CHAPTER 42

Filtered Iteration

If you think you've arrived, you're ready to be shown the door.

—Steve Forbes

I can give you my word, but I know what it's worth and you don't.

—Nero Wolfe

42.1 Introduction

We saw in Chapter 36 that transforming an iterator is a matter of applying a unary function to the dereference. Can we do this with a predicate, to filter out items? We might imagine something like the following:

```
using recls::stl::search_sequence;
search_sequence  files(".", "*", recls::FILES | recls::RECURSIVE);

std::copy(filter(files.begin(), is_readonly())
    , filter(files.end(), is_readonly())
    , std::ostream_iterator<search_sequence::value_type>(std::cout
                                                        , "\n"));
```

42.2 An Invalid Version

How would this work? Naturally, `filter()` will be a creator function that returns an instance of a (suitably specialized) filtering iterator type. We might imagine an iterator class template such as that shown in Listing 42.1.

Listing 42.1 Invalid Version of `filter_iterator`

```
template< typename I // The adapted iterator
        , typename P // Unary predicate that will select items
        , typename T = adapted_iterator_traits<I>
        >
class filter_iterator
{
public: // Member Types
  typedef I                                    base_iterator_type;
  typedef P                                    filter_predicate_type;
  typedef T                                    traits_type;
```

```
  typedef filter_iterator<I, P, T>                class_type;
  typedef typename traits_type::iterator_category iterator_category;
  typedef typename traits_type::value_type        value_type;
  . . . // And so on, for usual members (from adapted_iterator_traits)
public: // Construction
  filter_iterator(I it, P pr)
    : m_it(it)
    , m_pr(pr)
  {
    for(; !m_pr(*m_it); ++m_it) // Get first "selected" position
    {}
  }
public: // Forward Iterator Methods
  class_type& operator ++()
  {
    for(++m_it; !m_pr(*m_it); ++m_it) // Advance, then get next pos
    {}
    return *this;
  }
  class_type operator ++(int);        // Usual implementation
  reference operator *();             // Usual implementation
  const_reference operator *() const; // Usual implementation
private: // Member Variables
  I m_it;
  P m_pr;
};
```

Alas, the statement outputting read-only files shown in Section 42.1 will fail, probably in a crash. In fact, just about any use of this iterator will fail. There are two problems.

First, in the constructor for the first iterator, the active iterator, it uses the predicate and increment operator to ensure that the `filter_iterator` instance has the correct position before it is used. This correct position is the first one that matches the predicate, and that may be outside the given range [`files.begin()`, `files.end()`).

Second, the constructor for the second iterator, the one that adapts the endpoint iterator, dereferences its base iterator instance. It's a strict part of the STL *iterator* concept (Section 1.3) that we can "never [assume] that past-the-end values are dereferenceable" (C++-03: 24.1;5). (This also means that the implementation of `operator *()` is not well defined, but that's moot because we would have to go through an undefined constructor to get to a point where it could be invoked.)

42.3 Member Iterators Define the Range

It is clear that a filtering iterator instance must have access to a pair of iterators in order to avoid going outside the valid range. That being the case, our client code will be more verbose, for example:

```
search_sequence  files(".", "*", recls::FILES | recls::RECURSIVE);

std::copy(filter(files.begin(), files.end(), is_readonly())
    , filter(files.end(), files.end(), is_readonly())
    , std::ostream_iterator<search_sequence::value_type>(std::cout
                                                         , "\n"));
```

42.4 So . . . ?

One option might be to default the second endpoint iterator to `I()`, as follows:

```
template <. . .>
class filter_iterator
{
  . . .
public: // Construction
  filter_iterator(I it, P pr, I end = I())
    : m_it(it)
    , m_end(end)
    , m_pr(pr)
  {
    . . .
private: // Member Variables
  I m_it;
  I m_end;
  P m_pr;
};
```

However, this relies on the iterator type defining a default-constructed iterator as being equivalent to the endpoint iterator. Though this would work, on a case-by-case basis, for some iterators, including `readdir_sequence::const_iterator` (Section 19.3) and `findfile_sequence::const_iterator` (Section 20.5), it would not work for others, such as `glob_sequence::const_iterator` (Section 17.3). Or, if you prefer, it *might* work for `std::list`, `std::deque`, `std::map`, but it can't work for `std::vector` and, importantly, pointers.

Furthermore, providing this facility puts the onus on the user to know whether the assumption holds for a given iterator, which is both unreasonable and exceedingly likely to lead to failures. More leaking abstractions! Add the fact that such failures may never exhibit in testing, instead lurking until your product is out in the field, and this option is totally unacceptable.

"*Wait!*" you might say doggedly, "We can specialize the creator function to reject pointers." And so we can. However, there are plenty of iterators that are not pointers that do not satisfy the default-constructor/endpoint equivalence. For one, a random access iterator that is not a pointer will not do so.

Or you might wonder, "Can't we specialize to reject random access iterators?" Indeed, that would help, were it not for the fact that many of the iterators fulfilling other categories will also fail. In short, there's no getting away from the following rule and tip.

Rule: Never assume that a default-constructed instance of an iterator is equivalent to the endpoint iterator for the sequence or notional range for which the iterator acts.

Tip: Never use a filtering iterator adaptor that assumes, or allows the user to assume, that a default-constructed instance of the adapted iterator type is equivalent to the endpoint iterator.

With this in mind, let's see how to define a robust filtering iterator component.

42.5 stlsoft::filter_iterator

There's a fair bit to do in this class, so we'll tackle iterator refinements in a stepwise fashion. We'll start with input and forward iterators.

42.5.1 Forward Iterator Semantics

The handling of forward iterator semantics is shown in Listing 42.2.

Listing 42.2 Definition of `filter_iterator` Supporting Forward Iteration

```
template< typename I // The underlying iterator
        , typename P // The unary predicate that will select the items
        , typename T = adapted_iterator_traits<I>
        >
class filter_iterator
{
public: // Member Types
  . . . // All usual member types, most via T (adapted_iterator_traits)
public: // Construction
  filter_iterator(I begin, I end, P pr)
    : m_it(begin)
    , m_end(end)
    , m_pr(pr)
  {
    for(; m_it != m_end; ++m_it)
    {
      if(m_pr(*m_it))
      {
        break;
      }
    }
  }
public: // Forward Iterator Methods
  class_type& operator ++()
  {
    STLSOFT_MESSAGE_ASSERT( "Attempting to increment an endpoint
iterator", m_it != m_end);
```

```
    for(++m_it; m_it != m_end; ++m_it)
    {
      if(m_pr(*m_it))
      {
        break;
      }
    }
    return *this;
  }
  class_type& operator ++(int); // Canonical implementation
  effective_reference operator *()
  {
    return *m_it;
  }
  effective_const_reference operator *() const; // Same as operator *()
  effective_pointer operator ->()
  {
    enum { is_iterator_pointer_type
                        = is_pointer_type<base_iterator_type>::value };
    typedef typename
     value_to_yesno_type<is_iterator_pointer_type>::type  yesno_t;
    return invoke_member_selection_operator_(yesno_t());
  }
  effective_const_pointer operator ->() const; // Same as operator ->()
  . . .
private: // Member Variables
  I m_it;
  I m_end;
  P m_pr;
};
```

All member types are defined in terms of those provided by `adapted_iterator_traits` (just as is the case with `index_iterator`, described on the CD). The constructor takes the [`begin`, `end`) iterator pair defining the iterable range, followed by the predicate used for filtering. Note that the predicate comes last, as a reminder that defaulting the endpoint iterator is a crazy thing to attempt.

The constructor has to assume that the given base iterator instance specifying the start of the iterable range may not be one acceptable to the filter predicate and so tests it, possibly incrementing until finding one that is. Contrast this with the implementation of `operator ++()`, which knows that the current iteration point is acceptable to the filter and increments before it starts the loop. This is because the user *must* have previously tested it against the known endpoint filtered instance. This follows the basic idiom in STL that an iterator is determined to be viable by testing its equality against one that is known to not be.

The necessity to move to an acceptable point in the iteration implies the curious relationship whereby different start point iterators evaluate, in their filtered form, to be the same. Consider the

sequence of integers 0, 2, 4, 5, 6, 7, 8, 9. Using a filter, `is_odd`, which selects odd numbers, there are several ways to specify equivalent iterators:

```
int ints[] = { 0, 2, 4, 5, 6, 7, 8, 9 };

stlsoft::filter(&ints[0], &ints[0] + 8, is_odd());  // Is equivalent to:
stlsoft::filter(&ints[1], &ints[0] + 8, is_odd());  //  this
stlsoft::filter(&ints[2], &ints[0] + 8, is_odd());  //  and this
stlsoft::filter(&ints[3], &ints[0] + 8, is_odd());  //  and this
```

Each of the iterators in that case actually refers to the element at index 3, whose value is 5, since that's the first one in the series that has an odd value.

The remainder of the implementation shown is entirely normal, given what we learned in Section 36.4.5 about handling the member selection operator. So far, so good.

42.5.2 Bidirectional Iterator Semantics

I expect you're ahead of me here. Given the current member variables, we cannot implement bidirectional iterator semantics because we stand the same risk of stepping outside the iterable range as discussed in Section 42.2, only this time we would step out of the beginning rather than the end. The remedy in this case is to remember the starting point. Hence, we add another member variable of the base iterator type, `m_begin`, and adjust the implementation of the constructor accordingly, as shown in Listing 42.3.

Listing 42.3 Member Variables Supporting Bidirectional Iteration

```
template <typename I, typename P, typename T>
class filter_iterator
{
  . . .
public: // Construction
  filter_iterator(I begin, I end, P pr)
    : m_it(begin)
    , m_begin(begin)
    , m_end(end)
    , m_pr(pr)
  {
    . . .
  }
  . . .
private: // Member Variables
  I m_it;
  I m_begin;
  I m_end;
  P m_pr;
};
```

Using this member, the implementation of the bidirectional iterator methods is surprisingly simple, as shown in Listing 42.4.

Listing 42.4 Predecrement Operators

```
  . . .
public: // Bidirectional Iterator Methods
  class_type& operator --()
  {
    STLSOFT_MESSAGE_ASSERT( "Attempting to increment an endpoint
iterator", m_it != m_begin);
    for(--m_it; m_it != m_begin; --m_it)
    {
      if(m_pr(*m_it))
      {
        break;
      }
    }
    return *this;
  }
  class_type& operator --(int); // Canonical implementation
  . . .
```

Now we can enumerate forwards as well as backwards:

```
template <typename I>
void fn(I from, I to)
{
  ++it;
  --it;
}

struct is_odd;

int ints[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

fn(stlsoft::filter(&ints[0], &ints[0] + 8, is_odd())); // Well-formed
```

42.5.3 Random Access Iterator Semantics

This one is very simple. There's no reasonable way to implement random access iterator semantics in a filtering iterator, so we don't do it. The only conceivable way would be extremely expensive since each apparent random access operation would involve an element-by-element iteration to identify which elements match the predicate. Even if this weren't the case, I simply can't conceive of a scenario in which random access filtering is meaningful. I may be wrong, of course, in which case please write to me and disabuse me of my erroneous assumptions.

Recommendation: Eschew support for random access (and higher) iterators in filtering iterators.

42.6 Constraining the Iterator Category

Given that we've chosen to eschew random access iterator semantics, we actually have a problem on our hands. If we adapt a random access iterator, the adapted form will think it's a random access iterator—the `iterator_category` member type of the adapted type will be `std::random_access_iterator_tag`—even though we've supplied it only with bidirectional iterator semantics. This is a problem. As soon as we try to pass this off to an algorithm that has a specialized form for handling random access iterators, things are going to get ugly. What you tend to see in this case is an enormous list of error messages, and ensconced within, if you're lucky enough to spot it, will be some mention of a missing `operator -()`, or `operator +()`, or some other operation specific to random access iterators.

You might wonder why we've not come across this with the other adaptors. Well, `transform_iterator` (Chapter 36), `member_selector_iterator` (Chapter 38), and `index_iterator` (extra chapter on the CD) are all able to exhibit the iterator category of their base type; `filter_iterator`, by its very nature, cannot.

Thus, the final act of cunning is to use the `min_iterator_category` template and its 16 full specializations, each of which corresponds to a permutation of two standard iterator categories. The primary template and several of the specializations are shown in Listing 42.5. In each permutation, the member type `iterator_category` is defined as the lesser refinement of the two specializing types.

Listing 42.5 Primary Template and Some Specializations of `min_iterator_category`

```
template< typename C1 // First category
        , typename C2 // Second category
        >
struct min_iterator_category;

template <>
struct min_iterator_category< std::input_iterator_tag
                            , std::input_iterator_tag
                            >
{
  typedef std::input_iterator_tag iterator_category;
};
template <>
struct min_iterator_category< std::forward_iterator_tag
                            , std::input_iterator_tag
                            >
{
  typedef std::input_iterator_tag iterator_category;
};
. . .
```

```
template <>
struct min_iterator_category< std::bidirectional_iterator_tag
                            , std::random_access_iterator_tag
                            >
{
  typedef std::bidirectional_iterator_tag iterator_category;
};
template <>
struct min_iterator_category< std::random_access_iterator_tag
                            , std::random_access_iterator_tag
                            >
{
  typedef std::random_access_iterator_tag iterator_category;
};
```

The traits class is used to limit the `iterator_category` member type to the maximum sensible refinement that is supportable (Listing 42.6).

Listing 42.6 Definition of `filter_iterator`

```
template< typename I // The underlying iterator
        , typename P // The unary predicate that will select the items
        , typename T = adapted_iterator_traits<I>
        >
class filter_iterator
{
public: // Member Types
  . . .
  typedef filter_iterator<I, P, T>              class_type;
  typedef typename min_iterator_category<
                        typename traits_type::iterator_category
                      , std::bidirectional_iterator_tag
                      >::iterator_category      iterator_category;
  . . .
```

42.7 Summary

We've seen that a filtering iterator adaptor *must* be instantiated from an iterator pair defining the viable range of the adapted range. It's slightly inconvenient to the user but is the only workable solution. We've also seen that by applying the `adapted_iterator_traits` traits class, we can achieve a simple definition for what is a sophisticated iterator adaptation.

42.8 On the CD

The CD contains a preview of how filtering can be more simply achieved using the *ranges* concept, which will be described in Volume 2.