Item 2: Predicates, Part 1: What remove() Removes

.

ITEM 2: PREDICATES, PART 1: WHAT remove() REMOVES

DIFFICULTY: 4

This Item lets you test your standard algorithm skills. What does the standard library algorithm `remove()` actually do, and how would you go about writing a generic function to remove only the third element in a container?

1. What does the `std::remove()` algorithm do? Be specific.
2. Write code that eliminates all values equal to `3` from a `std::vector<int>`.
3. A programmer working on your team wrote the following alternative pieces of code to remove the *n*-th element of a container.

```
// Method 1: Write a special-purpose
// remove_nth algorithm.
//
template<typename FwdIter>
FwdIter remove_nth( FwdIter first, FwdIter last, size_t n )
{
  /* ... */
}
// 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_;
};

// Example invocation
... remove_if( v.begin(), v.end(), FlagNth(3) ) ...
```

a) Implement the missing part of Method 1.

b) Which method is better? Why? Discuss anything that might be problematic about either solution.

SOLUTION

What remove() Removes

1. What does the std::remove() algorithm do? Be specific.

The standard algorithm remove() does not physically remove objects from a container; the size of the container is unchanged after remove() has done its thing. Rather, remove() shuffles up the "unremoved" objects to fill in the gaps left by removed objects, leaving at the end one "dead" object for each removed object. Finally, remove() returns an iterator pointing at the first "dead" object, or, if no objects were removed, remove() returns the end() iterator.

For example, consider a vector<int> v that contains the following nine elements:

```
1 2 3 1 2 3 1 2 3
```

Say that you used the following code to try to remove all 3's from the container:

```
// Example 2-1
//
remove( v.begin(), v.end(), 3 );  // subtly wrong
```

What would happen? The answer is something like this:

```
1 2 1 2 1 2 ? ? ?

unremoved   "dead"
            objects
            ↑
            iterator returned by
            remove() points to
            the third-last object
            (because three were
            removed)
```

Three objects had to be removed, and the rest were copied to fill in the gaps. The objects at the end of the container may have their original values (1 2 3), or they may not; don't rely on that. Again, note that the size of the container is left unchanged.

If you're wondering why `remove()` works that way, the most basic reason is that `remove()` doesn't operate on a container, but rather on a range of iterators, and there's no such iterator operation as "remove the element this iterator points to from whatever container it's in." To do that, we have to actually get at the container directly. For further information about `remove()`, see also Andrew Koenig's thorough treatment of this topic in [Koenig99].

2. Write code that removes all values equal to 3 from a `std::vector<int>`.

Here's a one-liner to do it, where `v` is a `vector<int>`:

```
// Example 2-2: Removing 3's from a vector<int> v
//
v.erase( remove( v.begin(), v.end(), 3 ), v.end() );
```

The call to `remove( v.begin(), v.end(), 3 )` does the actual work, and returns an iterator pointing to the first "dead" element. The call to `erase()` from that point until `v.end()` gets rid of the dead elements so that the `vector` contains only the unremoved objects.

3. A programmer working on your team wrote the following alternative pieces of code to remove the *n*-th element of a container.

```
// Example 2-3(a)
//
// Method 1: Write a special-purpose
// remove_nth algorithm.
//
```

```
template<typename FwdIter>
FwdIter remove_nth( FwdIter first, FwdIter last, size_t n )
{
  /* ... */
}

// 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_;
};

// Example invocation
... remove_if( v.begin(), v.end(), FlagNth(3) ) ...
```

a) Implement the missing part of Method 1.

People often propose implementations that have the same bug as the following code. Did you?

```
// Example 2-3(c): Can you see the problem(s)?
//
template<typename FwdIter>
FwdIter remove_nth( FwdIter first, FwdIter last, size_t n )
{
  for( ; n > 0; ++first, --n )
    ;
  if( first != last )
  {
    FwdIter dest = first;
    return copy( ++first, last, dest );
  }
  return last;
}
```

There is one problem in Example 2-3(c), and that one problem has two aspects:

1. Correct preconditions: We don't require that `n <= distance( first, last )`, so the initial loop may move `first` past `last`, and then `[first,last)` is no longer a valid iterator range. If so, then in the remainder of the function, Bad Things will happen.
2. Efficiency: Let's say we decided to document (and test!) a precondition that `n` be valid for the given range, as a way of addressing problem #1. Then we should still dispense with the iterator-advancing loop entirely and simply write `advance( first, n )`. The standard `advance()` algorithm for iterators is already aware of iterator categories, and is automatically optimized for random-access iterators. In particular, it will take constant time for random-access iterators, instead of the linear time required for other iterators.

Here is a reasonable implementation:

```
// Example 2-3(d): Solving the problems
//
// Precondition:
//  - n must not exceed the size of the range
//
template<typename FwdIter>
FwdIter remove_nth( FwdIter first, FwdIter last, size_t n )
{
  // Check precondition. Incurs overhead in debug mode only.
  assert( distance( first, last ) >= n );

  // The real work.
  advance( first, n );
  if( first != last )
  {
    FwdIter dest = first;
    return copy( ++first, last, dest );
  }
  return last;
}
```

b) Which method is better? Why? Discuss anything that might be problematic about either solution.

Method 1 has two main advantages:

1. It is correct.
2. It can take advantage of iterator traits, specifically the iterator category, and so can perform better for random-access iterators.

Method 2 has corresponding disadvantages, which we'll analyze in detail in the second part of this miniseries.

.