CHAPTER 34

Functors and Ranges

34.1 Syntactic Clutter

Many of the standard library algorithms operate on ranges, where a range is defined as a pair of iterators [Aust1999]. This abstraction is very powerful, and has been exploited to the degree that much of the STL, and, therefore, much of modern C++, relies upon it.

An example of this might be in a simple program to read in integers into a vector:

```
std::fstream f("integers.dat", std::ios::in | std::ios::out);

std::copy( std::istream_iterator<int>(f)
         , std::istream_iterator<int>()
         , std::back_inserter(v2));
```

In this case, the second argument is a default-constructed iterator that acts as an indicator for the end of range. The two iterators are not connected in a physical sense; the implementation of `istream_iterator` is such that a default-constructed instance may be interpreted as the logic end point of the range.

Many times we use algorithms over a range of values acquired from a container or containerlike, object as in:

```
struct dump_string
{
  void operator ()(std::string const &) const;
};

std::vector<std::string> >   strings = . . .;
std::for_each(strings.begin(), strings.end(), dump_string());
```

As such, it can become tedious, since we replicate the same calls to `begin()` and `end()` time and time again. This is miles away from being an imperfection—barely even a minor gripe—but there are circumstances in which it can be a pain. Consider the case when using (pseudo)containers such as `glob_sequence` (see section 20.6.3) that are only created in order to elicit their range.[1] Let's imagine we want to determine how many *Imperfect C++*

[1]Note that I'm kind of creating my own problem to solve in this specific case, because I tend to extend STL in this way. It is possible in many cases to follow the example of `istream_iterator` and encapsulate the start of the enumeration in the constructor of the nondefault iterator; Boost's file-system enumeration component works in this way. But this is a logical disconnect that troubles me too much to follow the lead—you may well see it differently.

header files are larger than 1024 bytes, because we want to really immerse ourselves in the magic:

```
struct is_large
  : public std::unary_function<char const *, bool>
{
  bool operator ()(char const *file) const
  {
    . . . // Return true if "file" > 1024 bytes
  }
};

glob_sequence gs("/usr/include/", "impcpp*");
size_t n = std::count_if(gs.begin(), gs.end(), is_large());
```

`gs` is not needed for any other purpose, and its presence otherwise serves only to pollute the local namespace.

An analogous situation occurs when we want to enumerate two or more ranges in the same scope. We can end up introducing several variables for the different begin and end conditions of each range into the current scope, as we saw in section 17.3.2, or using the double-scoping trick from section 17.3.1.

34.2 for_all() ?

Given the common enumeration of full ranges, it is a simple matter to create full-range equivalents of the standard algorithms. We might create a container-compatible version of `std::for_each()`, which we'll call `for_all()` for the moment:

```
template< typename  C
        , typename  F
        >
inline F for_all(C &c, F f)
{
  return std::for_each(c.begin(), c.end(), f);
}
```

This is a reasonable implementation for the general case of such an algorithm since most containers—standard and otherwise—provide `begin()` and `end()` methods.

As well as reducing the tedium of specifying the two iterator method calls, it can also further reduce the eyestrain when dealing with (pseudo)containers. Our `glob_sequence` can be declared as an anonymous temporary, giving us the pleasingly condensed:

```
n = std::count_if( glob_sequence("/usr/include/", "impcpp*")
                 , is_large());
```

When you're dealing with production code, such savings of syntactic clutter can be a real boost in readability.

Some critics may counter that the considerable amount of code involved in that single statement is another example of how C++ has strayed from the Spirit of C [Como-SOC]; it's certainly hard to argue "nothing is hidden." I would concede this point, but only as far as the functor is concerned. I think arguing that the "hidden" nature of the range access and enumeration is wrong is a specious argument; one may as well argue that we should all eschew the use of library functions and write everything in assembler. The STL *Iterator* concept [Aust1999, Muss2001] is designed to facilitate maximum efficiency of enumeration—usually the only way to "misuse" sequences is to use higher level *Iterator* concept [Aust1999, Muss2001] behavior, but `for_each()` requires only input iterators.

34.2.1 arrays

As we saw in Chapter 14, the fact that we've got two places of definition of the array size is a potential source of errors. Even when they both use the same constant, there are still, in effect, two definitions.

```
int    ari[10] = { . . . };

std::for_each(&ari[0], &ari[10], print_int);
```

We also saw that the foolproof way to deal with arrays is to use static array size determination in the form of `dimensionof()` or a similar construct.

```
std::for_each(&ari[0], &ari[dimensionof(ari)], print_int);
```

However, given our definitions of `for_all()`, we can easily specialize for arrays using the similar facilities, as in:

```
template< typename  T
        , size_t    N
        , typename  F
        >
inline F for_all(T (&ar)[N], F f)
{
  return std::for_each(&ar[0], &ar[N], f);
}
```

34.2.2 Naming Concerns

Let's now turn to the issue of the naming of such algorithms. I deliberately chose an unsuitable name, so that we wouldn't get comfy with it. The problem with the name `for_all()` is all too obvious: it doesn't transfer to other algorithms. Do we have a `fill_all()`, `accumulate_all()`, and so on? It's not an attractive option. Ideally we'd like to call it for_each(), so why not do that? Unfortunately, this is a bit of a conundrum: we must select from equally unpleasant options.

We are only allowed to specialize templates that already exist in the standard namespace for new types; we are not allowed to add any new template (or nontemplate) functions or types. I would say that the array form of for_all(), were we to call it for_each(), could not be reasonably classed as a specialization based on new types, although I confess the issue's hardly equivocal.

The alternative is to define it as for_each() within our own namespace. In that case, we must remember to "use" it, via a using declaration, whenever we want to use it outside our namespace. Unfortunately, we'll also have to remember to "use" std::for_each() whenever that's what's required, since any using declaration for our for_each() will mask out the std one. Failure to do this can lead to some very perplexing error messages. But despite this, the problems raised for writing generalized code when the algorithms are of different names are such that, in general, this approach should be seriously considered[2] despite the namespace "use" hassles.

We'll look at just one example of the namespace problems. Consider that you are using your own library components en masse via a using directive, using namespace acmelib. That's okay, you reason, because in this client code you'll be using heaps of things from the acmelib namespace. One of these is your array version of for_each(). You go along swimmingly with this for a while, and then need to use std::for_each() in some part of the implementation file, so you "use" that one function via a using declaration:

```
using namespace acmelib; // Using directive
using std::for_each;     // Using declaration

int   ai[10];

for_each(ai, print_int);
```

Now your code will not compile, because the specific introduction of std::for_each() via the using declaration takes precedence over the one "used" via the general introduction of the using directive. What's to be done: introduce std::for_each() via a using directive? What happens if there're conflicting definitions in the two namespaces of a given function or type that is being used in our code?

What we're forced to do is make an additional using declaration for acmelib::for_each(). Given that, why not simply use using declarations in the first place? It may be a little bit more work initially, but it's a big saving in the long run, and we all know that the cost of the initial coding phase of software is pretty irrelevant in comparison to its maintenance [Glas2003]. This is just one of the reasons why I refuse to use using directives under just about any circumstances.[3]

[2]This is the approach taken by John in his Boost-compatible implementation of RangeLib, although he relies more on the explicit qualification of the RangeLib algorithm namespace boost::rtl::rng::for_each.

[3]Much hard-won experience with Java's import x.* also contributes to my loathing for the indiscriminate introduction of names into namespaces. Herb Sutter and I have an ongoing difference of opinion on this point. Fortunately for Herb, he is vastly more knowledgeable and erudite on C++ than I am. Fortunately, for me, this is my book and I can write what I like.

In this case, our new `for_each()` and `std::for_each()` have different numbers of arguments, so we could not be writing generalized code that could work with both anyway. Hence, we could just call the algorithms `for_each_c()`, `fill_c()`, and so on.

We'll come back to the naming problem toward the end of this chapter.

34.3 Local Functors

So far our concerns have been largely syntactic. But there's a second, and more significant, troubling aspect of using STL algorithms, which represent a more serious imperfection. When you have a suitable function or functor available, then you can get code that is very succinct, as well as being powerful:

```
std::for_each(c.begin(), c.end(), fn());
```

However, as soon as you need to do something more sophisticated or specific to the elements in a range, you have two choices, neither of which is particularly attractive. Either you unroll the loop and provide the functionality yourself, or you wrap up that functionality in a custom function or functor.

34.3.1 Unrolled Loops

The normal option is to unroll the algorithm and do it longhand, as in the following code from an early version of the Arturius compiler multiplexer (see Appendix C):

Listing 34.1

```
void CoalesceOptions(. . .)
{
  . . .
  { OptsUsed_map_t::const_iterator  b = usedOptions.begin();
    OptsUsed_map_t::const_iterator  e = usedOptions.end();
  for(; b != e; ++b)
  {
    OptsUsed_map_t::value_type const &v = *b;
    if( !v.second &&
        v.first->bUseByDefault)
    {
      arcc_option option;
      option.name              = v.first->fullName;
      option.value             = v.first->defaultValue;
      option.bCompilerOption   = v.first->type == compiler;
      arguments.push_back(option);
    }
  }}
  . . .
```

Naturally, such things can get quite verbose; I could have grabbed another from the same file that was considerably longer.

34.3.2 Custom Functors

The alternative to unrolling loops is to write custom functors to provide the behavior you need. If the behavior is something that can be used in a variety of places that's great, but oftentimes you're writing a new class for just one, or a small number of cases.

Because it's a separate class, it will be physically separate from where it is being used, leading to the code being hard to comprehend and maintain. The best way you can handle this is to define the functor class in the compilation unit as the code that's going to be using it, preferably just before the function where it is used.

Listing 34.2

```
struct argument_saver
{
public:
  argument_saver(ArgumentsList &args)
    : m_args(args)
  {}
  void operator ()(OptsUsed_map_t::value_type const &o) const
  {
    if( !o.second &&
        o.first->bUseByDefault)
    {
      arcc_option option;
      . . .
      m_args.push_back(option);
    }
  }
private:
  ArgumentsList &m_args;
};

void CoalesceOptions(. . .)
{
  . . .
  std::for_each( usedOptions.begin(), usedOptions.end()
               , argument_saver(arguments));
  . . .
```

But the domain specific code encapsulated within the functor is physically separate from the only place(s) where it is used, and meaningful, and this is not ideal. The separation reduces maintainability and, worse, encourages overeager engineers to try and reuse it or to refactor it.

34.3.3 Nested Functors

One thing that could be done to improve the problem of separation of specific functors from their point of its use might be to allow them to be defined within the function in which they're used. For example, we might want `argument_saver` to be defined within `fn()`, as in:

```
void CoalesceOptions(. . .)
{
  . . .
  struct argument_saver
  {
    . . .
  };
  std::for_each( usedOptions.begin(), usedOptions.end()
               , argument_saver(arguments));
  . . .
```

Alas, local functor classes are not legal in C++. This is because a template argument must refer to an entity with external linkage (C++-98: 14.3.2).[4]

Imperfection: C++ does not support local functor classes (to be used with template algorithms).

Notwithstanding the illegality, some compilers do allow them: CodeWarrior, Digital Mars, and Watcom. Furthermore, Borland and Visual C++ can be tricked into supporting it by the simple rouse of encapsulating the next class within another nested class, as in

Listing 34.3

```
void CoalesceOptions(. . .)
{
  . . .
  struct X
  {
    struct argument_saver
    {
      . . .
    };
  };

  std::for_each( usedOptions.begin(), usedOptions.end()
               , X::argument_saver(arguments));
  . . .
}
```

Table 34.1 summarizes the support for both forms for several popular compilers. Comeau, GCC, and Intel do not supported nested functions in either guise.[5]

If you're using only one or more of Borland, CodeWarrior, Digital Mars, Visual C++, and Watcom, then you might choose this approach. But it's not legal, and your portability will be compromised if you do so.

[4]This applies to all templates, not just template algorithms and functors.

[5]If those three tell you your code is wrong, the odds are good that you're doing something wrong.

Table 34.1

Compiler	Local class	Nested local class
Borland	No	Yes
CodeWarrior	Yes	Yes
Comeau	No	No
Digital Mars	Yes	Yes
GCC	No	No
Intel	No	No
Visual C++	No	Yes
Watcom	Yes	Yes

34.3.4 Legally Bland

The only legal way I know of to make such things work is turgid beyond bearing. Since the template's parameterizing type must be an external type, we define the function type outside the function. Given that we want to specialize the behavior in a local class, we must relate the internal and external classes. Since we cannot use templates, we can fall back on the old C++ workhorse: polymorphism. The local class `argument_saver` inherits from the external class `argument_processor`, and overrides its `operator ()() const`, as in:

Listing 34.4

```
struct argument_processor
{
public:
  virtual void operator ()(OptsUsed_map_t::value_type const &o) const = 0
};

void CoalesceOptions(. . .)
{
  . . .
  struct argument_saver
    : argument_processor
  {
    virtual void
          operator ()(OptsUsed_map_t::value_type const &o) const
    {
      . . .
    }
  };
```

That doesn't seem so bad, I suppose. However, to use it in a template algorithm requires that the parameterization be done in terms of the external (parent) class. And since the parent is an abstract class, the functor must be passed as a (`const`) reference, rather than by value.

Further, `std::for_each()` takes the functor type as the second template parameter, so it is necessary to explicitly stipulate the iterator type as well. Thus, using the "convenient" `for_each()` isn't very succinct, or general, and it certainly isn't pretty:

```
for_each< OptsUsed_map_t::const_iterator
        , argument_processor const &>( &ari[0], &ari[10]
                                     , argument_saver());
```

You'd have to agree that manual enumeration would be preferable. There's one last gasp at making this prettier, which is to define a `for_each()` equivalent that takes the template parameters in the reverse order so that the function type can be deduced implicitly:

```
template< typename F
        , typename I
        >
inline F for_each_l(I first, I last, F fn)
{
  return std::for_each<I, F>(first, last, fn);
}
```

which brings us to the final barely chewable form:

```
for_each_l<argument_processor const &>( &ari[0], &ari[10]
                                      , argument_saver());
```

But for my money this is still nowhere near good enough. Imagine the poor maintenance programmer trying to follow this stuff![6]

34.3.5 Generalized Functors: Type Tunnelling

If we can't make functors more local, maybe we can improve the situation by making them more general? Let's look at an example where we can expand the generality of the `is_large` functor (see section 34.1). We can use this with a sequence whose `value_type` is, or may be implicitly converted to, `char const*`, such as `glob_sequence`. Unfortunately, it can *only* be used with such types. If we want to use the same function with a type that uses Unicode encoding and the `wchar_t` type, it won't work.

One answer to this is to make `is_large` a template, parameterizable via its character type, as in:

```
template <typename C>
  : public std::unary_function<C const *, bool>
struct is_large
{
```

[6]There were a few weeks between starting this chapter and doing the final version. In that short time I forgot how this worked, and I wrote it!

```
  bool operator ()(C const *file) const;
};
```

Now this will work with sequences using either `char` or `wchar_t` (using the fictional Unicode `globw_sequence`), so long as we stipulate the appropriate instantiation:

```
glob_sequence  gs("/usr/include/", "impcpp*");
n = std::count_if(gs.begin(), gs.end(), is_large<char>());

globw_sequence gsw(L"/usr/include/", L"impcpp*");
n = std::count_if(gsw.begin(), gsw.end(), is_large<wchar_t>());
```

That's a lot more useful, but it's still not the full picture. Looking back to section 20.6.3, we also looked at another file system enumeration sequence `readdir_sequence`, whose `value_type`—`struct dirent const*`—is not implicitly convertible to `char const*`. The solution for the problem in that section was to use *Access Shims* (see section 20.6.1), and we can apply them here to the same end. However, it's a little more complex now, because we've got templates involved, as shown in Listing 34.5.

Listing 34.5

```
template< typename C
        , typename A = C const *
        >
struct is_large
       : public std::unary_function<A, bool>
{
  template <typename S>
  bool operator ()(S const &file) const
  {
    return is_large_(c_str_ptr(file)); // apply c_str_ptr shim
  }
private:
  static bool is_large_(C const *file)
  {
    . . . // determines whether large or not
  }
};
```

The function-call operator—`operator ()() const`—is now a template member function, which attempts to convert whatever type is applied to it via the `c_str_ptr()` shim to `C const*`, which is then passed to the `static` implementation method `is_large_()`. Now we can use the functor with any type for which a suitable `c_str_ptr()` definition exists and is visible, hence:

```
readdir_sequence  rs("/usr/include/");
n = std::count_if(rs.begin(), rs.end(), is_large<char>());
```

I call this mechanism *Type Tunneling*.

Definition: *Type Tunneling* is a mechanism whereby two logically related but physically unrelated types can be made to interoperate via the use of *Access Shims*. The shim allows an external type to be *tunneled* through an interface and presented to the internal type in a recognized and compatible form.

I've used this mechanism to great effect throughout my work over the last few years. As well as facilitating the decoupled interoperation of a large spectrum of physically unrelated types via C-string forms, there is also the generalized manipulation of handles, pointers, and even synchronization objects. Type tunneling (and shims in general) goes to town on the principal of making the compiler one's batman. We saw another example of type tunneling in section 21.2, whereby virtually any COM-compatible type can be *tunneled* into the logging API through the combination of generic template constructors and `InitialiseVariant()` overloads, which combine to act as an access shim.

34.3.6 A Step too Far, Followed by a Sure Step Beyond

You may be wondering whether we can take this one further step, and remove the need to stipulate the character type. The answer is that we can, and with ease, as shown in Listing 34.6.

Listing 34.6

```
struct is_large
  : public std::unary_function<. . ., bool>
{
  template <typename S>
  bool operator ()(S const &file) const
  {
    return is_large_(c_str_ptr(file));
  }
private:
  static bool is_large_(char const *file);
  static bool is_large_(wchar_t const *file);
};
```

It is now simpler to use, as in:

```
n = std::count_if(rs.begin(), rs.end(), is_large());
n = std::count_if(gs.begin(), gs.end(), is_large());
n = std::count_if(gsw.begin(), gsw.end(), is_large());
```

However, there's a good reasons why we don't do this. This functor is a predicate—a functor whose function-call operator returns a Boolean result reflecting some aspect of its argument(s). One important aspect of predicates is that they may be combined with adaptors [Muss2001], as in the following statement that counts the number of *small* files:

```
n = std::count_if( gs.begin(), gs.end()
                 , std::not1(is_large<char>()));
```

In order for adaptors to work with predicates, they must be able to elicit the member types argument_type and result_type from the predicate class. This is normally done by deriving from std::unary_operator. Now we can see why the final refinement shown in Listing 34.6 cannot be done. There's no way to specify the argument type, other than to define the predicate class as a template with a single template parameter to define the predicate. But this would have to be provided in every use as there's no sensible default, which would be onerous to use and confusing to read.

This is why the actual functor definition is a two-parameter template, where the first parameter C represents the character type and the second parameter A, which defaults to C const*, represents the argument_type of the predicate.

```
template< typename C
        , typename A = C const *
        >
struct is_large
       : public std::unary_function<A, bool>
{
  . . .
```

Now when we want to use this with an adaptor and a sequence whose value_type is not C const*, we do something like the following:

```
n = std::count_if( rs.begin(), rs.end() // rs: readdir_sequence
            , std::not1(is_large<char, struct dirent const*>()));
```

It doesn't have beauty that will stop a man's heart, but it's bearable considering the fact that the combination of sequence and adaptor that necessitates it is low incidence, and the advantages of the generality and consequent reuse are high. It also facilitates a high degree of generality, since we can write a template algorithm that would be perfectly compatible with any sequence, and maintain the type tunneling, as in:

```
template< typename C // character type
        , typename S // sequence type
        >
void do_stuff(. . .)
{
  S s = . . .;
  size_t n = std::count_if( s.begin(), s.end()
            , std::not1(is_large<C, typename S::value_type>()));
  . . .
```

Okay, before you think I've gone completely cracked, I confess that this is hardly something that gambols into the brain through the merest wisp of rapidly clearing mental fog. It's something that you definitely have to think about. But there are times when we simply *have* to have complex bits of code; check out some of your friendly neighborhood open-source C++ libraries if you don't believe me.

The point is that we have a mechanism for writing highly generalized—reusable, in other words—components, which are very digestible—in that they follow accepted idiomatic forms—in most cases where they are used. This generality is bought for the acceptable, in my opinion, cost of requiring the specification of the given sequence's `value_type` when used with adaptors.

34.3.7 Local Functors and Callback APIs

Just because local functors are not allowed for STL algorithms does not mean that they cannot find use in enumeration. In fact, when dealing with callback enumeration APIs, local classes are eminently solutions. Consider the implementation of the function `FindChildById()` (see Listing 34.7), which provides a deep-descendent equivalent to the well-known Win32 function `GetDlgItem()`. `GetDlgItem()` returns a handle to an immediate child window bearing the given id. `FindChildById()` provides the same functionality, but is able to locate the id in *any* descendent windows, not just immediate children.

Listing 34.7

```
HWND FindChildById(HWND hwndParent, int id)
{
  if(::GetDlgCtrlID(hwndParent) == id)
  {
    return hwndParent; // Searching for self
  }
  else
  {
    struct ChildFind
    {
      ChildFind(HWND hwndParent, int id)
        : m_hwndChild(NULL)
        , m_id(id)
      {
        // Enumerate, passing "this" as identity structure
        ::EnumChildWindows( hwndParent,
                            FindProc,
                            reinterpret_cast<LPARAM>(this));
      }

      static BOOL CALLBACK FindProc(HWND hwnd, LPARAM lParam)
      {
        ChildFind &find = *reinterpret_cast<ChildFind*>(lParam);

        return (::GetDlgCtrlID(hwnd) == find.m_id)
                  ? (find.m_hwndChild = hwnd, FALSE)
                  : TRUE;
      }
```

ch34.qxd 8/26/04 8:46 AM Page 521

```
      HWND      m_hwndChild;
      int const m_id;

    } find(hwndParent, id);

    return find.m_hwndChild;
  }
}
```

The class `ChildFind` is declared within the function, maximizing encapsulation. The instance `find` is passed the parent window handle and the id to find. The constructor records the id in the `m_id` member, and sets the search result member `m_hwndChild` to `NULL`. It then calls the Win32 callback enumeration function `EnumChildWindows()`, which takes the parent window handle, a callback function, and a caller-supplied parameter. The instance passes the static method `FindProc()` and itself as the parameter. `FindProc()` then responds to each callback by determining whether the desired id has been located, and, if so, it records the handle and terminates the search.

When the construction of `find` is complete, it will either contain the requested handle, or `NULL`, in the `m_hwndChild` member. In either case this is returned to the caller of `FindChildById()`. The entire enumeration has been carried out in the constructor of the local class, whose definition is not accessible to any outside context. `FindChildById()` perfectly encapsulates the `ChildFind` class.

34.4 Ranges

We've seen that there are two issues in the enumeration of ranges and the use of functors. First, we have a desire to avoid the general, though not universal, repeated stipulation of the `begin()` and `end()` methods for sequence types. Second, we have a need to avoid writing overly specific "write-once, use nowhere else" functors.

In a discussion with my friend John Torjo,[7] we discovered that we've had similar disenchantments with these issues, and wanted a better solution. We came up with the *Range* concept. Naturally, he and I see the issue slightly differently;[8] the concept definition and component implementations presented here primarily reflect my view.

34.4.1 The Range Concept

The range concept is quite simple.

Definition: A *Range* represents a bounded collection of elements, which may be accessed in an incremental fashion. It encapsulates a logical range—that is, a beginning and end point, along with rules for how to walk through it (move from the beginning to

[7]John was also a one of the reviewers for this book.

[8]You can fit more angels on the head of a pin than you can find software engineers who wouldn't argue over a blade of grass. Notwithstanding that, John and I are in a reasonable state of agreement. You can see the differences in our interpretation of the concept, and understand the different design principles and implementation mechanisms in our implementations, because John's expansive and impressive Boost-compatible implementation of the Range concept is included on the CD along with my own STLSoft version.

the end point)—and embodies a single entity with which client code may access the values contained within the range.

Sounds almost too simple to be meaningful, doesn't it? Well, the salient part of the definition is that it's a single entity. The canonical form of use is:

```
for(R r = . . .; r; ++r)
{
  f(*r);
}
```

or, if you don't go for all that operator overloading:

```
for(R r = . . .; r.is_open(); r.advance())
{
  f(r.current());
}
```

There's noticeable brevity of form, which is part the raison d'être of ranges. A range has the characteristics shown in Table 34.2.

Let's look at a couple of ranges in action:

```
// A range over a sequence
glob_sequence gs("/usr/include/", "impcpp*");
for(sequence_range<glob_sequence> r(gs); r; ++r)
{
  puts(*r); // Print the matched file entry to stdout
}

// A "notional range"
for(integral_range<int> r(10, 20, 2); r; ++r)
```

Table 34.2 Characteristics of a *notional range*.

Name	Expressions	Semantics	Precondition	Postcondition
`Dereference`	`*r` or `r.current()`	Returns the value of represented at the current position	`r` has not reached its end condition	`r` is unchanged
`Advance`	`++r` or `r.advance()`	Advances `r`'s current position	`r` has not reached its end condition	`r` is dereferenceable or has reached its end condition
`State`	`r` or `r.is_open()`	Evaluates to true if `r` has reached its end condition, false otherwise	-	`r` is unchanged

```
{
  cout << *r << endl; // Print int value at current point
}
```

There's no way any pointer can fit the range syntax[9] so we don't have to worry about catering for any fundamental types; this provides us with a lot of extra flexibility in the implementation. In other words, we can rely on all range instances being of class types, and therefore we can rely on the presence or absence of class type characteristics in order to specialize behavior for the algorithms that operate on ranges (see section 34.4.4).

The way this works is similar to the implementation of algorithms that manipulate Iterators [Aust1999], only simpler. My implementation of ranges simply relies on the specific range types deriving from the appropriate range type tag structure:

```
struct simple_range_tag
{};

struct iterable_range_tag
  : public simple_range_tag
{};
```

We'll see how these are used in the next sections.

34.4.2 Notional Range

Sometimes you don't have a concrete range, that is, one defined by two iterators; rather you have a notional range. It might be the odd numbers between 1 and 99. In this simple case you know that the number of odd numbers is 49 (1–97 inclusive), and so what you might do in classic STL is as shown in Listing 34.8:

Listing 34.8

```
struct next_odd_number
{
  next_odd_number(int first)
    : m_current(first - 2)
  {}
  int operator ()()
  {
    return ++++m_current; // Nasty, but it fits on one line ;/
  }
private:
  int m_current;
};

std::vector<int>  ints(49);
std::generate(ints.begin(), ints.end(), next_odd_number(1));
```

[9]Except if we were to point it high into the top of memory and then iterate until it reached 0, but this would result in an access violation on many platforms, so the issue is moot.

Books on STL tend to show such things as perfectly reasonable examples of STL best practice. That may indeed be the case, but to me it's a whole lot of pain. First, we have the creation of a functor that will probably not be used elsewhere.

Second, and more significantly, the technique works by treating the functor as a passive producer whose actions are directed by the standard library algorithm `generate()`, whose bounds are, in turn, defined by the vector. This is very important, because it means that we need to know the number of results that will be created beforehand, in order to create spaces for them in the vector. Hence, we need a deep appreciation of how the producer—the functor—works. In this simple case, that is relatively straightforward—although I must confess I did get it wrong when preparing the test program! But imagine if we were computing the members of the Fibonacci series up to a user-supplied maximum value. It would be impossible to anticipate the number of steps in such a range other than by enumerating it to completion, which would be a drag, not to mention inefficient.

What we want in such cases is for the producer to drive the process, in accordance with the criteria stipulated by the code's author. Time to crank out my favorite example range, the `integral_range` template, whose simplified definition is shown in Listing 34.9:

Listing 34.9

```
template <typename T>
class integral_range
  : public simple_range_tag
{
public:
  typedef simple_range_tag  range_tag_type;
  typedef T                 value_type;
// Construction
public:
  integral_range( value_type first, value_type last
                , value_type increment = +1)
  ~integral_range()
  {
    // Ensure that the parameterising type is integral
    STATIC_ASSERT(0 != is_integral_type<T>::value);
  }
// Enumeration
public:
  operator "safe bool"() const; // See Chapter 24
  value_type operator *() const;
  class_type &operator ++();
// Members
  . . .
};
```

The `integral_range` performs enumeration over its logical range via member variables of the given integral type, `T`.

We can now rewrite the instantiation of the vector with odd numbers as follows:

```
std::vector<int>  ints;
ints.reserve(49); // No problem if this is wrong.
r_copy(integral_range<int>(1, 99, 2), std::back_inserter(ints));
```

Note that the call to `reserve()` is an optimization and can be omitted without causing any change to the correctness of the code. `r_copy()` is a range algorithm (see section 34.4.3) that has the same semantics as the standard library algorithm `copy()` [Muss2001]. Now the producer, the instance of `integral_range<int>`, drives the process, which is as it should be. We could easily substitute a `Fibonacci_range` here, and the code would work correctly and efficiently, which cannot be said for the STL version.

34.4.3 Iterable Range

The *Iterable* range is an extension of the *Notional* range, with the additional provision of `begin()`, `end()` and `range()` methods, allowing it to be fully compatible with standard STL algorithms and usage (see section 34.4.4).

Iterable ranges usually maintain internal iterators reflecting the current and past-the-end condition positions, and return these via the `begin()` and `end()` methods. The `range()` method is provided to allow a given range to be cloned in its current enumeration state, although this is subject to the restrictions on copying iterators if the range's underlying iterators are of the *Input* or *Output* Iterator concepts [Aust1999, Muss2001]: only the *Forward* and "higher" models [Aust1999, Muss2001] support the ability to pass through a given range more than once.

It is possible to write iterable range classes, but the common idiom is to use an adaptor. In my implementation of the *Range* concept, I have two adaptors, the `sequence_range` and `iterator_range` template classes. Obviously the `sequence_range` adapts STL *Sequence*s, and the `iterator_range` adapts STL *Iterator*s.

Given a sequence type, we can adapt it to a range by passing the sequence instance to the constructor of an appropriate instantiation of `sequence_range`, as in:

```
std::deque<std::string> d = . . . ;

sequence_range< std::deque<std::string> > r(d);
for(; r; ++r)
{
  . . . // Use *r
}
```

Similarly, an `iterator_range` is constructed from a pair of iterators, as in:

```
vector<int> v = . . . ;

for(iterator_range<vector<int>::iterator> r(v.begin(), v.end());
    r; ++r)
{
  . . . // Use *r
}
```

Iterable ranges at first glance appear to be nothing more than a syntactic convenience for reducing the clutter of unwanted variables when processing iterator ranges in unrolled loops. I must confess that this in itself represents a big attraction to draw for me, but the concept offers much more as we'll see in the next couple of sections.

34.4.4 Range Algorithms and Tags

All the examples so far have shown ranges used in unrolled loops. Ranges may also be used with algorithms. That's where the separation of *Notional* and *Iterable* range concepts comes in.

Consider the standard library algorithm `distance()`.

```
template <typename I>
size_t distance(I first, I last);
```

This algorithm returns the number of elements represented by the range `[first, last)`. For all iterator types other than *Random Access* [Aust1999, Muss2001], the number is calculated by iterating `first` until it equals `last`. But the template is implemented to simply and efficiently calculate `(last - first)` for random access iterators. We certainly don't want to lose such efficiency gains when using algorithms with ranges.

The answer is very simple: we implement the range algorithms in terms of the standard library algorithms where possible. The range algorithm `r_distance()` is defined as shown in Listing 34.10:

Listing 34.10

```
template <typename R>
ptrdiff_t r_distance_1(R r, iterable_range_tag const &)
{
  return std::distance(r.begin(), r.end());
}
template <typename R>
ptrdiff_t r_distance_1(R r, simple_range_tag const &)
{
  ptrdiff_t d = 0;
  for(; r; ++r, ++d)
  {}
  return d;
}
template <typename R>
inline ptrdiff_t r_distance(R r)
{
  return r_distance_1(r, r);
}
```

`r_distance()` is implemented in terms of `r_distance_1()`,[10] which has two definitions: one for the iterable ranges that defers to a standard library algorithm, and one for notional ranges that carries out the enumeration manually. The two overloads of `r_distance_1()` are distinguished by their second parameter, which discriminate on whether the ranges as simple or iterable.

We use run time polymorphism (inheritance) to select the compile-time polymorphism (template type resolution), so we need to pass the range to `r_distance_1()` in two parameters, to preserve the actual type in the first parameter while at the same type discriminating the overload in the second. Since compilers can make mincemeat of such simple things, there're no concerns regarding inefficiency, and the mechanism ably suffices. We saw in section 34.4.2 that the `integral_range` template inherited from `simple_range_tag`. Iterable ranges inherit from `iterable_range_tag`. Hence the implementations of all range algorithms unambiguously select the most suitable implementation.

We can now come back and address the algorithm name problem (see section 34.2.2). Since we're clearly distinguishing between iterator-pair ranges and instances of a range class, we're never going to need the same name to facilitate generalized programming because the former ranges come as two parameters and the latter as one. Therefore, we can avoid all the namespace tribulations and just use algorithms with an `r_` prefix.

34.4.5 Filters

Another powerful aspect of the *Range* concept abstraction is that of filters. This aspect of ranges is only just developing, but is already promising great benefits. I'll illustrate with a simple filter `divisible`, which we'll apply to our `integral_range`.

Listing 34.11

```
template <typename R>
struct divisible
       : public R::range_tag_type
{
public:
  typedef R                         filtered_range_type;
  typedef typename R::value_type  value_type;
public:
  divisible(filtered_range_type r, value_type div)
    : m_r(r)
    , m_div(div)
  {
    assert(div > 0);
    for(; m_r && 0 != (*m_r % m_div); ++m_r)
    {}
  }
```

[10]This is a general implementation naming policy, which helps to avoid clashes that can occur when the "outer" function and the implementing "inner" functions have the same name, and there are multiple overloads of the "outer" function. Different compilers have different aptitudes for resolving such things, so the simplest solution is to avoid it, and name the implementing "inner" functions unambiguously.

```
public:
  operator "safe bool"() const; // See Chapter 24
  {
    . . . // implemented in terms of m_r's op safe bool
  }
  value_type operator *() const
  {
    return *m_r;
  }
  class_type &operator ++()
  {
    for(; m_r && 0 != (*++m_r % m_div); )
    {}
    return *this;
  }
private:
  filtered_range_type m_r;
  value_type          m_div;
};
```

When combined with `integral_range`, we can filter out all the odd numbers in a given range to only those that are wholly divisible by three, as in:

```
std::vector<int>    ints;
integral_range<int> ir(1, 99, 2);
r_copy( divisible<integral_range<int> >(ir, 3)
      , std::back_inserter(ints));
```

Naturally range filters can be a lot more sophisticated than this in the real world.

34.4.6 Hypocrisy?

As someone who is against the abuse of operator overloading,[11] I am certainly open to accusations of hypocrisy when it comes to the operators that ranges support. Clearly it can be argued, if one wished to be terribly uncharitable, that the unrealistic combination of operators provided by ranges is an abuse.

I can't really defend the concept in any purist sense, so I can fall back on the Imperfect Practitioner's Philosophy, and state that the fact that it works so well, and so simply, is worth the pricked conscience.

If you like the range concept, but don't want to go for the operator overloading, it is a simple matter to replace some or all of the operators with methods. The semantics and efficiency will be the same; only the syntax will be slightly heavier.

```
for(R r = . . .; r.is_open(); r.advance())
{
```

[11]Although you'll have noted that I managed to overcome my squeamishness in Chapter 25.

```
  . . . = r.current();
}
```

34.5 Functors and Ranges: Coda

This chapter has been taking a somewhat meandering tour of what is at best a mild, often unnoticed, gripe with having to make too many keystrokes, but at worst can be a serious detriment to maintainability.

We've looked at the trade-offs between unrolled loops and functors, and some of the imperfections in the language in terms of the difficulties of using local functors.

We've also seen how we can maximize the generality of functors, by making them compatible with a wider variety of types, via character type parameterizations, type tunneling, and adaptor argument type parameterization. The combination of these techniques represents a substantial resolution of the issues raised in the early part of the chapter.

Last, we looked at the *Range* concept. Not only do ranges provide a nice syntactic convenience, but they also facilitate the unified manipulation of iterator-bound ranges and what we might call purely logical ranges. Furthermore, the application of range filters, to both kinds of ranges, represents a powerful but simple-to-use mechanism for criteria-driven manipulation of ranges.

Update: The latest work on Ranges has incorporated callback enumeration APIs (in the form of the Indirect Range)—something that cannot be handled by STL iterators. See http://www.rangelib.org/ for the latest details.