# 8

# Containers, Iterators, and Algorithms

We have seen the template classes `vector` and `list`, the most heavily used containers in the C++ Standard Template Library. We will examine one more, class `map`, but will only glance at the others. The STL is so consistent that once you've seen the three big containers, you've seen them all. As evidence, look at the totally predictable classes `stack` (pp. 155−157), `string` (pp. 451−454), `queue` (pp. 798−799), and `multimap` (pp. 802−803).

## 8.1 Classes `map` and `pair`

Class `map` has member functions whose arguments and return values are of type `pair`, so we'll do that little class first.

### Class pair

We often need a class that does nothing but hold two public data members, possibly of different types. Since the members are public, we can declare the class as a `struct`. A C++ `struct` is the same as a `class` except that the members are public by default. In particular, a C++ `struct` could have member functions; an example is the constructor in line 7. But a `struct` should have no member functions beyond a constructor that merely copies its arguments into the data members. For anything more elaborate, we probably want a `class`.

```
 1 #include <string>
 2 #include "date.h"
 3 using namespace std;
 4
 5 struct point {
 6     double x, y;
 7     point(double initial_x, double initial_y): x(initial_x), y(initial_y) {}
 8 };
 9
10 struct name {
11     string first, last;
12
13     name(const string& initial_first, const string& initial_last)
14         : first(initial_first), last(initial_last) {}
15 };
16
17 struct event {
18     string name;
19     date d;
20
21     event(const string& initial_name, const date& initial_d)
22         : name(initial_name), d(initial_d) {}
23 };
```

Instead of the above classes, the standard library has one template class named `pair`. The data types `T1` and `T2` must be copy constructible because line 37 calls their copy constructors. (This restruction will come up on p. 800.) The typedefs will be used in the definition of the function objects `select1st` and `select2nd` on pp. 937−938.

```
24 //Excerpt from the header file <utility>.
25
26 //T1 and T2 must be copy constructible.
27
28 template <class T1, class T2>
29 struct pair {
30     typedef T1 first_type;
31     typedef T2 second_type;
32
33     T1 first;
34     T2 second;
35
36     pair(const T1& initial_first, const T2& initial_second)
37         : first(initial_first), second(initial_second) {}
38 };
```

The following program declares objects of two different `pair` types: a `pair<double, double>` in line 18, and a `pair<string, string>` in line 21.

### A helper function

There are three ways of passing a `pair` object to a function.

(1) The following line 25 passes the pair `A` to the function `f`.

(2) If a variable is used only once, it can be an anonymous temporary. Line 26 constructs one of type `pair<double, double>` by calling the constructor for that class.

(3) An easier way to construct an anonymous `pair` is to call the standard library function `make_pair` in line 27. It is a *helper function*, like the ones on pp. 781−783.

The expression `"Independence day"` in line 29 is of data type `const char[17]`, including the terminating `'\0'`. The `make_pair` in that line therefore constructs and returns a `pair<char[17], date>`. If that is what you want, fine. But to get a `pair<string, date>`, we must imitate line 30. It calls the constructors for classes `string` and `date` and then passes these two anonymous objects to `make_pair`.

To accept the diverse types of pairs passed to it in lines 10−14, the function `f` must obviously be very flexible. In fact, it is another template function, like `min` and `make_pair`. It will accept any type of pair to whose data members we can apply the `<<` operator.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/map/pair.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <string>
 4 #include <utility>   //for pair and make_pair
 5 #include "date.h"
 6 using namespace std;
 7
 8 //T1 and T2 must be puttable (i.e., able to be output with <<).
 9
```

```
10 template <class T1, class T2>
11 inline void f(const pair<T1, T2>& p)
12 {
13     cout << "(" << p.first << ", " << p.second << ")\n";
14 }
15
16 int main()
17 {
18     pair<double, double> A(3.0, 4.0);
19     cout << "Point A is (" << A.first << ", " << A.second << ").\n";
20
21     pair<string, string> signer("John", "Hancock");
22     cout << "The signer is " << signer.first << " "
23         << signer.second << ".\n";
24
25     f(A);
26     f(pair<double, double>(3.0, 4.0));
27     f(make_pair(3.0, 4.0));               //construct a pair<double, double>
28
29     f(make_pair(       "Independence Day" , date(date::july, 4, 1776)));
30     f(make_pair(string("Independence Day"), date(date::july, 4, 1776)));
31
32     return EXIT_SUCCESS;
33 }
```

Here is the definition for `make_pair`.

```
34 //Excerpt from <utility>
35
36 //T1 and T2 must be copy constructible.
37
38 template <class T1, class T2>
39 inline pair<T1, T2> make_pair(const T1& t1, const T2& t2)
40 {
41     return pair<T1, T2>(t1, t2);
42 }
```

```
Point A is (3, 4).              lines 18−19
The signer is John Hancock.     lines 21−23
(3, 4)                          line 25: the pair is a pair<double, double>
(3, 4)                          line 26
(3, 4)                          line 27
(Independence Day, 7/4/1776)    line 29: the pair is a pair<char[17], date>
(Independence Day, 7/4/1776)    line 30: the pair is a pair<string, date>
```

**A map is an array whose subscripts need not be integers.**

Like an array, a `map` is a container whose subscripts must all be of the same type. The subscripts of an array must be integers, and non-negative ones at that. But the subscripts of a `map` can be any strict weakly comparable type, with no restrictions as to the values. In our example, the `gravity` map in line 10, the subscript of each element will be a `string` and the value of each element will be a `double`. The <angle brackets> enclose two arguments.

Other languages have the same kind of generalized array. Awk calls it an associative array, Perl and Ruby call it a hash, and Java calls it a `Map` with an uppercase `M`.

The elements of a `map` are different from those of the other containers. Each element of a `vector` or a `list` is a single value, and an iterator for a `vector` or `list` gives us one value at a time. We dereference the iterator with the asterisk in line 4:

```
1       vector<int> v(argument(s) for constructor);
2       vector<int>::iterator it = v.begin();
3       if (!v.empty()) {
4           cout << *it << "\n";   //Output one integer.
5       }
```

But each element of a `map` is a `pair` of values, and an iterator for a `map` gives us a `pair` of values at a time. The `first` and `second` data members of each element are called the *subscript* (or *key*) and the *value.* The header file `<map>` in line 5 includes the `<utility>` header file for class `pair`.

We could dereference a `map` iterator with the asterisks and dots in line 57. But the arrows in 58 and 59 do the job more easily.

Line 12 inserts an element into the map. The element's subscript is `"Mercury"` and its value is `.27`. When we apply a subscript to an object, we are calling the object's `operator[]` member function.

The elements of a `map` do not remain in the order in which they were inserted. By default, they are rearranged by applying the `<` operator to the subscript of each new element. Specifically, we never have a later element whose subscript is `<` that of an earlier element. The subscripts of our `gravity` elements, for example, are `string`'s, and the `<` operator applied to two `string`'s checks for alphabetical order. That's why the loop in lines 55−60 visits the planets alphabetically. The subscripts must be strict weakly comparable. To sort them in a different order, see pp. 793−794. To use a non-built-in data type as the subscript of a `map`, we must first make it possible to apply the `<` operator to it (p. 753).

Having the elements in order allows the lookup in line 42 to be faster. Assuming a map with $n$ elements in no particular order, we would have to loop through $\frac{n}{2}$ of them, on the average, before finding the one we are looking for. But since the elements are in order, we have to examine only $\log_2 n$ of them, even in the worst case, to find the desired one. We first examine the element in the middle, and then divide and conquer. If there were 32 elements, for example, we would have to examine at most only five of them. For logarithms, see p. 773.

Even faster would be a `map` where the elements were looked up by hashing. The official C++ Standard Library doesn't have a `hash_map`, but many vendors supply it.

A `map` assumes that the subscripts can be compared with the `<` operator, but it does not assume they can be compared with `==`. Instead of checking two subscripts for equality, it only attempts to check them for *equivalence.* Two values are said to be equivalent if neither one is less than the other (p. 778). When searching for a subscript, either with the `operator[]` in line 42 or with the `find` on pp. 791−792, the `map` is satisfied when it finds a subscript equivalent to the one being sought.

Line 24 checks for input failure because `weight` would be left holding garbage if the input failed. Line 32 does the same check, because `name` would be left holding the null string. Lines 33−35 also accept an end-of-file as a legitimate way to break out of the loop.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/map/main1.C`

```
 1 #include <iostream>
 2 #include <iomanip>
 3 #include <cstdlib>
 4 #include <string>
 5 #include <map>   //for map; include <utility> for class pair
 6 using namespace std;
 7
 8 int main()
 9 {
10     map<string, double> gravity; //Default constructor constructs empty map.
```

```

    gravity["Mercury"] =  .27;   //gravity.operator[]("Mercury") = .27;
    gravity["Venus"]   =  .85;
    gravity["Earth"]   = 1.00;
    gravity["Mars"]    =  .38;
    gravity["Jupiter"] = 2.33;
    gravity["Saturn"]  =  .92;
    gravity["Uranus"]  =  .85;
    gravity["Neptune"] = 1.12;
    gravity["Pluto"]   =  .44;

    cout << "How many pounds do you weigh on Earth? ";
    double weight;
    if (!(cin >> weight)) {   //if (cin.operator>>(weight).operator!()) {
        return EXIT_FAILURE;
    }

    for (;;) {
        cout << "Type name of planet, or q to quit, and press RETURN: ";

        string name;
        if (!(cin >> name)) {
            if (cin.eof()) {
                break;
            }
            return EXIT_FAILURE;
        }
        if (name == "q") {
            break;
        }

        const double factor = gravity[name]; //gravity.operator[](name);

        if (factor == 0.0) {
            cout << "No planet is named \"" << name << "\".\n";
        } else {
            cout << "You would weigh " << weight * factor
                << " pounds on " << name << ".\n";
        }
    }

    cout << "\n";
    cout << setprecision(2) << fixed; //two digits to right of decimal point

    for (map<string, double>::const_iterator it = gravity.begin();
        it != gravity.end(); ++it) {
        //cout << (*it).first << " " << (*it).second << "\n";
        cout << left << setw(7) << it->first << right << " "
            << setw(4) << it->second << "\n";
    }

    return EXIT_SUCCESS;
}
```

```
How many pounds do you weigh on Earth? 150
Type name of planet, or q to quit, and press RETURN: Mars
You would weigh 57 pounds on Mars.
Type name of planet, or q to quit, and press RETURN: Mongo
No planet is named "Mongo".
Type name of planet, or q to quit, and press RETURN: q

Earth   1.00    applying the operator < to string's yields alphabetical order
Jupiter 2.33
Mars    0.38
Mercury 0.27
Mongo   0.00    Line 42 unintentionally inserted Mongo.
Neptune 1.12
Pluto   0.44
Saturn  0.92
Uranus  0.85
Venus   0.85
```

**Two typedefs that make a map easier to use**

Insert lines 64 and 65 immediately before the declaration for `gravity` in the above line 10.

```
64      typedef map<string, double> map_t;
65      typedef map_t::value_type pair_t; //another name for pair<string, double>
```

The `map_t` typedef in the above line 64 will let us simplify the above line 10 to

```
66      map_t gravity;                  //Default constructor constructs empty map.
```

We can also simplify the above lines 55−56 to

```
67      for (map_t::const_iterator it = gravity.begin(); it != gravity.end(); ++it) {
```

Every container in the C++ Standard Library has a public member named `value_type`, giving the data type of each element stored in the container. Our own containers also had the `value_type` member:

(1) class `stack`, pp. 153−154

(2) class `node`, p. 214

(3) We assume that any `CONTAINER` class has a `value_type` in line 18 of `typename.C` on p. 675.

(4) class `terminal`, pp. 742−743, ¶ (12).

(5) The next version of class `node` should have had a `value_type` on p. 805.

(6) The final version of class `terminal` will have a `value_type` in line 20 of `terminal.h` on p. 970.

A `map` element is actually a pair of values, so a `map` `value_type` is a `pair`. In our case, it is a `pair<string, double>`, for which the typedef in the above line 65 is a convenient name. I could have written 65 as

```
68      typedef pair<string, double> pair_t;
```

but I didn't want to repeat the arguments `<string, double>` in lines 64 and 65.

**A faster way to insert an element into a map**

The above program showed a fast and dirty way to construct a map, insert elements, and look them up. (Fast to write, that is, but slow to execute.) We can improve all three operations.

We would expect that the member function `operator[]` in the above line 12 would create an element for Mercury and initialize its value to `.27`. Unfortunately, this is not what happens. It creates an element whose value is initialized to `0.0`, and then assigns `.27` to the value. Where did the momentary `0.0` come from? This is a `map` that holds `double`’s, so the `operator[]` called the default constructor for type `double`. See the default constructors for the built-in types on p. 660.

Our values are merely `double`’s, so the initialization and reassignment don’t take long. For other data types, however, it would be faster to initialize to the correct value once and for all. To do this, change the above line 12 to the call to the `insert` member function in lines 69−78.

In the following line 69, the argument of `insert` is a `pair_t` consisting of a subscript and a value: a `pair<string, double>`. The expression `pair_t("Mercury", .27)` constructs an anonymous object of this type, which is then passed to `insert`. Note that line 69 could not have said
`make_pair("Mercury", .27)`, because that would have constructed a
`pair<char[8], double>`.

The return value of `insert` is a `pair` of a different type, consisting of a `map_t::iterator` and a `bool`. The `bool` will be true if the insertion was successful. If so, the iterator will refer to the newly-inserted pair containing `"Mercury"` and `.27`:

```
69      pair<map_t::iterator, bool> p = gravity.insert(pair_t("Mercury", .27));
70
71      if (p.second) {                        //p.second is a bool
72          map_t::iterator it = p.first;      //p.first is a map_t::iterator
73          string s = it->first;              //subscript
74          double d = it->second;             //value
75          cout << "Inserted the pair \"" << s << "\", " << d << ".\n";
76      } else {
77          cerr << "Not inserted.  \"Mercury\" must have already been in the map.\n";
78      }
```

The above lines 72−74 stored the iterator, subscript, and value into the variables `it`, `s`, and `d`. We can also use them directly in lines 83−84.

```
79      pair<map_t::iterator, bool> p = gravity.insert(pair_t("Mercury", .27));
80
81      if (p.second) {
82          cout << "Inserted the pair \""
83              << p.first->first << "\", "
84              << p.first->second << ".\n";
85      } else {
86          cerr << "Not inserted.  \"Mercury\" must have already been in the map.\n";
87      }
```

### Find an element without contaminating the map

We’re lucky that no planet has zero gravity. The above line 44 is unable to distinguish between an unsuccessful lookup and a planet whose gravity is `0.0`. Even worse, if the user types a nonexistent name such as `Mongo`, line 42 will create an element for `Mongo` and initialize its value to `0.0`. (As before, the initial value comes from the default constructor for type `double`.) In a future example, it may be beneficial for `operator[]` to construct a new element and initialize it to the default value (p. 796). But for the present, we want to look up a string without inadvertently creating a new element.

To do this, change the above lines 42−49 to

```
88          map_t::const_iterator it = gravity.find(name);
89
90          if (it == gravity.end()) {
91              cout << "No planet is named \"" << name << "\".\n";
92          } else {
93              //it->first is the subscript (a string),
94              //it->second is the value (a double).
95              cout << "You would weigh " << weight * it->second
96                  << " pounds on " << name << ".\n";
97          }
```

### A faster way to construct a map

The default constructor in the above line 10 constructed an empty map, which was then populated by the calls to `operator[]` in lines 12−20. But the two-argument constructor in the following line 25 is a faster way to make a map: it will be born with the nine elements already in it. (Older versions of Microsoft did not have the two-argument constructor for class `map`.)

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/map/main2.C`

```
 1 #include <iostream>
 2 #include <iomanip>
 3 #include <cstdlib>
 4 #include <string>
 5 #include <map>   //includes <utility>
 6 using namespace std;
 7
 8 int main()
 9 {
10     typedef map<string, double> map_t;
11     typedef map_t::value_type pair_t; //another name for pair<string,double>
12
13     const pair_t a[] = {
14         pair_t("Mercury",  .27),
15         pair_t("Venus",    .85),
16         pair_t("Earth",   1.00),
17         pair_t("Mars",     .38),
18         pair_t("Jupiter", 2.33),
19         pair_t("Saturn",   .92),
20         pair_t("Uranus",   .85),
21         pair_t("Neptune", 1.12),
22         pair_t("Pluto",    .44)
23     };
24     const size_t n = sizeof a / sizeof a[0];
25     const map_t gravity(a, a + n);
26
27     cout << "How many pounds do you weigh on Earth? ";
28     double weight;
29     if (!(cin >> weight)) {
30         return EXIT_FAILURE;
31     }
32
33     for (;;) {
34         cout << "Type name of planet, or q to quit, and press RETURN: ";
35
36         string name;
37         if (!(cin >> name)) {
38             if (cin.eof()) {
39                 break;
40             }
41             return EXIT_FAILURE;
```

```
        }
        if (name == "q") {
            break;
        }

        const map_t::const_iterator it = gravity.find(name);
        if (it == gravity.end()) {
            cout << "No planet is named \"" << name << "\".\n";
        } else {
            cout << "You would weigh " << weight * it->second
                << " pounds on " << name << ".\n";
        }
    }

    cout << "\n";
    cout << setprecision(2) << fixed;

    for (map_t::const_iterator it = gravity.begin();
        it != gravity.end(); ++it) {
        cout << left << setw(7) << it->first << right << " "
            << setw(4) << it->second << "\n";
    }

    return EXIT_SUCCESS;
}
```

```
How many pounds do you weigh on Earth? 150
Type name of planet, or q to quit, and press RETURN: Mars
You would weigh 57 pounds on Mars.
Type name of planet, or q to quit, and press RETURN: Mongo
No planet is named "Mongo".
Type name of planet, or q to quit, and press RETURN: q

Earth   1.00
Jupiter 2.33
Mars    0.38
Mercury 0.27
Neptune 1.12    find did not insert Mongo.
Pluto   0.44
Saturn  0.92
Uranus  0.85
Venus   0.85
```

**▼ Homework 8.1a: sort the subscripts in a different order**

Add a third template argument, `greater<string>`, to the data type of the `gravity` map. Note that the argument is the name of a data type; we saw it on pp. 769−770. Given this template argument, the third function argument of the constructor will default to `greater<string>()`.

```
#include <string>
#include <map>
#include <functional>   //for class greater
using namespace std;

    map<string, double, greater<string> > gravity(a, a + n);
```

Instead of applying the `<` operator to the subscripts, the map will now apply the `operator()` member function of an object of class `greater<string>` to the subscripts. In what order does the `for` loop visit the elements now?

▲

**▼ Homework 8.1b: why doesn't this map compile?**

Line 18 constructs an empty map. Line 19 looks for an element whose subscript is `key(10)`. There is no such element, so line 19 should create one just as the above `main1.C` created an element for `Mongo`.

What do we have to do to make the program compile? Hint: we don't need the `operator==`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/map/comparable.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <map>
 4 using namespace std;
 5
 6 class key {
 7     int i;
 8 public:
 9     key(int initial_i): i(initial_i) {}
10
11     friend bool operator==(const key& key1, const key& key2) {
12         return key1.i == key2.i;
13     }
14 };
15
16 int main()
17 {
18     map<key, int> m;
19     cout << m[key(10)] << "\n";
20     return EXIT_SUCCESS;
21 }
```

▲

**▼ Homework 8.1c:**
**Version 4.4 of the Rabbit Game: a map instead of a searching loop**

Remove the array of structures from `manual::decide`. Replace it by a map takes a `char` and gives us back a pair of integers.

```
 1     map<char, pair<int, int> > keypad;
```

`manual::decide` will no longer need the searching loop in lines 36−55 of `wolf.C` on pp. 198−199. `manual.C` will have to include the header file `<map>` and say `using namespace std;` if it does not already.

The above line 1 shows how natural it is for templates to nest. If this freaks you out, however, we can build the data type of the `map` with the typedefs in lines 4 and 6. Each pair of `dx`, `dy` offsets will be stored in the `pair<int, int>` in line 4. Each element of the `map` in line 6 will be the `pair` in line 7.

```
 2 //Excerpt from manual.C, showing part of manual::decide.
 3
 4     typedef pair<int, int> step_t;
 5
 6     typedef map<char, step_t> map_t;
 7     typedef map_t::value_type pair_t; //another name for pair<char, step_t>
 8                                       //which is another name for
 9                                       //pair<char, pair<int, int> >
10
11     static const pair_t a[] = {
12         pair_t('h', step_t(-1,  0)),           //left,
13         pair_t('j', step_t( 0,  1)),           //down,
14         //etc.
15     };
16     static const size_t n = //etc.
17     static const map_t keypad(a, a + n);
18
19     if (const char k = get the keystroke, if any) {
20         const map_t::const_iterator it = keypad.find(k);
21         if (we found k) {
22             put the two offsets into *dx and *dy
23                 (hint: it->second.first is the horizontal offset);
24             return;
25         }
26
27         punish();   //Punish user who pressed an illegal key.
28     }
29
30     //Arrive here if user pressed no key, or pressed an illegal key.
31     *dx = *dy = 0;
32 }
```

The `typedef` `step_t` in the above line 4 is only temporary. In the world to come, a variable that holds a `dx`, `dy` offset will eventually be of data type `terminal_t::difference_type` (p. 967). This will also be the return type of the functions `wabbit::decide`, `difference`, and `step`.
▲

▼ **Homework 8.1d:**
**Version 4.5 of the Rabbit Game: a map instead of a counting loop**

To make the program faster, let `game::count` be a map.

```
1     map<char, int> count;
```

Its member function `operator[]` will take a `char` representing a species (`'r'` for `rabbit`) and return the number of objects of this species that currently exist in the game. We will call only `operator[]`, not `find`; see below.

First, however, note that the above declaration has been simplified. Our game is no longer hardwired to run on a terminal that holds only `char`'s. And we should not assume that an `int` is big enough to hold the maximum number of animals on the master list. A professional would therefore declare the map as

```
2     map<terminal_t::value_type, master_t::size_type> count;
```

employing four typedefs:

(1) `terminal_t` in line 112 on p. 744;

(2) `value_type` on p. 790;

(3) `master_t` on p. 465;

(4) `size_type` on pp. 433−434 and 434.

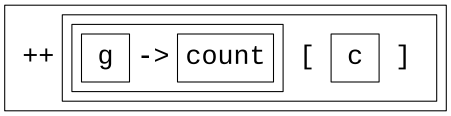
See how natural it is to have data types with standardized names (`value_type`, `size_type`) for each container?

`count` will be a private, non-static data member of class `game`. Construct it after the master list: it would make no sense to count the animals on the list before there was a list. The constructor for class `game` will pass no arguments to the constructor for `count`, just as it passes no arguments to the constructor for `master`. Since no array is passed to the constructor for `count`, don't bother to create the `pair_t` typedef for it. And since we will not be not iterating through `count`, don't bother to create the `map_t` typedef for it either.

Every constructor for class `wabbit` will say

```
    ++g->count[c];                          //++g->count.operator[](c);
```

immediately after inserting the address of the newborn `wabbit` into the master list. (We currently have only one constructor for class `wabbit`; the copy constructor is undefined.) Note that the `++` does not increment `g`. It increments `g->count[c]`.



The first time that the member function `operator[]` is called with a given character, it will create an element for that character and initialize its value to 0. In fact, a naïve implementation of the `operator[]` function for class `map<char, int>` would be

```
int& map<char, int>::operator[](char c)
{
    const map<char, int>::iterator it = find(c);
    if (it != end()) {
        //it->first is the char, it->second is the int.
        return it->second;
    }

    const pair<map<char, int>::iterator, bool> p =
        insert(pair<char, int>(c, int()));

    if (p.second) {   //The insertion was successful.
        //p.first->first is the char, p.first->second is the int.
        return p.first->second;
    }

    //Let's hope we never get here.
}
```

Back on pp. 791−792, `operator[]` contaminated the map when it constructed a new element ("Mongo"). Here, though, it is exactly what we want. The first call to the constructor for `rabbit_t` will pass an `'r'` to the constructor for class `wabbit`, which will pass the `'r'` to the `operator[]` member function of the game's `count`, which will create an element for `'r'` and initialize it to 0. The `++` will then increment it to 1. Subsequent calls to the constructor for `rabbit_t` will not perform the initialization to 0, but they will perform the increment.

The destructor for class `wabbit` will say

```
    --g->count[c];                          //--g->operator.operator[](c);
```

immediately before removing the address of the dying `wabbit` from the master list.

Remove the member function `game::count`. The code that recognizes when the game is over (lines 19−27 of `game.C` on p. 570) will now say `count['r']` instead of `count('r')`.
▲

**▼ Homework 8.1e:**
**Version 4.6 of the Rabbit Game: a map instead of the big switch**

The animals of different species were constructed with a big `switch` statement; see lines 21−41 of `game.C` on p. 569. But a `switch` should be used only when each `case` contains different code. Our `case`’s were almost identical.

With a map, the big `switch` can be reduced to the statements in lines 23−29 below. Even better, these statements will not have to change when a new species is added.

(1) Our map will take a character and return a pointer to a function that constructs an animal of the corresponding species. I wish the function could be the constructor for each species. But although there are pointers to other member functions, there is no such thing as a pointer to a constructor or destructor.

Define the following template function in the `grandchild.h` header file, inspired by the `make_pair` function in pp. 786−787. It is not a member function or friend of any class.

```
 1 template <class MOTION, class RANK, char C>
 2 inline void make_grandchild(game *initial_g,
 3     unsigned initial_x, unsigned initial_y) {
 4
 5     new grandchild<MOTION, RANK, C>(initial_g, initial_x, initial_y);
 6 }
```

There will be one instantiation of this function for each species of animal. To point to an instantiation, we can use a plain old pointer to a function: mercifully, there is no such thing as a “pointer to an instantiation of a template function”. Here is the declaration for a pointer `p` that can point to the instantiation of `make_grandchild` for any species:

```
 7     //p is a pointer to function.
 8     void (*p)(game *, unsigned, unsigned);
```

To get the name of the data type of this pointer, we remove the semicolon and the name of the pointer.

```
 9     void (*)(game *, unsigned, unsigned)
```

This is the data type plugged into line 10.

(2) Define the following map at the start of `game::game`, before the rectangular array of characters. It takes a `char` and returns a pointer to the instantiation of `make_grandchild` for the corresponding species.

The `map_t` and `pair_t` in lines 10 and 11 are the two typedefs from p. 790. The first `’W’` in line 14 is the character in the rectangular array in `game::game`; the second is the character that the user sees on the screen.

```
10     typedef map<char, void (*)(game *, unsigned, unsigned)> map_t;
11     typedef map_t::value_type pair_t;
12
13     static const pair_t species[] = {
14         pair_t(’W’, make_grandchild<manual, predator_t, ’W’>), //wolf
15         pair_t(’r’, make_grandchild<brownian, victim_t, ’r’>), //rabbit
16         //etc.
17     };
18     static const size_t n = sizeof species / sizeof species[0];
19     static const map_t m(species, species + n);
```

(3) Change the nested loops in lines 18−44 of `game.C` on p. 569 to

```
20     for (size_t y = 0; y < ymax; ++y) {
21         for (size_t x = 0; x < xmax; ++x) {
22             if (term.in_range(x, y) && a[y][x] != ’.’) {
23                 const map_t::const_iterator it = m.find(a[y][x]);
```

```
                if (we didn’t find the character a[y][x]) {
                    construct and throw an exception;
                }

                //Call the make_grandchild function for this species.
                it->second(this, x, y);   //or (*it->second)(this, x, y);
            }
        }
    }
```

Class `game` curently “knows about” every class derived from class `wabbit`. The file `game.C` includes the headers for the derived classes, and has to be recompiled whenever they change. But we are now in a position to eliminate these dependencies. The rectangular array of characters and the `map` can be passed as arguments to the constructor for class `game`. Think about this but don’t do it.

▲

## Class queue

Thus far, our game has been limited to at most one `manual` animal. We will now permit more than one, allowing us to have more than one human player. To do this, we will need a container called a *queue*. A stack is “last hired, first fired”; a queue is “first hired, first fired”.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/queue.C`

```
#include <iostream>
#include <cstdlib>
#include <stack>
#include <queue>
using namespace std;

int main()
{
    stack<int> s;   //Construct an empty stack.

    s.push(10);
    s.push(20);
    s.push(30);

    cout << s.top() << "\n";
    s.pop();

    cout << s.top() << "\n";
    s.pop();

    cout << s.top() << "\n";
    s.pop();

    cout << "\n";

    queue<int> q;   //Construct an empty queue.

    q.push(10);
    q.push(20);
    q.push(30);

    cout << q.front() << "\n";
33      q.pop();
34
35      cout << q.front() << "\n";
36      q.pop();
37
38      cout << q.front() << "\n";
39      q.pop();
40
41      cout << "\n" << q.empty() << "\n";
42
43      return EXIT_SUCCESS;
44  }
```

```
30    stack is LIFO: last in, first out
20
10

10    queue is FIFO: first in, first out
20
30

1    line 41: a bool is output as a 1 or 0.
```

**▼ Homework 8.1f: use a map to allow more than one animal to be manual**

Each `manual` animal will have to respond to a different set of keystrokes. For example, one might respond to

| | |
|---|---|
| `h` | *left* |
| `j` | *down* |
| `k` | *up* |
| `l` | *right* |

Another might respond to eight of the keys on the numeric keypad:

| | |
|---|---|
| `1` | *lower left* |
| `2` | *down* |
| `3` | *lower right* |
| `4` | *left* |
| `6` | *right* |
| `7` | *upper left* |
| `8` | *up* |
| `9` | *upper right* |

But now we have a new problem. Suppose the user typed the keystroke `8` when it was the turn of the `h-j-k-l` `manual` to move. What should that manual do with the keystroke? How could the `h-j-k-l` `manual` deliver the keystroke to the `1-2-3-4-6-7-8-9` `manual`? And how could the `h-j-k-l` `manual` beep if there were no `manual` authorized to receive this keystroke?

There will have to be a central clearinghouse for all incoming keystrokes; the place to put it is in the `game` object. The `game` will dispatch each keystroke to the appropriate `manual`, issuing a beep if there is none.

(1) Give class `game` the following four private members. As on p. 794, a "step" will be a pair of numbers describing the direction in which a `manual` should go in response to a keystroke.

```
1       typedef pair<int, int> step_t;
```

A “dispatch” is a marching order. It tells one particular `manual` to take a step in a certain direction. The dispatch can contain only an address, not the `manual` itself, because a `manual` is not copy constructible (p. 786).

```
typedef pair<manual *, step_t> dispatch_t;
```

`game.h` will need a forward declaration for the word `manual`, just like the one for the word `wabbit`.

A “dispatcher” issues a dispatch in response to a character from the keyboard.

```
typedef map<char, dispatch_t> dispatcher_t;
```

The `game` should have one dispatcher.

```
dispatcher_t dispatcher; //non-static private data member of class game
```

`game.h` will have to include the header file `<map>`. The constructor for `game` should pass no arguments to the constructor for the dispatcher.

(2) The rectangular array of characters in `game::game` gives us the `x`, `y` coördinates of each animal we construct, including the `manual` animals. But there is no pleasing way for this array to also contain each `manual`’s list of keystrokes and the corresponding steps. We will have to put this information into a separate data structure.

And there’s another problem. An object must be ready to assume its responsibilities by the time its constructor has completed. This means that each `manual` object must its list of keystrokes and steps before this happens. We could pass the list as an extra argument to the constructor for `manual`, but it would be simpler if the number of constructor arguments for every species of animal remained the same. The constructor for `manual` will have to claim its keystrokes and steps by calling a function, rather than by receiving an extra argument.

Add the following private, non-static, non-inline member function to class `game`.

```
void claim(manual *m, unsigned x, unsigned y);
```

The constructor for class `manual` will call `claim`, passing it the arguments `this`, `initial_x`, `initial_y`. Since the function is private, class `manual`, like class `wabbit`, will have to be a friend of class `game`.

The function `game::claim` will contain arrays such as the following.

```
typedef pair<char, step_t> keystroke_t;

static const keystroke_t wolf1[] = {
    keystroke_t('h', step_t(-1,  0)), //left
    keystroke_t('j', step_t( 0,  1)), //down
    //etc.
};

static const keystroke_t wolf2[] = {
    keystroke_t('1', step_t(-1,  1)), //lower left
    keystroke_t('2', step_t( 0,  1)), //down
    //etc.
};

//etc.

struct animal {
    size_t x; //subscripts of the manual in char array in game::game
    size_t y;
    const keystroke_t *begin;
    const keystroke_t *end;
27     };
28
29     static const animal a[] = {
30         {10, 10, wolf1, wolf1 + sizeof wolf1 / sizeof wolf1[0]},
31         {15, 15, wolf2, wolf2 + sizeof wolf2 / sizeof wolf2[0]},
32         //etc.
33     };
34     static const size_t n = sizeof a / sizeof a[0];
```

`game::claim` will load the above values into the dispatcher. Lines 47−48 construct and insert a pair whose first member is a `char` and whose second is a `dispatch_t`. We then check the return value's `second` member, which is a `bool`.

```
35     for (const animal *p = a; p < a + n; ++p) {
36         if (p->x == x && p->y == y) {
37             for (const keystroke_t *q = p->begin; q < p->end; ++q) {
38                 const dispatcher_t::const_iterator it =
39                     dispatcher.find(q->first);
40
41                 if (it != dispatcher.end()) {
42                     construct and throw an exception:
43                     there's already a manual
44                     that responds to the character q->first;
45                 }
46
47                 if (!dispatcher.insert(make_pair(q->first,
48                     dispatch_t(m, q->second))).second) {
49                     construct and throw an exception:
50                     the insert failed for some other reason;
51                 }
52             }
53             return;
54         }
55     }
56
57     construct and throw an exception:
58         there should be no manual at coördinates x, y;
59 }
```

(3) Give class `manual` the following two private members. `manual.h` will have to include the header file `<queue>`, and say `using namespace std;`. `manual::push` can mention the private member `game::step_t` because class `manual` is a friend of class `game`.

```
60     queue<game::step_t> q;
61     void push(const game::step_t& step) {q.push(step);}
```

(4) Just before giving every animal a chance to move, let `game::play` distribute all the outstanding keystrokes to the manuals.

```
62     for (;; term.wait(250) {
63
64         while (const char c = term.key()) {
65
66             const dispatcher_t::const_iterator it =
67                 dispatcher.find(c);
68
69             if (it == dispatcher.end()) {
```

```
70                  term.beep(); //No manual responds to this key.
71              } else {
72                  //it->first is the character c
73                  //it->second is a dispatch_t
74                  //it->second.first is a pointer to a manual
75                  //it->second.second is a step_t
76                  const dispatch_t& dispatch = it->second;
77                  dispatch.first->push(dispatch.second);
78              }
79          }
80
81          for (master_t::const_iterator it = master.begin(); //etc.
```

You can insert code at the above line 65 to make the `game::play` function return if the user has pressed `q` for “quit”. The `dispatch` reference in the above line 76 is merely a notational convenience. Without it, line 77 would have to be written as follows.

```
82                  it->second.first->push(it->second.second);
```

Declare `game::play` to be a friend of class `manual`, with a comment saying that it is to allow `game::play` to call `manual::push`.

(5) Finally, `manual::decide` will contain only the following. The `step` in line 86 can be a reference as long as we do not attempt to use it after the `pop` in 89. See pp. 156−157.

```
83      if (q.empty()) {
84          *dx = *dy = 0;
85      } else {
86          const game::step_t& step = q.front();
87          *dx = step.first;
88          *dy = step.second;
89          q.pop();
90      }
```

(6) The `wabbit::key` function is no longer used, so you can remove it.

One last problem. The `pop` in the above line 89 changes the `q` data member of class `manual`. This will not compile, since `decide` is a `const` member function of that class. One choice would be to let the `decide` member function of class `wabbit` and all of its descendants be non-`const`. The alternative would be to declare `q` to be a *mutable* data member of class `manual`:

```
91      mutable queue<game::step_t> q;
```

This permits the value of the data member `q` to be changed by a `const` member function of class `manual`. Another `mutable` data member will be on p. 751.
▲

### ▼ Homework 8.1g: fly in formation

If two or more `manual`’s responded to the same keystroke, they would fly in formation. To realize this vision, change the `dispatcher` from a `map` to a `multimap`. A `multimap` is like a `map`, except that it can contain two or more pairs with the same `first` data member.

All of these pairs will be stored consecutively in the `multimap`, since the elements of a `multimap`, like those of a `map`, are ordered (by default) by applying the `<` operator to the subscripts.

The `insert` member function of a `multimap` returns an iterator referring to the newly-inserted element. Ignore the return value. Just assume that the insertion was successful.

Instead of calling the `find` member function of a `map` dispatcher, call the `equal_range` member function of a `multimap` dispatcher. It will return a pair of iterators referring to the beginning and end of the range of elements that have the desired subscript.

Include the header file `<map>` for class `multimap`.

```
//Excerpt from game::play

const pair<dispatcher_t::const_iterator, dispatcher_t::const_iterator>
    range = dispatcher.equal_range(c);

if (range.first == range.second) {   //The range is empty.
    term.beep();   //No manual responds to this key.
} else {
    for (dispatcher_t::const_iterator it = range.first;
        it != range.second; ++it) {

        //as in the above lines 76-77,
        const dispatch_t dispatch = it->second;
        dispatch.first->push(dispatch.second);
    }
}
```

▲

## 8.2 Endow a Data Structure with an Iterator

A *data structure* is any source of or destination for data, consisting of a series of values all of the same data type. An array or linked list in memory, an input or output file on the disk, or a TCP/IP connection to another host are all examples of data structures. The data structure may be read-only, write-only, or read/write; sequential access or random access. The values read from or written to the data structure are called the *elements.*

The Standard Template Library (STL) contains a wealth of functions, including `sort`, `copy`, `find`, and my own favorite, `random_shuffle`. They can read and write the elements in any data structure that complies with the library's requirements. These may be stated simply: the data structure must have a type of *iterator* that can loop through it. Classes `vector`, `list`, and `map` all have iterators. Even the humble array has an iterator, for a pointer is a perfectly legitimate iterator.

A data structure endowed with an iterator is called a *container.* Each type of container requires a different type of iterator. A template function which will accept iterators of many types is called an *algorithm.* The functions in the STL are algorithms. Note that the arguments passed to an algorithm are not the containers themselves, but iterators that refer to elements in the containers.

We will turn three data structures into STL-compliant containers by endowing them with iterators. The ideal to which we aspire is to take an iterator looping through a container, and dress it up with the operators `!=`, `*`, and `++` to make it look like a pointer looping through an array.

```
int a[] = {10, 20, 30};
const size_t n = sizeof a / sizeof a[0];

for (int *p = a; p != a + n; ++p) {
    cout << *p << "\n";
}
```

Having a uniform notation for all containers is desirable in itself. It will also make their content accessible to the algorithms.

The three data structures were chosen to show different approximations to our ideal. Our purpose is show that each data structure can be turned into a container by means of a few superficial additions, without disturbing any existing code.

### 8.2.1 A Singly-Linked List

Consider the following circa-1985 linked list. The nodes are defined in lines 6−8 and 24 of `node.h` on p. 806, and the list of them is created and destroyed in lines 10−18 and 58−62 of `main.C` on pp. 807−808. The whole thing could have been a C programming exercise from a generation ago.

Our purpose is to make it STL-compliant with the least possible modification. We will resist the temptation to rewrite the data structure in C++. As a concession to contemporary expectations of comfort, we provide only two amenities: the constructor in lines 10−11 of `node.h`, and the calls to `new` instead of `malloc` in lines 11−13 of `main`.

The loop in lines 15−17 of `main.C` is the traditional way to access this data structure. The implementation of the linked list lies naked to our gaze: the pointer `p` to a structure, the arrows that dereference the pointer, the names of the fields, and the comparison to zero. The loop demands specialized expertise with structure and pointers thereto, and the `node` structure in particular.

To make the data structure STL-compliant, we create the `iterator` class in lines 13−29 of `node.h`. We give it the last name `node` by nesting it inside of class `node`, just as class `bill` was nested inside of class `clinton` on p. 420. The iterator class has the following five trimmings. Some are members, some are friends, and some are neither. The first four are used in our first exhibit, the loop in lines 24−26 of `main.C`. The last one will be used by our second exibit, the calls to the algorithms in lines 33−56.

(1) The `begin` iterator refers to the first element of the container. The `end` element refers to the slot in this container where the non-existent ‘‘element’’ after the last element would be if there was one (which of course there isn’t). See lines 20−21 of `main.C`. If the container holds no elements, `begin` has the same value as `end`.

(2) The `==` and `!=` operators compare two iterators (lines 20−22 and 33−36 of `node.h`).

(3) The `*` operator returns a read/write reference to the `value` in the node to which the iterator refers (line 17 of `node.h`). Do not apply the `*` to the `end` iterator.

(4) The prefix and postfix `++` operators move the iterator forward one element (lines 18 and 26−31 of `node.h`). Do not apply the `++` to the `end` iterator.

(5) We also write a specialization of the template class `iterator_traits` for this type of iterator, containing five public typedefs (lines 38−47 of `node.h`).

With these trimmings, we can write the loop in lines 24−26 of `main.C`. The comments in 23 and 25 show what these lines are actually doing. The pointer `p` that lay exposed in lines 15−17 is still there, but is now discretly hidden as the private data member `p` in line 14 of `node.h`. The code that bristled with arrows is still there, but is now packaged in the bodies of the functions (member functions, friends, and neither) in lines 16−36 of `node.h`. We are left with a loop that is totally generic. The loop would still work, completely unchanged, if the name of any other container were inserted in front of the double colon in line 24: `vector<int>`, `list<date>`, etc.

We can do more with the expression `*it` than just print it in line 25 of `main.C`. We can assign to it in line 31, because the `operator*` member function in line 17 of `node.h` returns a read/write reference. Incidentally, lines 30−31 of `main.C` can be combined to

```
	*++it = 20;                    //it.operator++().operator*() = 20;
```

But lines 29−30 cannot be combined to

```
	node::iterator it = begin + 1;  //operator+(begin, 1)
```

since we have not written an `operator+`.

Two `++` operators for class `node::iterator` are implemented in lines 18 and 26−31 of `node.h`. As usual, the postfix `++` calls the prefix `++` to do most of its work (line 29). Similarly, `operator!=` calls `operator==` to do most of *its* work (line 35).

Our iterators give us access to the `value` member of each `node`, but not to the address of each node. The loop in lines 58−62 of `main.C` must therefore be written in terms of pointers, not iterators. We take care to avoid the increment of death (pp. 444−445).

### A specialization of class iterator_traits

When we create a new class of iterator, we must also, at least for now, create a specialization of class `iterator_traits` for it. The `<iterator>` header file in line 3 of `node.h` contains the general template for class `iterator_traits`. Lines 39−46 of `node.h` create the specialization `iterator_traits<node::iterator>`.

Like `cin` and `cout`, the general template belongs to namespace `std` (p. 20). A specialization must belong to the same namespace as its general template, so we enclose it in lines 38 and 47. See the namespace declarations on p. 1021.

Class `iterator_traits` should be used *only* by an algorithm. It gives the algorithm five vital facts about the data type of an iterator that the algorithm receives as an argument.

(1) To find out if iterators of this type can be decremented, the algorithm can check the `iterator_category` member in line 41 of `node.h`. In our example, the answer is no. The ‘‘iterator categories’’ and their ‘‘tags’’ will be explained later in this chapter.

(2) An algorithm can use its iterator argument to access an element in a container. The algorithm can store a copy of the element, a pointer to the element, or a reference to the element, in variables of the types given by the members `value_type`, `pointer`, and `reference` in lines 42, 44, and 45 of `node.h`. For an example, see the `value_type` in line 6 of the algorithm `iter_swap` on p. 764.

(3) Our linked list in `main.C` contained three elements, originally holding the values `10`, `15`, `30`. The directed distance from the first to the last was +2 elements; from the last to the first was −2 elements. Thus, a signed integral value must be used to hold the distance between the elements to which two iterators refer. Should the value be `int`, `long`, or something more exotic? The `difference_type` member in line 43 of `node.h` gives the data type that should be used. This type will be large enough to hold the longest possible distance, but no larger than necessary. I decided that our `difference_type` would be a typedef for `ptrdiff_t`, the data type of the distance between two pointers in C or C++. `ptrdiff_t` in turn is a typedef in the header file `<cstddef>`, which is included by the header file `<iterator>` in line 3 of `node.h`. Examples are the variable in line 64, and the function return value in line 61, in the algorithms on p. 810. A `difference_type` can also used to count the number of elements. Examples are the variable in line 77, and the function return value in line 74, on p. 810.

The five members of `iterator_traits` will correspond to the five arguments of the template class `iterator` on p. 813. Three of these arguments—the ones corresponding to our members `difference_type`, `pointer`, and `reference`—will have default values, so they must be declared last. The members of `iterator_traits` could have been declared in any order, but for consistency we order them in agreement with the arguments of `iterator`.

### Resist the temptation to rewrite everything!

The data type of the payload (`int`) was mentioned in lines 7, 10, 17, 42, and 44−45 of `node.h`. We should have written it only once, in a typedef named `value_type` at line 6½. Better yet, a `node` should have been a `node<int>` so we could have nodes for other data types.

Furthermore, we could have avoided the litany of

```
head =
head =
head =
```

when lines 11−13 of `main.C` created the list. We should have given class `node` a constructor that inserts the newborn node at the head of the list. The variable `head` should have been a static data member of class `node`, updated by the constructor.

Even better, we should change the scheme to let us have more than (or less than) one list. We should have made a new class, `singly_linked_list`. `head` should have been a non-static, private data member of this class; and `begin` and `end` should have been non-static, public member functions, like the `begin` and `end` of classes `vector`, `list`, and `map`. Classes `iterator` and `node` should have been members of `singly_linked_list`.

But we don’t want to rewrite the whole program in contemporary C++. We just want to slap on an iterator to make the content of the list accessible to the STL algorithms.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/node_container/node.h`

```
#ifndef NODEH
#define NODEH
#include <iterator>   //for iterator_traits and forward_iterator_tag
using namespace std;

struct node {
    int value;
    node *next;

    node(int initial_value, node *initial_next)
        : value(initial_value), next(initial_next) {}

    class iterator {
        node *p;
    public:
        iterator(node *initial_p = 0): p(initial_p) {}
        int& operator*() const {return p->value;}
        iterator& operator++() {p = p->next; return *this;}   //prefix

        friend bool operator==(const iterator& it1, const iterator& it2) {
            return it1.p == it2.p;
        }
    };
};

inline const node::iterator operator++(node::iterator& it, int)//postfix
{
    const node::iterator old = it;
    ++it;   //it.operator++();
    return old;
}

inline bool operator!=(const node::iterator& it1, const node::iterator& it2)
{
    return !(it1 == it2);   //return !operator==(it1, it2);
}

namespace std {
    template <>
    struct iterator_traits<node::iterator> {
        typedef forward_iterator_tag iterator_category;
        typedef int value_type;
        typedef ptrdiff_t difference_type;
        typedef int *pointer;
        typedef int& reference;
    };                         //semicolon at end of class
}                              //no semicolon at end of namespace
#endif
```

**Pass the content of a container to an algorithm**

Now that our data structure is an STL-compliant container, we can pass its content to (most of) the algorithms in the STL. Lines 33−56 are a preview. The tasks commonly done with simple loops and `if` statements—sorting, searching, counting, comparing—have all been written once and for all in the STL. We will never have to write these loops again.

Why do we have to provide the zero for the `accumulate` in line 43? Why isn't zero the default starting point? Well, we don't always want to start at zero. For multiplication, we would want to start at 1. Observe that the template class `multiplies` in lines 55−58 of the excerpts on p. 810 is just like the template class `greater` on p. 770. The expression `multiplies<int>()` in line 46 of `main.C` constructs an anonymous object of this class. The object is then passed to `accumulate`, which calls the object's `operator()` member function in line 49 of the excerpts. For another way to accumulate, see line 30 of `valarray.C` on p. 899.

The `min_element` algorithm in line 50 returns an iterator, so we must apply a `*` to dereference it. Of course, line 49 must first check that the iterator refers to an element. Line 53−54 show that we can store a returned iterator in a variable for later dereferencing.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/node_container/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>     //for abs
 3 #include <algorithm>   //for find, distance, count, min_element, max_element
 4 #include <numeric>     //for accumulate
 5 #include "node.h"
 6 using namespace std;
 7
 8 int main()
 9 {
10     //Construct a list containing 10, 15, 30.
11     node *head = new node(30, 0);
12     head = new node(15, head);   //Insert 15 ahead of 30.
13     head = new node(10, head);   //Insert 10 ahead of 15.
14
15     for (const node *p = head; p != 0; p = p->next) {
16         cout<< p->value << "\n";
17     }
18     cout << "\n";
19
20     const node::iterator begin(head);
21     const node::iterator end;
22
23     //for (node::iterator it = begin; operator!=(it, end); it.operator++()) {
24     for (node::iterator it = begin; it != end; ++it) {
25         cout << *it << "\n";   //cout << it.operator*() << "\n";
26     }
27     cout<< "\n";
28
29     node::iterator it = begin;
30     ++it;       //it.operator++();
31     *it = 20;   //it.operator*() = 20; overwrite the 15.
32
33     const node::iterator found = find(begin, end, 20);
34
35     if (found == end) {   //if (operator==(found, end)) {
36         cout << "20 was not found.\n";
37     } else {
38         cout << "20 is at position " << distance(begin, found) << ".\n";
39     }
40
41     cout << "Value 20 occurs " << count(begin, end, 20) << " times.\n"
42         << "There are " << distance(begin, end) << " values.\n"
43         << "Sum of the values is " << accumulate(begin, end, 0) << ".\n"
44
45         << "Product of the values is "
46             << accumulate(begin, end, 1, multiplies<int>())
47             << ".\n";
48
49     if (begin != end) {
50         cout << "Smallest value is " << *min_element(begin, end)
51         <<".\n";
52
53         const node::iterator biggest = max_element(begin, end);
54         cout << "Biggest value is " << *biggest << ", at position "
55             << distance(begin, biggest) << ".\n";
56     }
57
58     for (const node *p = head; p != 0;) {
59         const node *const prev = p;
60         p = p->next;
61         delete prev;   //can do this even though prev is a const *
62     }
63
64     return EXIT_SUCCESS;
65 }
```

```
10                                 lines 15−17
15
30

10                                 lines 24−26
15
30

20 is at position 1.              lines 33−39
Value 20 occurs 1 times.           line 40
There are 3 values.                line 42
Sum of the values is 60.           line 43
Product of the values is 6000.     lines 45−47
Smallest value is 10.              lines 49−51
Biggest value is 30, at position 2. lines 53−54
```

### Simple definitions for the algorithms

The algorithms are template functions. Most of them take a pair of iterators, conventionally named `first` and `last`.

If these iterators are equal, the algorithm will process no elements at all. Otherwise, the algorithms assume that `last` is *accessible* from `first`, i.e., that we can get from `first` to `last` with a finite number of increments. It is the programmer's responsibility to make sure that `first` and `last` refer to elements in the *same* container, and that `last` comes after `first`. Failure to do so may result in an infinite loop or program crash.

`find` and `min_element` return an iterator of the type that was passed to them. `accumulate` returns a `T`. `distance` and `count` return a `difference_type` for the type of iterator that was passed to them. It would make more sense for `count` to return an unsigned result, since its return value will never be negative. And we could do it if `iterator_traits` had a typedef giving this unsigned type (`size_type` would be a good name). But you go to war with the `iterator_traits` you have, not the `iterator_traits` you might want or wish to have.

The STL always assumes that an iterator is fast enough to pass and return by value (line 6). This creates a local copy of the iterator, which we can then increment without disturbing the original (line 8). Similarly, lines 61 and 74 assume that a `difference_type` can be passed and returned by value. (61 is the return type of the function in 62; I'm sorry they wouldn't fit on the same line.)

A `T`, on the other hand, is passed and returned by reference whenever possible (lines 6, 75). The template function doesn't know what `T` is; it could be a type that is expensive (read: slow) or impossible to copy. The anonymous `T` constructed in line 57 must be returned by value, since it is an automatic variable. We want to pass the `T` in line 36 by value, since we have to create and return a new `T` anyway. Also, the *numeric* algorithms such as `accumulate` assume that a `T` is a type such as such as `float`, `double`, or `complex<double>`, which are fast enough to pass by value. See pp. 962−964 for the numeric algorithms.

Look for these definitions in the header files `<algorithm>`, `<numeric>`, `<functional>`. Unofficially, they may be in other headers included by these ones.

```
 1 //Excerpts from <algorithm>, <numeric> (accumulate), <functional> (multiplies)
 2
 3 #include <iterator>   //for iterator_traits
 4
 5 template <class IT, class T>
 6 IT find(IT first, IT last, const T& t)
 7 {
 8     for (; first != last; ++first) {
 9         if (*first == t) {
10             break;
11         }
12     }
13
14     return first;
15 }
16
17 template <class IT>
18 IT min_element(IT first, IT last)
19 {
20     if (first == last) {
21         return last;
22     }
23
24     IT it = first;
25
26     while (++first != last) {
27         if (*first < *it) {
28             it = first;
29         }
30     }
31
32     return it;
33 }
```

```

template <class IT, class T>
T accumulate(IT first, IT last, T t)
{
    for (; first != last; ++first) {
        t += *first;
    }

    return t;
}

template <class IT, class T, class OPERATION>
T accumulate(IT first, IT last, T t, OPERATION op)
{
    for (; first != last; ++first) {
        t = op(t, *first);
    }

    return t;
}

template <class T>
struct multiplies: public binary_function<T, T, T> {
        T operator()(const T& a, const T& b) const {return a * b;}
};

template <class IT>
typename iterator_traits<IT>::difference_type
distance(IT first, IT last)
{
    typename iterator_traits<IT>::difference_type d = 0;

    for (; first != last; ++first) {
        ++d;
    }

    return d;
}

template <class IT, class T>
typename iterator_traits<IT>::difference_type
count(IT first, IT last, const T& t)
{
    typename iterator_traits<IT>::difference_type n = 0;

    for (; first != last; ++first) {
        if (*first == t) {
            ++n;
        }
    }

    return n;
}
```

If the `op` in the above line 46 is a pointer to a function, line 49 will call the function. If the `op` is an object, line 49 will call the `operator()` member function of the object. It will behave as if we had written

```
87     t = op.operator()(t, *first);
```

Will `min_element` still work if we give it a range of elements that are not sorted? What will `min_element` return if we give it a range containing two or more elements tied for being the smallest? What will `min_element` return when line 96 gives it an empty range?

```
88 #include <vector>
89 #include <algorithm>
90 using namespace std;
91 
92     int a[] = {10, 20, 10};
93     const size_t n = sizeof a / sizeof a[0];
94     vector<int> v(a, a + n);
95     vector<int>::iterator it1 = min_element(v.begin(), v.end());
96     vector<int>::iterator it2 = min_element(v.begin(), v.begin());
```

**A simpler way to create an iterator_traits for class node::iterator**

There is a simpler way to create class `iterator_traits<node::iterator>`. If the five typedefs are public members of class `node::iterator` (lines 16−20), we will no longer have to define a specialization of class `iterator_traits`.

The word `int` in line 23 could now be written as `value_type`. In fact, the entire `int&` could be written as `reference` if you think the code would be clearer (it wouldn't, at least not at this stage). But these five typedefs are not primarily intended for use in class `node::iterator`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/node_container/node2.h`

```
 1 #ifndef NODEH
 2 #define NODEH
 3 #include <iterator>   //for forward_iterator_tag
 4 using namespace std;
 5 
 6 struct node {
 7     int value;
 8     node *next;
 9 
10     node(int initial_value, node *initial_next)
11         : value(initial_value), next(initial_next) {}
12 
13     class iterator {
14         node *p;
15     public:
16         typedef forward_iterator_tag iterator_category;
17         typedef int value_type;
18         typedef ptrdiff_t difference_type;
19         typedef int *pointer;
20         typedef int& reference;
21 
22         iterator(node *initial_p = 0): p(initial_p) {}
23         int& operator*() const {return p->value;}
24         iterator& operator++() {p = p->next; return *this;}   //prefix
25 
26         friend bool operator==(const iterator& it1, const iterator& it2) {
27             return it1.p == it2.p;
28         }
29     };
30 };
31
32 inline const node::iterator operator++(node::iterator& it, int)//postfix
33 {
34     const node::iterator old = it;
35     ++it;
36     return old;
37 }
38
39 inline bool operator!=(const node::iterator& it1, const node::iterator& it2)
40 {
41     return !(it1 == it2);   //return !operator==(it1, it2);
42 }
43 #endif
```

The general template for class `iterator_traits`, in the following lines 7−14, will now suffice for class `node::iterator`. Line 10 mentions the data type `typename IT::value_type`. The template argument `IT` must therefore stand for a class with a member named `value_type` that is the name of a data type. Our new class `node::iterator` has the required member in the above line 17. The following line 10 creates another `value_type`, this one a member of class `iterator_traits<IT>`. It is a typedef that stands for the same data type as `typename IT::value_type`. One by one, the five typedef members of class `IT` are replicated as members of the general template. A specialization for class `node::iterator` is no longer needed.

Are there any types of iterator that would still need a specialization? Well, a pointer is a completely legitimate iterator. But a pointer does not have the five typedef members (only objects have members), so a pointer data type would *not* be a legal template argument `IT` for the general `iterator_traits`. The library therefore has specializations for iterators that are pointers, in lines 16 and 25. As we saw on p. 643, separate specializations are needed for read/write pointers and read-only pointers.

```
 1 //Excerpts from <iterator>.
 2 #include <cstddef>      //for ptrdiff_t
 3
 4 //IT must be a class that has five public members that are data types,
 5 //named iterator_category, value_type, difference_type, pointer, reference.
 6
 7 template <class IT>
 8 struct iterator_traits {
 9     typedef typename IT::iterator_category iterator_category;
10     typedef typename IT::value_type value_type;
11     typedef typename IT::difference_type difference_type;
12     typedef typename IT::pointer pointer;
13     typedef typename IT::reference reference;
14 };
15
16 template <class T>
17 struct iterator_traits<T *> {
18     typedef random_access_iterator_tag iterator_category;
19     typedef T value_type;
20     typedef ptrdiff_t difference_type;
21     typedef T *pointer;
22     typedef T& reference;
23 };
24
25 template <class T>
26 struct iterator_traits<const T *> {
27     typedef random_access_iterator_tag iterator_category;
28     typedef T value_type;         //read/write
29     typedef ptrdiff_t difference_type;
30     typedef const T *pointer;     //read-only pointer
31     typedef const T& reference;   //read-only reference
32 };
```

It would seem that the `value_type` in the above line 28 should be a `const T`; after all, the members in 30 and 31 are `const`. But even if the iterator is read-only, a value that has been copied out of a container does not have to be held in a read-only variable (line 36). On the other hand, the pointer `p` in line 37 points to a value that is still in the container; it must be a read-only pointer. Similarly for the reference in line 38.

In line 37, the `*` and `&` are built-in operators. They cancel each other out and can be removed. But for iterators that are not pointers, the `*` would stand for a call to the iterator's `operator*` function. In that case, the `&` and `*` would both have to be written.

```
33    int a[] = {10, 20, 30};
34    const int *it = a;
35
36          int  i =  *it;       //No need to make i a const int,
37    const int *p = &*it;       //but p must be a const int *
38    const int& r =  *it;       //and r must be a const int&.
```

**An even simpler way to create an iterator_traits for class node::iterator**

Instead of writing the five typedefs in lines 16−20 of the above `node2.h`, there is an easier way to give these members to class `node::iterator`. We can simply derive this class from a base class that already has the members. The base class is the following template class. We first saw this technique on pp. 769−770, where the base class was an instantiation of the template class `binary_function`.

```
//Another excerpt from <iterator>.
#include <cstddef>   //for ptrdiff_t

template <
    class CATEGORY,
    class T,
    class DIFFERENCE = ptrdiff_t,
    class POINTER = T *,
    class REFERENCE = T&
>
struct iterator {
    typedef CATEGORY iterator_category;
    typedef T value_type;
    typedef DIFFERENCE difference_type;
    typedef POINTER pointer;
    typedef REFERENCE reference;
};
```

Any class that is publicly derived from class

```
iterator<forward_iterator_tag, int, ptrdiff_t, int *, int&>
```

would inherit these five typedefs:

```
18     typedef forward_iterator_tag iterator_category;
19     typedef int value_type;
20     typedef ptrdiff_t difference_type;
21     typedef int *pointer;
22     typedef int& reference;
```

We derive our iterator from this base class in line 13, without bothering to write values for the last three template arguments.

Thanks to the `using namespace std` in line 4, we would normally not need to mention `std` in line 13. But we have two classes with the same first name, our `node::iterator` and the standard library `std::iterator`. Without the `std::`, the rightmost `iterator` in line 13 would be the local class `iterator` (class `node::iterator`), triggering a chain of disasters. First off, this class is not a template class; the angle brackets in of line 13 would not compile. And let's not even think about deriving a class from itself.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/node_container/node3.h`

```
 1 #ifndef NODEH
 2 #define NODEH
 3 #include <iterator>   //for std::iterator and forward_iterator_tag
 4 using namespace std;
 5
 6 struct node {
 7     int value;
 8     node *next;
 9
10     node(int initial_value, node *initial_next)
11         : value(initial_value), next(initial_next) {}
12
13     class iterator: public std::iterator<forward_iterator_tag, int> {
14         node *p;
15     public:
16         iterator(node *initial_p = 0): p(initial_p) {}
17         int& operator*() const {return p->value;}
18         iterator& operator++() {p = p->next; return *this;}   //prefix
19
20         friend bool operator==(const iterator& it1, const iterator& it2) {
21             return it1.p == it2.p;
22         }
23     };
24 };
25
26 inline const node::iterator operator++(node::iterator& it, int)//postfix
27 {
28     const node::iterator old = it;
29     ++it;
30     return old;
31 }
32
33 inline bool operator!=(const node::iterator& it1, const node::iterator& it2)
34 {
35     return !(it1 == it2);   //return !operator==(it1, it2);
36 }
37 #endif
```

**How closely have we approached our ideal?**

Can our iterator now read values from a linked list with the same notation as a pointer reading values from an array? Pretty much so. The operators `!=`, `*`, and `++` can be applied to the iterator. But the `--` operator, prefix and postfix, is conspicuously absent. A `--` is impractical (read: too slow) because the list is singly-linked. It would have been like a salmon fighting its way upstream.

Also absent is an `operator<` to compare two iterators. It would have to determine the relative positions on the list of the two elements to which the iterators refer. But the only way to do this would be to start at the left iterator's element, and loop along the list until we encounter either the right iterator's element or the end of the list, whichever comes first. This would not be a *constant time* operation; the time would depend on how many elements have to be visited.

For the same reason, we did not write an `operator-` to measure the distance between two iterators. The only way to do this would be to walk from one to the other, counting the elements as we go. Similarly, operators such as `+=` and `[]`, which we regularly apply to pointers, would be too slow for our linked list. Executing the expression `it += 20` would take twice as long as `it += 10`.

To sum up, we can apply the following binary operators to a pointer but not to our iterator.

```
-- + - += -= < <= > >= []
```

Because of these limitations, will see that our `node::iterator` will qualify as only a "forward" iterator (pp. 839−840), not a "bidirectional" one (pp. 840−841). This was the meaning of the `forward_iterator_tag` we saw in line 40 of `node.h` on p. 806.

A singly-linked list, class `slist`, has already been written. It's not officially in the STL, but many vendors supply it anyway. Include the header file `<slist>` and don't decrement the iterators.

**▼ Homework 8.2.1a: create class node::const_iterator**

We saw a `const_iterator` for class `vector` in line 14 of `const_iterator.C` on p. 436. Create a `const_iterator` for class `node`. Do not remove the existing class `iterator`.

Give it the last name `node`, like our class `iterator`. For the reason in the ¶ (4) below, we must define `const_iterator` before `iterator`. Both definitions will be inside the {curly braces} of class `node`.

Class `node::const_iterator` will be exactly like the existing class `node::iterator` (don't forget the postfix increment), but with the following changes.

(1) The data member `p` will be a read-only pointer.

```
1      const node *p;
```

The argument of the constructor for `node::const_iterator` will also be a read-only pointer.

(2) The whole point of a `const_iterator` is to prevent the assignment in line 4 from compiling.

```
2      node::const_iterator it = begin;
3      if (it != end) {               //if there is a node,
4          *it = 20;                  //won't compile
5      }
```

The `operator*` member function of `node::const_iterator` will therefore return a read-only reference. (A return by value would prevent the following line 11 from compiling.)

```
6          const int& operator*() const {return p->value;}
```

Similarly, if class `const_iterator` had an `operator->` member function (which it doesn't), the pointer that it returns would have to be read-only. There will be an `operator->` in our next example.

(3) As in the above line 4, a `const_iterator` cannot change the value of an `int` in a container. But we are free to change the value of an `int` that has been copied out of the container (lines 9−10 below). The `value_type` member of class `iterator_traits<node::const_iterator>` can therefore remain `int`. But the pointer in line 11 and the reference in line 12 must be `const` to point and refer to a

value still in the container.

```
    node::const_iterator it = begin;
    if (it != end) {
        int i = *it;
        ++i;
        const int *p = &*it;   //const int *p = &it.operator*();
        const int& r = *it;
    }
```

Class `const_iterator` will therefore be derived from class

```
    std::iterator<forward_iterator_tag, int, ptrdiff_t,
        const int *, const int&>
```

(4) Do not make it possible to convert a `node::const_iterator` to a `node::iterator`. That would be a breach of security.

```
    node *head = new node(30, 0);
    head = new node(20, head);
    head = new node(10, head);

    node::const_iterator it = head;

    //Try to change the 10 to 15.  Do not allow this to compile.
    *static_cast<node::iterator>(it) = 15;
```

But conversion in the other direction, from `node::iterator` to `node::const_iterator`, would be harmless and convenient. For example, an `operator==` and `operator!=` that take two `node::const_iterator`'s would be able to handle all of our comparisons between `node::iterator`'s, `node::const_iterator`'s, and any combinatin thereof.

Add the following public member function to class `node::iterator`

```
    operator const_iterator() const {return p;}
```

Define an `operator==` and `operator!=` for two `node::const_iterator`'s, and remove the `operator==` and `operator!=` for two `node::iterator`'s. An expression that compares a `node::iterator` and a `node::const_iterator`

```
it == cit
```

will now behave as if we had swritten the following, calling the `operator==` that compares two `node::const_iterator`'s.

```
    operator==(it.operator node::const_iterator(), cit)
```

▲

## 8.2.2 An Input File

The second data structure we turn into a container will be an `istream` such as a sequential input file (not a random access input file). The following is a text file named `infile`. It contains dates so we can demonstrate an `operator->` member function as well as an `operator*` for the iterator.

```
7/4/1776
10/29/1929
12/7/1941
```

The loop in lines 15−17 is the traditional way to access this data structure. The constructor for class `ifstream` opens the file in lines 9−13, the destructor closes the file in line 19, the `operator!` and `operator void *` member functions tell us if the file is healthy, and, if so, the `>>` operator reads from the file. The latter is the `operator>>` friend of class `date` that we wrote in lines 7−65 of `date.C` on pp. 338−339. If we break out of the loop because of end-of-file, line 19 returns `EXIT_SUCCESS`. For any other reason, we return `EXIT_FAILURE`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/infile/ifstream.C`

```
 1 #include <iostream>  //for cout and cerr
 2 #include <fstream>   //for ifstream
 3 #include <cstdlib>
 4 #include "date.h"
 5 using namespace std;
 6
 7 int main(int argc, char **argv)
 8 {
 9     ifstream ifs("infile");
10     if (!ifs) {              //if (ifs.operator!()) {
11         cerr << argv[0] << ": couldn’t open infile\n";
12         return EXIT_FAILURE;
13     }
14
15     for (date d; ifs >> d;) { //operator>>(ifs, d).operator void *();
16         cout << d << "\n";    //operator<<(operator<<(cout, d), "\n");
17     }
18
19     return ifs.eof() ? EXIT_SUCCESS : EXIT_FAILURE;
20 }
```

```
7/4/1776
10/29/1929
12/7/1941
```

**Read from the file with an iterator**

All of the above notation was specific to data structures that are input files. Let’s create an iterator that can read dates from an input file, or from another `istream`, with the same notation as a pointer reading dates from an array. We will return to the details later. For now, we hurry ahead and admire the following `main1.C` and `main2.C`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/infile/istream_iterator_date.h`

```
#ifndef ISTREAM_ITERATOR_DATEH
#define ISTREAM_ITERATOR_DATEH
#include <iostream>
#include <iterator>
#include "date.h"
using namespace std;

class istream_iterator_date:
    public iterator<input_iterator_tag, date, ptrdiff_t,
        const date *, const date&> {

    istream *ist;
    bool ok;   //true if this iterator has a healthy istream
    date d;    //the date read most recently from the file

    void read() {
        if (ok) {
            ok = *ist >> d; //ok=operator>>(*ist,d).operator void *();
        }
    }
public:
    istream_iterator_date(istream& initial_is)
        : ist(&initial_is), ok(true) {read();}
    istream_iterator_date(): ist(0), ok(false) {}

    const date& operator*() const {return d;}
    const date *operator->() const {return &**this;}

    istream_iterator_date& operator++() {read(); return *this;}

    friend bool operator==(const istream_iterator_date& it1,
                           const istream_iterator_date& it2) {
        return it1.ok == it2.ok && (!it1.ok || it1.ist == it2.ist);
    }
};

inline const istream_iterator_date operator++(istream_iterator_date& it, int)
{
    const istream_iterator_date old = it;
    ++it;
    return old;
}

inline bool operator!=(const istream_iterator_date& it1,
                const istream_iterator_date& it2) {
    return !(it1 == it2);
}
#endif
```

With our new iterator, the loop in lines 18−20 is totally generic, like the one in lines 24−26 of `main.C` on p. 807. The iterator whose constructor takes no argument (line 16) refers to the slot in the container where the non-existent “element” after the last element would be. It represents the end of this input file. In fact, it will represent the end of *any* input file, just as a `'\0'` represents the end of any array of characters.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/infile/main1.C`

```
 1 #include <iostream>
 2 #include <fstream>
 3 #include <cstdlib>
 4 #include "date.h"
 5 #include "istream_iterator_date.h"
 6 using namespace std;
 7
 8 int main(int argc, char **argv)
 9 {
10     ifstream ifs("infile");
11      if (!ifs) {
12          cerr << argv[0] << ": couldn’t open infile\n";
13          return EXIT_FAILURE;
14      }
15
16      const istream_iterator_date end;
17
18      for (istream_iterator_date it(ifs); it != end; ++it) {
19          cout << *it << "\n";
20      }
21
22      return EXIT_SUCCESS;
23  }
```

```
7/4/1776
10/29/1929
12/7/1941
```

**Call a member function of each object in the container**

The `operator*` member function of the iterator returns the value of the `date` object being read from the container. The above line 19 behaves as if we had written the following.

```
24      cout << it.operator*() << "\n";
```

To call the `print` member function of each `date`, we can change line 19 to

```
25      (*it).print();
26      cout << "\n";
```

It will behave as if we had written the following.

```
27      it.operator*().print();
28      cout << "\n";
```

Since our `print` does the same thing as `operator<<`, the output will be the same.

```
7/4/1776
10/29/1929
12/7/1941
```

But there’s a simpler way to call the `print` member function of each object. We can change line 19 to

```
29      it->print();
30      cout << "\n";
```

It will behave as if we had written the following, calling the `operator->` member function of the iterator.

```
31      it.operator->()->print();
32      cout << "\n";
```

We expect that the `operator->` function would take two arguments, because the `->` operator takes two operands. After all, the `operator==` function takes two arguments because the `==` operator takes two operands. But `operator->` takes no arguments at all. It simply returns the address of the `date` object to which the iterator refers. The address and the following member name (the `print` in line 29 above) are then used as operands of an extra `->`, supplied by the computer, which is the rightmost `->` in line 31 above. This is the ‘‘pointer to structure’’ operator built into C and C++. Once again, the output is the same.

```
7/4/1776
10/29/1929
12/7/1941
```

**Pass the data in the container to an algorithm.**

We can now pass our data to an algorithm (line 21).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/infile/main2.C`

```
 1 #include <iostream>
 2 #include <fstream>
 3 #include <cstdlib>
 4 #include <algorithm>
 5 #include "date.h"
 6 #include "istream_iterator_date.h"
 7 using namespace std;
 8
 9 int main(int argc, char **argv)
10 {
11     ifstream ifs("infile");
12     if (!ifs) {
13         cerr << argv[0] << ": couldn’t open infile\n";
14         return EXIT_FAILURE;
15     }
16
17     const istream_iterator_date end;
18     const date crash(date::october, 29, 1929);   //stock market
19
20     const istream_iterator_date it =
21         find(istream_iterator_date(ifs), end, crash);
22
23     if (it == end) {
24         cout << "The file does not contain " << crash << ".\n";
25     } else {
26         cout << "The file contains " << crash << ".\n";
27     }
28
29     return EXIT_SUCCESS;
30 }
```

```
The file contains 10/29/1929.
```

**Detect end-of-file**

We might expect that each element in a range will be read by the `operator++` member function of an iterator. But there are two reasons why `operator++` cannot do all the reading. Consider first an iterator that refers to the first element of a non-empty range. The first call to the iterator’s `operator*` must return the first value in the range, even if there was no previous call to `operator++`.

Next consider an iterator that refers to the first “element” of an empty range. (It actually refers to no element at all; its value is merely equal to that of the `last` iterator.) The first call to `!= last` must return false, even if there was no previous call to `operator++`. In fact, any call to `!= last` must be able to detect in advance if the next application of the `*` operator would attempt to access the non-existent “element” beyond the end of the range. In the loop in `main1.C` on p. 819, for example, the comparison to `end` in line 18 must be able to detect if the following `*` in line 19 would access the non-existent element. And in the `find` algorithm on p. 809, the comparison to `last` in line 8 must be able to detect if the next `*` in line 9 would access the non-existent element.

These two requirements are critical for every STL algorithm that reads from a range, and will complicate the design of the iterator. The problem is that an `ifstream` or other `istream` does not detect end-of-file until it has attempted to read beyond the end of the file.

Let's demonstrate this by trying to read from an empty file. We will use the `/dev/null` file: on my platform (Unix), it is always present but always empty. The attempted read in line 19 will fail, making no change to `d`. But the `ifstream` cannot detect this in advance (line 17).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/infile/eof1.C`

```
 1 #include <iostream>
 2 #include <fstream>
 3 #include <cstdlib>
 4 #include "date.h"
 5 using namespace std;
 6
 7 int main(int argc, char **argv)
 8 {
 9     ifstream ifs("/dev/null");
10     if (!ifs) {
11         cerr << argv[0] << ": couldn’t open /dev/null\n";
12         return EXIT_FAILURE;
13     }
14
15     cout << boolalpha;
16
17     cout << ifs.eof() << "\n";
18     date d;
19     ifs >> d;   //reads nothing because it reaches end-of-file
20     cout << ifs.eof() << "\n";
21
22     return EXIT_SUCCESS;
23 }
```

```
false    line 17 does not detect eof yet, even though there are no dates to be read
true     line 20
```

But an iterator is expected to detect end-of-container *before* the `*` which attempts to access the non-existent "element". The iterator must compare equal to the `end` iterator in line 17.

The `date` that line 22 copies into `d` was never not read from the file. It is the dummy `date` that was created when the constructor for the iterator passed no arguments to the constructor for class `date`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/infile/eof2.C`

```
1 #include <iostream>
2 #include <fstream>
3 #include <cstdlib>
4 #include "istream_iterator_date.h"
5 #include "date.h"
6 using namespace std;
7
8 int main(int argc, char **argv)
```

```
 9 {
10     ifstream ifs("/dev/null");
11     if (!ifs) {
12         cerr << argv[0] << ": couldn’t open /dev/null\n";
13         return EXIT_FAILURE;
14     }
15
16     istream_iterator_date it(ifs);
17     const istream_iterator_date end;
18
19     cout << boolalpha;
20
21     cout << (it == end) << "\n";
22     date d = *it;
23     cout << (it == end) << "\n";
24
25     return EXIT_SUCCESS;
26 }
```

```
true       line 21 detects eof even before the attempted read in line 22
true       line 23
```

Now we can explain the convoluted design of the iterator in `istream_iterator_date.h` on pp. 817−818. It would seem natural for the `operator*` to read each date from the file. But if we did that, there would be no way for a previous `!=` operator to detect end-of-file.

Our solution is to have the *constructor* attempt to read the first date from the file (line 23 in the `.h` file), storing it in a data member `d` (line 14). The `operator++` member function also attempts to read a date, storing it in the same data member (line 29). When the `!=` operator is called, the read has therefore already been attempted. If the `!=` finds that the read was successful, the algorithm calls `operator*`, which returns the data member (line 26).

The boolean data member `ok` in line 13 is true if the iterator has an `istream`, and if the `istream` has not yet encountered end-of-file or other errors. The value of the expression

```
*ist >> d
```

in line 18 is `*ist`, which is converted to a `bool` when stored in `ok`. The `bool` will be true if the `>>` was successful.

Of course, not every iterator has an `istream`; the `end` iterator in lines 16−18 of `main1.C` on p. 819 did not have one. This, incidentally, explains why the `ist` data member in line 12 of `istream_iterator_date.h` has to be a pointer, not a reference. It had to be one or the other since we are not allowed to copy a stream (pp. 324−326). Now a pointer can easily point to no variable, but we should never have a reference that refers to no variable. Since an `istream` is not always present, `ist` has to be a pointer.

What about the forbidding logical expression in line 39 of `istream_iterator_date.h`? Two iterators are considered equal if they are both at the end-of-file, or if they are both reading from the same input stream. Let’s consider these two cases separately.

The date returned by `operator*` is read in a previous call to the iterator’s constructor or `operator++`. If that read encountered end-of-file, line 18 set the `ok` data member to false. Meanwhile, an `end` iterator always has its `ok` set to false (line 24). The `operator==` in line 37 considers any two iterators to be equal if their `ok`’s are both false. This means that the `end` iterator marks the end not only of our input stream, but of any input stream. It’s like the character `’\0’`, which marks the end of any string.

`operator==` also considers any two iterators reading from the same stream to be equal, provided that neither has encountered end-of-file yet. A comparison of two iterators reading from the same input stream is therefore not very illuminating. `operator==` and `operator!=` should be used only to compare an iterator that is reading and moving (the `it` in line 18 of `main1.C` on p. 819) with a stationary `end` iterator.

As usual, many functions call other functions to do their work. The `operator!=` in line 44 calls the `operator==` in line 31; the postfix `operator++` in 37 calls the prefix one in 29.

A new example is the `operator->` in line 27, which returns the address of the most recently read date. The `operator*` in line 26 returns the value of this date, so 27 simply returns the address of this value. The expression `this` in line 27 is the address of the iterator; `*this` is the value of the iterator; `**this` is the value returned by the `operator*` member function of the iterator; and `&**this` is the address of the value returned by the `operator*` member function of the iterator. We can take this address because the return value of `operator*` is a reference.

`operator->` could have been defined as follows, but I wanted it to be free of code specific to class `istream_iterator_date`.

```
27      const date *operator->() const {return &d;}
```

**How closely have we approached our ideal?**

Ideally, we would like to read from a container with an iterator with the same notation used to read from an array with a pointer. We accept that our `istream_iterator_date` suffers from the same limitations that plagued the `node::iterator`, starting with the absence of a `--` operator. But `istream_iterator_date` will turn out to be even more delicate.

The following line 17 shows that we can make a copy of this iterator, allowing us to pass it by value to a function. *But we must never increment one copy and compare or dereference the other.* The following program shows what goes wrong if we try this. The first date in the file is read by the constructor in line 16 and stored in the iterator. The second date is read in line 20. The loop in lines 23−25 misses the second date in the file, which has been siphoned off by the increment in line 20.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/infile/interfere.C`

```
 1 #include <iostream>
 2 #include <fstream>
 3 #include <cstdlib>
 4 #include "istream_iterator_date.h"
 5 using namespace std;
 6
 7 int main(int argc, char **argv)
 8 {
 9     ifstream ifs("infile");
10     if (!ifs) {
11         cerr << argv[0] << ": couldn’t open infile\n";
12         return EXIT_FAILURE;
13     }
14
15     const istream_iterator_date end;
16     istream_iterator_date it1(ifs);
17     istream_iterator_date it2 = it1;
18
19     if (it2 != end) {
20         date d = *++it2;
21     }
22
23     for (; it1 != end; ++it1) {
24         cout << *it1 << "\n";
25     }
26
27     return EXIT_SUCCESS;
28 }
```

```
7/4/1776       first date in the file
12/7/1941      third date in the file
```

A practical consequence is that the following program fails. Line 17 constructs the iterator `begin`. Line 20 passes the iterator by value, constructing a copy of it. The loop in the `distance` algorithm increments the copy until it reaches end-of-file. So far, all is well. But at line 23, `begin` has already been exhausted when we copy it again. This copy is born prematurely aged: the underlying input file is already at end-of-file, and the loop in the `find` algorithm is never entered.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/infile/main3.C`

```
 1 #include <iostream>
 2 #include <fstream>
 3 #include <cstdlib>
 4 #include <algorithm>
 5 #include "date.h"
 6 #include "istream_iterator_date.h"
 7 using namespace std;
 8
 9 int main(int argc, char **argv)
10 {
11     ifstream ifs("infile");
12     if (!ifs) {
13         cerr << argv[0] << ": couldn’t open infile\n";
14         return EXIT_FAILURE;
15     }
16
17     const istream_iterator_date begin(ifs);
18     const istream_iterator_date end;
19
20     cout << "The file contains " << distance(begin, end) << " dates,\n";
21
22     const date crash(date::october, 29, 1929);   //stock market
23     const istream_iterator_date it = find(begin, end, crash);
24
25     if (it == end) {
26         cout << "not including " << crash << ".\n";
27     } else {
28         cout <<     "including " << crash << ".\n";
29     }
30
31     return EXIT_SUCCESS;
32 }
```

```
The file contains 3 dates,    Line 20 correctly counted the elements,
not including 10/29/1929.     but line 23 did not search through them.
```

To rejuvinate `begin`, we would have to mess around with its private members. A better way to fix the program is to rewind the underlying `ifstream` back to the start of the file (lines 22−28) and create a fresh iterator at line 33. I wish we could rewind by calling `seekg` (p. 382), but we cannot ‘‘seek’’ a file that has already been driven to end-of-file. We have to rewind the file by closing and reopening it. Before line 23 can close the file successfully, line 22 must turn off the file’s `failbit`. This bit was turned on when line 20 encountered end-of-file.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/infile/main4.C`

```
 1 #include <iostream>
 2 #include <fstream>
 3 #include <cstdlib>
 4 #include <algorithm>
 5 #include "date.h"
 6 #include "istream_iterator_date.h"
 7 using namespace std;
 8
 9 int main(int argc, char **argv)
10 {
11     ifstream ifs("infile");
12     if (!ifs) {
13         cerr << argv[0] << ": couldn’t open infile\n";
14         return EXIT_FAILURE;
15     }
16
17     const istream_iterator_date end;
18
19     cout << "The file contains "
20         << distance(istream_iterator_date(ifs), end) << " dates,\n";
21
22     ifs.clear(ifs.rdstate() & ~ios_base::failbit);
23     ifs.close();
24     if (!ifs) {
25         cerr << argv[0] << ": couldn’t close infile\n";
26         return EXIT_FAILURE;
27     }
28     ifs.open("infile");
29     if (!ifs) {
30         cerr << argv[0] << ": couldn’t reopen infile\n";
31         return EXIT_FAILURE;
32     }
33
34     const date crash(date::october, 29, 1929);   //stock market
35
36     const istream_iterator_date it =
37         find(istream_iterator_date(ifs), end, crash);
38
39     if (it == end) {
40         cout << "not including " << crash << ".\n";
41     } else {
42         cout <<     "including " << crash << ".\n";
43     }
44
45     return EXIT_SUCCESS;
46 }
```

```
The file contains 3 dates,
including 10/29/1929.
```

Unfortunately, the call to `find` in the above lines 31−32 does not tell us all we wish to know. We have discovered that a certain date is in the file, but we don't know *where* in the file it is. The problem is that an `istream_iterator_date`, the return value of this call to `find`, is good at getting dates from a file, but does not mark a location in a file. We will have to write a new algorithm with a different return type; see p. 837.

More surprises happen when we compare two copies of an `istream_iterator_date`. First, look at the paradoxical output of line 17. The `==` operator believes that any two iterators are equal if they are reading from the same stream, provided that they have not reached end-of-input. Incidentally, if you change line 17 to

```
    cout << (*it1 == *++it1) << "\n";   //now we're comparing two dates
```

it will become false, at least on platforms that evaluate the `*it1` before the `*++it1`.

Next, observe that the iterators in line 20 are equal. Indeed, `it2` was newly minted by the copy constructor in the previous line. But the expressions in line 21 are unequal. Apparently, our iterators lack the *substitution property*, basic to Western Thought. How can this be?

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/infile/paradox.C`

```
 1 #include <fstream>
 2 #include <cstdlib>
 3 #include "istream_iterator_date.h"
 4 using namespace std;
 5
 6 int main(int argc, char **argv)
 7 {
 8     ifstream ifs("infile");
 9     if (!ifs) {
10         cerr << argv[0] << ": couldn't open infile\n";
11         return EXIT_FAILURE;
12     }
13
14     cout << boolalpha;
15
16     istream_iterator_date it1(ifs);
17     cout << (it1 == ++it1) << "\n";
18
19     istream_iterator_date it2 = it1;
20     cout << (  it1 ==   it2) << "\n";
21     cout << (++it1 == ++it2) << "\n";
22
23     return EXIT_SUCCESS;
24 }
```

```
true     line 17
true     line 20
false    line 21
```

The explanation is simple. The constructor in the above line 16 read the first date from the file; the `++` in line 17 read the second. Whichever iterator is incremented first in line 21 will read the third date, and its `ok` data member will remain true. The other iterator will encounter end-of-file, and its `ok` will be set to false. (There is no way to predict which `++` will execute first in line 21: precedence and associativity give us no decision since the increments are not adjacent. See pp. 14−16.)

The moral is that while an `istream_iterator_date` can be copied, but only one copy should be used. There is an elegant way to enforce this. We have often passed a function argument as an anonymous temporary, letting us avoid the bother of inventing a name for it. Our most recent example was the first argument of `find` in line 21 of `main2.C` on p. 820. Now we have another reason to make the temporary anonymous. If an `istream_iterator_date` is passed by value, there are two copies. But if the original is a anonymous, there is no way it can be used or even mentioned by the caller after it has been passed to an algorithm.

Because of these limitations—no decrement, use only one copy—we will see that our `istream_iterator_date` will qualify as only an ‘‘input iterator’’ (pp. 834−837). The STL already has an iterator like `istream_iterator_date`, but it is better because it is a template. Simply include the header file `<iterator>` and construct an `istream_iterator<date>`. See pp. 850−855.

## 8.2.3 An Output File

*The Moving Finger writes; and, having writ,*
*Moves on: nor all thy Piety nor Wit*
*Shall lure it back to cancel half a Line,*
*Nor all thy Tears wash out a Word of it.*

—*Rubáiyát of Omar Khayyám*, quatrain 51

The third data structure we turn into a container will be a sequential output file or other `ostream`. We will write three integers to a text file named `outfile`.

The traditional way to access this data structure is with the loop in lines 17−22. The constructor for class `ofstream` opens the file in lines 8−12, the destructor closes the file in lines 20 or 24, the `operator!` member function tells us if the file is healthy in lines 9 and 19, and the `<<` operator in line 19 writes an `int` to to the file. If all the integers and newlines were successfully written, line 24 returns `EXIT_SUCCESS`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/outfile/ofstream.C`

```
 1 #include <iostream>
 2 #include <fstream>
 3 #include <cstdlib>
 4 using namespace std;
 5
 6 int main(int argc, char **argv)
 7 {
 8     ofstream ofs("outfile");
 9     if (!ofs) {
10         cerr << argv[0] << ": couldn’t open outfile\n";
11         return EXIT_FAILURE;
12     }
13
14     const int a[] = {10, 20, 30};
15     const size_t n = sizeof a / sizeof a[0];
16
17     for (const int *p = a; p < a + n; ++p) {
18         //if (operator<<(ofs.operator<<(*p), "\n").operator!()) {
19         if (!(ofs << *p << "\n")) {
20             return EXIT_FAILURE;
21         }
22     }
```

```

    return EXIT_SUCCESS;
}
```

The program creates an output file, `outfile`, containing the three integers.

```
10
20
30
```

**Write to the file with an iterator**

All of the above notation was specific to data structures that are output files. Let’s create an iterator that can write integers to an output file, or to another `ostream`, with the same notation as a pointer writing integers to an array. The end result will be the loop in the following lines 18−20.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/outfile/main.C`

```
 1 #include <fstream>
 2 #include <cstdlib>
 3 #include "ostream_iterator_int.h"
 4 using namespace std;
 5
 6 int main(int argc, char **argv)
 7 {
 8     ofstream ofs("outfile");
 9     if (!ofs) {
10         cerr << argv[0] << ": couldn’t open outfile\n";
11         return EXIT_FAILURE;
12     }
13     ostream_iterator_int it(ofs);
14
15     const int a[] = {10, 20, 30};
16     const size_t n = sizeof a / sizeof a[0];
17
18     for (const int *p = a; p < a + n; ++p, ++it) {
19         *it = *p;   //it.operator*().operator=(*p);
20     }
21
22     return EXIT_SUCCESS;
23 }
```

The output file is

```
10
20
30
```

**A proxy object**

When we apply the `*` operator to an object, we are really calling the `operator*` member function of that object. For example, the above line 19 calls the `operator*` of `it`. We would expect this function to write an integer to the output file.

Here’s why it can’t. The above line 19 behaves as if we has said the following.

```
    it.operator*() = *p;
```

The `operator*` is a member function of `it`, so it has access to the `ostream *` data member thereof. But the `operator*` receives no arguments. In particular, it never receives the integer value `*p`. To write the value to a file, a function must have access to the `ostream *` in the iterator and to the value.

It would be nice if the `*` and the `=` could call a single member function of `it`, taking the `*p` as its argument. The above line 19 would then do the following, and the function would have access to the `ostream *` and to the integer value.

```
2     it.single(*p);                  //I wish *it = *p; could do this.
```

But this is wishful thinking. The two operators `*` and `=` will not turn into a single function call, at least not in this language. Is there any way the same effect could be obtained with the machinery at our disposal?

Let's brainstorm. The `operator*` member function of `it` will construct and return an anonymous object that contains a copy of all the data in `it`. Then the `=` in the above line 19 will call the `operator=` member function of the anonymous object, as in the comment in that line. The `operator=` will have access to the `ostream *` and to the integer value `*p`.

The anonymous object will be of the data type `proxy` in lines 11−16 of the following `ostream_iterator_int.h`. For convenience, we give it the last name `ostream_iterator_int`. When an iterator's `operator*` constructs and returns a `proxy`, the proxy will hold a copy of the `ostream *` that was in the iterator. The actual write to the file will take place in the `proxy`'s `operator=`.

The definition of class `proxy` (lines 11−16 in `ostream_iterator_int.h`) had to come before the definition of `operator*` (line 20). After all, `operator*` can't create a `proxy` unless it knows what a `proxy` is.

The `ost` data member of the `ostream_iterator_int` in line 9 of `ostream_iterator_int.h` is not `*const`, allowing us to assign one iterator to another:

```
1     it1 = it2;     //it1.operator=(it2);
```

But the `ost` data member of the `proxy` object in line 12 is `*const`, because we never want to assign one proxy to another. The following expression `*it2` cannot be used as the right operand of an assignment.

```
2     //won't compile: it1.operator*().operator=(it2.operator*());
3     *it1 = *it2;
```

In fact, the only thing it *can* be used as is the left operand of an assignment whose right operand is an `int` or convertible thereto. Class `proxy` has no other member functions.

For another proxy object, see p. 968.

### Machinery not needed by an output iterator

Some of the machinery of class `istream_iterator_date` becomes irrelevant in our new class `ostream_iterator_int`. An input file can be exhausted, but we boldly assume that an output file can absorb any amount of data. We never need to compare an `ostream_iterator_int` to an end-of-file iterator, so there are no `operator==` or `operator!=` functions. We can never use a pair of `ostream_iterator_int`'s to delimit a range of elements passed to an algorithm. (For an algorithm that will accept a single `ostream_iterator_int`, see the `copy` on p. 844.)

For the same reason, the `iterator_traits` for class `ostream_iterator_int` has no `difference_type`. The output file is infinite, so we would need an infinitely large variable to count how many writes we have performed. If we do need to count the number of writes, we can usually use the `difference_type` of some other container. In the above `main.C`, for example, the number of integers to write was determined by the number of integers in the container `a` in line 15. Since this container is an array, we could tally this number with a variable of data type `size_t`.

An `ostream_iterator_int` does not let us use the values written to the container. Once the value is written, it's gone. The `iterator_traits` therefore has no `value_type`, `pointer`, or `reference`. And if the value were an object, rather than an `int`, an `ostream_iterator_` would not let us use any member of the object. It therefore has no `operator->` member function either.

The four unnecessary typedef members of `itarator_traits` are still present, but all of them are `void` and should never be used (line 8 of the `ostream_iterator_int.h`).

The prefix `operator++`, in line 21 of `ostream_iterator_int.h`, has no work to do. We define it anyway, because our iterator will be passed to algorithms that apply a `++` to the iterator. The `operator++` is non-`const`, even though it changes no data members, because people would be puzzled if they found that they could increment a `const` iterator. The postfix `operator++`, in line 24 of `ostream_iterator_int.h`, is even more superfluous. I gave it the customary definition only from habit. It could just as well have been defined as

```
    const ostream_iterator_int& operator++(int) {return *this;}
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/outfile/proxy/ostream_iterator_int.h`

```
 1 #ifndef OSTREAM_ITERATOR_INTH
 2 #define OSTREAM_ITERATOR_INTH
 3 #include <iostream>
 4 #include <iterator>
 5 using namespace std;
 6
 7 class ostream_iterator_int:
 8     public iterator<output_iterator_tag, void, void, void, void> {
 9     ostream *ost;
10
11     class proxy {
12         ostream *const ost;
13     public:
14         proxy(ostream *initial_ost): ost(initial_ost) {}
15         void operator=(int i) const {*ost << i << "\n";}
16     };
17
18 public:
19     ostream_iterator_int(ostream& initial_ost): ost(&initial_ost) {}
20     const proxy operator*() const {return ost;}   //means return proxy(ost);
21     ostream_iterator_int& operator++() {return *this;}
22 };
23
24 inline const ostream_iterator_int operator++(ostream_iterator_int& it, int)
25 {
26     const ostream_iterator_int old = it;
27     ++it;
28     return old;
29 }
30 #endif
```

### ▼ Homework 8.2.3a: make sure there are no unauthorized proxies

An `ostream_iterator_int::proxy` should be constructed only by the function `ostream_iterator_int::operator*`. Enforce this by making the constructor for class `proxy` private. To call the constructor, `operator*` will now have to be a friend of class `proxy`.

(1) Let the constructor for class `proxy` be private. And, of course, class `proxy` should remain a private member of class `ostream_iterator_int`.

(2) To allow `operator*` to construct a `proxy`, add the following declaration to the definition of class `proxy`.

```
1       friend const proxy ostream_iterator_int::operator*() const;
```

(3) Line 20 of the above `ostream_iterator_int.h` on p. 830 was both a declaration and definition for the function `operator*`. It will have to be split in two. The definition of class `proxy` now mentions `operator*`, so the *declaration* of this function will have to come before the definition of class `proxy`. And `operator*` creates a `proxy`, so the *definition* of this function will have to come after the definition of class `proxy`.

The declaration of `operator*` and the definition of `proxy` can remain within the {curly braces} of the definition of class `ostream_iterator_int`. The declaration of `operator*` will look like this:

```
2       const proxy operator*() const;
```

But the definition of `operator*` will have to be moved to a point after the braces. It will look like this:

```
3   inline const ostream_iterator_int::proxy ostream_iterator_int::operator*() const
4   {
5       return ost;  //means return proxy(ost);
6   }
```

The above line 5 could call `proxy` by its first name, since it is inside the body of a member function of class `ostream_iterator_int`. But the above line 3 has to call `proxy` by its full name `ostream_iterator_int::proxy`, since it is outside the body of a member function of class `ostream_iterator_int`.

(4) The declaration for `operator*` now mentions `proxy` before the computer has seen the definition for this class. You will have to write a forward declaration for `proxy` (pp. 465−466), in the `private` section of the definition of class `ostream_iterator_int`, but before the declaration for `operator*`.

Was it worth it?

▲

**Eliminate the proxy class**

We could avoid the separate `proxy` class by letting `ostream_iterator_int` be its own proxy:

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/outfile/noproxy/ostream_iterator_int.h`

```
#ifndef OSTREAM_ITERATOR_INTH
#define OSTREAM_ITERATOR_INTH
#include <iostream>
#include <iterator>
using namespace std;

class ostream_iterator_int:
    public iterator<output_iterator_tag, void, void, void, void> {
    ostream *ost;
public:
    ostream_iterator_int(ostream& initial_ost): ost(&initial_ost) {}
    const ostream_iterator_int& operator*() const {return *this;}
    void operator=(int i) const {*ost << i << "\n";}
    ostream_iterator_int& operator++() {return *this;}
};

inline const ostream_iterator_int operator++(ostream_iterator_int& it, int)
{
    const ostream_iterator_int old = it;
20     ++it;
21     return old;
22 }
23 #endif
```

With the same `main.C`, we get the same output file.

```
10
20
30
```

Unfortunately, this way is less secure. With a separate proxy class, the `it` in line 19 of `main.C` on p. 828 must have exactly one asterisk. Without the proxy class, the `it` could be written with any number of asterisks, or even with none at all. But we don't want to allow that freedom. Line 19 might be transplanted into an algorithm someday, causing a bug when the algorithm is passed an iterator of a type that requires exactly one asterisk. (Let's hope that no one forgets the `++` in line 18.)

**How closely have we approached our ideal?**

Ideally, we would like to write to a container with an iterator with the same notation used to write to an array with a pointer. Like `istream_iterator_date` our `ostream_iterator_int` is missing the following operators:

```
-- + - += -= < <= > >= []
```

Similarly, we can copy an `ostream_iterator_int` but we cannot use both copies. Furthermore, the result of applying an `*` to an `ostream_iterator_int` can be used only as the left operand of an assignment.

For these reasons, our `ostream_iterator_int` will qualify only as an "output iterator" (pp. 837−839). The STL already has an iterator like `ostream_iterator_int`, but it is better because it is a template. Simply include the header file `<iterator>` and construct an `ostream_iterator<int>`. See pp. 850−855.

**Different induction variables for different data structures**

The variable whose value changes during each iteration of a loop is called the *induction variable.* Below are four data structures with four loops. The loops have different types of induction variables: the integer `i` in line 4 vs. the pointer to a structure `p` in line 14 vs. the pair of `unsigned`'s in line 21 vs. the stream `cin` in line 32. Each loop applies different code to the induction variable to access an element of a container: the `[`square brackets`]` in line 5 vs. the `->` line in 15 vs. the `term_put` in line 22 vs. the `.get` in line 32. Each loop has different code to update the induction variable, underlined in each example. Each loop has different code to test the induction variable: the `< n` in line 4 vs. the `!= 0` in line 14 vs. the `if` statement in lines 24−27 vs. the `.get` in line 32.

(1) This loop accesses each element by applying `[`square brackets`]` to the induction variable `i`, and updates `i` by applying `++`:

```
1     int a[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
2     const size_t n = sizeof a / sizeof a[0];
3
4     for (size_t i = 0; i < n; ++i) {
5         cout << a[i] << "\n";
6     }
```

(2) This loop accesses each element by applying `->value` to the induction variable `p`, and updates `p` by applying `= p->next`:

```
7     struct {
8         int value;
```

```
 9         node *next;
10     } node;
11
12     node *head = 0;
13
14     for (node *p = head; p != 0; p = p->next) {
15         cout << p->value << "\n";
16     }
```

(3) This loop calls the C functions on pp. 85−89, and therefore has two induction variables `x` and `y`. It accesses each element by passing them to the function `term_get`. The code to update them is too complicated to fit at the end of line 21, so I moved it to lines 24−28:

```
17 extern "C" {
18 #include "term.h"
19 }
20
21     for (unsigned x = 0, y = 0; y < term_height();) {
22         cout << term_get(x, y);
23
24         if (++x >= term_width()) {
25             x = 0;
26             ++y;
27             cout << "\n";
28         }
29     }
```

(4) This loop copies the standard input to the standard output, one character at a time. It accesses each element by calling the `.get` member function of `cin`. Although it is not obvious, the induction variable of this loop is `cin` since its internal state is changing as we read each character.

```
30 #include <iostream>
31 using namespace std;
32
33     for (char c; cin.get(c);) {//for (char c; cin.get(c).operator*();) {
34         cout.put(c);
35     }
```

The equivalent loop in C must have a `c` that is wider than a `char`, because a `char` is not big enough to hold every possible return value of `getchar`.

```
36 #include <stdio.h>
37
38     int c;
39
40     while ((c = getchar()) != EOF) {
41         putchar(c);
42     }
```

But once the induction variable—or variables—has been hidden as a private data member of an iterator, our loops will be identical except for the names of the data types:

```
43     for (my_container::const_iterator it = c.begin(); it != c.end(); ++it) {
44         cout << *it << "\n";
45     }
46
47     for (your_container::const_iterator it = c.begin(); it != c.end(); ++it){
48         cout << *it << "\n";
```

```
    }
```

We can then write them once and for all as a template.

```
template <class CONTAINER>
void print_loop(const CONTAINER& c)
{
    for (typename CONTAINER::const_iterator it = c.begin(); it != c.end();
        ++it) {
        cout << *it << "\n";
    }
}
```

## 8.3 Iterator Categories

An algorithm is a template function that receives an iterator. But it has no idea what the iterator's data type is. All it has to work with is the opaque word `IT`:

```
template <class IT>
void myalgorithm(IT it)
{
```

What can the algorithm do with the iterator? If the algorithm applies the `--` operator to an iterator of type "`IT`", will it compile? The risk is real: we have seen three iterators cannot be decremented, starting with our `node::iterator` on p. 806. How can an algorithm find out what an iterator of type `IT` can, and can not, do?

The iterator data types are divided into five *categories*, depending on the operators that can be applied to the iterator and how much time they take. A category is not a data type; it is an infinite set of data types.

**Input iterator**

For a data type `IT` to qualify as an *input iterator,* it must be able to do the following.

(1) An iterator must be copy constructible and assignable. We must be able to compare two iterators with `==` and `!=`, dereference an iterator with `*`, and increment an iterator with `++` (prefix and postfix).

When we dereference the iterator, we must get a value. If the iterator is a pointer, it cannot be a pointer to `void`. If the iterator is an object, its `operator*` cannot return `void`.

For example,

```
template <class IT>
void my_algorithm(IT first, IT last) //pass by value
{
    for (; first != last; ++first) {
        cout << *first << "\n"; //use value of *first, e.g., output it
    }
```

There is no requirement that we be able to compare iterators with the four other comparison operators <, <=, >, or >=. There is no requirement that we be able to dereference an iterator with the other dereferencing operator `[]`, or decrement it with `--` (prefix or postfix). There is no requirement that anything sane will happen if we dereference or increment an iterator that is equal to a container's `end` iterator. "No requirement" means that anything can happen without disqualifying the iterator from being an input iterator. "Anything" includes, but is not limited to, failure to compile, crashing the program, undefined behavior, and working as naïvely expected.

(2) The `operator++` in our class `istream_iterator_date` had a side effect: it discarded the previous `date` and input the next one. The `operator*` had no side effect. But for other classes of input

iterators, it might be the `operator*` that has a (possibly destructive) side effect. Consequently, there is no requirement that we be able to dereference an input iterator more than once. Each element can be read from the container only once. It’s like reading a byte from a Unix pipe: the read is destructive.

```
template <class IT>
void my_algorithm(IT first, IT last)
{
    for (; first != last; ++first) {
        cout << *first << "\n";
        cout << *first << "\n";   //no guarantee that this will work
    }
```

To use the value of an element more than once, we could copy it. `iterator_traits<IT>` must have a `value_type` member:

```
template <class IT>
void my_algorithm(IT first, IT last)
{
    for (; first != last; ++first) {
        typename iterator_traits<IT>::value_type t = *it;
        cout << t << "\n";
        cout << t << "\n";
    }
```

Or we could make a pointer to the value read from the container, and dereference the pointer more than once. `iterator_traits<IT>` must have a `pointer` member or a `reference` member. Just be careful to apply the `*` operator to the iterator only *once.*

```
template <class IT>
void my_algorithm(IT first, IT last)
{
    for (; first != last; ++first) {
        typename iterator_traits<IT>::pointer p = &*first;
        cout << *p << "\n";
        cout << *p << "\n";   //this will work
    }
```

There is no requirement, however, that the pointer will still point to the same element after the iterator has been incremented. (And anything might happen if we attempt to increment or dereference an iterator that is equal to `last`.)

```
template <class IT>
void my_algorithm(IT first, IT last)
{
    while (first != last) {
        typename iterator_traits<IT>::pointer p = &*first;

        //this *p will work
        cout << *p << "\n";

        ++first;

        //no guarantee that this *p will still work
        cout << *p << "\n";
    }

    cout << *first << "\n";   //unpredictable behavior
    ++first;                  //unpredictable behavior
```

(3) We must have a reference data type

`iterator_traits<IT>::reference`

that can refer to each element of the container, subject to the same caveat as the `pointer`.

```
template <class IT>
void my_algorithm(IT first, IT last)
{
    for (; first != last; ++first) {
        typename iterator_traits<IT>::reference r = *first;
        cout << r << "\n";
    }
```

(4) We must have a data type

`iterator_traits<IT>::value_type`

that can hold the values in the container. If this type is assignable, we can say

```
template <class IT>
void my_algorithm(IT first, IT last)
{
    for (; first != last; ++first) {
        typename iterator_traits<IT>::value_type t = *first;
        cout << t << "\n";
    }
```

(5) The data type

`iterator_traits<IT>::difference_type`

must be a signed integral data type (p. 61) that can hold the distance in elements between any two iterators referring to elements in the same container, even the largest possible container. For example, a variable of type `difference_type` must be big enough—but no bigger than necessary—to count the number of elements read from any container.

```
template <class IT>
void my_algorithm(IT first, IT last)
{
    typename iterator_traits<IT>::difference_type n = 0;
    for (; first != last; ++first) {
        ++n;
    }
```

(6) The expression `*it++` must perform the dereference *before* the increment, despite the higher precedence of the postfix `++` operator. The expression

```
    x = *it++;
```

must behave as if we had said

```
    temp = *it;
    ++it;
    x = temp;
```

To get the desired effect, the postfix `operator++` would perform no increment. It would return a proxy object (pp. 828−829) containg a pointer or reference to the iterator. The proxy's `operator*` would then dereference the iterator; the proxy's destructor would increment the iterator. Disaster would result from our normal procedure of copying the iterator, incrementing the original, and then dereferencing the copy:

```
    //Can’t increment one copy and dereference the other copy.
    temp = it;
    ++it;
    x = *temp;
```

(7) The data type

`iterator_traits<IT>::iterator_category`

must be a typedef for the data type `input_iterator_tag`.

If we copy an input iterator, there is no requirement that we be able to use both copies. (“Use” means test, dereference, and increment.) There is no requirement that

`it1 == it2`

must imply

`++it == ++it2`

Our class `istream_date_iterator` was an input iterator; the template class `istream_iterator` will be another example. As we shall see, these classes will not qualify as iterators of any other category.

Our classes `node::iterator` and `node::const_iterator` also qualify as input iterators. So will `list<int>::iterator`, `vector<int>::iterator`, and the data type `int *`. These types will qualify as iterators of other categories as well. But the data type `int *const` is not an input iterator. We cannot apply the `++` operator to it, so it is merely a *trivial* iterator.

**▼ Homework 8.3a: write an algorithm that will accept input iterators**

We found a date in an input file by calling the `find` algorithm in lines 32−39 of `main4.C` on p. 825. But we did not find where in the file the date was located. The problem is that the return value of this call to `find` was merely an input iterator. This category of iterator can read values from a container, but does not mark a position in the container.

Write another algorithm, `find_distance`, that will give us this information. It will be a template function like the `find` on p. 809, accepting the same three arguments. The return type of `find_distance` should be the `difference_type` for the type of iterators passed to `find_distance`. You have already seen how to do this: the return type of the `distance` algorithm on p. 810 was the `difference_type` for the type of iterators passed to `distance`. `find_distance` will return the position of the desired value in the container, or −1 if the value is not found. (The position numbers should start at zero.)

Be sure that `find_distance` will work correctly if its first two arguments are merely input iterators. For example, do not try to copy an iterator and then use both copies. Do not try to read the same value more than once.

▲

**Output iterator**

For a data type `IT` to qualify as an *output iterator,* we must be able to do the following with iterators of that type.

(1) An iterator must be copy constructible and assignable. We must be able to use `*it` as the left operand of an assignment (only `=`, not `+=` or the other assignment operators). Finally, we must be able to increment `it` with `++` (prefix and postfix). For example,

```
template <class IT>
void my_algorithm(IT it)   //pass by value
{
    for (; as long as we want to loop; ++it) {
        *it = some value, maybe a different one each time;
    }
```

There is no requirement that we be able to compare two iterators. There is no requirement that we be able to use the expression `*it` when it is *not* the left operand of an assignment. There is no requirement that we be able to assign more than once to the same element:

```
template <class IT>
void my_algorithm(IT it)
{
    for (; as long as we want to loop; ++it) {
        *it = some value;
        *it = some value; //no guarantee that this will still work
    }
```

There is no requirement that we can have two consecutive increments without an intervening assignment:

```
template <class IT>
void my_algorithm(IT it)
{
    for (; as long as we want to loop; ++it) {
        *it = some value;
        ++it;   //no guarantee that we can skip an element
    }
```

Similarly, there is no requirement that we can skip the first element.

```
template <class IT>
void my_algorithm(IT it)
{
    //no guarantee that we can skip the first element
    for (++it; as long as we want to loop; ++it) {
        *it = some value;
    }
```

An input iterator, on the other hand, can definitely skip over an element that it doesn't want to read.

(2) The expression `*it++ = t` must behave as if we has said

```
    *it = t;
    ++it;
```

In other words, the dereference and assignment must be performed *before* the increment.

In some cases this would be awkward to implement, because the expression `*it++` always executes the `++` before the `*`. (The postfix `++` has higher precedence than the `*`.) To get the desired effect, we would have to let the postfix `++` perform no increment at all, and let it return a proxy object (pp. 828−829). The `operator*` of the proxy will then return another proxy object. The `operator=` of the second proxy object will increment the iterator, dereference it, and perform the assignment. Fortunately, however, none of this was necessary for our class `ostream_iterator_int`. For unrelated reasons, the `operator++` functions of that class did nothing, so it didn't matter whether they execute before or after the dereference and assignment.

(3) The data type

```
iterator_traits<IT>::iterator_category
```

must be a typedef for the data type `output_iterator_tag`. There is no requirement that `iterator_traits<IT>` have any other members.

Our class `ostream_iterator_int` was an output iterator; the template class `ostream_iterator` will be another example. As we shall see, these classes will not qualify as iterators of any other category.

Our class `node::iterator` also qualifies as an output iterator. So will `list<int>::iterator`, `vector<int>::iterator`, and the data type `int *`. These types will qualify as iterators of other categories as well. But the `const_iterator` classes are not output iterators. Neither is the data type `const int *`.

**Forward Iterator**

For a data type `IT` to qualify as a *forward iterator,* we must be able to do the following with iterators of that type.

(1) A forward iterator must be able to do everything that an input iterator or an output iterator can do. It must therefore *be* an input iterator and an output iterator. (There is one exception; see below.) For example,

```
template <class IT>
void my_algorithm(IT first, IT last) //pass by value
{
    for (; first != last; ++first) {                        //compare
        typename iterator_traits<IT>::value_type t = *first; //read
        *first = typename iterator_traits<IT>::value_type(); //write

        if (++first == last) {                               //skip
            break;
        }
    }
```

(2) A forward iterator cannot exhibit the abnormal behaviors that would be tolerated in an input or output iterator. To start with,

```
it1 == it2
```

must imply

```
++it == ++it2
```

If we copy a forward iterator, we can use both copies without them interfering with each other. Finally, we can read or write the same element more than once, in any order. We can therefore use two or more forward iterators to loop through the same container at the same time. An example is the pair of iterators `first` and `previous` in the following homework.

There is one exception to the requirement that a forward iterator be able to do all the work of an output iterator. A forward iterator can be read-only and still qualify as being forward. A read/write forward iterator is said to be *mutable;* a read-only one is *immutable.* (These terms will also apply to ‘‘bidirectional’’ and ‘‘random access’’ iterators.) An example of an immutable forward iterator was our `node::const_iterator` back on pp. 815−816. But even if it is immutable, a forward iterator can still do many things that a mere input iterator cannot. It can read the same value twice.

```
    for (node::const_iterator it = begin; it != end; ++it) {
        cout << *it << "\n"
            << *it << "\n";
    }
```

(3) The data type

```
iterator_traits<IT>::iterator_category
```

must be a typedef for the data type `forward_iterator_tag`. Since a forward iterator is an input iterator, its `iterator_traits` must also gave the members `value_type`, `difference_type`, `pointer`, and `reference`.

Our classes `node::iterator` and `node::const_iterator` were input iterators and forward iterators. The same is true of `slist<int>::iterator`, which some vendors supply as part of the STL. As we shall see, these classes will not qualify as iterators of any other category.

Classes `list<int>::iterator`, `vector<int>::iterator`, and the data type `int *` are also input and forward iterators. These types will qualify as iterators of other categories too. But `istream_iterator_date` and `ostream_iterator_int` are not forward iterators. We cannot copy them and use both copies.

### ▼ Homework 8.3b: adjacent find

The `adjacent_find` algorithm in the standard library takes a pair of forward iterators referring to a range of elements. It searches for the first occurence of two adjacent equal values. Although the iterators do not write into the container, they must be forward, not merely input. This is because they are copied (`previous = first` in line 9), and then one copy is dereferenced (`*previous` in line 10) after the other copy has been incremented (`++first` in line 9).

Our input iterator `istream_iterator_date` just happens to work as an argument to `adjacent_find`, but only because we were lucky. `istream_iterator_date` reads from the input stream in `operator++`. Its `operator*` does nothing, so there is no harm in copying an iterator and calling the `operator*` of both copies. But there may be other input iterators whose `operator*` performs detectable work. In this case, we could not access the same element twice by calling `operator*` twice.

```
 1 //Excerpt from <algorithm>
 2 //IT must be a forward iterator, and
 3 //typename iterator_traits<IT>::value_type must be equality comparable.
 4
 5 template <class IT>
 6 IT adjacent_find(IT first, IT last)
 7 {
 8     if (first != last) {   //if there are elements,
 9         for (IT previous = first; ++first != last; previous = first) {
10             if (*first == *previous) {
11                 return previous;
12             }
13         }
14     }
15
16     return last;
17 }
```

To make the algorithm applicable to many more types of iterators, rewrite it to accept iterators that are merely input iterators. Instead of saving a copy of the previous iterator, save a copy of the previous value that was read from the container. The price you will pay is that the data type `iterator_traits<IT>::value_type` will now have to be copy constructible and assignable. List these requirements in the comment. To avoid conflict, give the algorithm a different name.

▲

### Bidirectional Iterator

A *bidirectional iterator* meets all the qualifications of a forward iterator; every bidirectional iterator *is* a forward iterator. In addition, a bidirectional iterator must also accept the operator `--`, both prefix and postfix. The data type

```
iterator_traits<IT>::iterator_category
```

must be a typedef for the data type `bidirectional_iterator_tag`.

Classes `list<int>::iterator`, `vector<int>::iterator`, and the data type `int *` are input, forward, and bidirectional iterators. The last two types will qualify as iterators of another category as well. But `node::iterator` was not bidirectional.

**Random Access Iterator**

A *random access iterator* meets all the qualifications of a bidirectional iterator; every random access iterator *is* a bidirectional iterator. In addition, we must be able to apply three groups of additional operators to the iterator.

(1) We must be able to make the iterator jump. We must be able to say

```
 it + d
 it - d
it += d
it -= d
```

where `d` is an expression of type `iterator_traits<IT>::difference_type`.

(2) We must be able to find the relative position and distance between two iterators that refer to elements in the same container. We must be able to compare iterators with all six relational operators:

```
==     <     >
!=    >=    <=
```

An algorithm must also be able to find the distance in elements between two iterators by subtracting them (`it1 - it2`), yielding a result of type `iterator_traits<IT>::difference_type`.

(3) We must be able to apply the `[]` operator to the iterator. Any iterator can access the element to which it refers: `*it`. A random access iterator must also be able to access other elements: `it[0]`, `it[10]`, `it[-10]`.

To qualify as random access, however, the iterator must be able to do still more. For any value `n` of type `iterator_traits<IT>::difference_type`, we must be able to execute `it += n` just as fast as `++it`. We can certainly do this if the iterator is a pointer to an array element, for example. In fact, a pointer is the classic example of a random access iterator. But for an iterator that refers to an element in a linked list, `it += n` has to be slower than `++it`. We cannot travel from one element to another without visiting every intervening element. As the distances become greater, the travel time increases. The standard library `list<int>::iterator` is therefore merely a bidirectional iterator. The `map` iterator is also merely bidirectional because its underlying data structure is a tree. Again, we cannot go from one element to another without visiting every intervening one.

All the other operators must work in constant time. Cor example, the operators `<`, `<=`, `>`, `>=`, and the `−` that measures the distance between two iterators, must be as fast as `==` and `!=`. It must be possible to tell which of two iterators is first without visiting all the intervening elements. And `it[10]` must be as fast as `it[0]` and `*it`.

**▼ Homework 8.3c: a sort that accepts bidirectional iterators**

The `sorter` template function in `sorter.h` on p. 762 accepted only random access iterators, since it applied the operators `<` and `[]` to them. Observe the error messages you get when passing it a pair of bidirectional iterators such as `list<int>::iterator`’s.

Rewrite the algorithm to accept iterators that are merely bidirectional. The `<` in line 17 of `sorter.h` can stay, but the ones in line 15 and 16 will have to go. The `it[0]` in line 17 can be changed to `*it`. How would you get rid of the `[1]` in `it[1]`?

Test the algorithm by passing it a pair of bidirectional iterators such as `list<int>::iterator`’s.

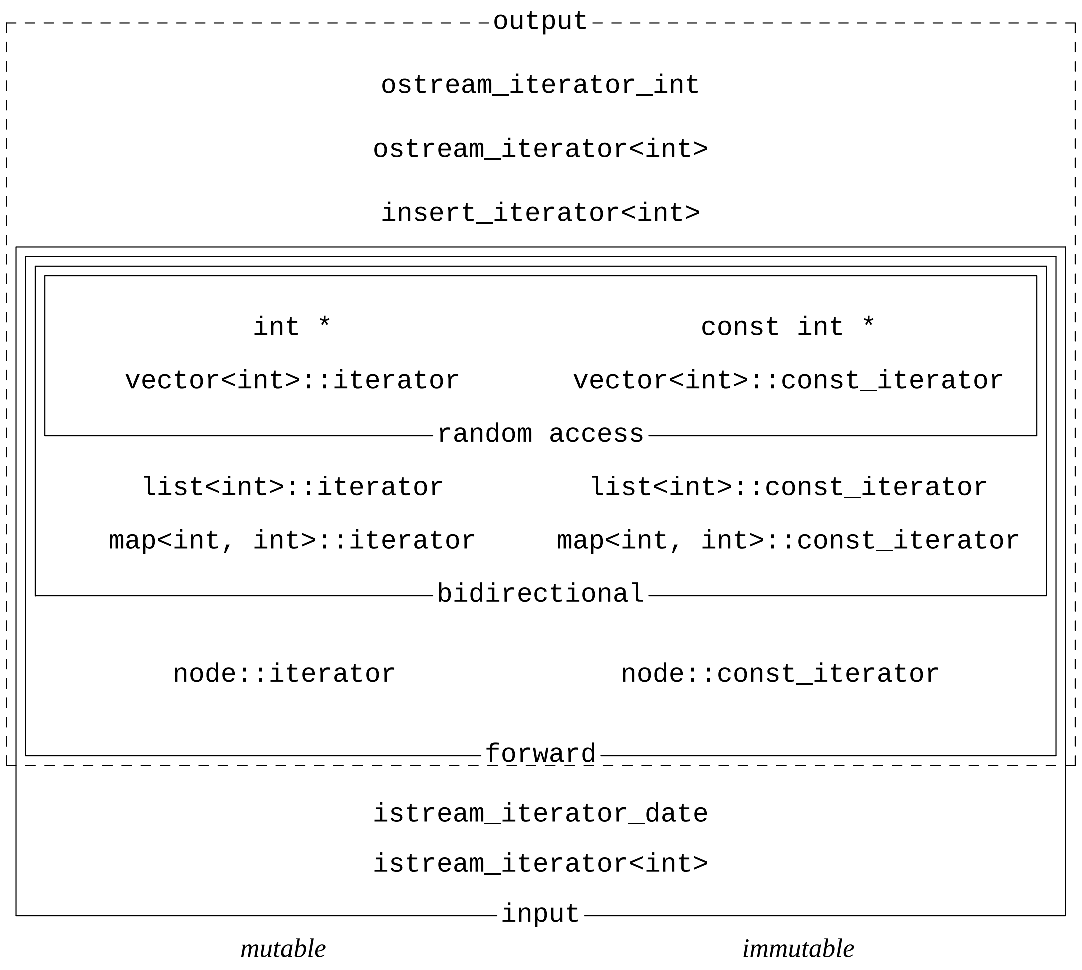
▲

▼ **Homework 8.3d: a sort that accepts forward iterators**

Now rewrite the `sorter` template function to accept iterators that are merely forward. You will no longer be able to apply the operator `--` (prefix or postfix) to the iterators. Hint: create a variable of the iterator's `difference_type` to count how many times your loops have iterated.
▲

## The hierarchy of iterator categories

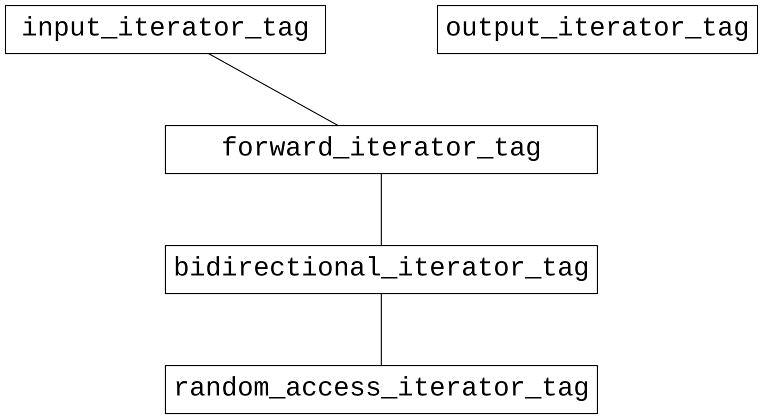
Each iterator category is an infinite set of data types. The five sets are overlapping and nested. For legibility, the set of output iterators is dashed



An iterator is allowed to be overqualified for its job. For example, a read/write pointer and a standard library `vector<int>::iterator` are random access iterators. But they are also completely legitimate bidirectional iterators. In fact, they belong to all five categories. We say that they are *models* of all five.

The standard library has definitions for five *tag classes,* one for each iterator category. Even though they have no members, each tag class is a different data type and can be used for function name overloading. For example, we could have two functions with the same name if one took an argument of class `random_access_iterator_tag` and the other an argument of class `bidirectional_iterator_tag`. The tags will be used for no other purpose.

The inheritance relationships between the tag classes correspond to the inclusion relationships between the iterator categories. For example, every `random_access_iterator_tag` object is also a `bidirectional_iterator_tag` object, and every random access iterator is also a bidirectional iterator. (Although every forward iterator is both an input iterator and an output iterator, class `forward_iterator_tag` is derived only from class `input_iterator_tag`. No one remembers why it was not also derived from class `output_iterator_tag`.)



```
//Excerpt from <iterator>

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 `iterator_category` member of an iterator's `iterator_traits` must be a typedef for one of the five tag classes. For example, class `iterator_traits<node::iterator>` originally had the following member (line 40 of `node.h` on p. 806).

```
    typedef forward_iterator_tag iterator_category;
```

A simpler way to accomplish the same result is to derive the iterator from a base class (line 13 of `node3.h` on p. 814).

```
    class iterator: public std::iterator<forward_iterator_tag, int> {
```

**▼ Homework 8.3e: define a category tester**

Define a class that acts as an input iterator, with a specialization of `iterator_traits` to go with it. The class must output an error message or fail to compile if the user tries to make it do anything that an input iterator does not need to do. This includes dereferencing the iterator before checking for end-of-range; reading the same value more than once; copying the iterator and then dereferencing and/or incrementing both copies. Do the same for the other four categories.

▲

## 8.4 Algorithms in the Standard Template Library

### 8.4.1 `copy`, Inserters, Stream Iterators, and Reverse Iterators

The algorithms are template functions. The ones in the standard library are defined in the header files `<algorithm>` and `<numeric>`. Here is a simple definition for `copy`; compare the other algorithms on pp. 808−811.

The template argument `INPUT` represents a data type that is at least an input iterator. The other conventional names are `OUTPUT`, `FORWARD`, `BIDIRECTIONAL`, and `RANDOM`. It's okay if `INPUT` is more than just an input iterator. It could also be forward, bidirectional, or random access. Similarly, `OUTPUT` could be more than just an output iterator as long as it is mutable (line 5).

`INPUT` and `OUTPUT` do not have to refer to values of exactly the same type. But we must be able to assign a value of type `typename iterator_traits<INPUT>::value_type` to the expression `*result`. For example, `INPUT` and `OUTPUT` could be `short *` and `int *`, or `vector<short>::const_iterator` and `int *`. But if they were `wabbit *` and `int *`, the call to the `copy` function would not compile.

An algorithm always assumes that an iterator is small enough to pass and return by value (line 2.) We also assume that an iterator is incrementable (line 4), which is why our class `output_iterator_int` had to have an `operator++` even though it did nothing (p. 830).

```
1 template <class INPUT, class OUTPUT>
2 OUTPUT copy(INPUT first, INPUT last, OUTPUT result)
3 {
4     for (; first != last; ++first, ++result) {
5         *result = *first;
6     }
7
8     return result;
9 }
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/copy/copy.C`

```
#include <iostream>
#include <cstdlib>
#include <vector>
#include <list>
#include <algorithm>  //for copy
using namespace std;

int main()
{
    int a[] = {10, 20, 30};
    const size_t n = sizeof a / sizeof a[0];

    vector<int> v(3);                     //born containing 0, 0, 0
    copy(a, a + n, v.begin());            //Can copy an array into a vector.
    for (vector<int>::const_iterator it = v.begin(); it != v.end(); ++it) {
        cout << *it << "\n";
    }
    cout << "\n";

    list<int> li(3);                      //born containing 0, 0, 0
    copy(v.begin(), v.end(), li.begin()); //Can copy a vector into a list.
    for (list<int>::const_iterator it = li.begin(); it != li.end(); ++it) {
        cout << *it << "\n";
    }
    cout << "\n";

    vector<int> big(5);         //Can copy part of a container into another.
    copy(v.begin(), v.begin() + 2, big.begin() + 3);
    for (vector<int>::const_iterator it=big.begin(); it != big.end(); ++it){
        cout << *it << "\n";
    }
    cout << "\n";

    //copy(big.begin(), big.end(), v.begin());   //may crash the program
    return EXIT_SUCCESS;
}
```

```
10
20
30

10
20
30

0
0
0
10
20
```

**▼ Homework 8.4.1a: call the copy algorithm**

(1) When we put a pointer data member `p` into class `stack`, we had to write a copy constructor (p. 153) and an `operator=` (p. 311) for that class. These member functions have `for` loops to copy the data member `a`. Replace each loop with a call to the `copy` algorithm.

(2) The `next` member function of class `life` has nested `for` loops copy to one array into another (pp. 144−147). Replace the two loops with a single call to the `copy` algorithm.

The first argument of `copy` should be the address of the first element of the array. Since the array is two-dimenstional, the first element has two subscripts. The static data members `ymax` and `xmax` were created on pp. 239 and 423−424.

```
    copy(&newmatrix[0][0], &newmatrix[ymax][0], &matrix[0][0]);
```

As in C, a leading & and a trailing `[0]` will cancel each other out.

```
    copy(newmatrix[0], newmatrix[ymax], matrix[0]);
```

The `copy` algorithm should also be called by the `life` constructor that takes an array as an argument.
▲

**Overwrite with an iterator**

The common ground on which he had at last brought both sides together was not ground he had discovered, but ground he had created.

—Robert A. Caro, *The Years of Lyndon Johnson: Master of the Senate*, p. 1005

We will use the copy algorithm to introduce three new kinds of iterators: inserters, stream iterators, and reverse iterators.

Let's recall what the normal kind of iterator does. The `v.begin()` in line 13 returns an anonymous iterator referring to the first element of the vector. We can apply a `+` to this iterator because a vector iterator is random access. The sum is another anonymous iterator, referring to the second element of the vector.

When line 14 writes into a container using this iterator, the element `21` to which the iterator refers is overwritten. This has no effect on the vector's size or capacity.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/inserter/overwrite.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 using namespace std;
 5
 6 int main()
 7 {
 8     const int a[] = {10, 21, 30};
 9     const size_t n = sizeof a / sizeof a[0];
10     vector<int> v(a, a + n);
11
12     cout << "size == " << v.size() << ", capacity == " << v.capacity() << "\n";
13     vector<int>::iterator it = v.begin() + 1;   //Refer to the 21.
14     *it = 20;                                   //Overwrite the 21.
15     cout << "size == " << v.size() << ", capacity == " << v.capacity() << "\n";
16
17     for (it = v.begin(); it != v.end(); ++it) {
18         cout << *it << "\n";
19     }
20
21     return EXIT_SUCCESS;
22 }
```

```
size == 3, capacity == 3
size == 3, capacity == 3
10
20
30
```

The new kind of iterator is called an *inserter;* its data type has the formidable name in line 14. An easier way to approach it is through the values that its constructor puts into it: a reference to the vector `v` and the iterator `v.begin() + 1` which we saw in the above line 13. The inserter refers to the element `30`.

When line 15 writes into a container with an inserter, the element to which the inserter refers is not overwritten. Instead, a new element is inserted in front of it and the container becomes larger. After the insertion, the iterator continues to refer to the 30.

Performing the insertion in front of the inserter allows us to insert the new element anywhere in the container. For example, we can insert a new element at the end of a container by using an inserter that refers to the `end` ‘‘element’’. An inserter that inserted the new element after the inserter would not be able to insert the new element at the beginning of a container.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/inserter/inserter.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <vector>
4 #include <iterator>   //for insert_iterator
5 using namespace std;
6
7 int main()
8 {
 9     const int a[] = {10, 30, 40};
10     const size_t n = sizeof a / sizeof a[0];
11     vector<int> v(a, a + n);
12
13     cout << "size == " << v.size() << ", capacity == " << v.capacity() << "\n";
14     insert_iterator<vector<int> > in(v, v.begin() + 1);  //Refer to the 30.
15     *in = 20;                                 //Insert 20 in front of the 30.
16     cout << "size == " << v.size() << ", capacity == " << v.capacity() << "\n";
17
18     for (vector<int>::const_iterator it = v.begin(); it != v.end(); ++it) {
19         cout << *it << "\n";
20     }
21
22     return EXIT_SUCCESS;
23 }
```

```
size == 3, capacity == 3
size == 4, capacity == 6
10
20
30
40
```

**Three types of inserters**

There are three types of inserter:

(1) the plain old *inserter* in lines 14−20, for inserting anywhere in a container;

(2) the *front inserter* in lines 21−24, for inserting new elements at the front of a container;

(3) the *back inserter* in lines 26−28, for inserting new elements at the end of a container.

There is no such thing as a front inserter for a `vector`, so this time we'll have to demonstrate with a `list`. Bear in mind that a `list` iterator is not random access, so the `+` in the above line 13 will no longer compile. We'll have to use the `++` in the following line 14.

Note that the constructor for a plain old inserter takes two arguments (line 14), while the ones for the other types take one argument each (lines 22 and 26).

An inserter is an output iterator. Code that receives an inserter will therefore increment it after each dereference. This is harmless because incrementing an inserter does nothing (line 19).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/inserter/inserters.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <list>
 4 #include <iterator> //insert_iterator, front_insert_iterator, back_insert_iterator
 5 using namespace std;
 6
 7 int main()
 8 {
 9     const int a[] = {30, 70, 80};
10     const size_t n = sizeof a / sizeof a[0];
11     list<int> li(a, a + n);
12     list<int>::iterator it = li.begin();
13
14     insert_iterator<list<int> > in(li, ++it);  //Refer to the 70.
```

```
    *in = 40;   //Insert 40 immediately in front of the 70.
                //At this point, in still refers to the 70.
    *in = 50;   //Insert 50 immediately in front of the 70.
                //At this point, in still refers to the 70.
    ++in;       //++ does nothing to an inserter: in still refers to the 70.
    *in = 60;   //Insert 60 immediately in front of the 70.

    front_insert_iterator<list<int> > fi(li);
    *fi = 20;   //Insert 20 at the front of the list.
    *fi = 10;   //Insert 10 at the front of the list.

    back_insert_iterator<list<int> > bi(li);
    *bi = 90;   //Insert 90 at the end of the list.
    *bi = 100;   //Insert 100 at the end of the list.

    for (it = li.begin(); it != li.end(); ++it) {
        cout << *it << " ";
    }
    cout << "\n";

    return EXIT_SUCCESS;
}
```

```
10 20 30 40 50 60 70 80 90 100
```

**Construct an anonymous inserter and pass it to a function**

We can always give a name to a variable. For example, lines 8−10 declare three inserters, which we then pass to an algorithm.

```
 1 #include <list>
 2 #include <iterator>   //for inserter, front_inserter, back_inserter
 3 using namespace std;
 4
 5     list<int> li(a, a + n);
 6     list<int>::iterator it = li.begin();
 7
 8           insert_iterator<list<int> > in(li, ++it);
 9     front_insert_iterator<list<int> > fi(li);
10      back_insert_iterator<list<int> > bi(li);
11
12     my_algorithm(in);
13     my_algorithm(fi);
14     my_algorithm(bi);
```

But if an inserter is used only once, there's no reason to declare a name for it. We can simply call its constructor, which returns an anonymous inserter to us. We then pass the newborn inserter to `f`:

```
15     my_algorithm(      insert_iterator<list<int> >(li, ++it));
16     my_algorithm(front_insert_iterator<list<int> >(li));
17     my_algorithm( back_insert_iterator<list<int> >(li));
```

Here is an even easier way to do the same thing. The following functions construct and return the same three kinds of inserters. They are template functions, like `min` and `make_pair`, whose arguments tell them what type of return value we want. For example, the argument `li` of data type `list<int>` in line 19 tells `front_inserter` to construct and return an anonymous `front_insert_iterator<list<int> >`.

```
18     my_algorithm(      inserter(li, ++it)); //construct insert_iterator<list<int> >
19     my_algorithm(front_inserter(li)); //construct front_insert_iterator<list<int> >
20     my_algorithm( back_inserter(li)); //construct back_insert_iterator<list<int> >
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/inserter/copy.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include <iterator>    //for inserter and back_inserter
 5 #include <algorithm>   //for copy
 6 using namespace std;
 7
 8 int main()
 9 {
10     const int a[] = {10, 21, 31, 41, 50, 90};
11     const size_t na = sizeof a / sizeof a[0];
12     vector<int> v(a, a + na);
13
14     //Overwrite the 21, 31, 41 with 20, 30, 40.
15     //The third argument in line 18 refers to the 21.
16     const int b[] = {20, 30, 40};
17     const size_t nb = sizeof b / sizeof b[0];
18     copy(b, b + nb, v.begin() + 1);
19
20     //Insert 60, 70, 80 in front of the 90.
21     //The third argument in line 24 refers to the 90.
22     const int c[] = {60, 70, 80};
23     const size_t nc = sizeof c / sizeof c[0];
24     copy(c, c + nc, inserter(v, v.begin() + 5));
25
26     //Insert 100, 110, 120 at the end of the vector.
27     const short d[] = {100, 110, 120};
28     const size_t nd = sizeof d / sizeof d[0];
29     copy(d, d + nd, back_inserter(v));
30
31     for (vector<int>::const_iterator it = v.begin(); it != v.end(); ++it) {
32         cout << *it << " ";
33     }
34     cout << "\n";
35     return EXIT_SUCCESS;
36 }
```

```
10 20 30 40 50 60 70 80 90 100 110 120
```

**When not to use an inserter**

If all you want to do is insert a value into a container, it's faster to call the container's `insert` member function. We saw class `list`'s on p. 444; class `vector` has one too (line 12). In fact, there is also an `insert` function that can do many insertions at once (line 16). This is much faster than applying a `*` to an inserter over and over again.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/inserter/insert.C`

```
#include <iostream>
#include <cstdlib>
#include <vector>
using namespace std;

int main()
{
    const int a[] = {10, 60};
    const size_t n = sizeof a / sizeof a[0];
    vector<int> v(a, a + n);

    v.insert(v.begin() + 1, 20);

    const int b[] = {30, 40, 50};
    const size_t nb = sizeof b / sizeof b[0];
    v.insert(v.begin() + 2, b, b + nb);

    for (vector<int>::const_iterator it = v.begin(); it != v.end(); ++it) {
        cout << *it << " ";
    }
    cout << "\n";

    return EXIT_SUCCESS;
}
```

```
10 20 30 40 50 60
```

An inserter should be used only as an argument of an algorithm that must be capable of either overwriting or inserting. We pass a normal iterator to the algorithm when we want to overwrite one or more elements; we pass an inserter when we want to insert new ones. Our example will be the `copy` algorithm.

## Stream iterators

An `ostream_iterator` is a conduit leading to an output stream: `cout`, `cerr`, `clog`, or to an output file. An `istream_iterator` is a conduit leading in from an input stream: `cin` or an input file. We wrote our own stream iterators on pp. 816−832, but the ones in the standard libary are better because they are templates. They can read and write any data type. Of course, any given `ostream_iterator` can write values of only one data type to the output stream. For example, the `ostream_iterator<int>` in line 9 can output only `int`'s. (See pp. 1047−1048 for the rarely-used second argument of the template.)

The constructor for class `ostream_iterator` takes two arguments. The first is the output stream; the optional second argument is a string to be output after each item.

On some platforms, the `++` in line 12 is required between every pair of assignments to `*it`. (Fortunately, `copy` always applies a `++` to its third argument.) On other platforms, the `++` may do nothing. For portability, the `++` should always be written.

Line 19 will not compile because an `ostream_iterator` is not an input iterator. This means that an `ostream_iterator` can be used as the third argument of `copy`, but not as the first or second.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stream_iterator/ostream_iterator1.C`

```
1 #include <iostream>
2 #include <fstream>    //for ofstream
```

```
 3 #include <cstdlib>
 4 #include <iterator>   //for ostream_iterator
 5 using namespace std;
 6 
 7 int main()
 8 {
 9     ostream_iterator<int> it(cout, " bottles of beer on the wall\n");
10 
11     *it = 100;
12     ++it;
13     *it = 99;
14     ++it;
15     *it = 98;
16     ++it;
17 
18     //--it;         //won't compile: this class has no operator-- function
19     //int i = *it; //won't compile: an ostream_iterator is not an input iterator
20 
21     ofstream of("outfile");
22     ostream_iterator<int> os(of, "\n");
23     *os = 100;
24     ++os;
25     *os = 99;
26     ++os;
27     *os = 98;
28     ++os;
29 
30     return EXIT_SUCCESS;
31 }
```

The above lines 11−12 may be combined to

```
32     *it++ = 100;
```

Instead of declaring the `ostream` object `of` in the above line 21 and using it only in 22, we should make it an anonymous temporary. Change lines 21−22 to

```
33     ostream_iterator<int> os(ofstream("outfile"), "\n");
```

The standard output produced by lines 9−16 is

```
100 bottles of beer on the wall
99 bottles of beer on the wall
98 bottles of beer on the wall
```

The `outfile` produced by lines 21−28 is

```
100
99
98
```

**Pass a stream iterator to an algorithm**

The above output could be done faster by writing directly to the output with `<<`. A stream iterator, like an inserter, is intended only for use by an algorithm.

The following call to the `copy`algorithm will process the vector of `date`’s in line 19. The normal way to output the contents of a container is with the `for` loop and `cout` in lines 21−23.

To output the same data with a stream iterator, the type we will need is the `ostream_iterator<date>` in line 26. We could then loop through the container in 27−29. But these lines are for demo purposes only. Since we know that we’re writing to the standard output, it’s faster to use the `cout` loop in lines 21−23. An output stream iterator should be used only as an argument of a template function that must be capable of writing either to a container or to an output stream. We pass a normal iterator to the function when we want to write to a container; we pass an output stream iterator when we want to write to an output stream.

For example, the call to the `copy` algorithm in line 32, with the output stream iterator `os` in line 26, does all the work of the loop in 27−29.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stream_iterator/ostream_iterator2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include <algorithm>   //for copy and fill_n
 5 #include <iterator>    //for ostream_iterator
 6 #include "date.h"
 7 using namespace std;
 8
 9 int main()
10 {
11     const date a[] = {
12         date(date::july,       4, 1776),
13         date(date::october,   29, 1929),
14         date(date::december,   7, 1941),
15         date(date::july,      20, 1969),
16         date(date::september, 11, 2001)
17     };
18     const size_t n = sizeof a / sizeof a[0];
19     vector<date> v(a, a + n);
20
21     for (vector<date>::const_iterator it = v.begin(); it != v.end(); ++it) {
22         cout << *it << "\n";
23     }
24     cout << "\n";
25
26     ostream_iterator<date> os(cout, "\n");
27     for (vector<date>::const_iterator it = v.begin(); it != v.end(); ++it, ++os) {
28         *os = *it;
29     }
30     cout << "\n";
31
32     copy(v.begin(), v.end(), os);
33     cout << "\n";
34
35     fill_n(ostream_iterator<char>(cout), 80, ’*’);
36     cout << "\n";
37
38     return EXIT_SUCCESS;
39 }
```

```
7/4/1776         lines 21−23
10/29/1929
12/7/1941
7/20/1969
9/11/2001

7/4/1776         lines 26−29
10/29/1929
12/7/1941
7/20/1969
9/11/2001

7/4/1776         line 32
10/29/1929
12/7/1941
7/20/1969
9/11/2001

********************************************************************************
```

Here is a simple definition for the `fill_n` algorithm. Usually the first two arguments of an algorithm are a pair of iterators, `first` and `last`. But since we require nothing more of `IT` than that it be an output iterator, there would be no guarantee that a comparison of `first` and `last` would even compile. We therefore pass a count `n`, of any type that can be compared and decremented with `>`. We assume that `n` is integral and therefore fast enough to pass by value. See p. 881 for another `_n` algorithm.

```
//Excerpt from <algorithm>

template <class IT, class N, class T>
IT fill_n(IT it, N n, const T& t)
{
    for (; n > 0; --n) {
        *it = t;
        ++it;
    }

    return it;
}
```

The following program copies its standard input directly to the standard output, integer by integer. Along the way, it condenses all the whitespace between successive input integers into a single newline. Since each iterator is mentioned only once, it can be an anonymous temporary.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stream_iterator/copy.C`

```
#include <iostream>
#include <cstdlib>
#include <iterator>
#include <algorithm>
using namespace std;

int main()
{
    copy(
        istream_iterator<int>(cin),
        istream_iterator<int>(),
12         ostream_iterator<int>(cout, "\n")
13     );
14     return EXIT_SUCCESS;
15 }
```

Warning. The `istream_iterator<int>` in the above line 10 delivers only the integers read from input. It discards the whitespace between them. This is because the iterator calls an `operator>>`, which discards whitespace.

For integers, this is probably what we want. For characters, we will probably be dismayed when the whitespace eliminated.

```
16     copy(
17         istream_iterator<char>(cin),
18         istream_iterator<char>(),
19         ostream_iterator<char>(cout)
20     );
```

To read every character, whitespace or not, do the following before calling `copy`. See p. 359.

```
21     cin >> noskipws;
```

Another way to copy every character was in p. 329.

**Pass a stream iterator to a constructor**

I'd like to read integers from the standard input and store them into a vector `v`, stopping when the input is exhausted. It should ‘‘slurp’’ the entire input like the following statement in the language Perl.

```
#Perl example.
#The expression @v provides an "array context" for the expression <STDIN>.
@v = <STDIN>;
```

The two arguments of the vector's constructor are the beginning and end of the standard input.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stream_iterator/constructor.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include <iterator>
 5 #include <algorithm>
 6 using namespace std;
 7
 8 int main()
 9 {
10     vector<int> v(istream_iterator<int>(cin), istream_iterator<int>());
11     sort(v.begin(), v.end());
12     copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
13
14     return EXIT_SUCCESS;
15 }
```

Unfortunately, the syntax in the above line 10 did not define a `vector` named `v`. It declared a function named `v`. Then line 11 complained because a function has no members.

```
constructor.C: In function 'int main()':
constructor.C:11:9: error: request for member 'begin' in 'v', which is of
non-class type 'std::vector<int>(std::istream_iterator<int>,
std::istream_iterator<int> (*)())'
constructor.C:11:20: error: request for member 'end' in 'v', which is of
```

Why did line 10 think `v` was a function? The following is a simpler example of the same problem. The first four declarations declare the same function `f`, returning a `vector<int>`. Its first argument is an integer; the second is a pointer to a function that takes no arguments and returns `int`.

The declaration in line 5 omits the asterisk. Now that it's gone, the surrounding parentheses are no longer needed. The declaration in line 6 omits the name of `p`. The one in line 7 adds unnecessary but permissible parentheses around the name of `i` (p. 671). This is exactly the syntax we have in the above line 10. It declares a function. It does not call the constructor for a vector. Given a statement with two possible interpretations, declaration or function call, the language always treats it as a declaration. See pp. 671 and 807−808 for simpler examples.

An obscure rule of grammar lets us fix this. We cannot have parentheses around an argument in a declaration for a function. But we can have parentheses around an actual argument when the function is called. In particular, when calling a constructor, we can have parentheses around an argument of the constructor. In this example, we have three choices. Line 8 has parentheses around the entire first argument, `int (i)`. Line 9 has parentheses around the second argument, `int ()`. Line 10 has parentheses around both. Lines 8−10 are definitions for an object named `f`. The object's constructor takes two arguments, which in this case are the anonymous temporaries returned by the one- and zero-argument constructors for the data type `int`. We saw the one-argument constructor on p. 134 and the zero-argument one on p. 660.

```
 1 #include <vector>
 2 using namespace std;
 3
 4 vector<int> f( int  i  ,  int (*p)() );   //Declare a function named f.
 5 vector<int> f( int  i  ,  int   p () );   //Declare a function named f.
 6 vector<int> f( int  i  ,  int     () );   //Declare a function named f.
 7 vector<int> f( int (i) ,  int     () );   //Declare a function named f.

 8 vector<int> f((int (i)),  int     () );   //Define an object named f.
 9 vector<int> f( int (i) , (int     ()));   //Define an object named f.
10 vector<int> f((int (i)), (int     ()));   //Define an object named f.
```

To fix line 10 of the above `constructor.C`, parentheses around the first argument would be fine.

```
11     vector<int> v((istream_iterator<int>(cin)), istream_iterator<int>());
```

I love anonymous temporaries as much as the next man. But we can avoid the whole issue by giving names to the iterators.

```
12     const istream_iterator<int> begin(cin);
13     const istream_iterator<int> end;
14
15     vector<int> v(begin, end);
```

Although the above lines 11 and 15 now compile, they cannot know how many elements are in the vector until the input has been exhausted. Only then can they allocate a block of memory of the correct size. The values may have to be copied through a series of blocks of geometrically increasing sizes until they reach their final resting place. If the values are objects, this will be done by calling their copy constructor.

**Reverse iterators**

When you apply the operator `++` to a *reverse iterator,* it goes backward. When you apply `--`, it goes forward. A reverse iterator has to be bidirectional.

The `rbegin` member function of a container returns a reverse iterator that refers to the last element of the container. The `rend` function returns a reverse iterator that refers to the empty slot where the element before the first element would be. If the container is empty, `rbegin` would return the same iterator as `rend`, just as `begin` would return the same iterator as `end`.

Classes `vector`, `list`, `map`, and `string` have a `reverse_iterator` member (line 14). Classes `stack` and `queue` have no iterators at all, reverse or otherwise. There's also a `reverse_iterator` template class (lines 23−25) for creating a reverse iterator out of a pointer or any other bidirectional iterator.

The loop in line 14 is for demonstration purposes only. A `reverse_iterator` is intended for use only as the argument of an algorithm.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/reverse_iterator/reverse_iterator.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include <iterator>  //for ostream_iterator and reverse_iterator template
 5 #include <algorithm>
 6 using namespace std;
 7
 8 int main()
 9 {
10     const int a[] = {10, 20, 30};
11     const size_t n = sizeof a / sizeof a[0];
12     vector<int> v(a, a + n);
13
14     for (vector<int>::reverse_iterator it = v.rbegin();
15         it != v.rend(); ++it) {
16         cout << *it << "\n";
17     }
18     cout << "\n";
19
20     copy(v.rbegin(), v.rend(), ostream_iterator<int>(cout, "\n"));
21     cout << "\n";
22
23     copy(reverse_iterator<const int *>(a + n),
24          reverse_iterator<const int *>(a),
25         ostream_iterator<int>(cout, "\n"));
26
27     return EXIT_SUCCESS;
28 }
```

```
30
20
10

30
20
10

30
20
10
```

### ▼ Homework 8.4.1b: const_reverse_iterator

A `reverse_iterator`, like a plain old `iterator`, also comes in a `const_` flavor. But when we try to use it, something goes wrong on some compilers.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/reverse_iterator-`
`/const_reverse_iterator.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 using namespace std;
 5
 6 int main()
 7 {
 8     const int a[] = {10, 20, 30};
 9     const size_t n = sizeof a / sizeof a[0];
10     vector<int> v(a, a + n);
11
12     for (vector<int>::const_reverse_iterator it = v.rbegin();
13         it != v.rend(); ++it) {
14         cout << *it << "\n";
15     }
16
17     return EXIT_SUCCESS;
18 }
```

```
30
20
10
```

The `rend` member function of a `const` container returns a `const_reverse_iterator`. But our vector `v` in the above line 12 is not `const`. Its `rend` function returns a plain old `reverse_iterator`. See p. 314 for a previous example of a `const` and non-`const` objects having different member functions.

There is no `operator!=` that will compare a `vector<int>::const_reverse_iterator` with a `vector<int>::reverse_iterator`. Apply one of the first three fixes.

(1) Cast the `v` in the above line 13 to a `const` vector and then call its `rend` function. We actually cast it to a reference to a `const` vector, to avoid making a copy; see p. 81.

```
19     it != static_cast<const vector<int>&>(v).rend();
```

(2) Cast the `v.rend()` in line 13 to a `vector<int>::const_reverse_iterator`. We assume that an iterator is fast to copy, so we don't bother casting it to a reference.

```
20      it != static_cast<vector<int>::const_reverse_iterator>(v.rend());
```

(3) The cleanest solution is to leave line 13 the way it is, and define an `operator!=` function that compares the two types of iterators. It packages the cast in the above ¶ (2).

```
21 inline bool operator!=(vector<int>::const_reverse_iterator it1,
22                        vector<int>::      reverse_iterator it2) {
23     return it1 != static_cast<vector<int>::const_reverse_iterator>(it2);
24 }
```

(4) Ideally `operator!=` should be a template function, a superficial change to the above ¶ (3),

```
25 template <class T>
26 inline bool operator!=(typename vector<T>::const_reverse_iterator it1,
27                        typename vector<T>::      reverse_iterator it2) {
28     return it1 !=
29         static_cast<typename vector<T>::const_reverse_iterator>(it2);
30 }
```

But we can't do it. See "template argument deduction" in pp. 977−979.
▲

▼ **Homework 8.4.1c: other copies**

(1) What goes wrong if the source and destination ranges of `copy` overlap?

```
    int a[] = {10, 20, 30, 40, 50, 60};
    const size_t n = sizeof a / sizeof a[0];

    //Want to move the 10 to where the 30 is, etc.
    copy(a, a + 4, a + 2);
```

Fix it by calling `copy_backward` with the same three arguments as `copy`. This time, the arguments must be bidirectional iterators.

(2) What happens if you pass the arguments of copy to `reverse_copy` or `unique_copy`? (The first two arguments of `reverse_copy` must be bidirectional iterators.)

```
    const string a[] = {"hello", "hello", "hello", "goodbye", "goodbye"};
    const size_t n = sizeof a / sizeof a[0];
    unique_copy(a, a + n, ostream_iterator<string>(cout, "\n"));
```

(3) `remove_copy` takes four arguments.

```
    const string a[] = {"hello", "", "", "goodbye", ""};
    const size_t n = sizeof a / sizeof a[0];
    //Skip the empty lines.
    remove_copy(a, a + n, ostream_iterator<string>(cout, "\n"), "");
```

(4) `replace_copy` gives us instant Cockney.

```
    char a[] = "Henry Higgins\n";
    const size_t n = sizeof a / sizeof a[0] - 1;
    replace_copy(a, a + n, ostream_iterator<char>(cout), 'H', '\'');
```

(5) `rotate_copy` cuts the deck.

```
    char a[] = "housedog";
    const size_t n = sizeof a / sizeof a[0] - 1;
```

```
18     rotate_copy(a, a + 5, a + n, ostream_iterator<char>(cout));
19     cout << "\n";
```

▲

### 8.4.2 `find`, `find_if`, and Function Objects

The `find` algorithm searches for a value in a container.

```
 1 template <class INPUT, class T>
 2 INPUT find(INPUT first, INPUT last, const T& t)
 3 {
 4     for (; first != last; ++first) {
 5         if (*first == t) {
 6             break;
 7         }
 8     }
 9
10     return first;
11 }
```

The argument `t` does not necessarily have to be of the same data type as the elements in the container: we can search for a `double` value in a container of `int`'s. Had the algorithm been defined as follows, any third argument we supply would be forcibly converted to the element type. We would then receive a warning as the `double` was truncated to `int`.

```
12 //Not the definition in the standard library.
13
14 template <class INPUT>
15 INPUT find(INPUT first, INPUT last,
16     const typename iterator_traits<INPUT>::value_type& t)
17 {
18     for (; first != last; ++first) {
19         if (*first == t) {
20             break;
21         }
22     }
23
24     return first;
25 }
```

`find` repeatedly uses the `==` operator in the above line 5 to find what it's looking for. For example, the following line 13 uses `==` to compare the integer 30 with the integers in the array `a`, and line 32 uses `==` to compare the date of the moon landing with the dates in the vector `v`. To compile line 32, we must make class `date` equality comparable. We could write an `operator==` that takes two `date`'s, or an `operator int` that converts a `date` to an integer. An `==` operator applied to two dates would then be the built-in `==` that compares two integers.

The third argument of `find` in line 32 is the object `moon` constructed in line 31. But `moon` is mentioned nowhere else, so there is no need to give it a name. It could have been an an anonymous object like the date in line 41.

If our iterators were forward iterators and our container was already sorted, we could perform a binary search. This is faster than the `find` algorithm. For example, the elements of a `map` are sorted by subscript, and the map's `find` member function will perform a binary search. For containers that have no `find` member function, you could call the `binary_search` algorithm. The `find` algorithm should be used only when the iterators are not forward or the container is not sorted.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/find.C`

```
#include <iostream>
#include <cstdlib>
#include <vector>
#include <algorithm>
#include "date.h"
using namespace std;

int main()
{
    const int a[] = {10, 50, 30, 40, 20};   //need not be sorted for find
    const size_t n = sizeof a / sizeof a[0];

    const int *const p = find(a, a + n, 30);

    if (p == a + n) {
        cout << "Didn't find 30.\n";
    } else {
        cout << "Found 30 at position " << p - a << ".\n";
    }

    const date d[] = {
        date(date::october,   29, 1929),
        date(date::july,       4, 1776),
        date(date::july,      20, 1969),
        date(date::september, 11, 2001),
        date(date::december,   7, 1941)
    };
    const size_t n1 = sizeof d / sizeof d[0];
    vector<date> v(d, d + n1);

    const date moon(date::july, 20, 1969);
    const vector<date>::const_iterator it1 = find(v.begin(), v.end(), moon);

    if (it1 == v.end()) {
        cout << "Didn't find " << moon << ".\n";
    } else {
        cout << "Found " << moon << " at position " << it1 - v.begin() << ".\n";
    }

    const vector<date>::const_iterator it2 =
        find(v.begin(), v.end(), date(date::july, 4, 1776));

    if (it2 == v.end()) {
        cout << "Didn't find it.\n";
    } else {
        cout << "Found it at position " << it2 - v.begin() << ".\n";
    }

    return EXIT_SUCCESS;
}
```

```
Found 30 at position 2.            lines 13−19
Found 7/20/1969 at position 2.     lines 31−38
Found it at position 1.            lines 40−46
```

▼ **Homework 8.4.2a: let cookie::operator new call find**

Let the `operator new` member function of class `cookie` call the `find` algorithm to find the first `false` in the array of `bool`'s. See p. 419.

▲

▼ **Homework 8.4.2b: let class life's `operator-` call find**

On pp. 441−442 we wrote a `operator-` function to measure the distance between two `life` objects. It contains a `for` loop that searches for a `life` in a `vector<life>` by calling `operator==`. Perform the search by calling `find`.

▲

▼ **Homework 8.4.2c: let the `find` member function call the `find` algorithm**

Our rudimentary versions of classes `set` and `map` had a `find` member function on pp. 696−702. Let this member function do its work by calling the `find` algorithm.

▲

▼ **Homework 8.4.2d: an operator< for the template class set**

Implement the `operator<` in pp. 777 and 778 that takes two of the standard library `set` objects, `a` and `b`, and returns true if `a` is a proper subset of `b`. Your `operator<` should be a template function whose two arguments are read-only references to `set<T>`.

▲

**Binders**

For more complicated searching tasks, we will need function objects and combinations thereof. We saw the function object `greater` on pp. 769−770. It inherits three typedef members named `first_argument_type`, `second_argument_type`, and `result_type` from its base class.

```
//Excerpt from <functional>

template <class T>
struct greater: public binary_function<T, T, bool> {
    bool operator()(const T& a, const T& b) const {return a > b;}
};
```

An object of this class, such as the `g` in the following line 18, can do only one thing for us: it takes two arguments and tells us if the first is greater than the second. As is usual when an object has only one significant member function, not counting whatever constructor or destructor it might have, the member function is named `operator()`. It is called in line 20. It takes two `double` arguments and returns `true` if the first argument is greater than the second.

Of course, this kind of object is intended for use only within an algorithm. Line 20 is just a demonstration. If all we want to do is compare `100` and `98.6`, we can simply the line 20 to

```
    if (100.0 > 98.6) {
```

If `98.6` is the most common second argument, it would be convenient if we didn't have to write it all the time. I wish we had an object just like `g`, except that it would be hardwired to use `98.6` as the second argument of its `operator()` member function.

That's what the object named `fever` is in line 26. It has a public member function named `operator()`, called in line 28, which takes one `double` argument and returns `true` if the argument is greater than `98.6`.

The easiest way to understand `fever` is to look at its two data members. We can't see them directly—they're private—but we can see the two arguments passed to its constructor in line 26. `fever` has a copy of `g` and a `98.6` stored permanently inside it. In line 28, the `operator()` member function of `fever` passes its argument `100.0`, and the `98.6` data member of `fever`, to the `operator()` member function of the `g` data member of `fever`. The `operator()` member function of `fever` then returns the return value of the `operator()` member function of `g`.

Line 34 passes the `fever` object to a function `f`. But there is no need to give a name to the `binder2nd` object. An object that is used only once should be an anonymous temporary. Line 35 constructs one and passes it to `f`.

The helper function `bind2nd` in line 36 is an easier way to construct a `binder2nd` object. Its return type is dictated by the data type of its arguments, just like the functions `make_pair`, `inserter`, `front_inserter`, and `back_inserter`. (In fact, the same was true of our very first template function, `min`.) For example, the two arguments in line 36 make it construct and return a `binder2nd<greater<double> >`, which is then passed to `f`.

Line 37 is just like line 36, except it doesn't use the `greater<double>` object `g`. In its place, it constructs an anonymous `greater<double>` object by calling the constructor with no arguments for this class.

The function `f` in lines 6−14 is very forgiving. It will accept *any* predicate to which we can apply a `double` in parentheses and from which we can get a result that is `bool` or convertable thereto (line 9).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/find_if/bind.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <functional>   //for greater, binder2nd, and bind2nd
 4 using namespace std;
 5
 6 template <class PREDICATE>
 7 void f(const PREDICATE& predicate)
 8 {
 9     if (predicate(100.0)) {
10         cout << "100.0 passes the test.\n";
11     } else {
12         cout << "100.0 fails the test.\n";
13     }
14 }
15
16 int main()
17 {
18     greater<double> g;
19
20     if (g(100.0, 98.6)) {                //if (g.operator()(100.0, 98.6)) {
21         cout << "You have a fever.\n";
22     } else {
23         cout << "Your temperature is normal.\n";
24     }
25
26     binder2nd<greater<double> > fever(g, 98.6);
27
28     if (fever(100.0)) {                  //if (fever.operator()(100.0)) {
29         cout << "You have a fever.\n";
30     } else {
31         cout << "Your temperature is normal.\n";
32     }
33
34     f(fever);
35     f(binder2nd<greater<double> >(g, 98.6));
36     f(bind2nd(g, 98.6));
37     f(bind2nd(greater<double>(), 98.6));
38
39     return EXIT_SUCCESS;
40 }
```

```
You have a fever.        lines 18−24
You have a fever.        lines 26−32
100.0 passes the test.   line 34
100.0 passes the test.   line 35
100.0 passes the test.   line 36
100.0 passes the test.   line 37
```

Here are definitions for class `binder2nd` and the function `bind2nd`; line 56 is the *punchline.* The "smaller" function object is of class `F`; its `operator()` takes two arguments. The "larger" function object is of class `binder2nd`; its `operator()` takes one argument.

Now at last we can see one use for the typedef members `first_argument_type`, `second_argument_type`, and `result_type` of class `greater` on pp. 769−770. The `second_argument_type` of the smaller function object becomes the type of the data member `x2` of the larger function object (line 48). We can also see why the `second_argument_type` cannot be a reference. The `initial_x2` in line 51 is a reference to the `second_argument_type`, and there is no such thing as a reference to a reference.

```
41 //Excerpt from <functional>
42
43 template <class F>
44 class binder2nd: public unary_function<typename F::first_argument_type,
45                                        typename F::result_type> {
46 protected:
47     F f;
48     typename F::second_argument_type x2;
49 public:
50     binder2nd(const F& initial_f,
51         const typename F::second_argument_type& initial_x2)
52         : f(initial_f), x2(initial_x2) {}
53
54     typename F::result_type
55     operator()(const typename F::first_argument_type& x1) const {
56         return f(x1, x2);
57     }
58 };
59
60 template <class F, class X2>
61 inline binder2nd<F> bind2nd(const F& f, const X2& x2)
62 {
63     return binder2nd<F>(f, typename F::second_argument_type(x2));
64 }
```

The bigger function object also has two typedefs of its own, `argument_type` and `result_type`, which it inherits from its base class `unary_function`. This would allow the bigger function object be part of an even bigger one.

```
//Excerpt from <functional>

template <class T1, class T2>
struct unary_function {
    typedef T1 argument_type;
    typedef T2 result_type;
};
```

▼ **Homework 8.4.2e: define class binder1st and a function bind1st**

Define a template class `binder1st` similar to `binder2nd`. Like `binder2nd`, it will make a larger function object (with one argument) out of a smaller one (with two arguments). This time, it will be the smaller function object's *first* argument that is hardwired in. Also make a `bind1st` helper function.

The standard library already has a `binder1st` and `bind1st` belonging to namespace `std`, so use a double colon to specify that yours belong to no namespace. When you test them, remember to change `greater` to `less`.

```
    f(bind2nd(greater<double>(), 98.6));
    f(::bind1st(less<double>(), 98.6));   //should do the same thing
```

▲

**Search an array with find_if**

If we know the exact value we're looking for, we call `find`. If we're looking for any value that satisfies a predicate, i.e., that makes an `if` true, we call `find_if`.

Here is a simple definition for `find_if` in the header file `<algorithm>`. The first two arguments must be input iterators, like the first two arguments of `find`. The third must be a predicate that can take one argument of the type (or convertible to the type) read by the input iterators. For example, the input iterators can be pointers to `int`, and the predicate can be a pointer to a function that takes an `int` and returns a `bool`.

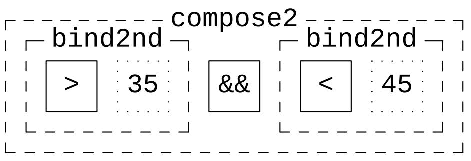
```
template <class INPUT, class PREDICATE>
INPUT find_if(INPUT first, INPUT last, PREDICATE predicate)
{
    for (; first != last; ++first) {
        if (predicate(*first)) {
            break;
        }
    }

    return first;
}
```

We can search a container of `int`'s either by calling `find` with a third argument that is an `int`, or by calling `find_if` with a third argument that is a predicate accepting an `int`.

One example of a predicate would be the function `greater_than_30` in line 8. To search a container for the first number greater than 30, give the address of `greater_than_30` to the `find_if` in line 15.

The function `greater_than_30` has the threshold 30 hardwired in. But we don't have to write a separate function for each threshold. As in the previous program, the `bind2nd` function in line 22 will construct and return an anonymous object that can act as a predicate. Its `operator()` member function will do exactly the same thing as the function `greater_than_30`. It's the same kind of anonymous predicate as the one in the above line 37.

The `compose2` function in lines 30−34 builds a big function object out of three smaller ones. The big object's `operator()` member function takes one `int` argument and returns `true` if the argument lies between 35 and 45. This `operator()` does its work by passing its argument to the `operator()`'s of the "greater than 35" object and the "less than 45" object; the results are passed to the `operator()` of the "and" object. In the diagram, the binary operators are solid, the unary operators are dashed, and the values that are not function objects are dotted. We had to use `compose2` because `logical_and` is a binary operator. It has nothing to do with the fact that `greater` and `less` are binary operators.



`compose2` is not part of the Standard Template Library, so line 5 had to include `<ext/functional>` and line 30 had to mention the namespace `__gnu_cxx` (with a double underscore before the `g`).

Since there are so many ways to compose functions, I think a better name for `compose2` would have been `compose_fg1x_g2x` after the mathematical expression

$$f(g_1(x),\ g_2(x))$$

See the punchline (line 23) of the fragment after the following program; compare with line 56 above and the $f(g_1(x_1),\ g_2(x_2))$ on p. 909. In our case, `g1` and `g2` would be the "greater than 35" and "less than 45" functions, and `f` would be the "logical and" function.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/find_if/find_if1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <algorithm>
 4 #include <functional>       //for greater, less, logical_and, bind2nd
 5 #include <ext/functional>   //for compose2
 6 using namespace std;
 7
 8 inline bool greater_than_30(int i) {return i > 30;}
 9
10 int main()
11 {
12     const int a[] = {10, 30, 20, 40, 50};   //need not be sorted for find_if
13     const size_t n = sizeof a / sizeof a[0];
14
15     const int *p = find_if(a, a + n, greater_than_30);
16     if (p == a + n) {
17         cout << "Found no int greater than 30.\n";
18     } else {
19         cout << "Found " << *p << " at position " << p - a << ".\n";
20     }
21
22     p = find_if(a, a + n, bind2nd(greater<int>(), 30));
23     if (p == a + n) {
24         cout << "Found no int greater than 30.\n";
25     } else {
26         cout << "Found " << *p << " at position " << p - a << ".\n";
27     }
28
29     p = find_if(a, a + n,
30         __gnu_cxx::compose2(
31              logical_and<bool>(),
32              bind2nd(greater<int>(), 35),
33              bind2nd(   less<int>(), 45)
34          )
35      );
36
37      if (p == a + n) {
38          cout << "Found no int in the range 35 to 45 exclusive.\n";
39      } else {
40          cout << "Found " << *p << " at position " << p - a << ".\n";
41      }
42
43      return EXIT_SUCCESS;
44  }
```

```
Found 40 at position 3.       lines 15−20
Found 40 at position 3.       lines 22−27
Found 40 at position 3.       lines 29−41
```

Here is a definition for class `logical_and`; it's analogous to class `greater` (pp. 769−770). We could not have named it `and` because this is a C++ keyword, a synonym for the && operator.

```
 1  //Excerpt from <functional>
 2
 3  template <class T>
 4  struct logical_and: public binary_function<T, T, bool> {
 5      bool operator()(const T& x, const T& y) const {return x && y;}
 6  };
```

Here is a definition for class `binary_compose`. It is called "binary" because the `f` in line 23 takes two arguments. This line is the punchline. I think a better name for this class would have been `composer_fg1x_g2x`.

```
 7  //Excerpt from <ext/functional>
 8  //Compose three functions f, g1, and g2 like this: f(g1(x), g2(x))
 9
10  template <class F, class G1, class G2>
11  class binary_compose: public
12      unary_function<typename G1::argument_type, typename F::result_type> {
13      F f;
14      G1 g1;
15      G2 g2;
16  public:
17      binary_compose(const F& initial_f,
18          const G1& initial_g1, const G2& initial_g2)
19          : f(initial_f), g1(initial_g1), g2(initial_g2) {}
20
21      typename F::result_type
22      operator()(const typename G1::argument_type& x) const {
23          return f(g1(x), g2(x));
24      }
25  };
```

`compose2` is like the functions `make_pair`, `inserter`, and `bind2nd`: it constructs and returns an anonymous `binary_compose` object whose data type depends on the three arguments we gave to `compose2`.

```
26 template <class F, class G1, class G2>
27 inline binary_compose<F, G1, G2>
28 compose2(const F& f, const G1& g1, const G2& g2)
29 {
30     return binary_compose<F, G1, G2>(f, g1, g2);
31 }
```

**An ideal language**

The function body in line 8 of the above `find_f1.C` is simple, but is far from its point of use in line 15. The anonymous object in lines 30−34 is used on the spot*−but the notation is dreadful. To get the best of both worlds, use the newer version of C++ called C++0x. The empty pair square brackets indicates that the anonymous function (a ‘‘lambda function’’) does not use any variables from the surrounding function, in this case `main`.

```
 1     p = find_if(a, a + n, [] (int x) -> int {return x > 35 && x < 45;});
```

**Search a container of objects with find_if**

Before we can compile `fox`, we must make class date ‘‘greater than or equal’’ comparable. We could define an `operator>=` function or an `operator int` function.

But as in the last program, we don’t have to bother writing `fox`. The `bind2nd` function in line 35 will construct and return an anonymous predicate whose `operator()` member function will do the same thing as `fox`. We saw an analogous object in line 22 of the previous program.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/find_if/find_if2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include <algorithm>
 5 #include <functional>       //for greater, less, logical_and, bind2nd
 6 #include <ext/functional>   //for compose2
 7 #include "date.h"
 8 using namespace std;
 9
10 inline bool fox(const date& d) {
11     static const date turn_of_the_century(date::january, 1, 1901);
12     return d >= turn_of_the_century; //return operator>=(d, turn_of_the_century);
13 }
14
15 int main()
16 {
17     date d[] = {
18         date(date::july,       4, 1776),
19         date(date::october,   29, 1929),
20         date(date::july,      20, 1969),
21         date(date::december,   7, 1941),
22         date(date::september, 11, 2001)
23     };
24     const size_t n = sizeof d / sizeof d[0];
25     vector<date> v(d, d + n);
26
27     vector<date>::const_iterator it = find_if(v.begin(), v.end(), fox);
28     if (it == v.end()) {
```

```
29         cout << "Found no date greater than or equal to January 1, 1901.\n";
30     } else {
31         cout << "Found " << *it << " at position " << it - v.begin() << ".\n";
32     }
33
34     it = find_if(v.begin(), v.end(),
35         bind2nd(greater_equal<date>(), date(date::january, 1, 1901)));
36
37     if (it == v.end()) {
38         cout << "Found no date greater than or equal to January 1, 1901.\n";
39     } else {
40         cout << "Found " << *it << " at position " << it - v.begin() << ".\n";
41     }
42
43     it = find_if(v.begin(), v.end(),
44         __gnu_cxx::compose2(
45             logical_and<bool>(),
46             bind2nd(greater_equal<date>(), date(date::january, 1, 1901)),
47             bind2nd(         less<date>(), date(date::january, 1, 2001))
48         )
49     );
50
51     if (it == v.end()) {
52         cout << "Found no twentieth century date.\n";
53     } else {
54         cout << "Found " << *it << " at position " << it - v.begin() << ".\n";
55     }
56
57     return EXIT_SUCCESS;
58 }
```

```
Found 10/29/1929 at position 1.     lines 27−32
Found 10/29/1929 at position 1.     lines 34−41
Found 10/29/1929 at position 1.     lines 43−55
```

▼ **Homework 8.4.2f: should game::get call find_if?**

`game::get` could perform its search by calling `find_if` if we passed an object of the following class `at_location` to `find_if`. To let us mention the `operator()` member of class `at_location` in line 17, we must define class `at_location` *before* class `wabbit`. To let us mention the `x` and `y` members of class `wabbit` in line 21, we must define the function `at_location::operator()` *after* class `wabbit`.

```
 1 //Excerpt from wabbit.h
 2
 3 class wabbit;    //Forward declaration lets line 12 mention wabbit.
 4
 5 class at_location {
 6     const unsigned x;
 7     const unsigned y;
 8 public:
 9     at_location(unsigned initial_x, unsigned initial_y)
10         : x(initial_x), y(initial_y) {}
11
12     bool operator()(const wabbit *p) const;
};

class wabbit {
    //etc.;
    friend bool at_location::operator()(const wabbit *p) const;
};

inline bool at_location::operator()(const wabbit *p) const {
    return x == p->x && y == p->y;
}
```

The first two arguments passed to `find_if` will be of type `game::master_t::const_iterator`, causing the return value of `find_if` to be of the same type. The return value of `game::get`, however, will continue to be a `game::master_t::value_type`.

The Homework does not ask you to make `game::get` call `find_if`. It asks you to decide if it's worth it.

Is there an easier way to call an algorithm to do the work of `wabbit::get`? Let's brainstorm. Suppose we made it possible to compare (with a `==`) a `wabbit *` and a `pair<unsigned, unsigned>`. Then we could pass the `pair` to `find`. Instead of having `x` and `y` data members in a `wabbit`, should class `wabbit` be derived from class `pair<unsigned, unsigned>`?
▲

**Build a function object out of a pointer to a function**

The third argument of `find_if` is usually a predicate that returns a `bool`. But it could also be the `strlen` in line 14: it returns a `size_t`, which is convertible to `bool`. This line searches for a string whose length is not zero.

Line 23 will search for a string whose length is 7. To do this, we will build a bigger function object out of the two smaller ones in lines 25 and 26. Something will go wrong, however, if we try to build the bigger object directly out of `strlen`.

The second argument of the function `compose1` in line 24 must be an object with a public member named `argument_type`. We have already seen that the function `compose2` had similar requirements: its second and third arguments had to have the same member. We saw what the `argument_type` was used for, in line 21 of the definition for class `binary_compose` on p. 866.

But suppose we want to use a plain old pointer to `strlen`, not a function object, as an argument of `compose1` or `compose2`? A pointer has no `argument_type` member; in fact, it has no members at all. Only objects have members. To supply the necessary members, we can wrap the pointer in the following function object.

`X` is the data type of each object in the container. `p` is a pointer to a function whose argument is a `X` and whose return value is a `Y`. Class `pointer_to_unary_function` is derived from class `unary_function`, which gives it the typedefs `argument_type` and `result_type`.

The `x` in line 7 is passed by value to the `operator()` to ensure that the function `*p` cannot change the objects in the container.

```
1 template <class X, class Y>
2 class pointer_to_unary_function: public unary_function<X, Y> {
3 protected:
4     Y (*p)(X);
5 public:
6     explicit pointer_to_unary_function(Y (*initial_p)(X)): p(initial_p) {}
7     Y operator()(X x) const {return (*p)(x);}
8 };
```

The function `ptr_fun` constructs and returns a `pointer_to_unary_function`, just as `make_pair` (pp. 786−787) constructs and returns a `pair`.

```
 9 template <class X, class Y>
10 inline pointer_to_unary_function<X, Y> ptr_fun(Y (*p)(X)) {
11     return pointer_to_unary_function<X, Y>(p);
12 }
```

Because of the `explicit` in the above line 6, the above line 11 cannot be changed to

```
13     return p;
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/find_if/ptr_fun.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cstring>         //for strlen
 4 #include <algorithm>       //for find_if
 5 #include <functional>      //for bind2nd, ptr_fun
 6 #include <ext/functional>  //for compose1
 7 using namespace std;
 8
 9 int main()
10 {
11     const char *const a[] = {"", "hello", "goodbye"};
12     const size_t n = sizeof a / sizeof a[0];
13
14     const char *const *p = find_if(a, a + n, strlen);
15
16     if (p == a + n) {
17         cout << "Every string was of length 0.\n";
18     } else {
19         cout << "The first non-empty string was a["
20             << p - a << "] == \"" << *p << "\".\n";
21     }
22
23     p = find_if(a, a + n,
24         __gnu_cxx::compose1(
25             bind2nd(equal_to<size_t>(), 7),
26             ptr_fun(strlen)
27         )
28     );
29
30     if (p == a + n) {
31         cout << "No string was of length 7.\n";
32     } else {
33         cout << "The first string of length 7 was a["
34             << p - a << "] == \"" << *p << "\".\n";
35     }
36
37     return EXIT_SUCCESS;
38 }
```

In the above line 14, the `size_t` return value of `strlen` is implicitly converted to `bool` by the `if` inside the `find_if` algorithm. If your compiler complains about this, use the code in lines 23−28 with the `7` changed to `0`.

```
The first non-empty string was a[1] == "hello".             lines 14−21
The first string of length 7 was a[2] == "goodbye".         lines 23−35
```

### ▼ Homework 8.4.2g: define unary_compose and compose1

The helper function `compose1` constructs and returns an object of class `unary_compose`, just as the helper function `compose2` constructs and returns an object of class `binary_compose`.

Define class `unary_compose` and the function `compose1`. Class `unary_compose` will be derived from the template class `unary_function`, just like class `binary_compose`. Class `unary_compose` will have two data members `f` and `g`. Hint: the punchline (i.e., the body of the `operator()` member function of class `unary_compose`) will be

```
    return f(g(x));
```

The class is called “unary” because the `f` takes one argument. Better names for `unary_compose` and `compose1` might have been `composer_fgx` and `compose_fgx`, after the mathematical expression

$$f(g(x))$$

▲

### Call a member function of each object in a container of objects

In the above container of `char *`’s, each element was passed to the `strlen` function. But it would be more realistic to have a container of `string` objects, where each element has its `size` member function called.

Our original `find_if` passed each element to the predicate (line 5).

```
 1 template <class INPUT, class PREDICATE>
 2 INPUT find_if(INPUT first, INPUT last, PREDICATE predicate)
 3 {
 4     for (; first != last; ++first) {
 5         if (predicate(*first)) {
 6             break;
 7         }
 8     }
 9
10     return first;
11 }
```

It looks like we will need another version of `find_if` whose third argument is a predicate that is a pointer to a member function (line 19). Each element will have the pointed-to member function called.

```
12 //PREDICATE must be a pointer to a member function of
13 //typename iterator_traits<INPUT>::value_type.
14
15 template <class INPUT, class PREDICATE>
16 INPUT find_if(INPUT first, INPUT last, PREDICATE predicate)
17 {
18     for (; first != last; ++first) {
19         if ((*first.*predicate)()) {
20             break;
21         }
22     }
23
24     return first;
25 }
```

But a clever function object lets us do the job with the original definition of `find_if`. `T` is the data type of each object in the container. `p` is a pointer to a `const` member function of class `T`. `Y` is the data type of the return value of the member function to which `p` points. It's called a "`ref_`" because the `t` is passed by reference in line 10. This line is the punchline.

```
 1 //Excerpt from <functional>.
 2
 3 template <class Y, class T>
 4 class const_mem_fun_ref_t: public unary_function<T, Y> {
 5     Y (T::*p)() const;
 6 public:
 7     explicit const_mem_fun_ref_t(Y (T::*initial_p)() const)
 8         : p(initial_p) {}
 9
10     Y operator()(const T& t) const {return (t.*p)();}
11 };
```

The helper function `mem_fun_ref` constructs and returns a `const_mem_fun_ref_t`, just as the function `ptr_fun` constructs and returns a `pointer_to_unary_function`.

```
12 template <class Y, class T>
13 inline const_mem_fun_ref_t<Y, T> mem_fun_ref(Y (T::*p)() const) {
14     return const_mem_fun_ref_t<Y, T>(p);
15 }
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/find_if/mem_fun_ref.C`

```
#include <iostream>
#include <cstdlib>
#include <string>           //for class string
#include <algorithm>        //for find_if
#include <functional>       //for bind2nd, mem_fun_ref
#include <ext/functional>   //for compose1
using namespace std;

int main()
{
    const string a[] = {"", "hello", "goodbye"};
    const size_t n = sizeof a / sizeof a[0];

    const string *p = find_if(a, a + n, mem_fun_ref(&string::size));

    if (p == a + n) {
        cout << "Every string was of length 0.\n";
    } else {
        cout << "The first non-empty string was a["
            << p - a << "] == \"" << *p << "\".\n";
    }

    p = find_if(a, a + n,
        __gnu_cxx::compose1(
            bind2nd(equal_to<string::size_type>(), 7),
            mem_fun_ref(&string::size)
        )
    );
29
30     if (p == a + n) {
31         cout << "No string was of length 7.\n";
32     } else {
33         cout << "The first string of length 7 was a["
34             << p - a << "] == \"" << *p << "\".\n";
35     }
36
37     return EXIT_SUCCESS;
38 }
```

```
The first non-empty string was a[1] == "hello".             lines 14−21
The first string of length 7 was a[2] == "goodbye".         lines 23−35
```

**▼ Homework 8.4.2h: define class mem_fun_ref_t**

Define a class `mem_fun_ref_t`. It will be the same as class `const_mem_fun_ref_t`, except that it will hold a pointer to a non-`const` member function. Define another `mem_fun_ref` function to construct and return an object of class `mem_fun_ref_t`.

▲

**Call a member function of each object in a container of pointers to objects**

Instead of the above container of objects, it would be even more realistic to have a container of pointers to objects. It looks like we will need another version of `find_if`, this time with the `->*` operator in line 5.

```
 1 template <class INPUT, class PREDICATE>
 2 INPUT find_if(INPUT first, INPUT last, PREDICATE predicate)
 3 {
 4     for (; first != end; ++first) {
 5         if ((*first->*predicate)()) {
 6             break;
 7         }
 8     }
 9
10     return first;
11 }
```

But another clever function object lets the original definition of `find_if` do the job. As before, `T` is the data type of each object in the container. `p` is a pointer to a `const` member function of class `T`. `Y` is the data type of the return value of the member function to which `p` points. Line 10 is the punchline.

```
 1 //Excerpt from <functional>
 2
 3 template <class Y, class T>
 4 class const_mem_fun_t: public unary_function<const T *, Y> {
 5     Y (T::*p)() const;
 6 public:
 7     explicit const_mem_fun_t(Y (Y::*initial_p)() const)
 8         : p(initial_p) {}
 9
10     Y operator()(const T *pt) const {return (pt->*p)();}
11 };
```

The function `mem_fun` constructs and returns a `const_mem_fun_t`, just as the function `mem_fun_ref` constructs and returns a `const_mem_fun_ref_t`.

```
template <class Y, class T>
inline const_mem_fun_t<Y, T> mem_fun(Y (T::*p)() const) {
    return const_mem_fun_t<Y, T>(p);
}
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/find_if/mem_fun.C`

```
#include <iostream>
#include <cstdlib>
#include <string>          //for class string
#include <algorithm>       //for find_if
#include <functional>      //for bind2nd, mem_fun
#include <ext/functional>  //for compose1
using namespace std;

int main()
{
    const string *a[] = {
        new string(""),
        new string("hello"),
        new string("goodbye")
    };
    const size_t n = sizeof a / sizeof a[0];

    const string *const *p = find_if(a, a + n, mem_fun(&string::size));

    if (p == a + n) {
        cout << "Every string was of length 0.\n";
    } else {
        cout << "The first non-empty string was a["
            << p - a << "] == \"" << **p << "\".\n";
    }

    p = find_if(a, a + n,
        __gnu_cxx::compose1(
            bind2nd(equal_to<string::size_type>(), 7),
            mem_fun(&string::size)
        )
    );

    if (p == a + n) {
        cout << "No string was of length 7.\n";
    } else {
        cout << "The first string of length 7 was a["
            << p - a << "] == \"" << **p << "\".\n";
    }

    for (const string *const *p = a + n - 1; p >= a; --p) {
        delete *p;
    }
    return EXIT_SUCCESS;
}
```

```
The first non-empty string was a[1] == "hello".              lines 18−25
The first string of length 7 was a[2] == "goodbye".          lines 27−39
```

**▼ Homework 8.4.2i: define class mem_fun_t**

Define a class `mem_fun_t`. It will be the same as class `const_mem_fun_t`, except that it will hold a pointer to a non-`const` member function. Define another `mem_fun` function to construct and return an object of class `mem_fun_t`.

▲

**Find all of them, not just the first one**

How do we find *every* element of a container that satisfies a predicate, not just the first one? We have to call `remove_copy_if`, which copies all the items for which the predicate is *false*. It's similar to the `remove_copy` we saw on p. 858.

The contents of the source container (the array `a` in line 11) remain unchanged. The destination `v` is an empty container that will be expanded (line 14), so the third argument of `remove_copy_if` must be an inserter (line 15). A non-inserter iterator such as `v.begin()` would overwrite memory beyond the end of the container, causing the program to blow up (if we are lucky). Instead of inserting the output into a vector, we can write it directly to the standard output (line 21).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/remove_copy_if/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include <iterator>     //for back_inserter and ostream_iterator
 5 #include <functional>   //for not1
 6 #include <algorithm>    //for remove_copy_if
 7 using namespace std;
 8
 9 int main()
10 {
11     const int a[] = {10, 30, 40, 20, 50};   //need not be sorted for remove_copy_if
12     const size_t n = sizeof a / sizeof a[0];
13
14     vector<int> v;
15     remove_copy_if(a, a + n, back_inserter(v), bind2nd(greater<int>(), 30));
16     copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
17     cout << "\n";
18
19     remove_copy_if(
20         a, a + n,
21         ostream_iterator<int>(cout, "\n"),
22         bind2nd(greater<int>(), 30)
23     );
24
25     return EXIT_SUCCESS;
26 }
```

```
10   lines 14−17
30
20

10   lines 19−21
30
20
```

It’s annoying to have to write the opposite of the desired predicate in the above line 22. To search for the first value less than or equal to 30, I wish we could write the more natural predicate `bind2nd(less_equal<int>(), 30)`. We can do this with the one-argument function `not1`. It takes a predicate and constructs and returns one that yield the opposite result. The following line12 is the punchine.

```
 1 //Excerpts from <functional>.
 2
 3 template <class F>
 4 class unary_negate: public unary_function<typename F::argument_type, bool>
 5 {
 6 protected:
 7     F f;
 8 public:
 9     explicit unary_negate(const F& initial_f): f(initial_f) {}
10
11     bool operator()(const typename F::argument_type& x) const {
12         return !f(x);
13     }
14 };

15 template <class F>
16 inline unary_negate<F> not1(const F& f) {
17     return unary_negate<F>(f);
18 }
```

We can now change the above lines 19−23 to the following.

```
19     remove_copy_if(
20         a, a + n,
21         ostream_iterator<int>(cout, "\n"),
22         not1(bind2nd(less_equal<int>(), 30))
23     );
```

Even better, write your own `copy_if` algorithm and forget about `remove_copy_if` and the `not1`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/remove_copy_if/copy_if.h`

```
#ifndef COPY_IFH
#define COPY_IFH

template <class INPUT, class OUTPUT, class PREDICATE>
OUTPUT copy_if(INPUT first, INPUT last, OUTPUT result, PREDICATE predicate)
{
    for (; first != last; ++first) {
        if (predicate(*first)) {
            *result = *first;
10             ++result;
11         }
12     }
13 
14     return result;
15 }
16 #endif
```

```
17 #include "copy_if.h"
18 
19     copy_if(
20         a, a + n,
21         ostream_iterator<int>(cout, "\n"),
22         bind2nd(less_equal<int>(), 30)
23     );
```

Let's print the printable characters in a `string` `s`. We must convert each character to `unsigned char` before passing it to `isprint`. To see what would go wrong otherwise, see line 15 of `static_cast.C` on p. 65.

Recall that a built-in type has a one-argument constructor; our first example was in line 37 of `duo.C` on p. 136. With the template arguments in the following line 29, the `DEST(source)` in line 9 will call the constructor for type `unsigned char`. We must say `DEST(source)` instead of an unadorned `source` because the constructor for the `DEST` type might be `explicit`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/remove_copy_if/convert.h`

```
 1 #ifndef CONVERTH
 2 #define CONVERTH
 3 #include <functional>  //for unary_function
 4 using namespace std;
 5 
 6 template <class SOURCE, class DEST>
 7 class convert: public unary_function<SOURCE, DEST>  {
 8 public:
 9     DEST operator()(const SOURCE& source) const {return DEST(source);}
10 };
11 
12 #endif
```

```
13 #include <iostream>
14 #include <cctype>         //for isprint
15 #include <string>
16 #include <iterator>       //for ostream_iterator
17 #include <functional>     //for ptr_fun
18 #include <ext/functional> //for compose1
19 #include "convert.h"
20 #include "copy_if.h"
21 using namespace std;
22 
23     copy_if(
24         s.begin(),
25         s.end(),
26         ostream_iterator<char>(cout),
27         __gnu_cxx::compose1(
28             ptr_fun(static_cast<int (*)(int)>(isprint)),
29              convert<char, unsigned char>()
30          )
31      );
```

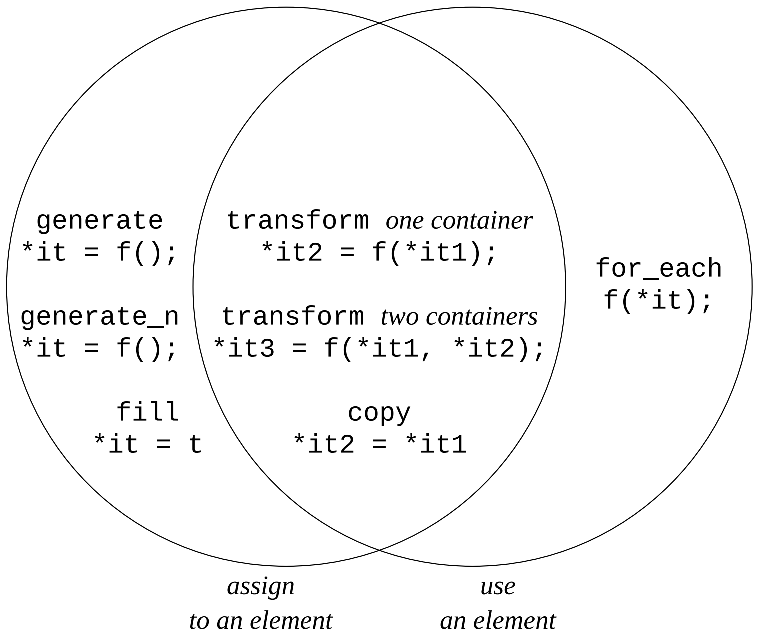
`isprint` returns an `int`, so the `compose1` in the above lines 22−25 constructs an anonymous object whose `operator()` returns an `int`. If line 8 of `copy_if.h` on page p. 876 complains about converting this `int` to a `bool`, you can define a `compose_fghx` and change the above lines 22−25 to

```
32      compose_fghx(
33          convert<int, bool>(),
34          static_cast<int (*)(int)>(isprint),
35          convert<char, unsigned char>()
36      );
```

## 8.4.3 Algorithms that call Functions, and Additional Function Objects

The five algorithms in this section call a user-supplied function `f` during each iteration of a loop. This keeps the body of the loop separate from the control structure that governs it. The body is written as the function `f`; the control structure is written in the algorithm. Any body can be plugged into any control structure.

The name of the algorithm acts as documentation. Instead of writing the keyword “`for`” at the head of every loop, we can write `for_each` for a loop that reads, `generate` for a loop that writes, and `transform` for a loop that does both. The following diagram shows what happens during each iteration.



**for_each**

The `for_each` algorithm passes each value in a container to a function. As usual, the `f` in line 2 could be a pointer to a function of one argument, or it could be a function object whose `operator()` takes one argument.

```
1 template <class INPUT, class FUNCTION>
2 FUNCTION for_each(INPUT first, INPUT last, FUNCTION f)
3 {
4     for (; first != last; ++first) {
5         f(*first);   //ignore the return value of f, if any
6     }
7
8     return f;
9 }
```

Let's print the elements in a container, numbering each one. The container is in lines 18−24; its elements are `string` objects.

Line 26 constructs an anonymous function object of the class `line_numberer<string, int>`, inserting the value `6` into it. We pass the container and the function object to `for_each`, which passes each element of the container to the `operator()` member function of the object. When `for_each` is finished, it passes the function object back to us in the above line 8. Our function object has the member function `operator int` in the following line 12. Line 18 calls this function to convert the object into an `int`.

The default initial value of the data member `i` should be `0` if `COUNTER` is `int`, `0L` if `COUNTER` is `long`, `date()` if `COUNTER` is `date`, etc. But there is no need to write a specialization for each data type. Line 10 simply calls the default constructor for data type `COUNTER`. See p. 660.

To print the elements of only one container, it would have been easier to write a traditional `for` loop. For many containers, each with elements of different types, it's easier to pass a function object to `for_each`.

Do not call `for_each` if the library has a more specific algorithm. To copy, find, count, or accumulate, it is easier to call `copy`, `find`, `count`, or `accumulate`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/line_numberer.h`

```
 1 #ifndef LINE_NUMBERERH
 2 #define LINE_NUMBERERH
 3 #include <iostream>
 4 using namespace std;
 5
 6 template <class DATA, class COUNTER = int>
 7 class line_numberer {
 8     COUNTER i;
 9 public:
10     line_numberer(COUNTER initial_i = COUNTER()): i(initial_i) {}
11     void operator()(const DATA& data) {cout << i++ << " " << data << "\n";}
12     operator COUNTER() const {return i;}
13 };
14 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/for_each.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <string>
 4 #include <algorithm>
 5 #include "line_numberer.h"
 6 using namespace std;
 7
 8 int main()
 9 {
10     const string a[] = {                      //Macbeth IV i
11         "Toad, that under cold stone",
12         "Days and nights has thirty one",
13         "Swelt’red venom, sleeping got,",
14         "Boil thou first i’ th’ charmed pot."
15     };
16     const size_t n = sizeof a / sizeof a[0];
17
18     const int i = for_each(a, a + n, line_numberer<string>(6));
19     cout << "\nThe next line number will be " << i << ".\n\n";
20
21     const char c = for_each(a, a + n, line_numberer<string, char>('A'));
22     cout << "\nThe next line number will be " << c << ".\n\n";
23     return EXIT_SUCCESS;
24 }
```

```
6 Toad, that under cold stone
7 Days and nights has thirty one
8 Swelt’red venom, sleeping got,
9 Boil thou first i’ th’ charmed pot.

The next line number will be 10.

A Toad, that under cold stone
B Days and nights has thirty one
C Swelt’red venom, sleeping got,
D Boil thou first i’ th’ charmed pot.

The next line number will be E.
```

`for_each` can do more than read the elements. The call in line 11 of the following `hal.h` can modify them. The `string::value_type` is just a hypercorrect way of saying `char`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/decrement.h`

```
 1 #ifndef DECREMENTH
 2 #define DECREMENTH
 3 #include <functional>  //for unary_function
 4 using namespace std;
 5
 6 template <class T>
 7 class decrement: public unary_function<T, void> {
 8 public:
 9     void operator()(T& t) const {--t;}   //read/write reference
10 };
11 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/hal.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <string>
 4 #include <algorithm>
 5 #include "decrement.h"
 6 using namespace std;
 7
 8 int main()   //HAL-9000 computer in "2001: a Space Odyssey"
 9 {
10     string s = "IBM";
11     for_each(s.begin(), s.end(), decrement<string::value_type>());
12     cout << s << "\n";
13     return EXIT_SUCCESS;
14 }
```

```
HAL
```

Examples of `for_each` modifying the elements of a container with an STL function object are in lines 28−29 of `mem_fun.C` on p. 942. Those elements will be objects and pointers thereto.

**generate**

The `generate` algorithm writes into a container, so its iterators must be output iterators. But it also compares the iterators, so they must also be input iterators. Together, they must be forward iterators.

```
15 template <class FORWARD, class FUNCTION>
16 void generate(FORWARD first, FORWARD last, FUNCTION f)
17 {
18     for (; first != last; ++first) {
19         *first = f();
20     }
21 }
```

If your iterators are merely output, not forward, call `generate_n`. As with the `fill_n` algorithm, `N` can be any data type that can be decremented and compared with `>`.

```
 1 template <class OUTPUT, class N, class FUNCTION>
 2 void generate_n(OUTPUT first, N n, FUNCTION f)
 3 {
 4     for ( ; n > 0; --n, ++first)
 5         *first = f();
 6     }
 7 }
```

To demonstrate generation, let's overwrite a range with random integers. The third argument of the `generate` in line 21 is a plain old pointer to a function, not a function object.

```
 8 #include <cstdlib>   //for rand
 9 #include <vector>
10 #include <algorithm>
11 using namespace std;
12
13     vector<int> v(argument(s) for constructor);
14     generate(v.begin(), v.end(), rand);
```

**▼ Homework 8.4.3a: class pointer_to_generator**

To overwrite a container with the negatives of random integers, I wish we could say the following.

```
1 #include <cstdlib>        //for rand
2 #include <vector>
3 #include <functional>     //for negate
4 #include <ext/functional> //for compose1
```

```
 5 #include <algorithm>
 6 using namespace std;
 7
 8     vector<int> v(argument(s) for constructor);
 9
10     generate(
11         v.begin(),
12         v.end(),
13         __gnu_cxx::compose1(
14             negate<int>(),
15             ptr_fun(rand)
16         )
17     );
```

The two `ptr_fun`'s in the standard library take a pointer to a function of one or two arguments. But `rand` is a function with no arguments. A function taking no arguments, or a function object whose `operator()` takes no arguments, is called a *generator.* Examples are the standard library function `rand` in the above line 15 and an object of the following class `consecutive`.

To get the above code to compile, we will have to define the following classes and functions.

(1) Define a template class named `generator`. It will be just like the template classes `unary_function` and `binary_function` on pp. 863−864 and 769−770, except that it will have only one template argument, and its only member will be `result_type`.

(2) Derive a template class named `pointer_to_generator`. It will be just like class `pointer_to_unary_function` on pp. 869−870, except that it will be derived from your class `generator`, contain a pointer to a generator, and have an `operator()` that takes no arguments. Class `pointer_to_generator` will need only one template argument.

(3) Define another helper function named `ptr_fun`. It will be just like the one on pp. 869−870, except that its argument will be a pointer to a function that is a generator, and its return value will be an object of class `generator`. This `ptr_fun` will need only one template argument.

(4) Define a template class named `composer_fg`. It will be just like class `unary_compose` on p. 871, except that it will be derived from class `generator` and `g` will be a generator. Class `composer_fg` will need only one template argument.

(5) Define a helper function named `compose_fg`. It will be just like the helper function `compose1` on p. 871, except that its second argument will be a generator. The `compose_fg` function will need only one template argument.

In place of the above lines 10−17, we can now say the following,

```
18     generate(
19         v.begin(),
20         v.end(),
21         compose_fg(
22             negate<int>(),
23             ptr_fun(rand)
24         )
25     );
```

▲

## An iterator instead of a generator

Whenever we need to generate a series of consecutive values, we write a `for` loop with a counter and an increment. The counter and the increment can be written once and for all in the following template class `consecutive`. A stride or increment can easily be added.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/consecutive/consecutive.h`

```
#ifndef CONSECUTIVEH
#define CONSECUTIVEH

//T must be copy constructable and incrementable.

template <class T = int>
class consecutive {
    T t;
public:
    consecutive(const T& initial_t = T()): t(initial_t) {}
    T operator()() {return t++;}
};
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/consecutive/main.C`

```
#include <iostream>
#include <cstdlib>
#include <iterator>
#include <algorithm>
#include "date.h"
#include "consecutive.h"
using namespace std;

int main()
{
    const size_t n = 10;
    int a[n];

    generate(a, a + n, consecutive<int>());
    copy(a, a + n, ostream_iterator<int>(cout, " "));
    cout << "\n";

    generate_n(ostream_iterator<char>(cout), 26, consecutive<char>('A'));
    cout << "\n";

    generate_n(ostream_iterator<date>(cout, "\n"), 3, consecutive<date>());

    //Output the address of each array element.
    generate_n(ostream_iterator<int *>(cout, "\n"), 3,
        consecutive<int *>(a));

    return EXIT_SUCCESS;
}
```

```
0 1 2 3 4 5 6 7 8 9         line 14: generate with a pair of forward iterators
ABCDEFGHIJKLMNOPQRSTUVWXYZ  line 18: generate_n with one output iterator
4/8/2014                    line 21
4/9/2014
4/10/2014
0xffbff140                  line 24: sizeof (int) == 4 on my machine
0xffbff144
0xffbff148
```

But when we do more than just generate the range of values, this approach wastes space. For example, let's pick out the prime numbers in a range of integers. A *prime number* is a positive integer that is greater than 1 and whose only factors are itself and 1.

Generating the entire range in lines 15−16 is senselessly profligate, since the prime numbers are so few and far between. Lines 33−34 commit the same sin. Finally, we have to hope that the `int n` in line 33 can fit into a `vector<int>::size_type`.

The function object

```
modulus<int>()
```

in line 39 takes two arguments, `dividend` and `divisor`, and returns `dividend % divisor`. The bigger function object

```
bind1st(modulus<int>(), n)
```

takes only one argument, and returns zero if the argument is a divisor of `n`. Since zero and non-zero can be implicitly converted to `false` and `true`, this function object can be used as a predicate. The even bigger predicate

```
not1(bind1st(modulus<int>(), n))
```

does just the opposite. It takes an argument and returns `true` if the argument is a divisor of `n`. Since we hope that `n` will be a prime, we hope that `find_if` will *not* find what it is looking for.

The code can be made even simpler by making a typedef for `t_iterator<int>`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/consecutive/prime.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <vector>
 4 #include <iterator>
 5 #include <functional>      //not1, ptr_fun, equal_to, modulus, bind1st, bind2nd
 6 #include <ext/functional>  //compose1
 7 #include <algorithm>
 8 #include "consecutive.h"
 9 using namespace std;
10
11 bool isprime(int n);
12
13 int main()
14 {
15     const size_t n = 30;
16     int a[30];
17     generate_n(a, n, consecutive<int>(1));
18
19     remove_copy_if(
20         a,
21         a + n,
        ostream_iterator<int>(cout, "\n"),
        not1(ptr_fun(isprime))
    );
    return EXIT_SUCCESS;
}

bool isprime(int n)
{
    if (n < 2) {
        return false;        //The smallest prime is 2.
    }

    vector<int> v(n - 2);    //a vector of n - 2 zeros
    generate(v.begin(), v.end(), consecutive<int>(2));

    return find_if(
        v.begin(),
        v.end(),
        __gnu_cxx::compose1(
            bind2nd(equal_to<int>(), 0),
            bind1st(modulus<int>(), n)
        )
    ) == v.end();
}
```

```
2
3
5
7
11
13
17
19
23
29
```

We can avoid the waste of space by writing the counting variable and the increment in an iterator instead of a generator. Pleasantly, the increment can now be prefix (line 16) instead of postfix (line 11 of the above `consecutive.h`). Class `t_iterator` has more lines of source code than `consecutive`, but it is totally stereotyped.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/t_iterator/t_iterator.h`

```
 1 #ifndef T_ITERATORH
 2 #define T_ITERATORH
 3 #include <iterator>
 4 using namespace std;
 5
 6 //T must be copy constructable (line 13), incrementable (line 16),
 7 //and equality comparable (line 25).
 8
 9 template <class T = int>
10 class t_iterator: public iterator<forward_iterator_tag, T> {
11     T t;
public:
    t_iterator(const T& initial_t = T()): t(initial_t) {}
    const T& operator*() const {return t;}

    t_iterator& operator++() {++t; return *this;}

    const t_iterator operator++(int) {
        const t_iterator old = *this;
        ++*this;
        return old;
    }

    friend bool operator==(const t_iterator& it1, const t_iterator& it2) {
        return it1.t == it2.t;
    }
};

template <class T>
inline bool operator!=(const t_iterator<T>& it1, const t_iterator<T>& it2) {
    return !(it1 == it2);
}
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/t_iterator/prime.C`

```
#include <iostream>
#include <cstdlib>
#include <iterator>
#include <functional>
#include <ext/functional>
#include <algorithm>
#include "t_iterator.h"
using namespace std;

inline bool isprime(int n)
{
    return n >= 2 && find_if(
        t_iterator<int>(2),
        t_iterator<int>(n),
        __gnu_cxx::compose1(
            bind2nd(equal_to<int>(), 0),
            bind1st(modulus<int>(), n)
        )
    ) == t_iterator<int>(n);
}

int main()
{
    remove_copy_if(
        t_iterator<int>(1),
        t_iterator<int>(30),
        ostream_iterator<int>(cout, "\n"),
        not1(ptr_fun(isprime))
    );
    return EXIT_SUCCESS;
}
```

```
2
3
5
7
11
13
17
19
23
29
```

**▼ Homework 8.4.3b: use t_iterator**

(1) Define a function to return the factorial of an integer.

```
int factorial(int n);
```

If `n` is less than or equal to 1, the function will return 1. (Why will we go into an almost infinite loop without this test?) Otherwise, the function will call the `accumulate` algorithm (include the header file `<numeric>`) and pass it a pair of `t_iterator<int>`’s representing the numbers from 2 to `n` inclusive. Also pass it an anonymous function object of class `multiplies<int>`.

(2) A *perfect number* is a positive integer that is the sum of all of its positive divisors that are smaller than it. Perfect numbers are extremely rare. The first three examples are

$$6 = 1 + 2 + 3$$

$$28 = 1 + 2 + 4 + 7 + 14$$

$$496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248$$

Define a function

```
bool isperfect(int n);
```

that returns `true` if its argument `n` is a perfect number. If `n` is less than 1, the function will return `false`. (Why will we go into an almost infinite loop without this test?) Otherwise, the function will create a vector of all the positive divisors of `n` that are less than `n`. Begin with the vector empty. Call the `remove_copy_if` algorithm and pass it a pair of `t_iterator<int>`’s representing the numbers from 1 to `n`−1 inclusive. Also pass it a back inserter to fill up the vector, and a function object for picking out the divisors of `n`. Sum up all the elements in the vector by calling `accumulate`. If `n` is perfect, it will be equal to this sum.

Find all the perfect numbers in the range 1 to 10,000 inclusive. Is there an odd perfect number?

Some numbers have a lot of factors. Is there a way to test if a number is perfect without storing all the factors simultaneously in a container? Could you make an iterator that loops through the factors of `n`?

▲

**▼ Homework 8.4.3c: make t_iterator random access**

Upgrade `t_iterator` to be a random access iterator, at least for the data types `T` to which the `+` and `<` operators can be applied.

Give class `t_iterator` an extra template argument `DIFFERENCE`, which will be used for the arguments and return value of the following member functions and friend. Like the `DIFFERENCE` template argument of class `iterator`, let it default to `ptrdiff_t`.

```
template <class T, class DIFFERENCE = ptrdiff_t>
```

```
class t_iterator: public iterator<random_access_iterator_tag, T, DIFFERENCE>
{
    //etc.
    const T& operator[](DIFFERENCE d) const {return t + d;}
    t_iterator& operator+=(DIFFERENCE d) {t += d; return *this;}

    friend DIFFERENCE operator-(t_iterator& it1, t_iterator& it2) {
        return it1.t - it2.t;
    }
```

You can then implement `operator++` by calling `operator+=`, and `operator*` by calling `operator[]`. Don't forget to define `operator<`, etc.
▲

▼ **Homework 8.4.3d: redesign t_iterator**

To loop through *every* value of a data type, the test has to go at the bottom of the loop. We don't want to increment a `char` that already contains the maximum value. On a platform where `char` is signed, that would result in undefined behavior.

```
#include <iostream>
#include <limits>
using namespace std;

    //Output every char.

    for (char c = numeric_limits<char>::min();; ++c) {
        cout << c;
        if (c == numeric_limits<char>::max()) {
            break;
        }
    }
```

Unfortunately, the standard algorithms have their test at the top of the loop.

```
#include <iostream>
#include <limits>
#include <iterator>
#include <algorithm>
using namespace std;

    //Doesn't output every char.
    //It fails to output the last one, numeric_limits<char>::max().

    copy(
        t_iterator<char>(numeric_limits<char>::min()),
        t_iterator<char>(numeric_limits<char>::max()),
        ostream_iterator<char>(cout)
    );
```

An approach to correcting this is suggested by class `istream_iterator`. Its default (no-argument) constructor gave us an iterator representing the end of a range.

```
#include <iostream>
#include <iterator>
#include <algorithm>
using namespace std;

```

```
32     //Copy the standard input to the standard output.
33
34     copy(
35         istream_iterator<char>(cin),
36         istream_iterator<char>(),   //end-of-input
37         ostream_iterator<char>(cout)
38     );
```

Let's create a similar `t_iterator` representing the end of a range of values. Decide which of the following designs is better.

(1) Line 49 specifies the starting value, but line 50 does not have to mention the ending value. The loop will stop automatically when it reaches `numeric_limits<T>::max()`.

```
39 #include <iostream>
40 #include <limits>
41 #include <iterator>
42 #include <algorithm>
43 #include "t_iterator2.h"
44 using namespace std;
45
46     //Output every char.
47
48     copy(
49         t_iterator<char>(numeric_limits<char>::min()),
50         t_iterator<char>(),
51         ostream_iterator<char>(cout)
52     );
```

Give the iterator the extra data member in line 10. It can be born `true` in line 13, or become `true` in line 32. The `is_specialized` static member in line 31 is `true` if there is a specialization of class `numeric_limits` for the data type `T`. If not, the `max` function returns no meaningful result.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/t_iterator/t_iterator2.h`

```
 1 #ifndef T_ITERATOTH
 2 #define T_ITERATOTH
 3 #include <cstdlib>   //for exit
 4 #include <iterator>
 5 #include <limits>
 6 using namespace std;
 7
 8 template <class T = int, class DIFFERENCE = ptrdiff_t>
 9 class t_iterator: public iterator<randoom_access_iterator_tag, T> {
10     bool at_end;   //true if we have reached end of range
11     T t;
12 public:
13     t_iterator(): at_end(true) {}
14     t_iterator(const T& initial_t): at_end(false), t(initial_t) {}
15
16     const T& operator*() const {
17         if (at_end) {
18             cerr << "dereference exhausted t_iterator\n";
19             exit(EXIT_FAILURE);
20         }
21         return t;
22     }
23
24     t_iterator& operator++() {
25         if (at_end) {
26             cerr << "increment exhausted t_iterator\n";
27             exit(EXIT_FAILURE);
28         }
29
30         typedef numeric_limits<T> limits;   //for convenience
31         if (limits::is_specialized && t == limits::max()) {
32             at_end = true;
33         } else {
34             ++t;
35         }
36         return *this;
37     }
38
39     friend bool operator==(const t_iterator<T, DIFFERENCE>& it1,
40                            const t_iterator<T, DIFFERENCE>& it2) {
41         return it1.at_end == it2.at_end &&
42             (it1.at_end || it1.t == it2.t);
43     }
44
45     //etc.
46 };
47
48 //etc.
49 #endif
```

To use the above `t_iterator` to loop through every possible `date`, we would have to define a specialization of class `numeric_limits` for class `date`.

```
50 #include <limits>
51 #include "date.h"
52 using namespace std;
53
54 namespace std {
55 template <>
56 class numeric_limits<date> {
57 public:
58     static const bool is_specialized = true;
59
60     static date min() throw () {
61         static date d(date::january, 1, numeric_limits<int>::min());
62         return d;
63     }
64
65     static date max() throw () {
66         static date d(date::december, 31, numeric_limits<int>::max());
67         return d;
68     }
69 };
70 }
```

(2) The following `t_iterator` has the two-argument constructor in line 83.

```
71 #include <iostream>
```

```
#include <limits>
#include <iterator>
#include <algorithm>
#include "t_iterator3.h"
using namespace std;

    //Output every char.

    typedef numeric_limits<char> limits;

    copy(
        t_iterator<char>(limits::min(), limits::max()),
        t_iterator<char>(),
        ostream_iterator<char>(cout)
    );
```

Line 11 has yet another data member.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/t_iterator/t_iterator3.h`

```
 1 #ifndef T_ITERATOTH
 2 #define T_ITERATOTH
 3 #include <cstdlib>   //for exit
 4 #include <iterator>
 5 using namespace std;
 6
 7 template <class T = int, class DIFFERENCE = ptrdiff_t>
 8 class t_iterator: public iterator<random_access_iterator_tag, T> {
 9     bool at_end;   //true if we have reached end of range
10     T t;
11     const T end;
12 public:
13     t_iterator(): at_end(true), end() {}
14
15     t_iterator(const T& initial_t, const T& initial_end):
16         at_end(false), t(initial_t), end(initial_end) {}
17
18     operator*() and operator== as in t_iterator2.h
19
20     t_iterator& operator++() {
21         if (at_end) {
22             cerr and exit;
23         }
24
25         if (t == end) {
26             at_end = true;
27         } else {
28             ++t;
29         }
30         return *this;
31     }
32
33     //etc.
34 };
35
```

```
//etc.
#endif
```

▲

**transform one input container**

Like `for_each`, the `transform` algorithm reads each element of an input container. Like `generate`, it writes to each element of an output container. The two containers could be the same one. But if they are not, they must be of the same length.

```
template <class INPUT, class OUTPUT, class FUNCTION>
OUTPUT transform(INPUT first, INPUT last, OUTPUT result, FUNCTION f)
{
    for (; first != last; ++first, ++result) {
        *result = f(*first);
    }

    return result;
}
```

Line 19 outputs a string in lowercase. Line 22 converts the characters to `unsigned char` to prevent them from sign extending when passed to `tolower`. For the `convert` template class, see p. 877. Line 21 needs the cast because the C++ Standard Library has more than one function with this name. The one we use here is inherited from the C Standard Library; the other takes a `locale` object as its second argument (p. 1041).

Line 26 demonstrates that the same range can be used for both input and output. Line 34 outputs the code number of each character in a string. We convert each character to `unsigned char` and then to `unsigned` to display the codes as non-negative integers. To see what would go wrong if we went directly to `unsigned`, look at line 15 of `static_cast.C` on p. 65.

Line 17 outputs the original `string`. The template class `identity` is not part of the standard library; its `operator()` member function takes one argument and returns its unchanged.

Had we wanted to be hypercorrect, we could have written `string::value_type` in place of the `char` in line 15.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/transform1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cctype>         //for toupper and tolower
 4 #include <string>         //for string
 5 #include <iterator>       //for ostream_iterator
 6 #include <algorithm>      //for transform
 7 #include <functional>     //for ptr_fun
 8 #include <ext/functional> //for compose1, identity
 9 #include "convert.h"      //for convert
10 using namespace std;
11
12 int main()
13 {
14     string s = "Hello\n";
15     ostream_iterator<char> it(cout);
16
17     transform(s.begin(), s.end(), it, __gnu_cxx::identity<char>());
18
19     transform(s.begin(), s.end(), it,
20         __gnu_cxx::compose1(
21             ptr_fun(static_cast<int (*)(int)>(tolower)),
22             convert<char, unsigned char>()
23         )
24     );
25
26     transform(s.begin(), s.end(), s.begin(),
27         __gnu_cxx::compose1(
28             ptr_fun(static_cast<int (*)(int)>(toupper)),
29             convert<char, unsigned char>()
30         )
31     );
32     cout << s;
33
34     transform(
35         s.begin(), s.end(),
36         ostream_iterator<unsigned>(cout, " "),
37         convert<char, unsigned char>()
38     );
39     cout << "\n";
40
41     return EXIT_SUCCESS;
42 }
```

```
Hello
hello
HELLO
72 69 76 76 79 10
```

The call to `transform` in the above line 26 does the same work as

```
43     for (string::iterator it = s.begin(); it != s.end(); ++it) {
44         *it = toupper(static_cast<unsigned char>(*it));
45     }
```

For examples of `transform` where the elements are objects and pointers thereto, see lines 33−34 of `mem_fun.C` on p. 942. To transform a `valarray`, see pp. 899−900.

**transform two input containers**

There is also a `transform` that takes two input containers of equal length.

```
 1 template <class INPUT1, class INPUT2, class OUTPUT, class FUNCTION>
 2 OUTPUT transform(INPUT1 first1, INPUT1 last1, INPUT2 first2, OUTPUT result,
 3           FUNCTION f)
 4 {
 5     for (; first1 != last1; ++first1, ++first2, ++result) {
 6         *result = f(*first1, *first2);
 7     }
 8
 9     return result;
10 }
```

In this case, the last argument of `transform` must be a binary function. We supply the start of the second input container in line 21.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/transform2.C`

```
#include <iostream>
#include <cstdlib>
#include <vector>
#include <iterator>
#include <functional>  //for plus
#include <algorithm>
using namespace std;

int main()
{
    const size_t n = 5;
    int a1[n] = {1700, 1900, 1900, 1900, 2000};
    int b1[n] = {  76,   29,   41,   69,    1};

    vector<int> a(a1, a1 + n);
    vector<int> b(b1, b1 + n);
    vector<int> c;

    transform(
        a.begin(), a.end(),
        b.begin(),
        back_inserter(c),
        plus<int>()
    );

    copy(c.begin(), c.end(), ostream_iterator<int>(cout, "\n"));
    return EXIT_SUCCESS;
}
```

```
1776
1929
1941
1969
2001
```

We have just seen one C++ equivalent for the *array operations* that are built into other languages. (An alternative is on pp. 897−900.) In PL/I, for example, we could have done the same thing with

```
    C = A + B;               /* PL/I example; A, B, C are arrays. */
```

Here are two more examples. We can do

```
    C = A ** B;              /* PL/I example: ** is exponentiation */
```

with

```
    transform(
        a.begin(), a.end(),   //a, b, c are vectors of double now
        b.begin(),
        back_inserter(c),
        static_cast<double (*)(double, double)>(pow)
    );
```

And we can do

```
37     C = (A + B) / 2;          /* PL/I example: take the average */
```

with

```
38     transform(
39         a.begin(), a.end(),
40         b.begin(),
41         back_inserter(c),
42         compose_fgx1_x2(
43             bind2nd(divides<double>(), 2.0),
44             plus<double>()
45         )
46     );
```

Did I mention that we would have to write our own `compose_fgx1_x2`? `f` will be a unary function (in this case, "divide by 2"), and `g` will be a binary function (in this case, "add"). In the `compose_fg1x_g2x` example on p. 865, `f` was the binary function and `g1` and `g2` were unary functions. The following line 61 is the punchline.

```
47 template <class F, class G>
48 class composer_fgx1_x2: public
49     binary_function<typename G::first_argument_type,
50                     typename G::second_argument_type,
51                     typename F::result_type> {
52     F f;
53     G g;
54 public:
55     composer_fgx1_x2(const F& initial_f, const G& initial_g)
56         : f(initial_f), g(initial_g) {}
57
58     typename F::result_type operator()(
59         const typename G:: first_argument_type& x1,
60         const typename G::second_argument_type& x2) const {
61         return f(g(x1, x2));
62     }
63 };
64
65 template <class F, class G>
66 inline class composer_fgx1_x2<F, G> compose_fgx1_x2(const F& f, const G& g)
67 {
68     return compose_fgx1_x2<F, G>(f, g);
69 }
```

Warning: there is no way to predict the order in which `transform` will process the elements of the ranges. The following `transform` behaves unpredictably.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/fibonacci.C`

```
#include <iostream>
#include <cstdlib>
#include <iterator>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{
    const size_t n = 10;
    int a[n] = {0, 1};   //initialize the first two elements

    transform(a, a + n - 2, a + 1, a + 2, plus<int>());
    copy(a, a + n, ostream_iterator<int>(cout, "\n"));
    return EXIT_SUCCESS;
}
```

```
0
1
1      The Fibonacci series: from this point onwards, each number is the sum of the two previous.
2
3
5
8
13
21
34
```

The two input containers given to `transform` can have elements of different types. Often one container holds objects; the other holds arguments for a member function of each object. In the following example, the containers holds `string` objects and subscripts for the member function `at`. Unfortunately, class `string` has more than one `at` function, for the same reason that class `mystring` had more than one function named `operator[]` on p. 314. To simplify line 21, lines 14−15 created a pointer named `at` to the member function named `string::at` that returns a `string::const_reference`, which is a hypercorrect way of saying `const char &`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/at.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <string>
 4 #include <iterator>     //for ostream_iterator
 5 #include <functional>   //for mem_fun_ref
 6 #include <algorithm>    //for transform
 7 using namespace std;
 8
 9 int main()
10 {
11     const size_t n = 3;
12     string a[n] = {"abe", "ike", "jake"}; //container of objects
13     string::size_type b[n] = {1, 1, 0};   //container of subscripts
14     string::const_reference (string::*const at)(string::size_type) const =
15         &string::at;
16
17     transform(
18         a, a + n,
19         b,
20         ostream_iterator<string::value_type>(cout, "\n"),
21         mem_fun_ref(at)
22     );
23
24     return EXIT_SUCCESS;
25 }
```

```
b
k
j
```

Without the pointer in the above lines 14−15, line 21 would have been

```
26      mem_fun_ref(
27          static_cast<
28          string::const_reference (string::*)(string::size_type) const>
29          (&string::at)
30      )
```

The call to `mem_fun_ref` in line 21 constructs and returns an anonymous object of class

```
const_mem_fun1_ref_t<string::value_type, string, string::size_type>
```

This class is similar to the class `const_mem_fun_ref_t` we saw on p. 872. `OBJECT` is the data type of each object in the container. `p` is a pointer to a `const` member function of class `OBJECT`. `X` and `Y` are the data types of the argument and return value of the member function to which `p` points. Line 36 is the punchline.

```
31 //Excerpt from <functional>.
32
33 template <class Y, class OBJECT, class X>
34 class const_mem_fun1_ref_t: public binary_function<OBJECT, X, Y> {
35 private:
36     Y (OBJECT::*p)(X x) const;
37 public:
38     explicit const_mem_fun_ref_t(Y (OBJECT::*initial_p)() const)
39         : p(initial_p) {}
40
41     Y operator()(const OBJECT& object, X x) const {return (object.*p)(x);}
42 };
```

The helper function `mem_fun_ref` constructs and returns a `const_mem_fun_ref_t`, just as the function `ptr_fun` constructs and returns a `pointer_to_unary_function`.

```
43 template <class Y, class OBJECT, class X>
44 inline const_mem_fun1_ref_t<Y, T> mem_fun_ref(Y (OBJECT::*p)(X x) const) {
45     return const_mem_fun1_ref_t<Y, OBJECT, X>(p);
46 }
```

## 8.4.4 Array Operations with valarray

A `valarray` is a vector of numbers for aggressively optimized, high-speed computation. Using a `valarray`, the array operation on pp. 894−895 can be done more simply with the `c = a+b` in the following line 17. In fact, any of the following statements could be written after line 17. For the math functions, we would change the `<int>`'s to `<double>`'s.

| | |
|---|---|
| `c = 10;` | *assign* `10` *to each element of* `c` |
| `c += 10;` | *add* `10` *to each element of* `c` |
| `c = a;` | *copy each element of* `a` *into the corresponding element of* `c` |
| `c += a;` | *add each element of* `a` *to the corresponding element of* `c` |
| `c = a + 10;` | *let each element of* `c` *be* `10` *greater than the corresponding element of* `a` |
| `c = (a + b) / 2;` | *let each element of* `c` *be the average of the corresponding elements of* `a` *and* `b` |
| `c = -a;` | |
| `c <<= 2;` | *left-shift each element of* `c` |
| `valarray<bool> equal = a == b;` | |
| | |
| `c = sqrt(a);` | |
| `c = pow(a, b);` | |
| `c = sin(a);` | |
| `c = atan(a, b);` | |
| `c = c.shift(2);` | *copy* `c[2]` *into* `c[0]`, `c[3]` *into* `c[1]`, *etc.* |

Like an array or `vector`, the elements of a `valarray` are stored consecutively in memory and can be accessed with a subscript (line 20). For a `valarray`, the subscript should be of type `size_t`. Like an array, the only available iterators are plain old pointers (line 25).

Since `c` is not a `const`, the `c[0]` in line 25 is an lvalue and we can take its address. (See the following Homework.) But the `&c[0]` cannot be rewritten as `c`. These brackets are the `operator[]` member function, which is not guaranteed to cancel with the `&`. Similarly, the `&c[n]` cannot be rewritten as `c+n`. Instead of doing pointer arithmetic, the expresison `c+n` would yield a new `valarray` each of whose elements is `n` greater than the corresponding element of `c`.

The `shift` and `cshift` member functions in lines 32 and 36 are `const`. But the `resize` in line 40 will overwrite the old values of the elements. Its second argument defaults to `T()`, where `T` is the data type stored in the `valarray`. In our example `T` is `int`, whose default constructor constructs a zero.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/valarray/valarray.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <valarray>   //for valarray
 4 #include <iterator>
 5 #include <algorithm>
 6 using namespace std;
 7
 8 int main()
 9 {
10     const size_t n = 5;
11
12     int a1[n] = {1700, 1900, 1900, 1900, 2000};
13     int b1[n] = {  76,   29,   41,   69,    1};
14
15     valarray<int> a(a1, n); //born containing 1700, 1900, 1900, 1900, 2000
16     valarray<int> b(b1, n);
17     valarray<int> c = a + b;
18
19     for (size_t i = 0; i < c.size(); ++i) {
20         cout << c[i] << " ";
21     }
22     cout << "\n";
23
24     ostream_iterator<int> it(cout, " ");
```

```
    copy(&c[0], &c[c.size()], it);
    cout << "\n";

    cout << "c.min() == " << c.min() << "\n"
        << "c.max() == " << c.max() << "\n"
        << "c.sum() == " << c.sum() << "\n";

    c = a + b.shift(2);    //left shift; negative argument for right
    copy(&c[0], &c[c.size()], it);
    cout << "\n";

    c = a + b.cshift(2);   //circular left shift; negative for right
    copy(&c[0], &c[c.size()], it);
    cout << "\n";

    c.resize(6, 1000);       //six 1000's
    copy(&c[0], &c[c.size()], it);
    cout << "\n";
    return EXIT_SUCCESS;
}
```

```
1776 1929 1941 1969 2001
1776 1929 1941 1969 2001
c.min() == 1776
c.max() == 2001
c.sum() == 9616
1741 1969 1901 1900 2000      left shift: zeroes enter from right end
1741 1969 1901 1976 2029
1000 1000 1000 1000 1000 1000
```

The following line 16 shows an equivalent for the `transform` algorithm. The standard library contains several different functions named `sqrt`, so the address of the `double` `sqrt` function would normally have to be written as

```
static_cast<double (*)(double)>(sqrt)
```

But the `apply` function of a `valarray<double>` will accept only a function whose argument is a `double`, so there is no ambiguity.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/valarray/apply.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cmath>
 4 #include <valarray>
 5 #include <iterator>
 6 #include <algorithm>
 7 using namespace std;
 8
 9 int main()
10 {
11     double a[] = {1, 16, 81};
12     const size_t n = sizeof a / sizeof a[0];
13     valarray<double> v(a, n);
14     ostream_iterator<double> it(cout, "\n");
15
16     v = v.apply(sqrt);                   //overwrite v
17     copy(&v[0], &v[n], it);
18     cout << "\n";
19
20     valarray<double> w = v.apply(sqrt);  //don’t overwrite v
21     copy(&w[0], &w[n], it);
22     return EXIT_SUCCESS;
23 }
```

```
1     lines 16−17
4
9

1     lines 20−21
2
3
```

**▼ Homework 8.4.4a: define an operator<< for valarray**

Class `valarray` has no `operator<<` function. Define one in a header file named `valarray_putto.h`. It will not need to be a member function or a friend of any class.

As usual, the second argument in line 8 will be a read-only reference to the variable being output.

```
 1 #ifndef VALARRAY_PUTTOH
 2 #define VALARRAY_PUTTOH
 3 #include <iterator>
 4 #include <algorithm>
 5 using namespace std;
 6
 7 template <class T>
 8 ostream& operator<<(ostream& ost, const valarray<T>& v)
 9 {
10     //The &v[0] and &v[v.size()] won’t compile.
11     copy(&v[0], &v[v.size()], ostream_iterator<T>(ost, "\n"));
```

The `operator[]` member function of a non-`const` `valarray<T>` returns a `T&`, allowing the return value to be used as an lvalue (pp. 12−13). The “address of” operator `&` can therefore be applied to the return value of this function. But the `operator[]` member function of a `const` `valarray<T>` returns a `T` without the `&`. (It returns the `T` by value.) Its return value is merely an rvalue, so the `&v[0]` in the above line 7 will not compile. Instead of calling `copy`, you will have to write a `for` loop to output the elements one by one. Could you call the `apply` member function?

See pp. 74−76 for functions that return lvalues and rvalues; p. 314 for classes with two different `operator[]` functions.

▲

### What is the data type of a+b?

What exactly is the data type of the expression `a+b` in line 17 of `valarray.C` on p. 898? The expression was used as if it were a `valarray`, but the truth is more complicated.

The `operator+` function, or any function that returns a `valarray` by value, returns an object that has the same friends and public members as a `const valarray`. It might even *be* a `const valarray`. But there is no guarantee of this, which means that the `operator<<` we just wrote may reject the argument `a+b` in the following line 10.

```
#include <iostream>
#include <valarray>
#include "valarray_putto.h"   //previous homework
using namespace std;

    valarray<int> a(argument(s) for constructor);
    valarray<int> b(argument(s) for constructor);

    cout << a;       //will compile: a is a valarray<int>
    cout << a + b;   //may not compile: a + b may not be a valarray<int>
    cout << valarray<int>(a + b);//will compile
```

**A slice of a valarray**

A *slice* is a set of elements having equally spaced subscripts in a `valarray`. The subscripts are stored in an object of class `slice`. Line 12 constructs a slice that holds the subscripts 0, 2, 4, 6, 8. The three arguments are the starting subscript, the number of subscripts, and the stride.

Watch what happens when line 15 uses the `slice` object as the subscript of a `valarray`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/valarray/slice.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <valarray>
 4 #include <iterator>
 5 #include <algorithm>
 6 using namespace std;
 7
 8 int main()
 9 {
10     const size_t n = 10;
11     valarray<int> v(n);   //born containing n int()’s, i.e., n zeroes
12     slice s(0, 5, 2);     //every other subscript, starting with 0
13
14     v[1] = 10;            //subscript is a size_t
15     v[s] = 20;            //subscript is a slice object
16
17     copy(&v[0], &v[n], ostream_iterator<int>(cout, "\n"));
18     return EXIT_SUCCESS;
19 }
```

```
20
10
20
0
20
0
20
0
20
0
```

Let’s use slices to build a Sieve of Eratosthenes for finding prime numbers. We start with a list of the integers from 0 to `n`−1 inclusive. Disregard the 0 and 1. Keep the 2, but remove its larger multiples: 4, 6, 8, etc. Keep the 3, but remove its larger multiples: 6, 9, 12, etc. The integer 4 has already been removed. Keep the 5, but remove its larger multiples: 10, 15, 20, etc. In each step, line 12 sets `p` to the next surviving integer. Line 13 keeps this integer, but removes its larger multiples. We leave it as an exercise for the reader to change the `for` loop and `if` in lines 16−22 into a call to a standard library algorithm.

Warning: if `v` were a `vector`, the two arguments in line 10 would be in the opposite order. See line 10 of `vector.C` on p. 430.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/valarray/sieve.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <valarray>
 4 #include <algorithm>
 5 using namespace std;
 6
 7 int main()
 8 {
 9     const size_t n = 30;
10     valarray<bool> v(true, n);   //born containing n true’s
11
12     for (size_t p = 2; 2*p < n; p = find(&v[p+1], &v[n], true) - &v[0]) {
13         v[slice(2 * p, n/p - 1, p)] = false; //remove the multiples of p
14     }
15
16     //Print the subscripts of the true elements.
17
18     for (size_t i = 2; i < n; ++i) {
19         if (v[i]) {
20             cout << i << "\n";
21         }
22     }
23
24     return EXIT_SUCCESS;
25 }
```

```
2
3
5
7
11
13
17
19
23
29
```

**What is the data type of v[s]?**

What exactly is the data type of the expression `v[s]` in line 15 of `slice.C` on p. 901? It depends on the data type of `v`. Class `valarray` has two `operator[]` member functions, one `const` and one non-`const`, returning different types. Since `c` is not `const`, the `v[s]` is an anonymous object of class `slice_array<int>`. This type of object acts as a reference to the selected elements of the `valarray`.

The constructors for `slice_array` are private. We can create one only by calling the `operator[]` member function of class `valarray`, which is a friend of `slice_array`. We have done this several times; the following line 24 does it again.

Only two groups of operators can be applied to a slice_array.

(1) We can assign a scalar to each element of the slice (line 24). Be warned that the assignments can happen in any order, depending on the hardware.

(2) We can assign the elements of a valarray<int> to each element of the slice (lines 27−30). Again, the assignments can happen in any order.

To do anything else with the elements to which a slice_array refers, they must first be copied into a valarray. This can be done by initialization (line 21) or by assignment (line 22). If the valarray is to be used only once, it can be an anonymous temporary (lines 10 and 32).

Here are three examples.

(1) There is a operator+= to add a valarray to a slice_array (line 28), but none to add a scalar to a slice_array (line 25). To get this line to compile, we had to define our own operator+= that copies the slice_array into a valarray (line 10).

Our operator+= would normally be a member function (p. 283). But class slice_array has already been written, and we don't want to modify a Standard Library class. To our relief, we find that our operator+= does not need to be a member function or a friend.

The assignment in line 10 assigns new values to the elements of the valarray to which s refers. But it does not change s itself, which merely acts as a reference to the members. This allows s to be const in line 9. In fact, it has to be const because the expression v[s] in line 25 is an anonymous temporary.

(2) There is also no operator[] for a slice_array. We don't want to define one, either, because an operator[] must always be a member function (p. 287). To apply the subscript [0] to the v[s], line 32 first had to construct a valarray from the v[s].

(3) There is no operator= that assigns one slice_array to another. Line 33 constructs an anonymous valarray and then calls the operator= we saw in line 27.

From these restrictions we conclude that a slice_array is intended to be only an intermediate result. The final result should reside in a valarray.

The assignment operators return void, forcing us to execute them in separate statements. Lines 29 and 33 cannot be combined to the following.

```
 1      v[t] = v[s] *= w;   //won't compile: the expression v[s] *= w is void
```

—On the Web at
http://i5.nyu.edu/~mm64/book/src/valarray/slice_array.C

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <valarray>
 4 #include <iterator>
 5 #include <algorithm>
 6 using namespace std;
 7
 8 template <class T>
 9 inline void operator+=(const slice_array<T>& s, const T& t) {
10     s = valarray<int>(s) + t;
11 }
12
13 int main()
14 {
15     int a[] = {0, 10, 20, 30, 40, 50, 60, 70, 80, 90};
16     const size_t n = sizeof a / sizeof a[0];
17     valarray<int> v(a, n);
18     slice s(0, 5, 2);
19     slice t(1, 5, 2);
20
21     valarray<int> w = v[s]; //initialization
22     w = v[s];               //assignment
23
24     v[s] = 10;
25     v[s] += 10;             //call line 9
26
27     v[s] = w;
28     v[s] += w;
29     v[s] *= w;
30     //etc.
31
32     cout << valarray<int>(v[s])[0] << "\n";
33     v[t] = v[s];  //behaves as if we had said v[t] = valarray<int>(v[s]);
34
35     copy(&v[0], &v[v.size()], ostream_iterator<int>(cout, "\n"));
36     return EXIT_SUCCESS;
37 }
```

```
0        line 32
0
0
800      = (20 + 20) × 20
800
3200
3200
7200
7200
12800
12800
```

The `v` in the above line 17 is non-`const`. Now let us suppose that `v` were `const`. The `operator[]` member function of a `const valarray` returns an object that has the same friends and public members as a `const valarray`. It might even *be* a `const valarray`. We can therefore help ourselves to the cornucopia of `valarray` operations we saw earlier.

```
38     valarray<int> c = v[s] + v[t];
39     valarray<double> d = sqrt(v[s]);   //if v were a valarray<double>
```

Remember, though, that `v` is now a `const`. The `v[s]` is no longer an lvalue, so we cannot say

```
40     v[s] = sqrt(c);
```

And `v[s]` is not necessarily an object of type `valarray`, which means there is no way to write a portable `operator<<` for `v[s]`. The moral, once again, is to copy the final result into a `valarray`.

```
41     cout << v[s];                      //No way to make this portable.
42     cout << valarray<int>(v[s]);       //Must say this instead.
```

### A multi-dimensional matrix

The Sieve example was intended only as a learning tool. The real purpose of slicing is to let us access a `valarray` as if its elements were arranged in rows and columns. This gives us the raw material for creating the vectors and matrices of Linear Algebra.

Since `v` is `const`, the expressions `v[row0]` and `v[col0]` can be multiplied together as if they were `valarray`'s. (They might even *be* `valarray`'s.) If `v` were not `const`, these expressions would be `slice_array`'s. They would have to be converted to `valarray`'s before the `*` operator could be applied to them. In both cases, the product would have all the friends and `const` member functions as a `valarray`. (It might even *be* a `valarray`.)

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/valarray/dimension.C`

```
#include <iostream>
#include <cstdlib>
#include <valarray>
using namespace std;

int main()
{
    int a[] = {2, 3, 5, 7};
    const valarray<int> v(a, sizeof a / sizeof a[0]);

    /*
    Treat the valarray as if it were the following 2 by 2 matrix.
    This interpretation, called "row-major order", is used in C and C++.
        2 3
        5 7
    */
    slice row0(0, 2, 1);         //row vector containing 2 and 3
    slice col0(0, 2, 2);         //column vector containing 2 and 5
    cout << (v[row0] * v[col0]).sum() << "\n";

    /*
    Treat the valarray as if it were the following 2 by 2 matrix.
    This interpretation, called "column-major order", is used in Fortran.
        2 5
        3 7
    */
    slice fortran_row0(0, 2, 2); //row vector containing 2 and 5
    slice fortran_col0(0, 2, 1); //column vector containing 2 and 3
    cout << (v[fortran_row0] * v[fortran_col0]).sum() << "\n";
    return EXIT_SUCCESS;
}
```

```
19           = 2 × 2 + 3 × 5  (dot product)
19           = 2 × 2 + 5 × 3
```

**A multi-dimensional slice**

Consider the matrix

```
 0  1  2  3  4
10 11 12 13 14
20 21 22 23 24
30 31 32 33 34
```

and its two-dimensional submatrix

```
11 12 13
21 22 23
```

The top row of the submatrix is

```
11 12 13
```

This row can be described by `slice(6, 3, 1)`. The left column of the submatrix is

```
11
21
```

This column can be described by `slice(6, 2, 5)`. Together, these two slices span the two-dimensional submatrix. Both start with the element whose value is 11 and whose subscript is 6. This subscript is the first argument in line 23. The two columns in lines 17−18 hold the remaining arguments of the constructors of the slices. There are two columns because the submatrix has two dimensions.

The `g` in line 23 is a *generalized slice,* which may have more or less than two dimensions. The alternatives for the data type of the `v[g]` in line 24 are similar to those for the `v[s]` in line 15 of `slice.C` on p. 901. Since `v` is `const`, `v[g]` has all the friends and public members of a `const valarray`, including the `sum` in line 24. If `v` were not `const`, `v[s]` would be of type `gslice_array<int>` and we would have to change line 24 to

```
    cout << valarray<int>(v[g]).sum() << "\n";
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/valarray/gslice.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <valarray>
 4 using namespace std;
 5
 6 int main()
 7 {
 8     int a[] = {
 9          0,  1,  2,  3,  4,
10         10, 11, 12, 13, 14,
11         20, 21, 22, 23, 24,
12         30, 31, 32, 33, 34
13     };
14     const valarray<int> v(a, sizeof a / sizeof a[0]);
15
16     const size_t n = 2;   //number of dimensions of submatrix
17     size_t length1[n] = {3, 2};
18     size_t stride1[n] = {1, 5};
19
20     valarray<size_t> length(length1, n);
21     valarray<size_t> stride(stride1, n);
22
23     gslice g(6, length, stride);
24     cout << v[g].sum() << "\n";
25     return EXIT_SUCCESS;
26 }
```

```
102            = 11 + 12 + 13 + 21 + 22 + 23
```

### mask_array and indirect_array

The subscript in line 18 is a `valarray<bool>`. The one in line 19 is a `valarray<size_t>`. The resulting `v[s]` and `v[t]` have data types similar to the `v[s]` in line 15 of `slice.C` on p. 901. Since `v` is not `const`, they cannot be subscripted. To print them, we must copy the referenced elements into a `valarray` and then print it.

See p. 958 for another way to permute the elements of a container.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/valarray/mask.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <valarray>
 4 #include <iterator>
 5 #include <algorithm>
 6 using namespace std;
 7
 8 int main()
 9 {
10     int v1[]  = {0,     10,    20,    30,    40,   50};
11     bool s1[] = {false, true,  false, true,  true, false};
12     size_t t1[] = {3, 4, 1};
13
14     valarray<int>    v(v1, sizeof v1 / sizeof v1[0]);
15     valarray<bool>   s(s1, sizeof s1 / sizeof s1[0]);
16     valarray<size_t> t(t1, sizeof t1 / sizeof t1[0]);
17
18     valarray<int> w = v[s];   //v[s] is a mask_array<int>
19     valarray<int> x = v[t];   //v[t] is an indirect_array<int>
20
21     ostream_iterator<int> it(cout, "\n");
22     copy(&w[0], &w[w.size()], it);
23     cout << "\n";
24     copy(&x[0], &x[x.size()], it);
25     return EXIT_SUCCESS;
26 }
```

```
10            elements in original order
30
40

30            elements in a permuted order
40
10
```

### ▼ Homework 8.4.4b: print the Sieve with a mask_array

Create a `valarray<int>` named `w`, containing `n` zeroes. Overwrite it with the integers from 0 to `n`−1 inclusive by passing a pair of `t_iterator`’s to the `copy` algorithm. Then copy `w[v]` into another `valarray<int>` and print it, where `v` is the `valarray<bool>` holding the Sieve of Eratosthenes (pp. 901−902).

▲

**▼ Homework 8.4.4c: play the game of life on a valarray**

Let the `matrix` data member of class `life`, and the `newmatrix` variable in `life::operator++`, be of type `valarray<bool>`. Do not change the type of `life::matrix_t`.

For convenience, give class `life` the following private, static member function. It returns the subscript in the `matrix` of the element that the user sees at column `x`, row `y`. Insert error checking if desired.

```
 1        static size_t xy(size_t x, size_t y) {return y * (xmax + 2) + x;}
```

The constructor will make the `matrix` big enough to hold the `xmax` columns and `ymax` rows that the user sees, plus the border. The `false` in line 3 is unnecessary, since it defaults to `bool()`. The `initial_matrix[y-1]` in line 7 is a pointer to a `bool`.

```
 2 life::life(const matrix_t initial_matrix)
 3         : g(0), matrix(false, (life_ymax + 2) * (life_xmax + 2))
 4 {
 5         for (size_t y = 1; y <= life_ymax; ++y) {
 6                 matrix[slice(xy(1, y), life_xmax, 1)]
 7                         = valarray<bool>(initial_matrix[y - 1], life_xmax);
 8         }
 9 }
```

For each `x`, `y` that the user sees, `life::operator++` should subscript the `matrix` with a `gslice` describing the 3 × 3 submatrix centered at `x`, `y`. The result of this subscripting will be a `valarray<bool>` of nine elements. Pass it to the `count` algorithm to count how many of the elements are `true`. (Since `life::operator++` already has a local variable named `count`, you will have to refer to the algorithm as `std::count`.) At the end of `life::operator++`, the `newmatrix` may be copied into `matrix` simply by saying

```
10         matrix = newmatrix;
```

Don't bother to allow the user to specify the `filled` and `empty` characters. Just `copy` the `bool`'s to the output stream with an `ostream_iterator<bool>`.

Another game that can be implemented with a `valarray` is Sudoku. It is played on a 9 × 9 matrix of integers. The elements are accessed one row at a time, one column at a time, or one 3 × 3 submatrix at a time.

▲

## 8.4.5 The `min_element` Algorithm and an Application

The `min_element` algorithm returns an iterator referring to the smallest element in a range of territory. If the range is empty, `min_element` returns its second argument. With two arguments (lines 26 and 30 of the following `min_element.C`), it compares the elements by applying the `<` operator to them. If you're not satisfied with `<`, a third argument can supply a different kind of comparison (lines 33, 36, 39). The third argument must be a predicate taking two arguments, returning `true` if its first argument is less than its second by your definition of "less than".

The iterators passed to `min_element` must do more than those passed to `find_if`. Line 10 copies an iterator and line 13 dereferences both copies. This means they cannot be merely input iterators. They must be at least forward iterators. They must also be forward iterators because each value is read more than once. Note that `min_element`, like our other forward iterator algorithm `adjacent_find` (p. 840), is careful not to copy any value of type `T`.

The predicate passed to `min_element` is more complicated that the one passed to `find_if`. The former is a predicate of two arguments; the latter, of one argument. As usual, the data type of the elements in the range must be the same as (or convertible to) the data type of the arguments of the predicate. Otherwise, the call to `min_element` will not compile.

```
 1 //Excerpt from <algorithm>
 2
 3 template <class FORWARD>
 4 FORWARD min_element(FORWARD first, FORWARD last)
 5 {
 6     if (first == last) {
 7         return last;   //range is empty
 8     }
 9
10     FORWARD it = first;
11
12     while (++first != last) {
13         if (*first < *it) {
14             it = first;
15         }
16     }
17
18     return it;
19 }
20
21 template <class FORWARD, class COMPARE>
22 FORWARD min_element(FORWARD first, FORWARD last, COMPARE compare)
23 {
24     if (first == last) {
25         return last;
26     }
27
28     FORWARD it = first;
29
30     while (++first != last) {
31         if (compare(*first, *it)) {
32             it = first;
33         }
34     }
35
36     return it;
37 }
```

The following class and function are named after the mathematical expression

$$f(g(x_1), g(x_2))$$

Line 21 is the punchline. Compare the $f(g_1(x), g_2(x))$ on p. 865.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/min_element/composer_fgx1_gx2.h`

```
 1 #ifndef COMPOSER_FGX1_GX2
 2 #define COMPOSER_FGX1_GX2
 3 #include <functional>   //for binary_function
 4 using namespace std;
 5
 6 //Compose the functions f(g(x1), g(x2)).
 7
 8 template <class F, class G>
 9 class composer_fgx1_gx2: public binary_function<typename G::argument_type,
10                                                 typename G::argument_type,
                                                      typename F::result_type> {
    F f;
    G g;
public:
    composer_fgx1_gx2(const F& initial_f, const G& initial_g)
        : f(initial_f), g(initial_g) {}

    typename F::result_type operator()(
        const typename G::argument_type& x1,
        const typename G::argument_type& x2) {
        return f(g(x1), g(x2));
    }
};

template <class F, class G>
inline composer_fgx1_gx2<F, G> compose_fgx1_gx2(
    const F& initial_f,
    const G& initial_g)
{
    return composer_fgx1_gx2<F, G>(initial_f, initial_g);
}
#endif
```

The expression `nearer_to_32_func` in line 33 is a straightforward predicate; it's the address of the free function in lines 10−11. Unfortunately, this function has the value `32` hardwired into it, so we'd have to write a different function to search for a different number. It's also called via a pointer, so it's slow.

A more convenient way to make predicates is by constructing objects of class `nearer_to` in lines 13−19. Instead of hardwiring the `32` into an object of this class, line 36 can pass the `32` as an argument to the object's constructor. The `32` is then used by the object's `operator()` function in line 18. Line 36 can just as easily put a different number into a different object of this class. Furthermore, the `operator()` is inline.

The header file `<cstdlib>` declares several functions named `abs`.

```
int abs(int);
long abs(long);
```

(There are also `float`, `double`, and `long double` versions, in `<cmath>`.) Had there been only one, we would not have needed the explicit template arguments `<int, int>` for the `ptr_fun` function in line 43.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/min_element/min_element.C`

```
 1 #include <iostream>
 2 #include <cstdlib>        //for abs and EXIT_SUCCESS
 3 #include <vector>
 4 #include <algorithm>      //for min_element
 5 #include <functional>     //for ptr_fun
 6 #include <ext/functional> //for compose1
 7 #include "composer_fgx1_gx2.h"
 8 using namespace std;
 9
10 //Return true if the first argument is nearer to 32 than the second argument is.
11 inline bool nearer_to_32_func(int a, int b) {return abs(a - 32) < abs(b - 32);}
12
13 //Return true if the first argument is nearer to 32 than the second argument is.
```

```
14 class nearer_to {
15     const int n;
16 public:
17     nearer_to(int initial_n): n(initial_n) {}
18     bool operator()(int a, int b) const {return abs(a - n) < abs(b - n);}
19 };
20
21 int main()
22 {
23     const int a[] = {50, 10, 30, 35, 40};
24     const size_t n = sizeof a / sizeof a[0];
25
26     const int *const p = min_element(a, a + n);
27     cout << "The smallest number in the array is " << *p << ".\n";
28
29     const vector<int> v(a, a + n);
30     vector<int>::const_iterator it = min_element(v.begin(), v.end());
31     cout << "The smallest number in the vector is " << *it << ".\n";
32
33     it = min_element(v.begin(), v.end(), nearer_to_32_func);
34     cout << "The number that’s nearest to 32 is " << *it << ".\n";
35
36     it = min_element(v.begin(), v.end(), nearer_to(32));
37     cout << "The number that’s nearest to 32 is " << *it << ".\n";
38
39     it = min_element(v.begin(), v.end(),
40         compose_fgx1_gx2(
41             less<int>(),
42             __gnu_cxx::compose1(
43                 ptr_fun<int, int>(abs),
44                 bind2nd(minus<int>(), 32))
45         )
46     );
47     cout << "The number that’s nearest to 32 is " << *it << ".\n";
48
49     return EXIT_SUCCESS;
50 }
```

```
The smallest number in the array is 10.          lines 26−27
The smallest number in the vector is 10.         lines 29−31
The number that’s nearest to 32 is 30.           lines 33−34
The number that’s nearest to 32 is 30.           lines 36−37
The number that’s nearest to 32 is 30.           lines 39−47
```

### ▼ Homework 8.4.5a: flee from the nearest enemy

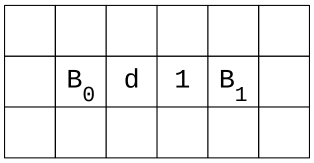
The `visionary::decide` on pp. 574−580 runs from the first enemy it finds. Let’s make it smart enough to run from the *nearest* enemy, or from one of the nearest enemies if two or more are equally near to the `visionary`.

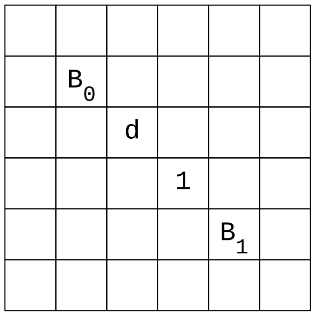
The predicates in lines 9, 32, and 36 have one argument; they can be passed to `find_if`. The predicate in line 21 has two arguments; it can be passed to `min_element`.

A random access container (one whose iterators are random access) is used in line 48 so it can be passed to the `sort` in line 52. The `begin` and `end` in line 49 are the member functions of class `wabbit` that return `wabbit::const_iterator`’s. We created them on p. 578.

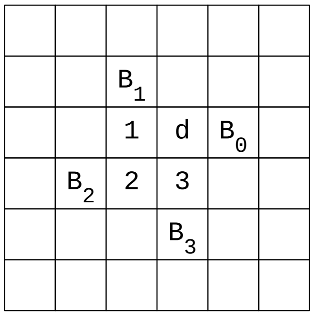
```
 1 //Excerpt from visionary.C.
 2
 3 /*
 4 Return true if the other wabbit is near enough to w to be visible, and is not w
 5 itself.  (No wabbit should be afraid of itself or should contemplate eating its
 6 own flesh.)
 7 */
 8
 9 class near_to: public unary_function<const wabbit *, bool> {
10     const wabbit *const w;
11 public:
12     near_to(const wabbit *initial_w): w(initial_w) {}
13
14     bool operator()(const wabbit *other) const {
15         return other != w && dist(w, other) <= 3;
16     }
17 };
18
19 //Return true if w1 is nearer to w than w2 is.
20
21 class nearer_to: public binary_function<const wabbit *, const wabbit *, bool> {
22     const wabbit *const w;
23 public:
24     nearer_to(const wabbit *initial_w): w(initial_w) {}
25
26     bool operator()(const wabbit *w1, const wabbit* w2) const {
27         //Imitate the nearer_to in lines 13-19 of above min_element.C,
28         //but instead of abs, use the dist function on pp. 577−578
29     }
30 };
31
32 class can_eat: public unary_function ...
33     //Left as an exercise: see how it’s used in line 55.
34 };
35
36 class can_be_eaten_by: ... {
37     //Left as an exercise; see how it’s used in line 62.
38 };
39
40 //Move one step away from the nearest enemy in visual range.
41 //If there are none, move one step toward the nearest food in visual range.
42
43 void visionary::decide(int *dx, int *dy) const
44 {
45     //Make a vector of all the wabbits that are near enough to be
46     //visible to this one, not counting this one.
47
48     vector<wabbit *> visibles;
49     remove_copy_if(begin(), end(), back_inserter(visibles),
50         not1(near_to(this)));
51
52     sort(visibles.begin(), visibles.end(), nearer_to(this));
53
54     vector<wabbit *>::const_iterator it =
        find_if(visibles.begin(), visibles.end(), can_eat(this));

    if (it != visibles.end()) {
        step(*it, this, dx, dy);    //Move one step away from the other wabbit.
        return;
    }

    it = find_if(visibles.begin(), visibles.end(), can_be_eaten_by(this));

    if (it != visibles.end()) {
        step(this, *it, dx, dy);    //Move one step toward the other wabbit.
        return;
    }

    *dx = *dy = 0;                  //lethargic in the absence of stimulation
}
```

To verify that the `deer` now flees from the nearest enemy, put more than one `black_hole` in visual range. Compare these diagrams with the ones on p. 575.

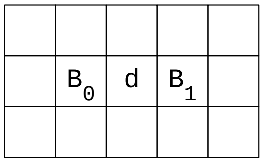




This `deer` has four enemies in visual range. It will be driven around and around the numbered path.



Unfortunately, this deer will step away from one `black_hole` right into the other.



And this `alien` will bump its head against the `boulder`.

| | | | | |
|---|---|---|---|---|
| | | | | |
| | A | b | s | |
| | | | | |

▲

## 8.5 The Rudiments of Dispatching

Each algorithm in the Standard Template Library requires a certain minimum category of iterator. `copy` requires its first two arguments to be at least input iterators; `min_element` requires forward iterators; `reverse_copy` requires bidirectional; and `sort` requires `random_access`.

The iterators passed to an algorithm may exceed the minimum requirements. `copy`, for example, will happily accept iterators that are forward, bidirectional, or random access. In fact, it will run faster with random access iterators.

We can write several implementations for each algorithm, and let the computer select the one that best exploits the category of iterator passed as an argument. To perform the selection at compile time, we will make sophisticated use of an elementary topic that has lain dormant since Chapter 1: function name overloading.

### 8.5.1 Dispatch the `advance` algorithm

Let's start with a simple algorithm. The library has one named `advance` that advances an input iterator by adding an integer to it. Here is a pseudo-code definition showing that each category of iterator has its own strengths and weaknesses.

(1) We hope that the iterator argument `it` is random access. If so, it can be advanced or retracted in a single bound (lines 6−8). Note that `d` could be negative.

(2) If the iterator is merely bidirectional, the `+=` operator cannot be applied to it. We can still get the job done, but more slowly (lines 10−18). Once again, `d` could be negative.

(3) If the iterator is merely a forward or input iterator, `d` must be non-negative. Now we have to do error checking (lines 20−29).

(4) If the argument is none of the above—merely an output iterator, or not an iterator at all—we should issue a compilation error (lines 31−33).

```
 1 //Pseudo-code excerpt from <iterator>
 2
 3 template <class ITERATOR, class DIFFERENCE_TYPE>
 4 void advance(ITERATOR& it, DIFFERENCE_TYPE d)   //read/write reference
 5 {
 6     if (it is a random access iterator) {
 7         it += d;
 8     }
 9
10     else if (it is a bidirectional iterator) {
11         for (; d > 0; --d) {
12             ++it;
13         }
14
15         for (; d < 0; ++d) {
16             --it;
17         }
18     }
19
20     else if (it is a forward or input iterator) {
21         if (d < 0) {
22             cerr << "Can’t move a non-bidir iterator backwards.\n";
23             exit(EXIT_FAILURE);
24         }
25
26         for (; d > 0; --d) {
27             ++it;
28         }
29     }
30
31     else {
32         compilation error: output iterator would need * before each ++
33     }
34 }
```

The `advance` algorithm is not needed if we already know the iterator’s category. In this case, we know enough to advance the iterator in the fastest way: line 49 for a random access iterator, lines 54−55 for a bidirectional. `advance` is needed only inside another algorithm. At line 40, for example, all we have is the opaque name `ITERATOR`. We don’t know the category of `it`, and `advance` must be called to select the best code.

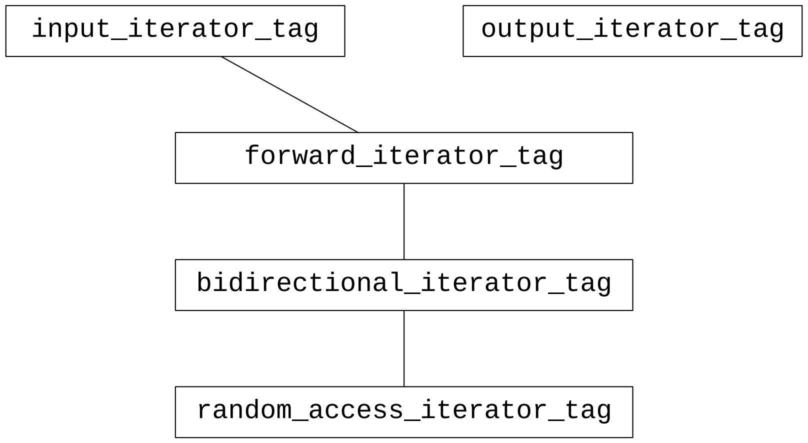
```
35 template <class ITERATOR>
36 void tiny_algorithm(ITERATOR it)
37 {
38     //Need to call advance here: the category of it may be different
39     //each time the tiny_algorithm is called.
40     advance(it, 2);
41 }
42
43 int main()
44 {
45     const int a[] = {10, 20, 30, 40, 50};
46     const size_t n = sizeof a / sizeof a[0];
47
48     int *it1 = a;
49     it1 += 2;  //No need to call advance here--we know it1 is random access.
50     tiny_algorithm(it1);
51
52     list<int> li(a, a + n)
53     list<int>::iterator it2 = li.begin();
54     ++it2;     //No need to call advance here--we know it2 is bidirectional.
55     ++it2;
56     tiny_algorithm(it2);
```

Now let’s make the pseudo-code compile. To avoid conflict with the standard library algorithm `advance`, we will name our function `my_advance`, in line 50 of the following `advance.C`.

For most applications, the best code is the fastest code. Selecting the best code for each possible category of iterator is called *dispatching,* performed at compile time by function name overloading. Instead of writing the pseudocode `if-else` with three clauses, we will give the same name to three functions. The functions will have to differ in the number or data type of their arguments.

Recall that we have five iterator category tag classes (p. 842). Although they are empty, they still count as different data types. Most of them are related by inheritance.

```
57 //Excerpt from <iterator>
58
59 struct input_iterator_tag {};
60 struct output_iterator_tag {};
61 struct forward_iterator_tag: public input_iterator_tag {};
62 struct bidirectional_iterator_tag: public forward_iterator_tag {};
63 struct random_access_iterator_tag: public bidirectional_iterator_tag {};
```



The `iterator_category` member of any `iterator_traits` class is a typedef (an alternative name) for one of the above five classes. The ones we will need for the following program are

```
iterator_traits<vector<int>::iterator>::iterator_category
```

which is a typedef for `random_access_iterator_tag`;

```
iterator_traits<list<int>::iterator>::iterator_category
```

which is a typedef for `bidirectional_iterator_tag`; and

```
iterator_traits<node::iterator>::iterator_category
```

which we made a typedef for `forward_iterator_tag` (p. 806).

When 63 passes a `vector<int>::iterator` to the template function `my_advance` in line 50, the computer behaves as if we had called a copy of this function with every occurrence of the name `INPUT` changed to `vector<int>::iterator`. For example, the data type

`typename iterator_traits<INPUT>::iterator_category`

in line 53 is changed to

`typename iterator_traits<vector<int>::iterator>::iterator_category`

As we remarked, this data type is a typedef for class `random_access_iterator_tag`. Line 53 therefore constructs an anonymous object of this class. Its constructor takes no arguments, which is why the parentheses are empty. (Class `random_access_iterator_tag` actually has no constructor at all, but we have to write the parentheses anyway to create an object of this class.) The anonymous object is passed to one of the `__my_advance` functions. Like an iterator and a `difference_type`, a tag object is small enough to be passed by value.

To show that the `__my_advance` functions should not be called directly by the user, their names start with a double underscore. There are several of them, in lines 10, 18, and 33, so their arguments must differ. Since line 63 called `my_advance` with a `vector<int>::iterator` argument, lines 52−53 call the `__my_advance` whose third argument is an `random_access_iterator_tag` (line 10). Line 10 doesn’t even bother to declare a name for the third argument because its value is never used. In fact, it has no value at all—it's an empty object. Only the data type of the argument is used, to navigate us from lines 52−53 to line 10. For other arguments whose value is not used, see pp. 585−587.

All the work of advancing the iterator is done in the `__my_advance` at line 10. The function in line 50 was merely a *dispatching function:* a trick to make the call that originates in line 63 end up at line 10. A dispatching function is an inline call-through. Even better, the decision to go from lines 52−53 to line 10 is made at compile time, not at runtime, since that is when the computer decides which function with an overloaded name to call.

If lines 9−15 were deleted, lines 52−53 would be happy to call the `__my_advance` in line 18 because a `random_access_iterator_tag` is also a `bidirectional_iterator_tag`. But given a choice between the `__my_advance`'s in lines 10 and 18, lines 52−53 prefer line 10 because function name overloading selects the closest match.

Line 68 passes a `list<int>::iterator` to the dispatching function in line 50. This time, line 53 will construct an anonymous object of class `bidirectional_iterator_tag` and pass it to the `__my_advance` in line 18.

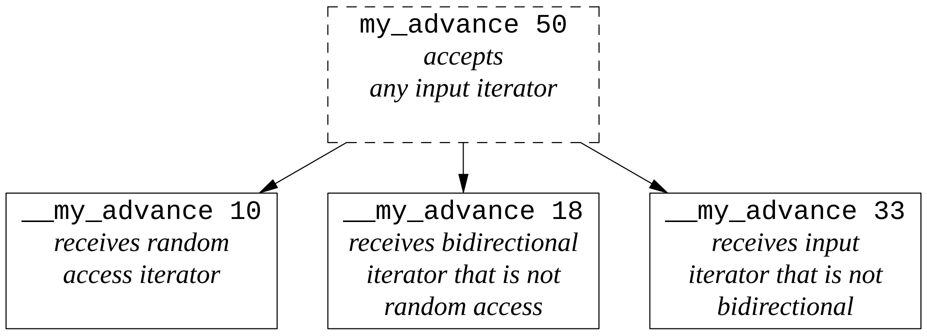
Lastly, line 78 passes a `node::iterator` to the dispatching function, and line 53 constructs an anonymous object of class `forward_iterator_tag`. If there were a `__my_advance` whose third argument was of type `forward_iterator_tag`, lines 52−53 would call it. There isn't, but 52−53 are happy to call the `__my_advance` in 33 because an `forward_iterator_tag` is also an `input_iterator_tag`. (The `forward_iterator_tag` will be sliced when it is received at line 33 [pp. 490−491]. But it has no value, so no one cares.)

Every forward iterator is an output iterator as well as an input iterator, so why wasn't class `forward_iterator_tag` derived from class `output_iterator` as well as from `input_iterator`? Well, it should have been, and no one remembers why it wasn't. Fortunately, there is no algorithm that requires at minimum an output iterator, but that can run faster with a forward, bidirectional, or random access.

If we passed an iterator that was not an input iterator to the dispatching function in line 50, lines 52−53 would have no suitable `__my_advance` to call and the program would not compile. And if we passed an argument that was not an iterator at all (i.e, that had no `iterator_traits`), line 53 itself would not compile.

All of the above machinery is hidden from the user. He or she simply calls `my_advance` in lines 63, 68, and 78, and automatically gets the fastest code. As with any template, there is always a price to pay. A separate instantiation of each function is created for each type of argument passed to it, and these instantiations take up space. But we don't care—we want the maximum speed.

The diagram shows the line number of each function definition. The dispatching function is dashed.


my_advance 50
*accepts any input iterator*
__my_advance 10
*receives random access iterator*
__my_advance 18
*receives bidirectional iterator that is not random access*
__my_advance 33
*receives input iterator that is not bidirectional*

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/dispatch/advance.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <vector>
```

```
#include <list>
#include <iterator>   //for iterator_traits
#include "node.h"
using namespace std;

template <class RANDOM, class DIFFERENCE_TYPE>
inline void __my_advance(RANDOM& it, DIFFERENCE_TYPE d,
    random_access_iterator_tag)
{
    cout << "random access iterator __my_advance\n";
    it += d;
}

template <class BIDIRECTIONAL, class DIFFERENCE_TYPE>
void __my_advance(BIDIRECTIONAL& it, DIFFERENCE_TYPE d,
    bidirectional_iterator_tag)
{
    cout << "bidirectional iterator __my_advance\n";

    for (; d > 0; --d) {
        ++it;
    }

    for (; d < 0; ++d) {
        --it;
    }
}

template <class INPUT, class DIFFERENCE_TYPE>
void __my_advance(INPUT& it, DIFFERENCE_TYPE d, input_iterator_tag)
{
    cout << "input iterator __my_advance\n";

    if (d < 0) {
        cerr << "Can’t move a non-bidirectional iterator backwards.\n";
        exit(EXIT_FAILURE);
    }

    for (; d > 0; --d) {
        ++it;
    }
}

//The dispatching function is always inline.

template <class INPUT, class DIFFERENCE_TYPE>
inline void my_advance(INPUT& it, DIFFERENCE_TYPE d)
{
    __my_advance(it, d,
        typename iterator_traits<INPUT>::iterator_category());
}

int main()
{
```

```
58     const int a[] = {10, 20, 30, 40, 50};
59     const size_t n = sizeof a / sizeof a[0];
60
61     vector<int> v(a, a + n);
62     vector<int>::iterator it1 = v.begin();
63     my_advance(it1, 4);
64     cout << *it1 << "\n";
65
66     list<int> li(a, a + n);
67     list<int>::iterator it2 = li.begin();
68     my_advance(it2, 4);
69     cout << *it2 << "\n";
70
71     node *begin = new node(50, 0);
72     begin = new node(40, begin);
73     begin = new node(30, begin);
74     begin = new node(20, begin);
75     begin = new node(10, begin);
76
77     node::iterator it3 = begin;
78     my_advance(it3, 4);
79     cout << *it3 << "\n";
80
81     return EXIT_SUCCESS;
82 }
```

The function call in the above lines 52−53 can be split into separate statements in lines 83−84.

```
83     typedef typename iterator_traits<ITERATOR>::iterator_category category;
84     __my_advance(it, d, category());
```

```
random access iterator __my_advance       lines 61−64
50
bidirectional iterator __my_advance       lines 66−69
50
input iterator __my_advance               lines 71−79
50
```

### 8.5.2 Dispatch the copy algorithm

Perhaps the most heavily used algorithm is `copy`. Here is the simplest possible definition, accepting any type of input iterator as its first two arguments.

```
template <class INPUT, class OUTPUT>
OUTPUT copy(INPUT first, INPUT last, OUTPUT result)
{
    for (; first != last; ++first, ++result) {
        *result = *first;
    }

    return result;
}
```

**A separate implementation for random access iterators**

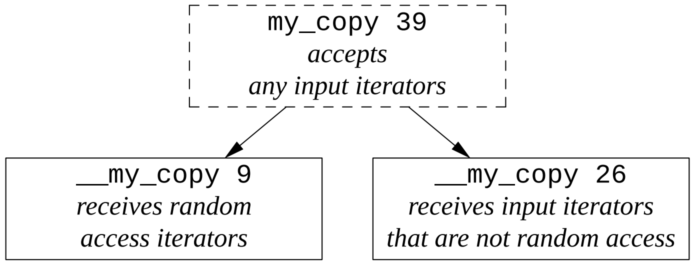
To avoid conflict with the standard library algorithm `copy`, we will name ours `my_copy`. The dispatching function in the following line 39 is just like the one in line 50 of the above `advance.C`. Based on the category of the first and second arguments, it will call line 9 for random access iterators, line 26 for other categories of input iterators, and give a compilation error for arguments that are none of the above.

Suppose the first two iterators passed to `my_copy` were of a random access type, represented by the `RANDOM` in line 9. If we were so inclined we could then compute their difference in line 16, yielding the number of times to iterate. To hold this result, we would need a variable of data type

```
typename iterator_traits<RANDOM>::difference_type
```

Lines 13−14 create a handy, one-word name for this type, which is just another name for `int`, `long`, etc. We saw this use of typedef in the above lines 83−86.

Why are we doing this? See “unrolling”, below.



—On the Web at
`http://i5.nyu.edu/~mm64/book/src/dispatch/copy1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <list>
 4 #include <iterator>
 5 #include <algorithm>
 6 using namespace std;
 7
 8 template <class RANDOM, class OUTPUT>
 9 OUTPUT __my_copy(RANDOM first, RANDOM last, OUTPUT result,
10     random_access_iterator_tag)
11 {
12     cout << "random access iterators\n";
13     typedef typename iterator_traits<RANDOM>::difference_type
14         difference_type;
15
16     for (difference_type i = last - first; i > 0; --i) {
17         *result = *first;
18         ++first;
19         ++result;
20     }
21
22     return result;
23 }
24
25 template <class INPUT, class OUTPUT>
26 OUTPUT __my_copy(INPUT first, INPUT last, OUTPUT result,
27     input_iterator_tag)
28 {
```

```
29     cout << "input iterators that are not random access\n";
30
31     for (; first != last; ++first, ++result) {
32         *result = *first;
33     }
34
35     return result;
36 }
37
38 template <class INPUT, class OUTPUT>
39 inline OUTPUT my_copy(INPUT first, INPUT last, OUTPUT result)
40 {
41     typedef typename iterator_traits<INPUT>::iterator_category
42         iterator_category;
43
44     return __my_copy(first, last, result, iterator_category());
45 }
46
47 int main()
48 {
49     const int a[] = {10, 20, 30};
50     const size_t n = sizeof a / sizeof a[0];
51
52     //First two arguments of my_copy are random access iterators.
53     my_copy(a, a + n, ostream_iterator<int>(cout, "\n"));
54     cout << "\n";
55
56     //1st 2 args of my_copy are input iterators that are not random access.
57     list<int> li(a, a + n);
58     my_copy(li.begin(), li.end(), ostream_iterator<int>(cout, "\n"));
59
60     return EXIT_SUCCESS;
61 }
```

**Unroll the loop**

Introducing the extra variable `i` in the random access version of `__my_copy` gives us a big advantage. A loop that iterates until an integer counts down to zero (`i > 0` in the above line 16) can run faster than one that compares two arbitrary variables (`first != last` in the above line 31). This is not because the expression `i > 0` can be evaluated faster than `first != last` (although it can be). It is because a loop that compares two iterators must perform the comparison during every iteration. But a loop that counts down to zero can be rewritten by a smart compiler to avoid the decrement and comparison during most iterations. The above lines 8−23 will be translated as if we had written the following function. The program is bigger but faster.

```
62 template <class RANDOM, class OUTPUT>
63 OUTPUT my_copy(RANDOM first, RANDOM last, OUTPUT result)
64 {
65     typedef typename iterator_traits<RANDOM>::difference_type
66         difference_type;
67
68     for (difference_type i = (last - first) / 8; i > 0; --i) {
69         *result = *first; ++first; ++result;
70         *result = *first; ++first; ++result;
71         *result = *first; ++first; ++result;
72          *result = *first; ++first; ++result;
73          *result = *first; ++first; ++result;
74          *result = *first; ++first; ++result;
75          *result = *first; ++first; ++result;
76          *result = *first; ++first; ++result;
77      }
78
79      switch (last - first) {
80      case 7: *result = *first; ++first; ++result;
81      case 6: *result = *first; ++first; ++result;
82      case 5: *result = *first; ++first; ++result;
83      case 4: *result = *first; ++first; ++result;
84      case 3: *result = *first; ++first; ++result;
85      case 2: *result = *first; ++first; ++result;
86      case 1: *result = *first;          ++result;
87      case 0: break;
88
89      default: cerr << "last - first == " << last - first << " in copy\n";
90          break;
91      }
92
93      return result;
94  }
```

```
random access iterators                          lines 52−54
10
20
30

input iterators that are not random access       lines 56−58
10
20
30
```

**Another way to dispatch copy:**
**Separate implementations for pointers to memmovable objects**

There is one case in which `my_copy` can be even faster. The `memcpy` and `memmove` functions from the C Standard Library make a literal copy of a block of memory, bit by bit. (This is called a *bitwise* copy.) They use specialized machine language instructions to squeeze the maximum speed out of the hardware.

The first two arguments are pointers; the third argument, a `size_t`, is the number of bytes to copy.

```
#include <cstring>   //for memcpy, memmove, size_t
using namespace std;

    const int source[] = {10, 20, 30};
    const size_t n = sizeof source / sizeof source[0];
    int dest[n];

    memcpy (dest, source, sizeof source);
    memmove(dest, source, sizeof source);
```

`memcpy` is faster, but we’ll use `memmove` because it works even when the source and destination overlap. We will temporarily remove the dispatching in the previous program.

Our my_copy can do its work by calling memmove whenever two conditions are met. First, the three arguments of my_copy must be pointers. memmove is a C function; it knows no other type of iterator.

Second, there are certain types of object that cannot be copied by memmove. The simplest example would be the following class introspect, whose only purpose is to demonstrate what can go wrong when an object is copied by memmove or memcpy. In this class, we require that the p data member of each object point to the object itself; that's why it's called introspect.

Paradoxically, memmove will disrupt this invariant precisely *because* it makes a bitwise copy of the object. The p data member of each copied object will be left with an exact copy of its original value—but this is the address of the original object, not the address of the copy. These objects can be copied, but only by their copy constructor or operator=, not by memmove or memcpy. At the end of each object's life, its destructor makes sure that it is still healthy (p. 164).

—On the Web at
http://i5.nyu.edu/~mm64/book/src/introspect/introspect.h

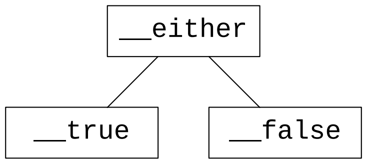
```
#ifndef INTROSPECTH
#define INTROSPECTH
#include <iostream>
using namespace std;

class introspect {
    const introspect *const p;
public:
    introspect(): p(this) {}
    introspect(const introspect&): p(this) {}
    introspect& operator=(const introspect&) {return *this;}

    friend ostream& operator<<(ostream& ost, const introspect& i) {
        return ost << "introspect at address " << &i
            << " contains " << i.p;
    }

    ~introspect() {
        if (p != this) {
            cerr << "Invariant disrupted: " << *this << "\n";
        }
    }
};
#endif
```

—On the Web at
http://i5.nyu.edu/~mm64/book/src/introspect/main.C

```
#include <iostream>
#include <cstdlib>
#include <cstring>   //for memmove
#include "introspect.h"
using namespace std;

int main()
{
    const size_t n = 3;
    const introspect source[n];
    introspect dest[n];

    memmove(dest, source, sizeof source);
    cout << "address of dest == " << dest << "\n";
    return EXIT_SUCCESS;
}
```

```
address of dest == 0xffbff18c
Invariant disrupted: introspect at address 0xffbff194 contains 0xffbff1a0
Invariant disrupted: introspect at address 0xffbff190 contains 0xffbff19c
Invariant disrupted: introspect at address 0xffbff18c contains 0xffbff198
```

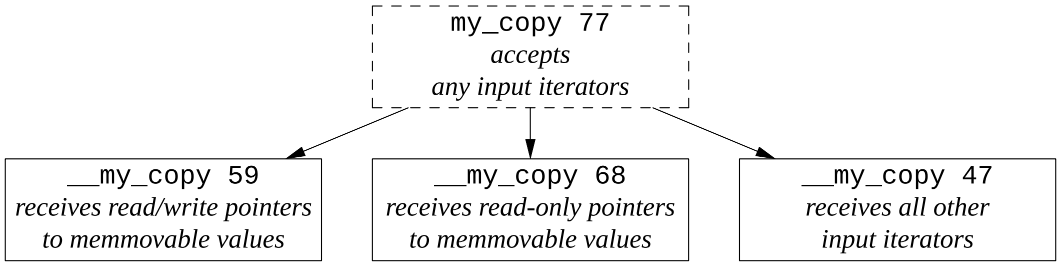
It’s up to us to tell the computer which types of objects can, and cannot, be copied by `memmove`. To do this, we first define the family of empty classes in lines 11−14. They will serve the same purpose as the family of empty iterator tag classes.



We then define the template class `__copy_traits` in lines 16−44. This is an altruistic class, like `numeric_limits` and `iterator_traits`, whose only purpose is to give us information about another data type `T`. It currently delivers only one fact. The member `is_memmovable` in line 24 is a `typedef` for `__true` if objects of type `T` can be copied by `memmove`; `__false` otherwise.

The general template in lines 22−25 errs on the side of safety: it assumes that no data type `T` can be copied by `memmove`. It is followed by a specialization for each data type that can be so copied. Any type of pointer can be; it is only a minor inconvenience that read/write and read-only pointers must be listed separately in lines 27−35. The built-in types can also be copied by `memmove`. To save paper, lines 37−41 defined specializations for only a few of them. Finally, line 43 indicates that class `date` can be copied by `memmove`.

Line 64 needs the parentheses in order to compile. Without them, we would be adding the pointers `result` and `last`.



—On the Web at
`http://i5.nyu.edu/~mm64/book/src/dispatch/copy2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cstring>   //for memmove
 4 #include <vector>
 5 #include <iterator>
 6 #include <algorithm>
 7 #include "introspect.h"
 8 #include "date.h"
 9 using namespace std;
```

```

//The is_memmovable member of class __copy_traits is a typedef for __true or __false.
struct __either {};
struct __true : public __either {};
struct __false: public __either {};

/*
struct __copy_traits takes a data type T and tells us if a variable of that type
can be copied with memmove.  If so, the is_memmovable member of __copy_traits<T>
will be a typedef for __true; otherwise it will be a typedef for __false.
*/

template <class T>
struct __copy_traits {
    typedef __false is_memmovable;
};

template <class T>
struct __copy_traits<T *> {         //partial specialization
    typedef __true is_memmovable;
};

template <class T>
struct __copy_traits<const T *> {   //same as above, but with const
    typedef __true is_memmovable;
};

template <> struct __copy_traits<bool       > {typedef __true is_memmovable;};
template <> struct __copy_traits<char       > {typedef __true is_memmovable;};
//etc.
template <> struct __copy_traits<double     > {typedef __true is_memmovable;};
template <> struct __copy_traits<long double> {typedef __true is_memmovable;};

template <> struct __copy_traits<date> {typedef __true is_memmovable;};
//Etc.: define a specialization for each type T that can be copied with memmove.

template <class INPUT, class OUTPUT>
OUTPUT __my_copy(INPUT first, INPUT last, OUTPUT result, __either)
{
    cout << "can’t use memmove\n";

    for (; first != last; ++first, ++result) {
        *result = *first;
    }

    return result;
}

template <class T>
T *__my_copy(T *first, T *last, T* result, __true)
{
    cout << "memmove with read/write source\n";

    memmove(result, first, (last - first) * sizeof (T));
    return result + (last - first);
}

template <class T>   //same as above, but with const’s
T *__my_copy(const T *first, const T *last, T* result, __true)
{
    cout << "memmove with read-only source\n";

    memmove(result, first, (last - first) * sizeof (T));
    return result + (last - first);
}

template <class INPUT, class OUTPUT>
inline OUTPUT my_copy(INPUT first, INPUT last, OUTPUT result)
{
    typedef typename iterator_traits<INPUT>::value_type value_type;
    typedef typename __copy_traits<value_type>::is_memmovable is_memmovable;

    return __my_copy(first, last, result, is_memmovable());
}

int main()
{
    const date source1[] = {
        date(date::july,       4, 1776),
        date(date::october,   29, 1929),
        date(date::december,   7, 1941)
    };
    const size_t n1 = sizeof source1 / sizeof source1[0];
    vector<date> v(source1, source1 + n1);
    my_copy(v.begin(), v.end(), ostream_iterator<date>(cout, "\n"));
    cout << "\n";

    const size_t n2 = 3;
    const introspect source2[n2];
    introspect dest2[n2];
    my_copy(source2, source2 + n2, dest2);
    copy(dest2, dest2 + n2, ostream_iterator<introspect>(cout, "\n"));
    cout << "\n";

    const size_t n3 = n1;
    date dest3[n3];
    my_copy(source1, source1 + n3, dest3);
    copy(dest3, dest3 + n3, ostream_iterator<date>(cout, "\n"));
    cout << "\n";

    date source4[] = {
        date(date::july,      20, 1969),
        date(date::september, 11, 2001),
        date()
    };
    const size_t n4 = sizeof source4 / sizeof source4[0];
    date dest4[n4];
    my_copy(source4, source4 + n4, dest4);
```

```
118     copy(dest4, dest4 + n4, ostream_iterator<date>(cout, "\n"));
119
120     return EXIT_SUCCESS;
121 }
```

The arguments in the above line 94 are not pointers, so we can’t call `memmove`. The ones in 100 are pointers, but they point to objects that cannot be copied by `memmove`. The conditions for `memmove` are fulfilled only in lines 106 and 117.

```
can’t use memmove                                          lines 87−95
7/4/1776
10/29/1929
12/7/1941

can’t use memmove                                          lines 97−102
introspect at address 0xffbff0b8 contains 0xffbff0b8
introspect at address 0xffbff0bc contains 0xffbff0bc
introspect at address 0xffbff0c0 contains 0xffbff0c0

memmove with read-only source                              lines 104−108
7/4/1776
10/29/1929
12/7/1941

memmove with read/write source                             lines 110−118
7/20/1969
9/11/2001
4/8/2014
```

**Combine the two above examples**

When the following line 117 calls the `my_copy` in line 103, the name `INPUT` in line 103 will stand for the data type `const int *`. The `value_type` in line 105 will be a `typedef` for `int`, and the `is_memmovable` in line 106 will be a typedef for `__true` thanks to lines 30−33. The expression `is_memmovable()` in line 108 therefore calls the constructor for class `__true`, passing it no arguments and constructing an anonymous object of this class. The anonymous object, and the three other arguments in line 108, are then passed to one of the functions named `__my_copy`.

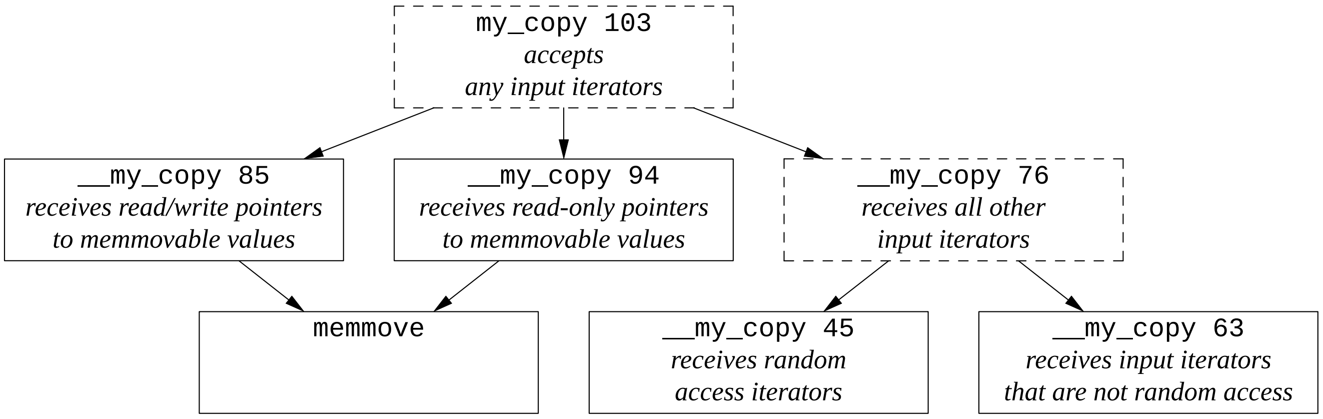
There are five functions with this name, in lines 45, 63, 76, 85, and 94. The one we just called from line 108 will be the one in line 94, because its fourth argument is `__true`. `my_copy` calls the `__my_copy` in line 94 only when the arguments are pointers to objects that can be copied by `memmove`.

On the other hand, when line 127 calls the `my_copy` in line 103, the name `INPUT` in line 103 will stand for the data type `list<int>::iterator`. Once again, the `value_type` in line 105 will be `int`, and the `is_memmovable` in line 106 will be `__true`. But this time, line 108 will call the `__my_copy` in line 76 rather than 85 or 94 because the first pair of arguments are not pointers.

The `iterator_category` in lines 78−79 will be `bidirectional_iterator_tag`, so line 81 will construct an anonymous object of this class. The line will then pass the anonymous object, and three other arguments, to the `__my_copy` in line 63, because its fourth argument is an `input_iterator_tag`. (Class `bidirectional_iterator_tag` is derived from class `input_iterator_tag`.) This `__my_copy` is the best we can do with a pair of `list<int>::iterator`’s, because they are input iterators that are not pointers or other random access iterators.

Finally when line 133 calls the `my_copy` in line 103, the name `INPUT` in line 103 will stand for the data type `const introspect *`. The `value_type` in line 105 will be `introspect`, and the `is_memmovable` in line 106 will be `__false`. The latter forces line 108 to call the `__my_copy` in line 76: the ones in lines 85 and 94 could be called only with a fourth argument of data type `true`.

This time, the `iterator_category` in lines 78−79 will be `random_access_iterator_tag`, so line 81 will construct an anonymous object of this class. The line will then pass the anonymous object, and three other arguments, to the `__my_copy` in line 45, because its fourth argument is a `random_access_iterator_tag`. This function takes advantage of our ability to subtract two `const introspect *`'s.



—On the Web at
`http://i5.nyu.edu/~mm64/book/src/dispatch/copy3.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cstring>   //for memmove
 4 #include <list>
 5 #include "introspect.h"
 6 #include "date.h"
 7 using namespace std;
 8
 9 //The is_memmovable member of class __copy_traits is a typedef for __true or __false.
10 struct __either {};
11 struct __true : public __either {};
12 struct __false: public __either {};
13
14 /*
15 struct __copy_traits takes a data type T and tells us if a variable of that type
16 can be copied with memmove.  If so, the is_memmovable member of __copy_traits<T>
17 will be a typedef for __true; otherwise it will be a typedef for __false.
18 */
19
20 template <class T>
21 struct __copy_traits {
22     typedef __false is_memmovable;
23 };
24
25 template <class T>
26 struct __copy_traits<T *> {
27     typedef __true is_memmovable;
28 };
29
30 template <class T>
31 struct __copy_traits<const T *> {   //same as above, but with const
```

```
    typedef __true is_memmovable;
};

template <> struct __copy_traits<char       > {typedef __true is_memmovable;};
template <> struct __copy_traits<int        > {typedef __true is_memmovable;};
//etc.
template <> struct __copy_traits<double     > {typedef __true is_memmovable;};
template <> struct __copy_traits<long double> {typedef __true is_memmovable;};

template <> struct __copy_traits<date> {typedef __true is_memmovable;};
//Etc.: define a specialization for each type T that can be copied with memmove.

template <class RANDOM, class OUTPUT>
OUTPUT __my_copy(RANDOM first, RANDOM last, OUTPUT result,
    random_access_iterator_tag)
{
    cout << "random access iterators\n";

    typedef typename
        iterator_traits<RANDOM>::difference_type difference_type;

    for (difference_type n = last - first; n > 0; --n) {
        *result = *first;   //(*result).operator=(*first);
        ++first;
        ++result;
    }

    return result;
}

template <class INPUT, class OUTPUT>
OUTPUT __my_copy(INPUT first, INPUT last, OUTPUT result,
    input_iterator_tag)
{
    cout << "input iterators that are not random access\n";

    for (; first != last; ++first, ++result) {
        *result = *first;   //(*result).operator=(*first);
    }

    return result;
}

template <class INPUT, class OUTPUT>
inline OUTPUT __my_copy(INPUT first, INPUT last, OUTPUT result, __either)
{
    typedef typename iterator_traits<INPUT>::iterator_category
        iterator_category;

    return __my_copy(first, last, result, iterator_category());
}

template <class T>
T *__my_copy(T *first, T *last, T* result, __true)
```

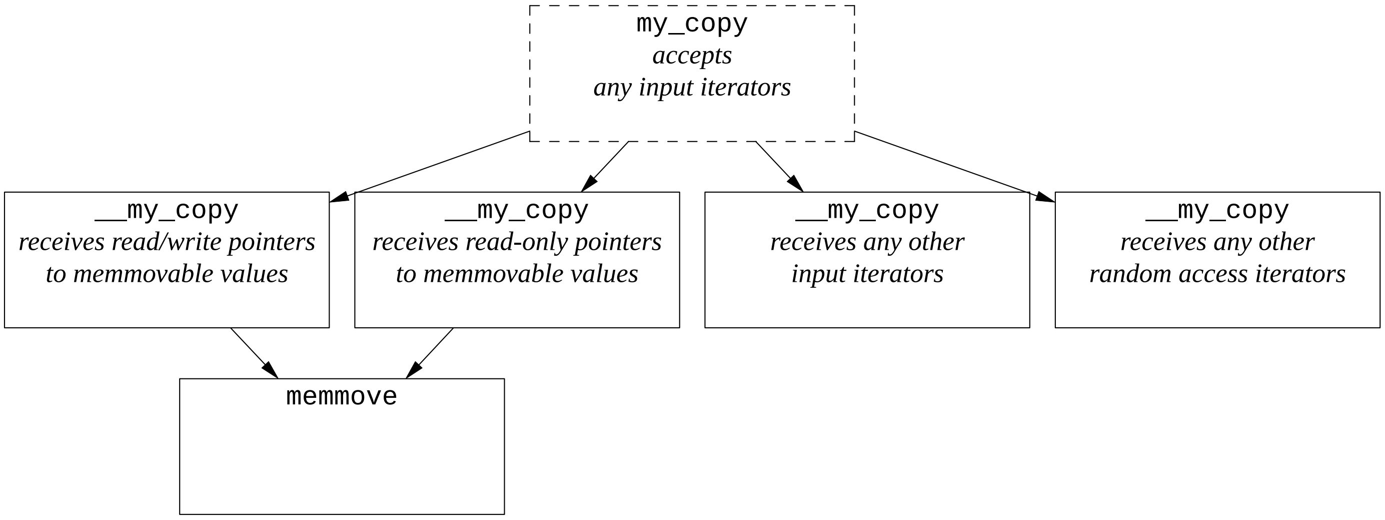
```
 86 {
 87     cout << "memmove with read/write source\n";
 88 
 89     memmove(result, first, (last - first) * sizeof (T));
 90     return result + (last - first);
 91 }
 92 
 93 template <class T>   //same as above, but with const's
 94 T *__my_copy(const T *first, const T *last, T* result, __true)
 95 {
 96     cout << "memmove with read-only source\n";
 97 
 98     memmove(result, first, (last - first) * sizeof (T));
 99     return result + (last - first);
100 }
101 
102 template <class INPUT, class OUTPUT>
103 inline OUTPUT my_copy(INPUT first, INPUT last, OUTPUT result)
104 {
105     typedef typename iterator_traits<INPUT>::value_type value_type;
106     typedef typename __copy_traits<value_type>::is_memmovable is_memmovable;
107 
108     return __my_copy(first, last, result, is_memmovable());
109 }
110 
111 int main()
112 {
113     //Can be copied with memmove.
114     const int source1[] = {10, 20, 30};
115     const size_t n = sizeof source1 / sizeof source1[0];
116     int dest1[n];
117     my_copy(source1, source1 + n, dest1);  //Line 108 calls 94.
118 
119     //Can be copied with memmove.
120     date source2[n];
121     date dest2[n];
122     my_copy(source2, source2 + n, dest2);  //Line 108 calls 85.
123 
124     //Can't be copied with memmove:
125     //the int's are memovable, but list iterators are not pointers.
126     const list<int> li(source1, source1 + n);
127     my_copy(li.begin(), li.end(), dest1);  //Line 108 calls 76; 81 calls 63.
128 
129     //Can't be copied with memmove:
130     //the iterators are pointers, but introspect's are not memmovable.
131     const introspect source4[n];
132     introspect dest4[n];
133     my_copy(source4, source4 + n, dest4);  //Line 108 calls 76; 81 calls 45.
134 
135     //Any type of pointer can be copied with memmove.
136     const introspect *const source5[] = {source4, source4 + 1, source4 + 2};
137     const introspect *dest5[n];
138     my_copy(source5, source5 + n, dest5);  //Line 108 calls 94.
139 
140     return EXIT_SUCCESS;
141 }
```

| | |
|---|---|
| `memmove with read-only source` | *lines 113−117* |
| `memmove with read/write source` | *lines 119−122* |
| `input iterators that are not random access` | *lines 124−127* |
| `random access iterators` | *lines 129−133* |
| `memmove with read-only source` | *lines 135−138* |

▲

**▼ Homework 8.5.2a: can we do it with only one dispatching function?**

Does the above program `copy3.C` really need two dispatching functions, in lines 102 and 75? Can we do it all with a single dispatching function?



```
 1 template <class INPUT, class OUTPUT>
 2 inline OUTPUT my_copy(INPUT first, INPUT last, OUTPUT result)
 3 {
 4     typedef typename iterator_traits<INPUT>::iterator_category
 5         iterator_category;
 6
 7     typedef typename iterator_traits<INPUT>::value_type value_type;
 8
 9     typedef typename __copy_traits<value_type>::is_memmovable is_memmovable;
10
11     return __my_copy(first, last, result,
12         iterator_category(), //one of the iterator tag classes
13         is_memmovable());    //__true or __false
14 }
```

Hint. Which `__my_copy` would be called by the “intersection of sets” algorithm on pp. 93−94 if the iterators were `vector<int>::iterator`’s?

The reality is even more complicated. On my platform, the elements of a vector are stored consecutively in memory, and a vector iterator is an object whose only data member is a pointer to an element. An vector should therefore be copied by `memmove`. But the iterators in line 19 are objects, not pointers, so `my_copy` doesn’t recognize the opportunity to call `memmove`.

```
15     int source[] = {10, 20, 30};
```

```
16      const size_t n = sizeof source / sizeof source[0];
17      vector<int> v(a, a + n);
18      int dest[n];
19      my_copy(v.begin(), v.end(), dest);
```

How could we make `my_copy` smart enough so that the above line 19 will call `memmove`? Note that the existing `my_copy` will call `memmove` if we change line 19 to

```
20      my_copy(&*v.begin(), &*v.end(), dest);
```

but we don't want to do that.

▲

**▼ Homework 8.5.2b: dispatch the find algorithm**

Dispatch the `find` algorithm. If the first and second arguments are pointers (read-only or read/write) to `char`, `unsigned char`, or `signed char`, have the `find` algorithm call the C Standard Library function `memchr`. If the arguments are another type of random access iterator, have `find` count a `difference_type` down to zero. Otherwise, compare two iterators during each iteration.

It will be simpler than `copy` because you won't have to worry about `__copy_traits`.

▲

**▼ Homework 8.5.2c: dispatch the find_distance algorithm**

The `find_distance` algorithm we wrote on p. 837 will accept any input iterators. It then increments a counter of type

```
typename iterator_traits<IT>::difference_type
```

during each iteration of a loop.

If the iterators are input iterators that are not random access, keep the existing code. But if the iterators are random access, we can get rid of the counter and do the job faster. After finding the desired element, we can find the answer by a single subtraction of two iterators. In fact, we can do the whole job simply by calling the `find` algorithm and the `distance` algorithm.

▲

## 8.6 Standard Template Library Summary

We have studied some of the STL components in depth: the containers `vector`, `list`, and `map`; the iterators for input streams, output streams, and the inserters; and the algorithms `sort`, `copy`, `find`, `find_if`, and `min_element`. The remaining components can be sketched in outline because the design of the STL is so consistent. Full documentation is online at

```
http://www.sgi.com/tech/stl/
```

This summary covers a few components that are not officially part of the STL, but provided by many vendors. The STL is the brainchild of Alexander Stepanov.

### 8.6.1 STL Containers

The following containers are not officially part of the STL.

```
slist
hash_set
hash_multiset
hash_map
hash_multimap
```

Class `string` does not belong to the STL, but we list it here because it is STL-compliant. Classes `bitset` and `valarray` are present because they have an `operator[]`.

| *name of class* | *header file* | *re-ordered* | *iterator category* | *has* `[]` | *other member functions for accessing elements* |
|---|---|---|---|---|---|
| `vector` | `<vector>` | *no* | *random* | *yes* | `at`, `front`, `back` |
| `string` | `<string>` | *no* | *random* | *yes* | `at`, `find`, `substr` |
| `deque` | `<deque>` | *no* | *random* | *yes* | *see below* |
| `list` | `<list>` | *no* | *bidirectional* | *no* | *see below* |
| `slist` | `<slist>` | *no* | *forward* | *no* | `front`, `previous` |
| `bitset` | `<bitset>` | *no* | *none* | *yes* | `set`, `reset`, `test` |
| `valarray` | `<valarray>` | *no* | *none* | *yes* | `[]` *takes* `slice`*'s* |
| `map` | `<map>` | *sorted* | *bidirectional* | *yes* | `find` |
| `multimap` | `<map>` | *sorted* | *bidirectional* | *no* | `equal_range` |
| `set` | `<set>` | *sorted* | *bidirectional* | *no* | `lower_bound` |
| `multiset` | `<set>` | *sorted* | *bidirectional* | *no* | `upper_bound` |
| `hash_map` | `<hash_map>` | *hashed* | *forward* | *yes* | |
| `hash_multimap` | `<hash_map>` | *hashed* | *forward* | *no* | `find` |
| `hash_set` | `<hash_set>` | *hashed* | *forward* | *no* | `equal_range` |
| `hash_multiset` | `<hash_set>` | *hashed* | *forward* | *no* | |
| `stack` | `<stack>` | *no* | *none* | *no* | `push`, `top`, `pop` |
| `queue` | `<queue>` | *no* | *none* | *no* | `push`, `front`, `pop` |
| `priority _queue` | `<priority _queue>` | *heap* | *none* | *no* | `push`, `top`, `pop` |

**Topology**

Every STL container is one-dimensional. The only hint of a non-linear data structure in the library is the binary tree in the ‘‘heap’’ algorithms on pp. 961−962. In Chapter 9, we will impose a two-dimensional shape on a one-dimensional container.

A container with input iterators can have an end; in fact, it can even be empty. A container whose iterators are merely output iterators is endless.

**Sequences vs. associative containers**

Classes `vector`, `string`, `deque`, `list`, and `slist` are called *sequences*. Think of each of their elements as having a non-negative integer subscript. Class `list` is doubly-linked, class `slist` is singly-linked. Even the humble array is a sequence, albeit one with no member functions or member types.

A `deque` is a double-ended queue. Classes `deque` and `list` have the following member functions.

| | *get element* | *push element* | *pop element* |
|---|---|---|---|
| *front* | `front` | `push_front` | `pop_front` |
| *back* | `back` | `push_back` | `pop_back` |

Classes `map`, `set`, and their `multi-` and `hash_` variants are called *associative containers*. Think of each element as having a subscript (the *key*) that can be any data type. The elements of the various `map` containers are `pair` objects, whose first data member is the subscript. The elements of the various `set` containers are just the subscripts themselves.

We look up a subscript by passing it to a member function of the container. The member function `find` returns an iterator referring to the element with the desired subscript. The `multi-` containers might have more than one element with a given subscript. `equal_range` constructs and returns a `pair` of iterators delimiting the range of elements with the desired subscript, or the empty range where they would have been had they existed. `lower_bound` and `upper_bound` return the first or second iterator in this pair, when we need to know only where the range begins or ends.

### The data type of the elements of a container

The elements of a given container must all be of the same data type. The data type must be assignable. This disqualifies stream objects (pp. 324−326), `type_info`'s (p. 1017), `facet`'s (p. 1036), and, for that matter, `rabbit`'s (pp. 200 and 311−312). A `pair` is assignable if its two data members are.

A reference is not assignable, but a pointer is. Containers usually hold pointers to objects, not the objects themselves, so we can insert and access the objects without making unwanted copies of them (pp. 440−441). Often a container holds pointers to objects of a family of data types, allowing us to call their virtual functions (pp. 487−489). But even in this case, the pointers themselves are all of the same data type. They are pointers to the common base class of all the objects.

### Sorted containers

The elements of a *sorted* container are always stored in increasing order of their subscripts. If inserted in the wrong order, they are automatically rearranged. We saw the planets rearranged to alphabetical order on p. 788.

Increasing order is defined negatively: we never have a later subscript that is less than an earlier one. Not surprisingly, ''less than'' means `<` by default. The `<` operator is applied to the subscript of each element in the container.

Two subscripts are said to be *equivalent* if neither one is less than the other. A `set` or `map` will not accept two or more elements with equivalent subscripts. Their `multi`- variants will.

In each sorted container, the choice of `<` for ''less than'' can be overridden (line 5). Beware: the `greater<int>` in the <angle brackets> of the `map` is the name of a data type; the `greater<int>()` in the `(`parentheses`)` of the `sort` in line 14 on p. 936 is an anonymous object.

```
 1 #include <map>
 2 #include <functional>   //for greater
 3 using namespace std;
 4 
 5     map<int, double> m1;                  //sorted in order of <
 6     map<int, double, greater<int> > m2;   //sorted in order of >
```

A `priority_queue` is implemented with the heap algorithms on pp. 961−962. Its elements are ordered in a heap, not a sequence; the biggest one is always at the `top`.

### Hashed containers

The elements of a *hashed* container are stored in a hash table. The hashing function accepts an element of the hashed container and returns a `size_t`. By default, the hashing function is the `operator()` member function of class `hash<T>`, where `T` is the data type of the subscript.

A `hash_set` or `hash_map` will not accept two or more elements with equal subscripts. Their `multi`- variants will. Not surprisingly, ''equal'' means `==` by default.

In each hashed container, the choice of hashing function and the choice of `==` for ''equals'' can be overridden. `myhash<int>` must be a class whose `operator()` takes an `int` and returns a `size_t`; `myequality<int>` must be a class whose `operator()` takes two `int`'s and returns a `bool`.

```
 7 #include <hash_map>
 8 using namespace std;
 9 
10     hash_map<int, double> m1;
11     hash_map<int, double, myhash<int>, myequality<int> > m2;
```

**Container adaptors**

Classes `stack`, `queue`, and `priority_queue` are merely *container adaptors*, interfaces that allow access to part of the functionality of an underlying container. We will build one ourselves on pp. 986−988.

By default, `stack` and `queue` are adaptors for class `deque`; `priority_queue` is an adaptor for class `vector`. In each container, the choice of underlying container can be overridden (line 16).

```
12 #include <stack>
13 #include <list>
14 using namespace std;
15
16     stack<int> s1;                  //adaptor for deque<int>
17     stack<int, list<int> > s2;      //adaptor for list<int>
```

**Relatives of containers**

We have listed classes `string` and `bitset` here, even though they do not belong to the STL. Class `string` could hold any type of values (see the typedef on p. 688), but it is fastest for characters. Class `bitset` has an `operator[]` member function, but no iterators or other container features.

Classes `pair` and `complex` contain two data members each. They have no iterators.

## 8.6.2 STL Function Objects

By definition, each function object has a public, non-static `operator()` member function, preferably inline. When we say that a function object takes certain arguments and returns a value, we mean that its `operator()` function does these things.

A *unary function object* takes one argument `x`, does something with it, and returns the result. These classes are derived from the template class `unary_function`, from which they inherit the two typedef members `argument_type` and `result_type`.

A *binary function object* takes two arguments `x1` and `x2`, does something with them, and returns the result. These classes are derived from the template class `binary_function`, from which they inherit the three typedef members `first_argument_type`, `second_argument_type`, and `result_type`.

A *generator* takes no arguments and returns a result. These classes are derived from the template class `generator`, from which they inherit the typedef member `result_type`. This template class is not in the library, but we wrote it ourselves on p. 882.

In addition to providing the typedefs, the names of the base classes `generator`, `unary_function`, and `binary_function` act as documentation.

Our first example of a function object was class `greater` (p. 769). In the following line 9, the first argument of its `operator()` is of type `const T&`. But the first template argument of its base class `binary_function` in line 7 is an unadorned `T`, and the template argument in line 14 is an unadorned `int`. The intent of the template argument is to show the data type of a variable that can be passed to the `operator()`, not the mechanism by which the variable is passed.

Function objects are intended only to be passed as arguments to an algorithm. They will probably be anonymous temporaries. Line 14 is an example.

```
 1 #include <vector>
 2 #include <functional>  //for greater
 3 #include <algorithm>
 4 using namespace std;
 5
 6 template <class T>
 7 struct greater: public binary_function<T, T, bool>
```

```
{
    bool operator()(const T& x1, const T& x2) const {return x1 > x2;}
};

    vector<int> v(argument(s) for constructor);
    sort(v.begin(), v.end());                //increasing order (<) by default
    sort(v.begin(), v.end(), greater<int>()); //decreasing order (>)
```

For each class of function object that contains data members, there is a *helper function* that constructs and returns an object of that class. The classes and helper functions in §8.6.2 are declared in the header file `<functional>`.

### 8.6.2.1 Function objects containing no data members

The following function objects are identical to the above `greater`, except for the operation performed. We saw `greater` on p. 769, `multiplies` on 810.

| *name of class* | *argument(s) of* `operator()` | *return value of* `operator()` |
|---|---|---|
| `equal_to` | `x1, x2` | `x1 == x2` |
| `not_equal_to` | `x1, x2` | `x1 != x2` |
| `less` | `x1, x2` | `x1 < x2` |
| `greater` | `x1, x2` | `x1 > x2` |
| `less_equal` | `x1, x2` | `x1 <= x2` |
| `greater_equal` | `x1, x2` | `x1 >= x2` |
| `plus` | `x1, x2` | `x1 + x2` |
| `minus` | `x1, x2` | `x1 - x2` |
| `multiplies` | `x1, x2` | `x1 * x2` |
| `divides` | `x1, x2` | `x1 / x2` |
| `modulus` | `x1, x2` | `x1 % x2` |
| `negate` | `x` | `-x` |
| `logical_and` | `x1, x2` | `x1 && x2` |
| `logical_or` | `x1, x2` | `x1 \|\| x2` |
| `logical_not` | `x` | `!x` |

The following group of function objects is not officially part of the STL. The ‘‘operation’’ they perform is merely to return an argument, or merely a data member of an argument. The argument of each `select` function is a `pair` object, whose data members are named `first` and `second`.

| *name of class* | *argument(s) of* `operator()` | *return value of* `operator()` |
|---|---|---|
| `identity` | `x` | `x` |
| `project1st` | `x1, x2` | `x1` |
| `project2nd` | `x1, x2` | `x2` |
| `select1st` | `x` | `x.first` |
| `select2nd` | `x` | `x.second` |

Here’s an example of `identity`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/identity.C`

```
#include <iostream>
#include <cstdlib>
#include <vector>
```

```
#include <functional>      //for not_equal_to and bind2nd
#include <ext/functional>  //for identity
#include <algorithm>
using namespace std;

int main()
{
    int a[] = {0, 0, 10, 20, 30};
    const size_t n = sizeof a / sizeof a[0];
    vector<int> v(a, a + n);

    vector<int>::iterator it =
        //Pedantic way to find the first non-zero element.
        //find_if(v.begin(), v.end(), bind2nd(not_equal_to<int>(), 0));

        //Simpler way to find the first non-zero element.
        find_if(v.begin(), v.end(), __gnu_cxx::identity<int>());

    cout << "Subscript of first non-zero element is "
        << distance(v.begin(), it) << ".\n";

    return EXIT_SUCCESS;
}
```

```
Subscript of first non-zero element is 2.
```

Strangely, the projection objects return by value, while the others return by reference. If they have any use, it lies with the `transform` algorithm that takes two input containers.

The selection objects are useful for a container of `pair`'s, such as the `map` in line 21.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/select1st.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <string>
 4 #include <map>
 5 #include <iterator>       //for ostream_iterator
 6 #include <ext/functional> //for select1st and select2nd
 7 #include <algorithm>      //for transform
 8 using namespace std;
 9
10 int main()
11 {
12     typedef map<string, double> map_t;
13     typedef map_t::value_type pair_t;
14
15     const pair_t a[] = {
16         pair_t("Mercury",  .27),
17         pair_t("Venus",    .85),
18         pair_t("Earth",   1.00)
19     };
20     const size_t n = sizeof a / sizeof a[0];
21     map_t m(a, a + n);
22
23     //Output the subscripts.  map_t::key_type is a typedef for string.
24     transform(
25         m.begin(), m.end(),
26         ostream_iterator<map_t::key_type>(cout, " "),
27         __gnu_cxx::select1st<pair_t>()
28     );
29     cout << "\n";
30
31     //Output the values.  map_t::mapped_type is a typedef for double.
32     transform(
33         m.begin(), m.end(),
34         ostream_iterator<map_t::mapped_type>(cout, " "),
35         __gnu_cxx::select2nd<pair_t>()
36     );
37     cout << "\n";
38
39     return EXIT_SUCCESS;
40 }
```

```
Earth Mercury Venus    lines 23−29: subscripts (in alphabetical order)
1 0.27 0.85            lines 31−37: the corresponding values
```

### 8.6.2.2 Function objects containing a pointer to a free function

A `pointer_to_unary_function<ARG, RETURN>` is a function object that contains a pointer to a free (i.e., non-member) function `p` whose argument and return value are of types `ARG` and `RETURN`. The function object takes one argument `x` and returns `(*p)(x)`.

A `pointer_to_binary_function<ARG1, ARG2, RETURN>` is a function object that contains a pointer to a function `p` whose arguments and return value are of types `ARG1`, `ARG2`, and `RETURN`. The function object takes two arguments, `x1` and `x2`, and returns `(*p)(x1, x2)`. See pp. 944−945 for a specialization you might have to write.

Instead of mentioning the template arguments in the <angle brackets>, it's simpler to call the template function `ptr_fun`. It will construct and return a `pointer_to_unary_function` or `pointer_to_binary_function` of the correct type.

| *name of class* | *data member* | *argument(s) of* `operator()` | *return value of* `operator()` | *helper function* |
|---|---|---|---|---|
| `pointer_to_unary_function` | `p` | `x` | `(*p)(x)` | `ptr_fun` |
| `pointer_to_binary_function` | `p` | `x1, x2` | `(*p)(x1, x2)` | `ptr_fun` |

A pointer to a function can be passed directly to an algorithm. Line 24 passes `abs` directly to `transform`.

A pointer to a function can be inserted into a function object, such as the one constructed by the `compose1` in line 32, only if the pointer to a function is first encased in a function object constructed by `ptr_fun`. Lines 33−34 do this to two of them: `sqrt` and `abs`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/ptr_fun.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cmath>          //for the double sqrt and abs
 4 #include <iterator>       //for ostream_iterator
 5 #include <functional>     //for ptr_fun
 6 #include <ext/functional> //for compose1
 7 #include <algorithm>
 8 using namespace std;
 9
10 int main()
11 {
12     double a[] = {9, -4, 2};
13     const size_t n = sizeof a / sizeof a[0];
14     ostream_iterator<double> it(cout, " ");
15
16     //Copy each number.
17     copy(a, a + n, it);
18     cout << "\n";
19
20     //Output the absolute value of each number.
21     transform(
22         a, a + n,
23         it,
24         static_cast<double (*)(double)>(abs)
25     );
26     cout << "\n";
27
28     //Output the square root of the absolute value of each number.
29     transform(
30         a, a + n,
31         it,
32         __gnu_cxx::compose1(
33             ptr_fun(static_cast<double (*)(double)>(sqrt)),
34             ptr_fun(static_cast<double (*)(double)>(abs ))
35         )
36     );
37     cout << "\n";
38
39     return EXIT_SUCCESS;
40 }
```

```
9 -4 2             lines 16−18
9 4 2              lines 20−26
3 2 1.41421        lines 28−37
```

The above lines 24 and 34 need the cast because the library has more than one function named `abs`; we want the one that takes and returns a `double`. Instead of repeating this every time we mention `abs`, line 15 could declare a pointer `abs` to the one we want.

```
41     double (*const abs)(double) = std::abs;
```

(Without the `std::`, our pointer would be initialized to itself.) Then in lines 24 and 34, we could change the

```
static_cast<double (*)(double)>(abs)
```

to `abs`.

The standard library has no `pointer_to_function`'s that take more than no arguments or more than two arguments. The `rand` function, for example, takes no arguments. To fill a container with the square roots of random numbers, you will have to write your own `pointer_to_zeroary_function` class, and another `ptr_fun` to construct a `pointer_to_zeroary_function`.

```
    transform(
        a, a + n,
        it,
        __gnu_cxx::compose1(
            ptr_fun(static_cast<double (*)(double)>(sqrt)),
            ptr_fun(rand)
        )
    );
```

See pp. 944−945 for another `pointer_to_function` class you might have to write.

#### 8.6.2.3 Function objects containing a pointer to a member function

A `mem_fun_ref_t<RETURN, OBJECT>` is a function object that contains a pointer `p` to a member function of class `OBJECT`. The member function taks no arguments and returns a `RETURN`. The function object takes one argument `obj` (an object of class `OBJECT` passed as a read/write reference, hence the name `ref_`) and returns `(obj.*p)()`.

A `mem_fun_t<RETURN, OBJECT>` is a function object that contains a pointer `p` to a member function of class `OBJECT`. The member function taks no arguments and returns a `RETURN`. The function object takes one argument `q` (a read/write pointer to an object of class `OBJECT`) and returns `(q->*p)()`.

A `mem_fun1_ref_t<RETURN, OBJECT, ARG>` is a function object that contains a pointer `p` to a member function of class `OBJECT`. The argument and the return value of the member function are of types `ARG` and `RETURN`. The function object takes two arguments, `obj` (an object of class `OBJECT` passed as a read/write reference) and `x` (of class `ARG`), and returns `(obj.*p)(x)`.

A `mem_fun1_t<RETURN, OBJECT, ARG>` is a function object that contains a pointer `p` to a member function of class `OBJECT`. The argument and the return value of the member function are of types `ARG` and `RETURN`. The function object takes two arguments, `q` (a read/write pointer to an object of class `OBJECT`) and `x` (of class `ARG`), and returns `(q->*p)(x)`.

There are also `const_` variants, in which the member function is a `const` member function. In this case, the object of class `OBJECT` is passed to the function object as a read-only reference or read-only pointer.

| *name of class* | *data member* | *argument of* `operator()` | *return value of* `operator()` | *helper function* |
|---|---|---|---|---|
| `mem_fun_ref_t` `const_mem_fun_ref_t` | `p` | `obj` | `(obj.*p)()` | `mem_fun_ref` |
| `mem_fun_t` `const_mem_fun_t` | `p` | `q` | `(q->*p)()` | `mem_fun` |

| *name of class* | *data member* | *arguments of* `operator()` | *return value of* `operator()` | *helper function* |
|---|---|---|---|---|
| `mem_fun1_ref_t` `const_mem_fun1_ref_t` | `p` | `obj, x` | `(obj.*p)(x)` | `mem_fun_ref` |
| `mem_fun1_t` `const_mem_fun1_t` | `p` | `q, x` | `(q->*p)(x)` | `mem_fun` |

Here are examples of `mem_fun_ref` and `mem_fun`.

(1) The `clear` in line 28 is a non-`const` member function of class `string`, taking no arguments and returning `void`. The `mem_fun_ref` in that line therefore constructs and returns an anonymous object of type

```
mem_fun_ref_t<void, string>
```

and the `mem_fun` in line 29 constructs and returns an anonymous object of type

```
mem_fun_t<void, string>
```

(2) The `size` in line 33 is a `const` member function of class `string`, taking no arguments and returning `string::size_type`. The `mem_fun_ref` in that line constructs and returns an anonymous object of type

```
const_mem_fun_ref_t<string::size_type, string>
```

and the `mem_fun` in line 34 constructs and returns an anonymous object of type

```
const_mem_fun_t<string::size_type, string>
```

The `at` in line 22 is a `const` member function of class `string`, because we're storing it's address into a pointer to that type in line 21. (There is another `at` function that is a non-`const` member function of class `string`, so we specify once and for all which one we want.) It takes an argument of type `string::size_type` and returns a `string::value_type`, which is just a hypercorrect way of saying `char`. The `mem_fun_ref` in line 24 constructs and returns an anonymous object of type

```
const_mem_fun1_ref_t<string::value_type, string, string::size_type>
```

and the `mem_fun` in line 25 constructs and returns an anonymous object of type

```
const_mem_fun1_t<string::size_type, string, string::size_type>
```

Lines 28−29 call the `for_each` algorithm because `string::clear` has no argument or return value. Lines 33−34 call the `transform` with one source range because `string::size` has a return value but no argument. The stream iterator in line 32 prints the `stream::size_type` returned by the calls to `size`. Lines 24−25 call the `transform` with two source ranges because `string::at` has an argument and a return value. The stream iterator in line 23 prints the `stream::size_type` returned by the calls to `size`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/mem_fun.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <string>
 4 #include <iterator>
 5 #include <functional>
 6 #include <algorithm>
 7 using namespace std;
 8
 9 int main()
10 {
11     const size_t n = 3;
12     string a[n] = {"abe", "ike", "jake"}; //container of objects
13     string *b[n] = {                      //container of pointers to objects
14         new string("abe"),
15         new string("ike"),
16         new string("jake")
17     };
18     string::size_type c[n] = {1, 2, 3};   //container of subscripts
19
20     //Call the at member function of each object, print the return values.
21     string::const_reference (string::*const at)(string::size_type) const =
22         &string::at;
23     ostream_iterator<string::value_type> it1(cout, " ");
24     transform(a, a + n, c, it1, mem_fun_ref(at)); cout << "\n";
25     transform(b, b + n, c, it1, mem_fun    (at)); cout << "\n";
26
27     //Call the clear member function of each object.  It returns no value.
28     for_each(a, a + n, mem_fun_ref(&string::clear));
29     for_each(b, b + n, mem_fun    (&string::clear));
30
31     //Call the size member function of each object, print the return values.
32     ostream_iterator<string::size_type> it2(cout, " ");
33     transform(a, a + n, it2, mem_fun_ref(&string::size)); cout << "\n";
34     transform(b, b + n, it2, mem_fun    (&string::size)); cout << "\n";
35
36     for (size_t i = 0; i < n; ++i) {
37         delete b[i];
38     }
39     return EXIT_SUCCESS;
40 }
```

```
b e e        line 24
b e e        line 25
0 0 0        line 33
0 0 0        line 34
```

### 8.6.2.4 Function objects containing one, two, or three other function objects

`compose1` and `compose2` are not officially part of the STL. The other functions in this group are ones we wrote ourselves. A `unary_compose<F, G>` contains two unary function objects, `f` and `g`, of types `F` and `G`. The `unary_compose` takes one argument `x` of type `G::argument_type` and returns `f(g(x))`.

A `binary_compose<F, G1, G2>` contains one binary function object `f`, and two unary function objects, `g1` and `g2`, of types `F`, `G1`, and `G2`. The `binary_compose` takes one argument `x` and returns `f(g1(x), g2(x))`. The argument `x` must be convertible to `G1::argument_type` and `G2::argument_type`.

A `composer_fg<F, G>` (p. 882) contains one unary function object `f` and generator object `g`, of types `F` and `G`. The `composer_fg` takes no arguments and returns `f(g())`.

A `composer_fgx1_x2<F, G>` (pp. 894−895) contains one unary function object `f` and binary unary function object `g`, of types `F` and `G`. The `composer_fgx1_x2` takes two arguments `x1` and `x2`, of type `G::first_argument_type` and `G::second_argument_type`, and returns `f(g(x1, x2))`.

A `composer_fgx1_gx2<F, G>` (p. 909) contains one binary function object `f` and one unary function object `g`, of types `F` and `G`. The `composer_fgx1_gx2` takes two arguments `x1` and `x2`, of type `G::argument_type`, and returns `f(g(x1), g(x2))`.

A `unary_negate<G>` contains a unary function object `g` of type `G`. The `unary_negate` takes one argument `x` of type `G::argument_type` and returns `!g(x)`. Class `unary_negate` is just a shorthand. If it did not exist, we could construct an object that does the same thing as `not1(g)` by saying

```
compose1(logical_not<G::result_type>(), g)
```

A `binary_negate<G>` contains a binary function object `g` of type `G`. The `binary_negate` takes two arguments, `x1` and `x2`, of types `G::first_argument_type` and `G::second_argument_type`, and returns `!g(x1, x2)`. Class `binary_negate` is just a shorthand. If it did not exist, we could construct an object that does the same thing as `not2(g)` by saying

```
compose_fgx1_x2(logical_not<G::result_type>(), g)
```

| *name of class* | *data member(s)* | *argument(s) of* `operator()` | *return value of* `operator()` | *helper function* |
|---|---|---|---|---|
| `composer_fg` | `f, g` | *none* | `f(g())` | `compose_fg` |
| `unary_compose` | `f, g` | `x` | `f(g(x))` | `compose1` |
| `composer_fgx1_x2` | `f, g` | `x1, x2` | `f(g(x1, x2))` | `compose_fgx1_x2` |
| `composer_fgx1_gx2` | `f, g` | `x1, x2` | `f(g(x1), g(x2))` | `compose_fgx1_gx2` |
| `binary_compose` | `f, g1, g2` | `x` | `f(g1(x), g2(x))` | `compose2` |
| `unary_negate` | `g` | `x` | `!g(x)` | `not1` |
| `binary_negate` | `g` | `x1, x2` | `!g(x1, x2)` | `not2` |

To construct an anonymous function object whose `operator()` takes an argument `x` and returns `f(g(h(x)))`, say either of the following. They do the same thing; a mathematician would say that "function composition is associative."

```
    compose1(f, compose1(g, h))
    compose1(compose1(f, g), h)
```

In fact, you could easily define the following helper function.

```
#include <ext/functional>
using namespace std;

template <class F, class G, class H>
inline __gnu_cxx::unary_compose<F, __gnu_cxx::unary_compose<G, H> >
compose_fghx(const F& f, const G& g, const H& h)
{
    return __gnu_cxx::compose1(f, __gnu_cxx::compose1(g, h));
}
```

and then say

```
    compose_fghx(f, g, h)
```

### 8.6.2.5 Function objects containing a function object and an argument for it

A `binder1st<F>` contains a binary function object `f` of type `F` and a value `x1` of type `F::first_argument_type`. The `binder1st` takes an argument `x2` of type `F::second_argument_type` and returns `f(x1, x2)`.

A `binder2nd<F>` contains a binary function object `f` of type `F` and a value `x2` of type `F::second_argument_type`. The `binder2nd` takes an argument `x1` of type `F::first_argument_type` and returns `f(x1, x2)`.

Binders appeared on pp. 861−864.

| *name of class* | *data members* | *argument of* `operator()` | *return value of* `operator()` | *helper function* |
|---|---|---|---|---|
| `binder1st` | `f, x1` | `x2` | `f(x1, x2)` | `bind1st` |
| `binder2nd` | `f, x2` | `x1` | `f(x1, x2)` | `bind2nd` |

### ▼ Homework 8.6.2.5a: a pointer_to_function specialization

The following code adds 10 to each integer in the vector. The calls to `bind2nd` are successful in lines 14 and 17. But why won't the one in line 20 compile?

```
 1 #include <iostream>
 2 #include <vector>
 3 #include <iterator>     //for ostream_iterator
 4 #include <functional>   //for ptr_fun, bind2nd, plus
 5 #include <algorithm>    //for transform
 6 using namespace std;
 7
 8 inline int sum(int i, int j) {return i + j;}
 9 inline int crsum(const int& i, const int& j) {return i + j;}
10
11     vector<int> v(argument(s) for constructor);
12
13     transform(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"),
14         bind2nd(ptr_fun(sum), 10));
15
16     transform(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"),
17         bind2nd(plus<int>(), 10));
18
19     transform(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"),
20         bind2nd(ptr_fun(crsum), 10));
```

The `crsum` in the above line 20 is a pointer to a function. The pointer is of the following data type. We underline the part that will eventually get us into trouble.

```
int (*)(const int&, const int&)
```

The data type of this pointer causes the `ptr_fun` to construct and return a `pointer_to_binary_function` object whose member `second_argument_type` is a typedef for `const int&`. This typedef causes the `bind2nd` to construct and return a `binder2nd` object whose second data member is of type `const int&`. (We saw the data member on line 48 of the definition for class `binder2nd` on p. 863.) Not surprisingly, this data member is initialized by the second argument passed to the `binder2nd`'s constructor (the `10` in the above line 20). The argument is passed by reference; as usual, it is a reference to the data type of the data member. But the data member is already a reference, and there is no such thing as a reference to a reference. The program does not compile.

Fix it by specializing the `pointer_to_binary_function` template class as follows. If its constructor receives a pointer to a function whose second argument is a reference to a `const` data type, the constructor will construct a `pointer_to_binary_function` object whose member `second_argument_type` is a typedef for the same data type, but without the "reference to" (line 24). In othr words, the data type `X2&` in line 23 is stripped down to plain old `X2` in line 24.

To get it to compile with `g++`, I had to strip the `const` out of the data type as well. The `g++` `binder2nd` has two `operator()` member functions, taking arguments of type `first_argument_type&` and `const first_argument_type&`. These types will be distinct only if `first_argument_type` is not `const`.

```
21 namespace std {
22 template <class X1, class X2, class Y>
23 class pointer_to_binary_function<const X1&, const X2&, Y>:
24     public binary_function<X1, X2, Y> {
25 protected:
26     Y (*p)(const X1&, const X2&);
27 public:
    explicit pointer_to_binary_function(Y (*initial_p)(const X1&, const X2&))
        : p(initial_p) {}

    Y operator()(const X1& x1, const X2& x2) const {return (*p)(x1, x2);}
};
}
```

▲

## 8.6.3 STL Algorithms

### A range of elements

Let `first` and `last` be a pair of iterators referring to elements in the same container, or to the empty slot after the last element where the next element would be. Then the notation

`[first, last)`

represents the range of elements from `first` to `last`, including the element to which `first` refers but *not* including the one to which `last` refers. If `first` and `last` are equal, the range is empty. Otherwise, `first` must refer to an element that is earlier than `last`.

These conventions allow us to use the notation `[first, last)` for all ranges, even empty ones. And, of course, the "container" need not be a container at all. The two iterators might be stream iterators referring to the standard input, an input file, or a TCP/IP socket.

Note, however, that a range cannot be delimited by iterators that are merely output iterators. There is no guarantee that output iterators can be compared, so our definition of when the range is empty or non-empty becomes meaningless. To define an output range, we specify one output iterator and an integer count. Examples are the `_n` algorithms: `generate_n`, `fill_n`, `uninitialized_fill_n`, `random_sample_n` (but not `search_n`).

Algorithms that search for an element in a range `[first, last)` do not return the element. They return an iterator that refers to the element, or `last` if the element is not found. The iterator gives us access to the element, and, if the iterator is not merely an input iterator (p. 837), tells us where in the container the element was located.

### The data type of the elements

Let `T` be the data type of each element in the range whose iterators are passed to an algorithm. The "numeric" algorithms (pp. 962−964) assume that `T` is a type such as `float`, `double`, or `complex<double>`: one that can be copied quickly and with no side effects. These algorithms pass and return a `T` by value. The other algorithms make no such assumption and *always* pass and return a `T` by reference.

Other arguments and return values—iterators, predicates and other function objects, and miscellaneous integers—are passed and returned by value. A `T` passed to or returned by a function object is passed by reference, except for the projection function objects.

The numeric algorithms are defined in the header file `<numeric>`. The other algorithms in §8.6.3 are defined in `<algorithm>`.

### Shorthand declarations for the algorithms

The algorithms are template functions, so each "declaration" should be preceded by a template preamble.

```
template <class FOR> FOR adjacent_find(FOR first, FOR last);
```

We omit the preamble to save space.

```
FOR adjacent_find(FOR first, FOR last);
```

The following conventional names are used for template arguments that stand for data types.

`IN` is an input iterator. `IN2` is another type of input iterator, not necessarily the same as `IN`. `OUT` is an output iterator. `FOR` is a forward iterator. `FOR2` is another type of forward iterator, not necessarily the same as `FOR`.

`UPRED` is a unary predicate: a function or function object that will take one of the elements and return a `bool` or a value convertible thereto. `BPRED` is a binary predicate. `FUNC2` is a binary function. `GENERATOR` is a function of no arguments. `UNARY` is a unary function; `BINARY` is a binary function.

`DIFFERENCE` is the `difference_type` of the iterators passed in. `N` is an integer. `RNG` is a random number generator (an object such as a `subtractive_rng` with an `operator(n)` member function that returns a random number greater than or equal to zero and less than `n`).

### 8.6.3.1 Read-only Algorithms

A read-only algorithm does not, by itself, assign a value to an element in a range. An assignment might performed, however, by a function or function object passed to the algorithm, and applied by the algorithm to each element in the range. We did this with the function objects passed to `for_each` on pp. 880−881. In fact, nothing prevents us from defining an `operator==` that assigns a new value to its operands, and having a read-only algorithm apply this `operator==` to the elements. Nothing, that is, except a decent respect for the opinions of mankind.

#### 8.6.3.1.1 Apply no predicate to the elements

**Algorithms that do not access the values of the elements at all**

```
template <class INPUT, class DIFFERENCE_TYPE d>
void advance(INPUT& it, DIFFERENCE_TYPE d);   //read/write reference

template <class INPUT>
typename iterator_traits<INPUT>::difference_type
void distance(INPUT first, INPUT last);
```

The `advance` algorithm moves the iterator forwards or backwards by the specified number of elements. `d` can be negative only if the iterator is at least bidirectional as well as forward. Beware: the first argument is passed as a read/write reference (pp. 73−74).

The `distance` algorithm returns the number of elements in a range. Its return type is the `difference_type` for the given type of iterator. For example, `int *` iterators will yield a `ptrdiff_t`; `vector<int>::iterator`'s will yield a `vector<int>::difference_type`. Other algorithms that return the `difference_type` are `count`, `count_if`, and the `find_distance` we wrote on p. 837.

`advance` and `distance` were given word names, rather than the names "`operator+=`" and "`operator-`", to remind the user of the cost of calling them. They are fast for random access iterators, but slower for other categories.

**Algorithms that call a function during each iteration of a loop**

```
template <class INPUT, class FUNCTION>
FUNCTION for_each(INPUT first, INPUT last, FUNCTION f);

template <class FORWARD, class FUNCTION>
void generate(FORWARD first, FORWARD last, FUNCTION f);

template <class OUTPUT, class N, class FUNCTION>
OUTPUT generate_n(OUTPUT first, N n, FUNCTION f);

```

```
template <class INPUT, class OUTPUT, class FUNCTION>
OUTPUT transform(INPUT first, INPUT last, OUTPUT result, FUNCTION f);

template <class INPUT1, class INPUT2, class OUTPUT, class FUNCTION>
OUTPUT transform(INPUT1 first1, INPUT1 last1,
                 INPUT2 first2, OUTPUT result, FUNCTION f);
```

See the diagram on p. 878 for a summary of these algorithms. The `f` passed to `for_each` and the one-input-range `transform` must accept an element of the input range as its argument. The `f` passed to the two-inpyt-range `transform` must accept an element of each range as its two arguments. The `f` passed to `generate`, `generate_n`, and `transform` must return a value that can be stored into each element of the output range.

To transform a `valarray`, see pp. 899−900.

### 8.6.3.1.2 Check the elements one at a time

```
template <class INPUT, class T>
INPUT find(INPUT first, INPUT last, const T& t);

template <class INPUT, class T>
typename iterator_traits<INPUT>::difference_type
count(INPUT first, INPUT last, const T& t);

template <class INPUT, class T, CLASS UNARY_PREDICATE>
INPUT find_if(INPUT first, INPUT last, UNARY_PREDICATE unary_predicate);

template <class INPUT, class T, CLASS UNARY_PREDICATE>
typename iterator_traits<INPUT>::difference_type
count_if(INPUT first, INPUT last, UNARY_PREDICATE unary_predicate);
```

`find` and `count` compare their third argument to each element of the range. By default, the comparison is performed with the `==` operator. To substitute an alternative comparison, pass a unary predicate to `find_if` and `count_if`.

`find` and `find_if` return `last` if nothing is found; otherwise they return an iterator referring to the first element that satisfied them. `find` has a simple definition on p. 859. The C function `strchr` does the same job for a range of characters; the member function `find` does the same job for a `string` object. (A well-implemented `find` would be dispatched to call these functions when possible.) `find_if` has a simple definition on on p. 864.

`count` has a simple definition on p. 810. There is no need for `count` and `count_if` to return a signed type, because they will never give us a negative value. Unfortunately, class `iterator_traits` has no unsigned `size_type` member corresponding to the signed `difference_type`.

Class `bitset` has a member function that counts how many bits are `1`.

```
#include <iostream>
#include <bitset>
using namespace std;

    bitset<32> b = 0x00000000;
    cout << b.count() << "\n";   //print 0
    b.flip();                    //flip all 32 bits
    cout << b.count() << "\n";   //print 32
```

`find` and `find_if` had to have different names. Otherwise there would be no way to know whether the third argument was the value to search for, or the unary predicate to which each value in the range should be passed. Ditto for `count` and `count_if`.

#### 8.6.3.1.3 Check pairs of elements for equality

```
//There is another declaration for each of these algorithms; see below.

templace <class FORWARD>
FORWARD adjacent_find(FORWARD first, FORWARD last);

templace <class FORWARD, class SIZE_TYPE, class T>
FORWARD search_n(FORWARD first, FORWARD last, SIZE_TYPE n, const T& t);

template <class INPUT, class FORWARD>
INPUT find_first_of(INPUT first1, INPUT last1, FORWARD first2, FORWARD last2);

template <class FORWARD1, class FORWARD2>
FORWARD1
search(FORWARD1 first1, FORWARD1 last1, FORWARD2 first2, FORWARD2 last2);

template <class FORWARD1, class FORWARD2>
FORWARD1
find_end(FORWARD1 first1, FORWARD1 last1, FORWARD2 first2, FORWARD2 last2);

template <class INPUT1, class INPUT2>
bool equal(INPUT1 first1, INPUT1 last1, INPUT2 first2);

template <class INPUT1, class INPUT2>
pair<IN, IN2> mismatch(INPUT1 first1, INPUT1 last1, INPUT2 first2);
```

By default, the algorithms in §8.6.3.1.3 apply the operator `==` to pairs of elements in the range. In each case, an optional final argument lets us substitute a different binary predicate. For example, there are two `adjacent_find`'s:

```
template <class FORWARD>
FORWARD adjacent_find(FORWARD first, FORWARD last);

template <class FORWARD>
FORWARD
adjacent_find(FORWARD first, FORWARD last, BINARY_PREDICATE binary_predicate);
```

`adjacent_find` finds the first occurrence in `[first, last)` of any two consecutive equal values, returning an iterator referring to the first value. If it doesn't find what it's looking for, it returns `last`. See the simple algorithm on p. 840.

`search_n` finds the first occurrence in `[first, last)` of `n` consecutive copies of `t`, returning an iterator referring to the first copy. If it doesn't find what it's looking for, it returns `last`. See the poker example in line 26 of `adjacent_difference.C` on p. 962.

`find_first_of` finds the first occurrence in `[first1, last1)` of any of the values in `[first2, last2)`. The C function `strpbrk` does the same job for a range of characters. The member functions `find_first_of` and `find_last_of` do the same job for a `string` object.

`search` finds the first occurrence in `[first1, last1)` of the entire range `[first2, last2)`, i.e., it finds a substring in a string, returning an iterator that refers to the first element in the substring. The C function `strstr` does the same job for a range of characters. The member function `find` does the same job for a `string` object.

`find_end` finds the last occurrence in `[first1, last1)` of the entire range `[first2, last2)`. It should have been named `search_end`. The member function `rfind` does the same job for a `string` object.

`equal` returns `true` if the range `[first1, last1)` and the range of the same length starting at `first2` have the same elements. In other words, it compares two strings for equality. The C function `strcmp` does the same job for two ranges of characters, returning zero if they are equal. The member function `compare` does the same job for two `string`'s. To compare entire objects, not just subsequences, use the `==` operator. Call the `equal` algorithm only when comparing subsequences of objects, or when comparing two arrays.

```
    vector<int> v1(argument(s) for constructor);
    vector<int> v2(argument(s) for constructor);

    //if the entire vectors are equal,
    if (v1 == v2) {

    //if the first five elements are equal,
    if (equal(v1.begin(), v1.begin() + 5, v2.begin(), v2.begin() + 5)) {
```

Examples are line 13 of `case_insensitive_equal_to.h` on p. 951, and line 38 of `datetime.h` on p. 953. The simplest way to implement `equal` is by calling `mismatch`.

`mismatch` finds the first element in `[first1, last1)` that is different from the corresponding element in the range of the same length starting at `first2`. It constructs and returns a `pair` of two iterators that refer to these elements. The Unix utility `cmp` does this for two sequences of bytes.

**▼ Homework 8.6.3.1.3a: let stack::operator== call mismatch**

Let the `operator==` friend of class `stack` do its work by calling `mismatch`.

▲

**▼ Homework 8.6.3.1.3b: let cookie::operator new[] call search_n**

Let the `operator new[]` member function of class `cookie` call the `search_n` algorithm to find the first `n` consecutive `false`'s in the array of `bool`'s. See p. 419.

▲

**A different binary predicate in place of ==**

Here is a binary predicate we could use in place of the operator `==`. Its name echoes that of the STL function object `equal_to`. Warning: this predicate is not an "equivalence relation". If `a` is approximately equal to `b`, and `b` is approximately equal to `c`, it is not necessarily true that `a` is approximately equal to `c`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/approximately_equal_to.h`

```
#ifndef APPROXIMATELY_EQUAL_TOH
#define APPROXIMATELY_EQUAL_TOH
#include <cstdlib>     //for abs functions that take int and long
#include <cmath>       //for abs functions that take float, double, long double
#include <functional>  //for binary_function
using namespace std;

//Return true if t1 and t2 are within .01 of each other.
//("Close enough for government work.")

template <class T>
struct approximately_equal_to: public binary_function<T, T, bool> {
    bool operator()(const T& t1, const T& t2) const {
        return abs(t1 - t2) < .01;
    }
};
#endif
```

We can use it as follows.

```
#include <vector>
#include <algorithm>
#include "approximately_equal_to.h"
using namespace std;

    vector<double> v(argument(s) for constructor);

    //Find the first pair of adjacent elements that are equal.
    vector<double>::const_iterator it = adjacent_find(v.begin(), v.end());

    //Find the first pair of adjacent elements that are approximately equal.
    it = adjacent_find(v.begin(), v.end(),
        approximately_equal_to<double>());
```

Or we can build a function object that does the same thing as `approximately_equal_to` by using the `compose_fgx1_x2` we wrote on pp. 894−895 to compute the average.

```
#include <cmath>
#include <vector>
#include <algorithm>
using namespace std;

    vector<double> v(argument(s) for constructor);

    //Find the first pair of adjacent elements that are approximately equal,
    //but without approximately_equal_to.
    vector<double>::const_iterator it = adjacent_find(v.begin(), v.end(),
        compose_fgx1_x2(
            bind2nd(less<double>(), .01),
            compose_fgx1_x2(
                ptr_fun(static_cast<double (*)(double)>(abs)),
                minus<double>()
            )
        )
    );
```

**Another binary predicate in place of ==**

Here is another binary predicate we could use in place of the operator `==`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/case_insensitive_equal_to.h`

```
#ifndef CASE_INSENSITIVE_EQUAL_TOH
#define CASE_INSENSITIVE_EQUAL_TOH
#include <cctype>      //for tolower
#include <string>      //for class string
#include <functional>  //for binary_function, equal_to, ptr_fun
#include <algorithm>   //for equal
using namespace std;

//Return true if the strings are equal, ignoring case.

```

```
struct case_insensitive_equal_to: public binary_function<string, string, bool> {
    bool operator()(const string& s1, const string& s2) const {
        return s1.size() == s2.size() && equal(
            s1.begin(), s1.end(),
            s2.begin(),
            compose_fgx1_gx2(
                equal_to<int>(),
                ptr_fun(static_cast<int (*)(int)>(tolower))
            )
        );
    }
};
#endif
```

We can use it as follows.

```
#include <vector>
#include <string>
#include <algorithm>
#include "case_insensitive_equal_to.h"
using namespace std;

    vector<string> v(argument(s) for constructor);

    //Find the first pair of adjacent strings that are equal.
    vector<string>::const_iterator it = adjacent_find(v.begin(), v.end());

    //Find the first pair of adjacent strings that are equal,
    //ignoring case.
    it = adjacent_find(v.begin(), v.end(), case_insensitive_equal_to());
```

#### 8.6.3.1.4 Check pairs of elements for <

```
//There is another declaration for each of these algorithms; see below.

template <class INPUT1, class INPUT2>
bool lexicographical_compare(INPUT1 first1, INPUT1 last1,
                             INPUT2 first2, INPUT2 last2);

template <class FORWARD>
FORWARD min_element(FORWARD first, FORWARD last);

template <class FORWARD>
FORWARD max_element(FORWARD first, FORWARD last);

templace class T
const T& min(const T& t1, const T& t2);

templace class T
const T& max(const T& t1, const T& t2);
```

By default, the algorithms in §8.6.3.1.4 apply the operator `<` to the elements in the range. (They do not attempt to apply the operator `==` to the elements.) In each case, an optional final argument lets us substitute a different binary predicate. For example, there are two `lexicographical_compare`'s.

```
template <class INPUT1, class INPUT2>
```

```
bool lexicographical_compare(INPUT1 first1, INPUT1 last1,
                             INPUT2 first2, INPUT2 last2);

template <class INPUT1, class INPUT2>
bool lexicographical_compare(INPUT1 first1, INPUT1 last1,
                             INPUT2 first2, INPUT2 last2,
                             BINARY_PREDICATE binary_predicate);
```

Here is a binary predicate we could use in place of the operator `<`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/case_insensitive_less.h`

```
#ifndef CASE_INSENSITIVE_LESSH
#define CASE_INSENSITIVE_LESSH
#include <cctype>      //for tolower
#include <string>      //for class string
#include <functional>  //for binary_function, less, ptr_fun
#include <algorithm>   //for lexicographic_compare
using namespace std;

//Return true if s1 is less than s2 (i.e., s1 is earlier in alphabetical order),
//ignoring case.

struct case_insensitive_less: public binary_function<string, string, bool> {
    bool operator()(const string& s1, const string& s2) const {
        return lexicographic_compare(
            s1.begin(), s1.end(),
            s2.begin(), s2.end()
            compose_fgx1_gx2(
                less<int>(),
                ptr_fun(static_cast<int (*)(int)>(tolower))
            )
        );
    }
};
#endif
```

`lexicographical_compare` returns true if `[first1, last1)` is less than `[first2, last2)` in *lexicographical order.* The following example sorts `date` objects in this order to achieve chronological order. The year of each object is the primary sort key; the month is the secondary sort key; the day of the month is the tertiary sort key; etc. In fact, we've been doing lexicographical sort all along whenever we compare two strings for alphabetical order.

To construct a `ostream_iterator<datetime>` (line 22 of `lexicographical_compare.C`), we must first define an `operator<<` for class `datetime` (line 28 of `datetime.h`).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/datetime.h`

```
 1 #ifndef DATETIMEH
 2 #define DATETIMEH
 3 #include <iostream>
 4 #include <iomanip>
 5 #include <algorithm>
 6 using namespace std;
 7
```

```
class datetime {
    static const size_t n = 6;
    int a[n];
public:
    enum {
        january = 1, february, march, april, may, june,
            july, august, september, october, november, december
    };

    //error checking omitted for brevity
    datetime(int initial_month, int initial_day, int initial_year,
        int initial_hour, int initial_minute, int initial_second) {
        a[0] = initial_year;
        a[1] = initial_month;
        a[2] = initial_day;
        a[3] = initial_hour;
        a[4] = initial_minute;
        a[5] = initial_second;
    }

    friend ostream& operator<<(ostream& ost, const datetime& d) {
        const char save = ost.fill();
        return ost << d.a[1] << "/" << d.a[2] << "/" << d.a[0]
            << " " << setfill('0')
            << setw(2) << d.a[3] << ":"
            << setw(2) << d.a[4] << ":"
            << setw(2) << d.a[5] << setfill(save);
    }

    friend bool operator==(const datetime& d1, const datetime& d2) {
        return equal(d1.a, d1.a + datetime::n, d2.a);
    }

    friend bool operator<(const datetime& d1, const datetime& d2) {
        return lexicographical_compare(
            d1.a, d1.a + datetime::n,
            d2.a, d2.a + datetime::n
        );
    }
};

inline bool operator!=(const datetime& d1, const datetime& d2){return!(d1==d2);}
inline bool operator>=(const datetime& d1, const datetime& d2){return !(d1<d2);}
inline bool operator> (const datetime& d1, const datetime& d2){return  d2 < d1;}
inline bool operator<=(const datetime& d1, const datetime& d2){return d2 >= d1;}
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/lexicographical_compare.C`

```
#include <iostream>
#include <cstdlib>
#include <iterator>
#include <algorithm>
#include "datetime.h"
```

```
using namespace std;

int main()
{
    datetime a[] = {
        datetime(datetime::february,  1, 2015, 23,  0,  0),
        datetime(datetime::february,  2, 2015,  1,  1,  1),
        datetime(datetime::february,  2, 2015,  0, 59,  0),
        datetime(datetime::march,     1, 2014,  0,  0,  0),
        datetime(datetime::february,  2, 2015,  1,  0, 59),
        datetime(datetime::february,  2, 2015,  1,  1,  0),
        datetime(datetime::january,   3, 2015,  0,  0,  0)
    };
    const size_t n = sizeof a / sizeof a[0];

    sort(a, a + n);
    copy(a, a + n, ostream_iterator<datetime>(cout, "\n"));
    return EXIT_SUCCESS;
}
```

```
3/1/2014 00:00:00    Sort by year.  2014 comes first.
1/3/2015 00:00:00    If years are equivalent, sort by month.  January comes first.
2/1/2015 23:00:00    If months are equivalent, sort by day.  The first day of the month comes first.
2/2/2015 00:59:00    If days are equivalent, sort by hour.  Midnight comes first.
2/2/2015 01:00:59    If hours are equivalent, sort by minute.
2/2/2015 01:01:00    If minutes are equivalent, sort by second.
2/2/2015 01:01:01
```

`min_element` and `max_element` find the first element with the maximum or minimum value in `[first, last)`. They return `last` if the range is empty. See the simple definition on pp. 908−909.

`min` and `max` take elements, not iterators that refer to elements. If neither argument is less than the other, `min` and `max` return their first argument. (We never consider the case where the arguments are equal. The algorithms in §8.6.3.1.4 do not apply the `==` operator to the elements.) See the simple definition on p. 641.

### 8.6.3.2 Read/write algorithms

A read/write algorithm can assign a value to an element in a range, even if no assignment is performed by the function or function object passed to the algorithm. But no algorithm can resize the container that holds the range, unless the arguments are insert iterators. Even the `remove` and `remove_if` algorithms remove no elements. They merely take the values we want to keep and move them up to the front of the range.

#### 8.6.3.2.1 Apply no predicate to the elements

The algorithms in section §8.6.3.2.1 apply no predicate to the elements.

```
template <class FORWARD, class T>
void fill(FORWARD first, FORWARD last, const T& t);

template <class OUTPUT, class SIZE_TYPE, class T>
OUT fill_n(OUTPUT first, N n, const T& t);

template <class OUTPUT, class INPUT>
OUTPUT copy(INPUT first, INPUT last, OUTPUT result);
```

```

template <class BIDIR1, class BIDIR2>
BIDIR2
copy_backward(BIDIR1 first, BIDIR1 last, BIDIR2 result);

template <class BIDIR>
void reverse(BIDIRL first, BIDIR last);

template <class BIDIR, class OUTPUT>
OUTPUT reverse_copy(BIDIR first, BIDIR last, OUTPUT result);

template <class FORWARD>
void rotate(FORWARD first, FORWARD middle, FORWARD last);

template <class FORWARD, class OUTPUT>
OUTPUT rotate_copy(FORWARD first, FORWARD middle, FORWARD last, OUTPUT result);

template <class RANDOM>
void random_shuffle(RANDOM first, RANDOM last);

template <class RANDOM, class RANDOM_NUMBER_GENERATOR>
void random_shuffle(RANDOM first, RANDOM last, R,
    RANDOM_NUMBER_GENERATOR& random_number_generator);

template <class INPUT, class RANDOM>
RANDOM random_sample(INPUT first1, INPUT last1, RANDOM first2, RANDOM last2);

template <class INPUT, class RANDOM_NUMBER_GENERATOR>
RANDOM random_sample(INPUT first1, INPUT last1, RANDOM first2, RANDOM last2,
    RANDOM_NUMBER_GENERATOR& random_number_generator);

template <class FORWARD, class OUTPUT, class SIZE_TYPE>
RANDOM random_sample_n(FORWARD first1, FORWARD last1, OUTPUT result,
    SIZE_TYPE n);

template <class FORWARD, class OUTPUT, class SIZE_TYPE>
RANDOM random_sample_n(FORWARD first1, FORWARD last1, OUTPUT result,
    SIZE_TYPE n, RANDOM_NUMBER_GENERATOR& random_number_generator);
```

`fill` copies the value `t` into each element of `[first, last)`. `fill_n` copies the value `t` into each element of the range of length `n` starting at `first`. See the simple definitions on pp. 966 and 853. To `fill` a `valarray`, just assign a value to it or to a slice thereof. `copy` and `copy_backwards` copy the input range into the output range. If the ranges do not overlap, we can call either algorithm. If the end of the input range overlaps with the start of the output range, call `copy`. It copies the input range from `first` to `last`, returning an iterator that refers to the element after the last element in the output range. (If the output range is empty, the return value is the third argument.) where the next value would be copied, if there were one more value. (By contrast, the C function `strcpy` returns the address of the *start* of the output range.) See the simple definition on p. 844 and the optimized one on pp. 919−932.

If the start of the input range overlaps with the end of the output range, call `copy_backwards`. It copies the first range from `last` to `first`. The third argument of `copy_backwards` refers to the *last* element in the result range; this is where the first element of the source range will be copied. `copy_backwards` returns an iterator that refers to the element before the first element in the output range. (If the output range is empty, the return value is the third argument.)

The `reverse` algorithm reverses the order of the values in a range, overwriting the original ones. `reverse_copy` writes the values into a different destination.

**rotate and rotate_copy**

```
void     rotate(FOR first, FOR middle, FOR last);
OUT rotate_copy(FOR first, FOR middle, FOR last, OUT result);
```

**random_shuffle**

```
void random_shuffle(RANDOM first, RANDOM last);
void random_shuffle(RANDOM first, RANDOM last, RNG& rng);
```

**random_sample**

```
RANDOM random_sample(IN first1, IN last1, RANDOM first2, RANDOM last2);
RANDOM random_sample(IN first1, IN last1, RANDOM first2, RANDOM last2, RNG& rng);
RANDOM random_sample_n(FOR first1, FOR last1, OUT result, N n);
RANDOM random_sample_n(FOR first1, FOR last1, OUT result, N n, RNG& rng);
```

**Swapping algorithms**

```
void swap(T& t1, T& t2);    //read/write references
void iter_swap(FOR it1, FOR it2);   //read/write iterators
FOR2 swap_ranges(FOR first1, FOR last1, FOR2 first2);
```

`swap` and `iter_swap` swap a pair of elements. Lines 8−10 all do the same thing; 10 is simpler than 9 because we don't have to write the asterisks.

`swap_ranges` swaps two ranges of equal size.

```
 1     int a[] = {10, 20, 30, 40};
 2     const size_t n = sizeof a / sizeof a[0];
 3     vector<int> v(a, a + n);
 4
 5     vector<int>::iterator it1 = v.begin();
 6     vector<int>::iterator it2 = v.begin() + 1;
 7
 8     swap(v[0], v[1]);    //arguments are elements passed by reference
 9     swap(*it1, *it2);    //arguments are elements passed by reference
10     iter_swap(it1, it2); //arguments are iterators passed by value
11
12     //Swap first two elements and last two elements.
13     //Return v.begin() + 4, which is the same as v.end().
14     swap_ranges(v.begin(), v.begin() + 2, v.begin() + 2);
```

Each container (except `bitset` and the container adaptors) has a `swap` member function. The `swap` algorithm will call the `swap` member function for each container.

#### 8.6.3.2.2 Check the elements one at a time

Each algorithm in §8.6.3.2.2 (except the `partition` algorithms) comes in two flavors. It can search for elements with the value `t`, or it can search for elements that satisfy the unary predicate.

```
void replace   (FOR first, FOR last, const T& t,  const T& tnew);
void replace_if(FOR first, FOR last, UPRED upred, const T& tnew);
```

```
OUT replace_copy   (FOR first, FOR last, OUT result, const T& t,  const T& tnew);
OUT replace_copy_if(FOR first, FOR last, OUT result, UPRED upred, const T& tnew);

FOR remove   (FOR first, FOR last, const T& t );
FOR remove_if(FOR first, FOR last, UPRED upred);

OUT remove_copy   (IN first, IN last, OUT result, const T& t);
OUT remove_copy_if(IN first, IN last, OUT result, UPRED upred);

BIDIR partition(BIDIR first, BIDIR last, UPRED upred);
FOR stable_partition(FOR first, BIDIR last, UPRED upred);
```

### 8.6.3.2.3 Check pairs of elements for equality

By default, the algorithms in §8.6.3.2.3 apply the operator `==` to the elements in the range. An optional final argument lets us substitute a different binary predicate.

#### uniq and uniq_copy

Unix program `uniq`; `list::uniq`.

```
FOR unique(FOR first, FOR last);
FOR unique(FOR first, FOR last, BPRED bpred);
OUT unique_copy(IN first, IN last, OUT result);
OUT unique_copy(IN first, IN last, OUT result, BPRED bpred);
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/unique.C`

```
#include <iostream>
#include <cstdlib>
#include <cctype>   //for isspace
#include <string>
#include <functional>
#include <ext/functional>
#include <algorithm>
using namespace std;

struct consecutive_blanks:
    public binary_function<string::value_type, string::value_type, bool> {
    bool operator()(string::value_type c1, string::value_type c2) const {
        return c1 == ’ ’ && c2 == ’ ’;
    }
};

int main()
{
    string s = "It   was\f\n\r\t\va miracle        of rare device,   ";

    replace_if(s.begin(), s.end(), static_cast<int (*)(int)>(isspace), ’ ’);
    cout << "\"" << s << "\"\n";

    string::iterator it = unique(s.begin(), s.end(), consecutive_blanks());
    cout << "\"" << s.substr(0, it - s.begin()) << "\"\n";
    cout << "\"" << s << "\"\n";

    return EXIT_SUCCESS;
}
```

```
"It   was     a miracle        of rare device,   "
"It was a miracle of rare device, "
"It was a miracle of rare device, rare device,   "
```

#### 8.6.3.2.4 Check pairs of elements for <

By default, the algorithms in §8.6.3.2.4 apply the operator `<` to the elements in the range. An optional final argument lets us substitute a different binary predicate.

**Permutation**

```
bool next_permutation(BIDIR first, BIDIR last);
bool prev_permutation(BIDIR first, BIDIR last);
```

By default, the permutation functions apply the operator `<` to the elements in the range. An optional final argument lets us substitute a different binary predicate. Since we are comparing characters in this example, the final argument could be `greater<char>()`.

`next_permutation` returns `false` when it has arranged the elements back into lexicographical order. `prev_permutation` returns `false` when it is given elements that are already in lexicographical order.

To permute the elements of a `valarray`, see p. 907.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/permute.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <string>
 4 #include <algorithm>
 5 using namespace std;
 6
 7 int main()
 8 {
 9     string s = "abc";   //"abcd" has 24 permutations; "abcde" has 120
10
11     do {
12         cout << s << "\n";
13     } while (next_permutation(s.begin(), s.end()));
14
15     cout << s << "\n";
16     return EXIT_SUCCESS;
17 }
```

```
abc        lines 11−13 output 6 = 1 × 2 × 3 permutations starting with and ending just before abc.
acb
bac
bca
cab
cba
abc        line 15
```

**Sorting algorithms**

The average-case complexity of `sort` is $O(N \log_2 N)$. This means that if we double the number of elements, we will more than double the time it takes to sort them. In the words of Leviticus 26:8,

> And five of you shall chase a hundred,
> And a hundred of you shall put ten thousand to flight.

The worst-case complexity of `sort` is not specified by the C++ Standard. In older versions of C++ it was $O(N^2)$ because it used C. A. R. Hoare's Quicksort (1962), but now it is is $O(N \log_2 N)$ because it uses David R. Musser's Introsort (1997).

`stable_sort` is guaranteed to make no unnecessary moves, leaving equivalent values in their original order. It's slower than plain old `sort`; worst case is $ON(N \log_2 N)^2$, but it will be $O(N \log_2 N)$ if enough memory is available.

Call `partial_sort` if you need to find only the winners for the first prize, second prize, third prize.

```
void sort(RANDOM first, RANDOM last);
void stable_sort(RANDOM first, RANDOM last);
void partial_sort(RANDOM first, RANDOM middle, RANDOM last);
void partial_sort_copy(IN first1, IN last1, RANDOM first2, RANDOM last2);
void nth_element(RANDOM first, RANDOM nth, RANDOM last);
bool is_sorted(FOR first, FOR last);
```

`list::sort`

### 8.6.3.2.5 Assume that < has already been applied to the elements

By default, the algorithms in §8.6.3.2.5 apply the operator `<` to the elements in the range. An optional final argument lets us substitute a different binary predicate.

```
bool binary_search        (FOR first, FOR last, const T& t);
FOR lower_bound           (FOR first, FOR last, const T& t);
FOR upper_bound           (FOR first, FOR last, const T& t);
pair<FOR, FOR> equal_range(FOR first, FOR last, const T& t);

OUT merge(IN first1, IN last1, IN2 first2, IN2 last2, OUT result)
void inplace_merge(BIDIR first, BIDIR middle, BIDIR last);
void inplace_merge(BIDIR first, BIDIR middle, BIDIR last, BPRED bpred);
```

`bsearch` in C Standard Library. `list::merge`

**Set operations on a pair of ranges**

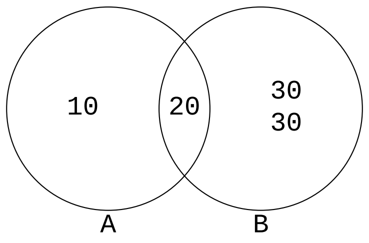
These algorithms take a pair of ranges and perform a classic set operation on them: union, intersection, etc. Each range is a pair of input iterators. The container to which each pair belongs could be a `set` object, but it does not have to be. As usual, it could also be a `vector`, `list`, or a plain old array.

By definition, a mathematical set contains only at most one copy of each value. The content of each range could be a mathematical set, but it does not have to be. To show that the algorithms will work correctly even if a range contains more than one copy of a value, we put two `30`'s in the container `B`.

The two ranges must be sorted before they are passed to the algorithms in §8.6.3.2.5. By default, the algorithms assume that the ranges have been sorted in order of the `<` operator. An optional final argument lets us substitute a different binary predicate. Note that a `multiset`, like a `map` or a `set`, will automatically rearrange its elements if necessary (line 13).

The algorithms in §8.6.3.2.5 assume that the elements are comparable with `<` or with the optional predicate, but they do not assume that the elements are comparable with `==`. Instead of testing for equality, they assume that two elements are equivalent if neither one is less than the other. Classes `map` and `set` compare in the same way; see p. 788.

```
bool includes               (IN first1, IN last1, IN2 first2, IN2 last2);
OUT set_union               (IN first1, IN last1, IN2 first2, IN2 last2, OUT result);
OUT set_intersection        (IN first1, IN last1, IN2 first2, IN2 last2, OUT result);
OUT set_difference          (IN first1, IN last1, IN2 first2, IN2 last2, OUT result);
OUT set_symmetric_difference(IN first1, IN last1, IN2 first2, IN2 last2, OUT result);
```



—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/set.C`

```
#include <iostream>
#include <cstdlib>
#include <set>
#include <iterator>
#include <algorithm>
using namespace std;

int main()
{
    const int A[] = {10, 20};
    const size_t nA = sizeof A / sizeof A[0];

    const int b[] = {30, 20, 30};
    const size_t nB = sizeof b / sizeof b[0];
    const multiset<int> B(b, b + nB);

    ostream_iterator<int> it(cout, " ");

    cout << boolalpha << includes(A, A + nA, B.begin(), B.end()) << "\n";

    cout << "union == ";
    set_union(A, A + nA, B.begin(), B.end(), it);
    cout << "\n";

    cout << "intersection == ";
    set_intersection(A, A + nA, B.begin(), B.end(), it);
    cout << "\n";

    cout << "difference: A - B == ";
    set_difference(A, A + nA, B.begin(), B.end(), it);
    cout << "\n";

    cout << "difference: B - A == ";
    set_difference(B.begin(), B.end(), A, A + nA, it);
    cout << "\n";

    cout << "symmetric difference == ";
    set_symmetric_difference(A, A + nA, B.begin(), B.end(), it);
    cout << "\n";

    return EXIT_SUCCESS;
}
```

```
false                                  Is it true that A ⊇ B?
union == 10 20 30 30                   A ∪ B
intersection == 20                     A ∩ B
difference: A - B == 10                the elements in A but not in B
difference: B - A == 30 30             the elements in B but not in A
symmetric difference == 10 30 30       the elements in either one but not in both
```

**Heap algorithms**

```
void make_heap(RANDOM first, RANDOM last);
void push_heap(RANDOM first, RANDOM last);
void  pop_heap(RANDOM first, RANDOM last);
void sort_heap(RANDOM first, RANDOM last);
bool   is_heap(RANDOM first, RANDOM last);
```

Class `consecutive` was on pp. 882−883.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/heap.C`

```
#include <iostream>
#include <cstdlib>
#include <algorithm>
#include <iterator>
#include "consecutive.h"
using namespace std;

int main()
{
    const size_t n = 10;
    int a[n];
    ostream_iterator<int> it(cout, " ");

    generate(a, a + n, consecutive<int>());
    copy(a, a + n, it);
    cout << "\n";

    random_shuffle(a, a + n);
    copy(a, a + n, it);
    cout << "\n";

    make_heap(a, a + n);
    copy(a, a + n, it);
    cout << "\n";

    return EXIT_SUCCESS;
}
```

```
0 1 2 3 4 5 6 7 8 9
4 5 9 8 1 3 6 0 2 7
9 8 6 5 7 3 4 0 2 1
```

### 8.6.3.3 Numeric algorithms

```
T accumulate(IN first, IN last, T init);
T accumulate(IN first, IN last, T init, FUNC2 func2);
OUT partial_sum(IN first, IN last, OUT result);
OUT partial_sum(IN first, IN last, OUT result, FUNC2 func2);
OUT adjacent_difference(IN first, IN last, OUT result);
OUT adjacent_difference(IN first, IN last, OUT result, FUNC2 func2);
```

The numeric algorithms are declared in the header file `<numeric>`. They expect `T` to be a built-in number or combination thereof (`float`, `double`, `complex<double>`, etc.), and therefore fast enough to pass by value.

For `accumulate`, see the simple definition on p. 810. To sum the values in a `valarray`, see line 30 of `valarray.C` on p. 899.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/adjacent_difference.C`

```
#include <iostream>
#include <cstdlib>
#include <iterator>
#include <algorithm> //for sort, search_n
#include <numeric>
using namespace std;

int main()
{
    //A poker hand.  11, 12, 13, 14 are J, Q, K, A.
    int a[] = {2, 6, 3, 5, 4};
    const size_t n = sizeof a / sizeof a[0];

    ostream_iterator<int> it(cout, " ");
    copy(a, a + n, it);
    cout << "\n";

    sort(a, a + n);
    copy(a, a + n, it);
    cout << "\n";

    adjacent_difference(a, a + n, a);
    copy(a, a + n, it);
    cout << "\n";

    if (search_n(a + 1, a + n, n - 1, 1) == a + 1) {
        cout << "It’s a straight.\n";
    } else {
        cout << "It’s not a straight.\n";
    }

    //Reconstruct the original hand.
    partial_sum(a, a + n, a);
    copy(a, a + n, it);
    cout << "\n";

    return EXIT_SUCCESS;
}
```

```
2 6 3 5 4
2 3 4 5 6
2 1 1 1 1
It’s a straight.
2 3 4 5 6
```

**▼ Homework 8.6.3.3a: let date::julian call accumulate**

In the three-data member version of class `date`, change the body of `date::julian` to the following.

```
        return accumulate(length + 1, length + month, day);
```

▲

**Inner product**

The `inner_product` algorithm returns the “dot product” of a pair of vectors, beloved of students of Linear Algebra. Instead of defaulting to zero as the staring value, you must supply it as the third argument. Instead of defaulting to multiplication and addition, you can supply your own operations as the fourth and fifth arguments.

```
T inner_product(IN first1, IN last1, IN2 first2, T t);
T inner_product(IN first1, IN last1, IN2 first2, T t, MULT mult, ADD add);
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/library/inner_product.C`

```
#include <iostream>
#include <cstdlib>
#include <cmath>     //for sqrt
#include <vector>
#include <numeric>   //for inner_product
using namespace std;

template <class T>
inline T length(const vector<T>& v)
{
    return sqrt(inner_product(v.begin(), v.end(), v.begin(), T()));
}

int main()
{
    const double a1[] = {3.0, 4.0};
    const size_t n1 = sizeof a1 / sizeof a1[0];
    vector<double> v1(a1, a1 + n1);
    cout << "v1: size == " << v1.size()
        << ", length == " << length(v1) << "\n";

    const double a2[] = {2.0, 3.0, 6.0};
    const size_t n2 = sizeof a2 / sizeof a2[0];
    vector<double> v2(a2, a2 + n2);
    cout << "v2: size == " << v2.size()
        << ", length == " << length(v2) << "\n";

    return EXIT_SUCCESS;
}
```

```
v1: size == 2, length == 5
v2: size == 3, length == 7
```

### 8.6.3.4 Algorithms for implementing a new container

These algorithms were used to implement classes `vector`, `list`, `map`, etc. For professional use only. Declared in the header file `<memory>`.

```
construct
destroy
uninitialized_copy
uninitialized_fill
uninitialized_fill_n
get_temporary_buffer
return_temporary_buffer
```

**Still to do in Chapter 8:**

```
int date::julian() const
{
    return accumulate(length + 1, length + month, day);
}
```