find_if() with a function pointer.
//
bool GreaterThanFive( int i )
{
```

ITEM1_11new.fm Page 3 Tuesday, November 27, 2001 12:41 PM

```
  return i > 5;
}

bool IsAnyElementGreaterThanFive( vector<int>& v )
{
  return find_if( v.begin(), v.end(), GreaterThanFive )
         != v.end();
}
```

Here's the same example, only using `find_if()` with a function object instead of a free function:

```
// Example 3-1(e):
// Using find_if() with a function object.
//
class GreaterThanFive
  : public std::unary_function<int, bool>
{
public:
  bool operator()( int i ) const
  {
    return i > 5;
  }
};

bool IsAnyElementGreaterThanFive( vector<int>& v )
{
  return find_if( v.begin(), v.end(), GreaterThanFive() )
         != v.end();
}
```

In this example, there's not much benefit to using a function object over a free function, is there? But this leads us nicely into our other questions, in which the function object shows much greater flexibility.

2. When would a "stateful" predicate be useful? Give examples.

Continuing on from Examples 3-1(d) and 3-1(e), here's something a free function can't do as easily without using something like a static variable:

```
// Example 3-2(a):
// Using find_if() with a more general function object.
//
class GreaterThan
  : public std::unary_function<int, bool>
{
public:
  GreaterThan( int value ) : value_( value ) { }
  bool operator()( int i ) const
  {
```

ITEM1_11new.fm Page 4 Tuesday, November 27, 2001 12:41 PM

```
    return i > value_;
  }
private:
  const int value_;
};

bool IsAnyElementGreaterThanFive( vector<int>& v )
{
  return find_if( v.begin(), v.end(), GreaterThan(5) )
         != v.end();
}
```

This GreaterThan predicate has member data that remembers a value, in this case the value it should compare each element against. You can already see that this version is much more usable—and reusable—than the special-purpose code in Examples 3-1(d) and 3-1(e), and a lot of the power comes from the ability to store local information inside the object like this.

Taking it one step further, we end up with something even more generalized:

```
// Example 3-2(b):
// Using find_if() with a fully general function object.
//
template<typename T>
class GreaterThan
  : public std::unary_function<T, bool>
{
public:
  GreaterThan( T value ) : value_( value ) { }

  bool operator()( const T& t ) const
  {
    return t > value_;
  }

private:
  const T value_;
};

bool IsAnyElementGreaterThanFive( vector<int>& v )
{
  return find_if( v.begin(), v.end(), GreaterThan<int>(5) )
         != v.end();
}
```

So we can see some usability benefits from using predicates that store value.

ITEM1_11new.fm Page 5 Tuesday, November 27, 2001 12:41 PM

The Next Step: Stateful Predicates

The predicates in both Examples 3-2(a) and 3-2(b) have an important property: Copies are equivalent. That is, if you make a copy of a `GreaterThan<int>` object, it behaves in all respects just like the original one and can be used interchangeably. This turns out to be important, as we shall see in Question #3.

Some people have tried to write *stateful* predicates that go further, by changing as they're used—that is, the result of applying a predicate depends on its history of previous applications. In Examples 3-2(a) and 3-2(b), the objects did carry some internal values, but these were fixed at construction time; they were not state that could change during the lifetime of the object. When we talk about stateful predicates, we mean primarily predicates having state *that can change* so that the predicate object is sensitive to what's happened to it over time, like a little state machine.[1]

Examples of such stateful predicates appear in books. In particular, people have tried to write predicates that keep track of various information about the elements they were applied to. For example, people have proposed predicates that remember the values of the objects they were applied to in order to perform calculations (for example, a predicate that returns true as long as the average of the values it was applied to so far is more than 50, or the total is less than 100, and so on). We just saw a specific example of this kind of stateful predicate in Item 2, Question #3:

```
// Example 3-2(c)
// (From Item 2, Example 2-3(b))
//
// Method 2: Write a function object which returns
// true the nth time it's applied, and use
// that as a predicate for remove_if.
//
class FlagNth
{
public:
  FlagNth( size_t n ) : current_(0), n_(n) { }

  template<typename T>
  bool operator()( const T& ) { return ++current_ == n_; }

private:
  size_t       current_;
  const size_t n_;
};
```

Stateful predicates like the above are sensitive to the way they are applied to elements in the range that is operated on. This one in particular depends on both the num-

1. As John D. Hickin so elegantly describes it: "The input `[first, last)` is somewhat like the tape fed to a Turing machine and the stateful predicate is like a program."

ITEM1_11new.fm Page 6 Tuesday, November 27, 2001 12:41 PM

ber of times it has been applied and on the order in which it is applied to the elements in the range (if used in conjunction with something like `remove_if()`, for example).

The major difference between predicates that are stateful and those that aren't is that, for stateful predicates, copies are *not* equivalent. Clearly an algorithm couldn't make a copy of a `FlagNth` object and apply one object to some elements and the other object to other elements. That wouldn't give the expected results at all, because the two predicate objects would update their counts independently and neither would be able to flag the correct *n*-th element; each could flag only the *n*-th element it itself happened to be applied to.

The problem is that, in Example 3-2(c), Method 2 possibly tried to use a `FlagNth` object in just such a way:

```
// Example invocation

... remove_if( v.begin(), v.end(), FlagNth(3) ) ...
```

"Looks reasonable, and I've used this technique," some may say. "I just read a C++ book that demonstrates this technique, so it must be fine," some may say. Well, the truth is that this technique may happen to work on your implementation (or on the implementation that the author of the book with the error in it was using), but it is *not* guaranteed to work portably on all implementations, or even on the next version of the implementation you are (or that author is) using now.

Let's see why, by examining `remove_if()` in a little more detail in Question #3:

3. What requirements on algorithms are necessary in order to make stateful predicates work correctly?

For stateful predicates to be really useful with an algorithm, the algorithm must generally guarantee two things about how it uses the predicate:

a) the algorithm must not make copies of the predicate (that is, it should consistently use the same object that it was given), and

b) the algorithm must apply the predicate to the elements in the range in some known order (usually, first to last).

Alas, the standard does not require that the standard algorithms meet these two guarantees. Even though stateful predicates have appeared in books, in a battle between the standard and a book, the standard wins. The standard does mandate other things for standard algorithms, such as the performance complexity and the number of times a predicate is applied, but in particular it never specifies requirement (a) for any algorithm.

For example, consider `std::remove_if()`:

a) It's common for standard library implementations to implement `remove_if()` in terms of `find_if()`, and pass the predicate along to `find_if()` by value. This will make the predicate behave unexpectedly, because the predicate object

ITEM1_11new.fm Page 7 Tuesday, November 27, 2001 12:41 PM

actually passed to `remove_if()` is not necessarily applied once to every element in the range. Rather, the predicate object *or a copy of the predicate object* is what is guaranteed to be applied once to every element. This is because a conforming `remove_if()` is allowed to assume that copies of the predicate are equivalent.

b) The standard requires that the predicate supplied to `remove_if()` be applied exactly `last - first` times, but it doesn't say in what order. It's possible, albeit a little obnoxious, to write a conforming implementation of `remove_if()` that doesn't apply the predicate to the elements in order. The point is that if it's not required by the standard, you can't portably depend on it.

"Well," you ask, "isn't there any way to make stateful predicates such as `FlagNth` work reliably with the standard algorithms?" Unfortunately, the answer is no.

All right, all right, I can already hear the howls of outrage from the folks who write predicates that use reference-counting techniques to solve the predicate-copying problem (a) above. Yes, you can share the predicate state so that a predicate can be safely copied without changing its semantics when it is applied to objects. The following code uses this technique (for a suitable `CountedPtr` template; follow-up question: provide a suitable implementation of `CountedPtr`):

```
// Example 3-3(a): A (partial) solution
// that shares state between copies.
//
class FlagNthImpl
{
public:
  FlagNthImpl( size_t nn ) : i(0), n(nn) { }
  size_t       i;
  const size_t n;
};

class FlagNth
{
public:
  FlagNth( size_t n )
    : pimpl_( new FlagNthImpl( n ) )
  {
  }

  template<typename T>
  bool operator()( const T& )
  {
    return ++(pimpl_->i) == pimpl_->n;
  }

private:
  CountedPtr<FlagNthImpl> pimpl_;
};
```

ITEM1_11new.fm Page 8 Tuesday, November 27, 2001 12:41 PM

But this doesn't, and can't, solve the ordering problem (b) above. That is, you, the programmer, are entirely dependent on the order in which the predicate is applied by the algorithm. There's no way around this, not even with fancy pointer techniques, unless the algorithm itself guarantees a traversal order.

Follow-Up Question

The follow-up question, above, was to provide a suitable implementation of `CountedPtr`, a smart pointer for which making a new copy just points at the same representation, and the last copy to be destroyed cleans up the allocated object. Here is one that works, although it could be beefed up further for production use:

```
template<typename T>
class CountedPtr
{
private:
  class Impl
  {
  public:
    Impl( T* pp ) : p( pp ), refs( 1 ) { }

    ~Impl() { delete p; }

    T*     p;
    size_t refs;
  };
  Impl* impl_;

public:
  explicit CountedPtr( T* p )
    : impl_( new Impl( p ) ) { }

  ~CountedPtr() { Decrement(); }

  CountedPtr( const CountedPtr& other )
    : impl_( other.impl_ )
  {
    Increment();
  }

  CountedPtr& operator=( const CountedPtr& other )
  {
    if( impl_ != other.impl_ )
    {
      Decrement();
      impl_ = other.impl_;
      Increment();
    }
```

ITEM1_11new.fm Page 9 Tuesday, November 27, 2001 12:41 PM

```
    return *this;
  }

  T* operator->() const
  {
    return impl_->p;
  }

  T& operator*() const
  {
    return *(impl_->p);
  }

private:
  void Decrement()
  {
    if( --(impl_->refs) == 0 )
    {
      delete impl_;
    }
  }

  void Increment()
  {
    ++(impl_->refs);
  }
};
```