# 18


# Algorithms and Function Objects

*Form is liberating.*
*– engineers´ proverb*

Introduction — overview of standard algorithms — sequences — function objects — predicates — arithmetic objects — binders — member function objects — ***for_each*** — finding elements — ***count*** — comparing sequences — searching — copying — ***transform*** — replacing and removing elements — filling a sequence — reordering — ***swap*** — sorted sequences — ***binary_search*** — ***merge*** — set operations — ***min*** and ***max***— heaps — permutations — C-style algorithms — advice — exercises.

## 18.1 Introduction [algo.intro]

A container by itself is really not that interesting. To be genuinely useful, a container must be supported by basic operations such as finding its size, iterating, copying, sorting, and searching for elements. Fortunately, the standard library provides algorithms to serve the most common and basic needs that users have of containers.

This chapter summarizes the standard algorithms and gives a few examples of their uses, a presentation of the key principles and techniques used to express the algorithms in C++, and a more detailed explanation of a few key algorithms.

Function objects provide a mechanism through which a user can customize the behavior of the standard algorithms. Function objects supply key information that an algorithm needs in order to operate on a user's data. Consequently, emphasis is placed on how function objects can be defined and used.

## 18.2 Overview of Standard Library Algorithms [algo.summary]

At first glimpse, the standard library algorithms can appear overwhelming. However, there are just 60 of them. I have seen classes with more member functions. Furthermore, many algorithms share a common basic behavior and a common interface style that eases understanding. As with language features, a programmer should use the algorithms actually needed and understood – and only those. There are no awards for using the highest number of standard algorithms in a program. Nor are there awards for using standard algorithms in the most clever and obscure way. Remember, a primary aim of writing code is to make its meaning clear to the next person reading it – and that person just might be yourself a few years hence. On the other hand, when doing something with elements of a container, consider whether that action could be expressed as an algorithm in the style of the standard library. That algorithm might already exist. If you don't consider work in terms of general algorithms, you will reinvent the wheel.

Each algorithm is expressed as a template function (§13.3) or a set of template functions. In that way, an algorithm can operate on many kinds of sequences containing elements of a variety of types. Algorithms that return an iterator (§19.1) as a result generally use the end of an input sequence to indicate failure. For example:

```
void f(list<string>& ls)
{
    list<string>::const_iterator p = find(ls.begin(),ls.end(),"Fred");

    if (p == ls.end()) {
        // didn't find "Fred"
    }
    else {
        // here, p points to "Fred"
    }
}
```

The algorithms do not perform range checking on their input or output. Range errors must be prevented by other means (§18.3.1, §19.3). When an algorithm returns an iterator, that iterator is of the same type as one of its inputs. In particular, an algorithm's arguments control whether it returns a *const_iterator* or a non-*const iterator*. For example:

```
void f(list<int>& li, const list<string>& ls)
{
    list<int>::iterator p = find(li.begin(),li.end(),42);
    list<string>::const_iterator q = find(ls.begin(),ls.end(),"Ring");
}
```

The algorithms in the standard library cover the most common general operations on containers such as traversals, sorting, searching, and inserting and removing elements. The standard algorithms are all in the *std* namespace and their declarations are found in *<algorithm>*. Interestingly, most of the really common algorithms are so simple that the template functions are typically inline. This implies that the loops expressed by the algorithms benefit from aggressive per-function optimization.

The standard function objects are also in namespace *std*, but their declarations are found in *<functional>*. The function objects are designed to be easy to inline.

Nonmodifying sequence operations are used to extract information from a sequence or to find the positions of elements in a sequence:

| Nonmodifying Sequence Operations (§18.5) <algorithm> | |
|---|---|
| *for_each()* | Do operation for each element in a sequence. |
| *find()* | Find first occurrence of a value in a sequence. |
| *find_if()* | Find first match of a predicate in a sequence. |
| *find_first_of()* | Find a value from one sequence in another. |
| *adjacent_find()* | Find an adjacent pair of values. |
| *count()* | Count occurrences of a value in a sequence. |
| *count_if()* | Count matches of a predicate in a sequence. |
| *mismatch()* | Find the first elements for which two sequences differ. |
| *equal()* | True if the elements of two sequences are pairwise equal. |
| *search()* | Find the first occurrence of a sequence as a subsequence. |
| *find_end()* | Find the last occurrence of a sequence as a subsequence. |
| *search_n()* | Find the ***n***th occurrence of a value in a sequence. |

Most algorithms allow a user to specify the actual action performed for each element or pair of elements. This makes the algorithms much more general and useful than they appear at first glance. In particular, a user can supply the criteria used for equality and difference (§18.4.2). Where reasonable, the most common and useful action is provided as a default.

Modifying sequence operations have little in common beyond the obvious fact that they might change the values of elements of a sequence:

| Modifying Sequence Operations (§18.6) <algorithm> | |
|---|---|
| *transform()* | Apply an operation to every element in a sequence. |
| *copy()* | Copy a sequence starting with its first element. |
| *copy_backward()* | Copy a sequence starting with its last element. |
| *swap()* | Swap two elements. |
| *iter_swap()* | Swap two elements pointed to by iterators. |
| *swap_ranges()* | Swap elements of two sequences. |
| *replace()* | Replace elements with a given value. |
| *replace_if()* | Replace elements matching a predicate. |
| *replace_copy()* | Copy sequence replacing elements with a given value. |
| *replace_copy_if()* | Copy sequence replacing elements matching a predicate. |
| *fill()* | Replace every element with a given value. |
| *fill_n()* | Replace first ***n*** elements with a given value. |
| *generate()* | Replace every element with the result of an operation. |
| *generate_n()* | Replace first ***n*** elements with the result of an operation. |
| *remove()* | Remove elements with a given value. |
| *remove_if()* | Remove elements matching a predicate. |

| Modifying Sequence Operations (continued) (§18.6) <algorithm> | |
|---|---|
| *remove_copy()* | Copy a sequence removing elements with a given value. |
| *remove_copy_if()* | Copy a sequence removing elements matching a predicate. |
| *unique()* | Remove equal adjacent elements. |
| *unique_copy()* | Copy a sequence removing equal adjacent elements. |
| *reverse()* | Reverse the order of elements. |
| *reverse_copy()* | Copy a sequence into reverse order. |
| *rotate()* | Rotate elements. |
| *rotate_copy()* | Copy a sequence into a rotated sequence. |
| *random_shuffle()* | Move elements into a uniform distribution. |

Every good design shows traces of the personal traits and interests of its designer. The containers and algorithms in the standard library clearly reflect a strong concern for classical data structures and the design of algorithms. The standard library provides not only the bare minimum of containers and algorithms needed by essentially every programmer. It also includes many of the tools used to provide those algorithms and needed to extend the library beyond that minimum.

The emphasis here is not on the design of algorithms or even on the use of any but the simplest and most obvious algorithms. For information on the design and analysis of algorithms, you should look elsewhere (for example, [Knuth,1968] and [Tarjan,1983]). Instead, this chapter lists the algorithms offered by the standard library and explains how they are expressed in C++. This focus allows someone who understands algorithms to use the library well and to extend it in the spirit in which it was built.

The standard library provides a variety of operations for sorting, searching, and manipulating sequences based on an ordering:

| Sorted Sequences (§18.7) <algorithm> | |
|---|---|
| *sort()* | Sort with good average efficiency. |
| *stable_sort()* | Sort maintaining order of equal elements. |
| *partial_sort()* | Get the first part of sequence into order. |
| *partial_sort_copy()* | Copy getting the first part of output into order. |
| *nth_element()* | Put the nth element in its proper place. |
| *lower_bound()* | Find the first occurrence of a value. |
| *upper_bound()* | Find the first element larger than a value. |
| *equal_range()* | Find a subsequence with a given value. |
| *binary_search()* | Is a given value in a sorted sequence? |
| *merge()* | Merge two sorted sequences. |
| *inplace_merge()* | Merge two consecutive sorted subsequences. |
| *partition()* | Place elements matching a predicate first. |
| *stable_partition()* | Place elements matching a predicate first, preserving relative order. |

| Set Algorithms (§18.7.5) <algorithm> | |
|---|---|
| *includes()* | True if a sequence is a subsequence of another. |
| *set_union()* | Construct a sorted union. |
| *set_intersection()* | Construct a sorted intersection. |
| *set_difference()* | Construct a sorted sequence of elements in the first but not the second sequence. |
| *set_symmetric_difference()* | Construct a sorted sequence of elements in one but not both sequences. |

Heap operations keep a sequence in a state that makes it easy to sort when necessary:

| Heap Operations (§18.8) <algorithm> | |
|---|---|
| *make_heap()* | Make sequence ready to be used as a heap. |
| *push_heap()* | Add element to heap. |
| *pop_heap()* | Remove element from heap. |
| *sort_heap()* | Sort the heap. |

The library provides a few algorithms for selecting elements based on a comparison:

| Minimum and Maximum (§18.9) <algorithm> | |
|---|---|
| *min()* | Smaller of two values. |
| *max()* | Larger of two values. |
| *min_element()* | Smallest value in sequence. |
| *max_element()* | Largest value in sequence. |
| *lexicographical_compare()* | Lexicographically first of two sequences. |

Finally, the library provides ways of permuting a sequence:

| Permutations (§18.10) <algorithm> | |
|---|---|
| *next_permutation()* | Next permutation in lexicographical order. |
| *prev_permutation()* | Previous permutation in lexicographical order. |

In addition, a few generalized numerical algorithms are provided in ***<numeric>*** (§22.6).

In the description of algorithms, the template parameter names are significant. ***In***, ***Out***, ***For***, ***Bi***, and ***Ran*** mean input iterator, output iterator, forward iterator, bidirectional iterator, and random-access iterator, respectively (§19.2.1). ***Pred*** means unary predicate, ***BinPred*** means binary predicate (§18.4.2), ***Cmp*** means a comparison function (§17.1.4.1, §18.7.1), ***Op*** means unary operation, and ***BinOp*** means binary operation (§18.4). Conventionally, much longer names have been used for template parameters. However, I find that after only a brief acquaintance with the standard library, those long names decrease readability rather than enhancing it.

A random-access iterator can be used as a bidirectional iterator, a bidirectional iterator as a forward iterator, and a forward iterator as an input or an output iterator (§19.2.1). Passing a type that doesn't provide the required operations will cause template-instantiation-time errors (§C.13.7). Providing a type that has the right operations with the wrong semantics will cause unpredictable run-time behavior (§17.1.4).

## 18.3 Sequences and Containers [algo.seq]

It is a good general principle that the most common use of something should also be the shortest, the easiest to express, and the safest. The standard library violates this principle in the name of generality. For a standard library, generality is essential. For example, we can find the first two occurrences of *42* in a list like this:

```
void f(list<int>& li)
{
    list<int>::iterator p = find(li.begin(),li.end(),42);      // first occurrence
    if (p != li.end()) {
        list<int>::iterator q = find(++p,li.end(),42);         // second occurrence
        // ...
    }
    // ...
}
```

Had *find*( ) been expressed as an operation on a container, we would have needed some additional mechanism for finding the second occurrence. Importantly, generalizing such an ''additional mechanism'' for every container and every algorithm is hard. Instead, standard library algorithms work on sequences of elements. That is, the input of an algorithm is expressed as a pair of iterators that delineate a sequence. The first iterator refers to the first element of the sequence, and the second refers to a point one-beyond-the-last element (§3.8, §19.2). Such a sequence is called ''half open'' because it includes the first value mentioned and not the second. A half-open sequence allows many algorithms to be expressed without making the empty sequence a special case.

A sequence – especially a sequence in which random access is possible – is often called a *range*. Traditional mathematical notations for a half-open range are [*first*, *last*) and [*first*, *last*[. Importantly, a sequence can be the elements of a container or a subsequence of a container. Further, some sequences, such as I/O streams, are not containers. However, algorithms expressed in terms of sequences work just fine.

### 18.3.1 Input Sequences [algo.range]

Writing *x*.*begin*( ),*x*.*end*( ) to express ''all the elements of *x*'' is common, tedious, and can even be error-prone. For example, when several iterators are used, it is too easy to provide an algorithm with a pair of arguments that does not constitute a sequence:

```
void f(list<string>& fruit, list<string>& citrus)
{
    typedef list<string>::const_iterator LI;

    LI p1 = find(fruit.begin(),citrus.end(),"apple");   // wrong! (different sequences)
    LI p2 = find(fruit.begin(),fruit.end(),"apple");    // ok
    LI p3 = find(citrus.begin(),citrus.end(),"pear");   // ok
    LI p4 = find(p2,p3,"peach");                        // wrong! (different sequences)
    // ...
}
```

In this example there are two errors. The first is obvious (once you suspect an error), but it isn't

easily detected by a compiler. The second is hard to spot in real code even for an experienced programmer. Cutting down on the number of explicit iterators used alleviates this problem. Here, I outline an approach to dealing with this problem by making the notion of an input sequence explicit. However, to keep the discussion of standard algorithms strictly within the bounds of the standard library, I do not use explicit input sequences when presenting algorithms in this chapter.

The key idea is to be explicit about taking a sequence as input. For example:

```
template<class In, class T> In find(In first, In last, const T& v)      // standard
{
    while (first!=last && *first!=v) ++first;
    return first;
}

template<class In, class T> In find(Iseq<In> r, const T& v)             // extension
{
    return find(r.first,r.second,v);
}
```

In general, overloading (§13.3.2) allows the input-sequence version of an algorithm to be preferred when an ***Iseq*** argument is used.

Naturally, an input sequence is implemented as a pair (§17.4.1.2) of iterators:

```
template<class In> struct Iseq : public pair<In,In> {
    Iseq(In i1, In i2) : pair<In,In>(i1,i2) { }
};
```

We can explicitly make the ***Iseq*** needed to invoke the second version of ***find()***:

```
LI p = find(Iseq<LI>(fruit.begin(),fruit.end()),"apple");
```

However, that is even more tedious than calling the original ***find()*** directly. Simple helper functions relieve the tedium. In particular, the ***Iseq*** of a container is the sequence of elements from its ***begin()*** to its ***end()***:

```
template<class C> Iseq<C::iterator_type> iseq(C& c)                     // for container
{
    return Iseq<C::iterator_type>(c.begin(),c.end());
}
```

This allows us to express algorithms on containers compactly and without repetition. For example:

```
void f(list<string>& ls)
{
    list<string>::iterator p = find(ls.begin(),ls.end(),"standard");
    list<string>::iterator q = find (iseq(ls),"extension");
    // ..
}
```

It is easy to define versions of ***iseq()*** that produce ***Iseq***s for arrays, input streams, etc. (§18.13[6]).

The key benefit of ***Iseq*** is that it makes the notion of an input sequence explicit. The immediate practical effect is that use of ***iseq()*** eliminates much of the tedious and error-prone repetition needed to express every input sequence as a pair of iterators.

The notion of an output sequence is also useful. However, it is less simple and less immediately useful than the notion of an input sequence (§18.13[7]; see also §19.2.4).

## 18.4 Function Objects [algo.fct]

Many algorithms operate on sequences using iterators and values only. For example, we can *find*( ) the first element with the value *7* in a sequence like this:

```
void f(list<int>& c)
{
    list<int>::iterator p = find(c.begin(),c.end(),7);
    // ...
}
```

To do more interesting things we want the algorithms to execute code that we supply (§3.8.4). For example, we can find the first element in a sequence with a value of less than *7* like this:

```
bool less_than_7(int v)
{
    return v<7;
}
void f(list<int>& c)
{
    list<int>::iterator p = find_if(c.begin(),c.end(),less_than_7);
    // ...
}
```

There are many obvious uses for functions passed as arguments: logical predicates, arithmetic operations, operations for extracting information from elements, etc. It is neither convenient nor efficient to write a separate function for each use. Nor is a function logically sufficient to express all that we would like to express. Often, the function called for each element needs to keep data between invocations and to return the result of many applications. A member function of a class serves such needs better than a free-standing function does because its object can hold data. In addition, the class can provide operations for initializing and extracting such data.

Consider how to write a function – or rather a function-like class – to calculate a sum:

```
template<class T> class Sum {
    T res;
public:
    Sum(T i = 0) : res(i) { }              // initialize
    void operator()(T x) { res += x; }     // accumulate
    T result() const { return res; }       // return sum
};
```

Clearly, *Sum* is designed for arithmetic types for which initialization by *0* and += are defined. For example:

```
void f(list<double>& ld)
{
    Sum<double> s;
    s = for_each(ld.begin(), ld.end(), s);          // invoke s() for each element of ld
    cout << "the sum is" << s.result() << '\n';
}
```

Here, ***for_each***( ) (§18.5.1) invokes ***Sum<double>::operator***( )(***double***) for each element of ***ld*** and returns the object passed as its third argument.

The key reason this works is that ***for_each***( ) doesn't actually assume its third argument to be a function. It simply assumes that its third argument is something that can be called with an appropriate argument. A suitably-defined object serves as well as – and often better than – a function. For example, it is easier to inline the application operator of a class than to inline a function passed as a pointer to function. Consequently, function objects often execute faster than do ordinary functions. An object of a class with an application operator (§11.9) is called a *function-like object*, a *functor*, or simply a *function object*.

### 18.4.1 Function Object Bases [algo.bases]

The standard library provides many useful function objects. To aid the writing of function objects, the library provides a couple of base classes:

```
template <class Arg, class Res> struct unary_function {
    typedef Arg argument_type;
    typedef Res result_type;
};

template <class Arg, class Arg2, class Res> struct binary_function {
    typedef Arg first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Res result_type;
};
```

The purpose of these classes is to provide standard names for the argument and return types for use by users of classes derived from ***unary_function*** and ***binary_function***. Using these bases consistently the way the standard library does will save the programmer from discovering the hard way why they are useful (§18.4.4.1).

### 18.4.2 Predicates [algo.pred]

A predicate is a function object (or a function) that returns a ***bool***. For example, ***<functional>*** defines:

```
template <class T> struct logical_not : public unary_function<T, bool> {
    bool operator()(const T& x) const { return !x; }
};

template <class T> struct less : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x<y; }
};
```

Unary and binary predicates are often useful in combination with algorithms. For example, we can compare two sequences, looking for the first element of one that is not less than its corresponding element in the other:

```
void f(vector<int>& vi, list<int>& li)
{
    typedef list<int>::iterator LI;
    typedef vector<int>::iterator VI;
    pair<VI,LI> p1 = mismatch(vi.begin(),vi.end(),li.begin(),less<int>());
    // ...
}
```

The *mismatch()* algorithm applies its binary predicate repeatedly to pairs of corresponding elements until it fails (§18.5.4). It then returns the iterators for the elements that failed the comparison. Because an object is needed rather than a type, *less<int>()* (with the parentheses) is used rather than the tempting *less<int>*.

Instead of finding the first element ***not less*** than its corresponding element in the other sequence, we might like to find the first element ***less*** than its corresponding element. We can do this by presenting the sequences to *mismatch()* in the opposite order:

```
pair<LI,VI> p2 = mismatch(li.begin(),li.end(),vi.begin(),less<int>());
```

or we can use the complementary predicate *greater_equal*:

```
p1 = mismatch(vi.begin(),vi.end(),li.begin(),greater_equal<int>());
```

In §18.4.4.4, I show how to express the predicate ''not less.''

### 18.4.2.1 Overview of Predicates [algo.pred.std]

In *<functional>*, the standard library supplies a few common predicates:

| Predicates <functional> | | |
|---|---|---|
| *equal_to* | Binary | arg1==arg2 |
| *not_equal_to* | Binary | arg1!=arg2 |
| *greater* | Binary | arg1>arg2 |
| *less* | Binary | arg1<arg2 |
| *greater_equal* | Binary | arg1>=arg2 |
| *less_equal* | Binary | arg1<=arg2 |
| *logical_and* | Binary | arg1&&arg2 |
| *logical_or* | Binary | arg1\|\|arg2 |
| *logical_not* | Unary | !arg |

The definitions of ***less*** and ***logical_not*** are presented in §18.4.2.

In addition to the library-provided predicates, users can write their own. Such user-supplied predicates are essential for simple and elegant use of the standard libraries and algorithms. The ability to define predicates is particularly important when we want to use algorithms for classes designed without thought of the standard library and its algorithms. For example, consider a variant of the ***Club*** class from §10.4.6:

```
class Person { /* ... */ };

struct Club {
        string name;
        list<Person*> members;
        list<Person*> officers;
        // ...
        Club(const name& n);
};
```

Looking for a ***Club*** with a given name in a ***list<Club>*** is clearly a reasonable thing to do. However, the standard library algorithm ***find_if()*** doesn't know about ***Club***s. The library algorithms know how to test for equality, but we don't want to find a ***Club*** based on its complete value. Rather, we want to use ***Club::name*** as the key. So we write a predicate to reflect that:

```
class Club_eq : public unary_function<Club,bool> {
        string s;
public:
        explicit Club_eq(const string& ss) : s(ss) { }
        bool operator()(const Club& c) const { return c.name==s; }
};
```

Defining useful predicates is simple. Once suitable predicates have been defined for user-defined types, their use with the standard algorithms is as simple and efficient as examples involving containers of simple types. For example:

```
void f(list<Club>& lc)
{
        typedef list<Club>::iterator LCI;
        LCI p = find_if(lc.begin(),lc.end(),Club_eq("Dining Philosophers"));
        // ...
}
```

### 18.4.3 Arithmetic Function Objects [algo.arithmetic]

When dealing with numeric classes, it is sometimes useful to have the standard arithmetic functions available as function objects. Consequently, in ***<functional>*** the standard library provides:

| **Arithmetic Operations <functional>** | | |
|---|---|---|
| ***plus*** | Binary | arg1+arg2 |
| ***minus*** | Binary | arg1−arg2 |
| ***multiplies*** | Binary | arg1*arg2 |
| ***divides*** | Binary | arg1/arg2 |
| ***modulus*** | Binary | arg1%arg2 |
| ***negate*** | Unary | −arg |

We might use ***multiplies*** to multiply elements in two vectors, thereby producing a third:

```
void discount(vector<double>& a, vector<double>& b, vector<double>& res)
{
    transform(a.begin(), a.end(), b.begin(), back_inserter(res), multiplies<double>());
}
```

The *back_inserter()* is described in §19.2.4. A few numerical algorithms can be found in §22.6.

### 18.4.4 Binders, Adapters, and Negaters [algo.adapter]

We can use predicates and arithmetic function objects we have written ourselves and rely on the ones provided by the standard library. However, when we need a new predicate we often find that the new predicate is a minor variation of an existing one. The standard library supports the composition of function objects:

- §18.4.4.1 A *binder* allows a two-argument function object to be used as a single-argument function by binding one argument to a value.
- §18.4.4.2 A *member function adapter* allows a member function to be used as an argument to algorithms.
- §18.4.4.3 A *pointer to function adapter* allows a pointer to function to be used as an argument to algorithms.
- §18.4.4.4 A *negater* allows us to express the opposite of a predicate.

Collectively, these function objects are referred to as ***adapters***. These adapters all have a common structure relying on the function object bases ***unary_function*** and ***binary_function*** (§18.4.1). For each of these adapters, a helper function is provided to take a function object as an argument and return a suitable function object. When invoked by its ***operator()()***, that function object will perform the desired action. That is, an adapter is a simple form of a higher-order function: it takes a function argument and produces a new function from it:

| **Binders, Adapters, and Negaters <functional>** | | |
|---|---|---|
| bind2nd(y) | binder2nd | Call binary function with ***y*** as 2nd argument. |
| ***bind1st(x)*** | ***binder1st*** | Call binary function with ***x*** as 1st argument. |
| ***mem_fun()*** | ***mem_fun_t*** | Call 0-arg member through pointer. |
| | ***mem_fun1_t*** | Call unary member through pointer. |
| | ***const_mem_fun_t*** | Call 0-arg const member through pointer. |
| | ***const_mem_fun1_t*** | Call unary const member through pointer. |
| ***mem_fun_ref()*** | ***mem_fun_ref_t*** | Call 0-arg member through reference. |
| | ***mem_fun1_ref_t*** | Call unary member through reference. |
| | ***const_mem_fun_ref_t*** | Call 0-arg const member through reference. |
| | ***const_mem_fun1_ref_t*** | Call unary const member through reference. |
| ***ptr_fun()*** | ***pointer_to_unary_function*** | Call unary pointer to function. |
| ***ptr_fun()*** | ***pointer_to_binary_function*** | Call binary pointer to function. |
| ***not1()*** | ***unary_negate*** | Negate unary predicate. |
| ***not2()*** | ***binary_negate*** | Negate binary predicate. |

### 18.4.4.1 Binders [algo.binder]

Binary predicates such as *less* (§18.4.2) are useful and flexible. However, we soon discover that the most useful kind of predicate is one that compares a fixed argument repeatedly against a container element. The *less_than_7()* function (§18.4) is a typical example. The *less* operation needs two arguments explicitly provided in each call, so it is not immediately useful. Instead, we might define:

```
template <class T> class less_than : public unary_function<T,bool> {
    T arg2;
public:
    explicit less_than(const T& x) : arg2(x) { }
    bool operator()(const T& x) const { return x<arg2; }
};
```

We can now write:

```
void f(list<int>& c)
{
    list<int>::const_iterator p = find_if(c.begin(),c.end(),less_than<int>(7));
    // ...
}
```

We must write *less_than<int>(7)* rather than *less_than(7)* because the template argument *<int>* cannot be deduced from the type of the constructor argument *(7)* (§13.3.1).

The *less_than* predicate is generally useful. Importantly, we defined it by fixing or binding the second argument of *less*. Such composition by binding an argument is so common, useful, and occasionally tedious that the standard library provides a standard class for doing it:

```
template <class BinOp>
class binder2nd : public unary_function<BinOp::first_argument_type, BinOp::result_type> {
protected:
    BinOp op;
    typename BinOp::second_argument_type arg2;
public:
    binder2nd(const BinOp& x, const typename BinOp::second_argument_type& v)
        : op(x), arg2(v) { }
    result_type operator()(const argument_type& x) const { return op(x,arg2); }
};

template <class BinOp, class T> binder2nd<BinOp> bind2nd(const BinOp& op, const T& v)
{
    return binder2nd<BinOp>(op,v);
}
```

For example, we can use *bind2nd()* to create the unary predicate ''less than *7*'' from the binary predicate ''less'' and the value *7*:

```
void f(list<int>& c)
{
    list<int>::const_iterator p = find_if(c.begin(),c.end(),bind2nd(less<int>(),7));
    // ...
}
```

Is this readable? Is this efficient? Given an average C++ implementation, this version is actually more efficient in time and space than is the original version using the function ***less_than_7()*** from §18.4! The comparison is easily inlined.

The notation is logical, but it does take some getting used to. Often, the definition of a named operation with a bound argument is worthwhile after all:

```
template <class T> struct less_than : public binder2nd<less<T>,T> {
    explicit less_than(const T& x) : binder2nd(less<T>(),x) { }
};

void f(list<int>& c)
{
    list<int>::const_iterator p = find_if(c.begin(),c.end(),less_than<int>(7));
    // ...
}
```

It is important to define ***less_than*** in terms of ***less*** rather than using < directly. That way, ***less_than*** benefits from any specializations that ***less*** might have (§13.5, §19.2.2).

In parallel to ***bind2nd()*** and ***binder2nd***, ***<functional>*** provides ***bind1st()*** and ***binder1st*** for binding the first argument of a binary function.

By binding an argument, ***bind1st()*** and ***bind2nd()*** perform a service very similar to what is commonly referred to as *Currying*.

### 18.4.4.2 Member Function Adapters [algo.memfct]

Most algorithms invoke a standard or user-defined operation. Naturally, users often want to invoke a member function. For example (§3.8.5):

```
void draw_all(list<Shape*>& lsp)
{
    for_each(c.begin(),c.end(),&Shape::draw);  // oops! error
}
```

The problem is that a member function ***mf()*** needs to be invoked for an object: ***p->mf()***. However, algorithms such as ***for_each()*** invoke their function operands by simple application: ***f()***. Consequently, we need a convenient and efficient way of creating something that allows an algorithm to invoke a member function. The alternative would be to duplicate the set of algorithms: one version for member functions plus one for ordinary functions. Worse, we'd need additional versions of algorithms for containers of objects (rather than pointers to objects). As for the binders (§18.4.4.1), this problem is solved by a class plus a function. First, consider the common case in which we want to call a member function taking no arguments for the elements of a container of pointers:

```
template<class R, class T> class mem_fun_t : public unary_function<T*,R> {
    R (T::*pmf)();
public:
    explicit mem_fun_t(R (T::*p)()) :pmf(p) {}
    R operator()(T* p) const { return (p->*pmf)(); } // call through pointer
};

template<class R, class T> mem_fun_t<R,T> mem_fun(R (T::*f)())
{
    return mem_fun_t<R,T>(f);
}
```

This handles the `Shape::draw()` example:

```
void draw_all(list<Shape*>& lsp)    // call 0-argument member through pointer to object
{
    for_each(lsp.begin(), lsp.end(), mem_fun(&Shape::draw));    // draw all shapes
}
```

In addition, we need a class and a `mem_fun()` function for handling a member function taking an argument. We also need versions to be called directly for an object rather than through a pointer; these are named `mem_fun_ref()`. Finally, we need versions for `const` member functions:

```
template<class R, class T> mem_fun_t<R,T> mem_fun(R (T::*f)());
// and versions for unary member, for const member, and const unary member (see table in §18.4.4)

template<class R, class T> mem_fun_ref_t<R,T> mem_fun_ref(R (T::*f)());
// and versions for unary member, for const member, and const unary member (see table in §18.4.4)
```

Given these member function adapters from `<functional>`, we can write:

```
void f(list<string>& ls)    // use member function that takes no argument for object
{
    typedef list<string>::iterator LSI;
    LSI p = find_if(ls.begin(), ls.end(), mem_fun_ref(&string::empty)); // find ""
}

void rotate_all(list<Shape*>& ls, int angle)
    // use member function that takes one argument through pointer to object
{
    for_each(ls.begin(), ls.end(), bind2nd(mem_fun(&Shape::rotate), angle));
}
```

The standard library need not deal with member functions taking more than one argument because no standard library algorithm takes a function with more than two arguments as operands.

### 18.4.4.3 Pointer to Function Adapters [algo.ptof]

An algorithm doesn't care whether a ''function argument'' is a function, a pointer to function, or a function object. However, a binder (§18.4.4.1) does care because it needs to store a copy for later use. Consequently, the standard library supplies two adapters to allow pointers to functions to be used together with the standard algorithms in `<functional>`. The definition and implementation

closely follows that of the member function adapters (§18.4.4.2). Again, a pair of functions and a pair of classes are used:

```
template <class A, class R> pointer_to_unary_function<A,R> ptr_fun(R (*f)(A));

template <class A, class A2, class R>
    pointer_to_binary_function<A,A2,R> ptr_fun(R (*f)(A, A2));
```

Given these pointer to function adapters, we can use ordinary functions together with binders:

```
class Record { /* ... */ };

bool name_key_eq(const Record&, const Record&);  // compare based on names
bool ssn_key_eq(const Record&, const Record&);   // compare based on number

void f(list<Record>& lr)   // use pointer to function
{
    typedef typename list<Record>::iterator LI;
    LI p = find_if(lr.begin(),lr.end(),bind2nd(ptr_fun(name_key_eq),"John Brown"));
    LI q = find_if(lr.begin(),lr.end(),bind2nd(ptr_fun(ssn_key_eq),1234567890));
    // ...
}
```

This looks for elements of the list *lr* that match the keys *John Brown* and *1234567890*.

#### 18.4.4.4 Negaters [algo.negate]

The predicate negaters are related to the binders in that they take an operation and produce a related operation from it. The definition and implementation of negaters follow the pattern of the member function adapters (§18.4.4.2). Their definitions are trivial, but their simplicity is obscured by the use of long standard names:

```
template <class Pred>
class unary_negate : public unary_function<typename Pred::argument_type,bool> {

    unary_function<argument_type,bool> op;
public:
    explicit unary_negate(const Pred& p) : op(p) { }
    bool operator()(const argument_type& x) const { return !op(x); }
};

template <class Pred>
class binary_negate : public binary_function<typename Pred::first_argument_type,
                                             typename Pred::second_argument_type, bool> {

    typedef first_argument_type Arg;
    typedef second_argument_type Arg2;

    binary_function<Arg, Arg2, bool> op;
public:
    explicit binary_negate(const Pred& p) : op(p) { }
    bool operator()(const Arg& x, const Arg2& y) const { return !op(x,y); }
};
```

```
template<class Pred> unary_negate<Pred> not1(const Pred& p);      // negate unary
template<class Pred> binary_negate<Pred> not2(const Pred& p);     // negate binary
```

These classes and functions are declared in *<functional>*. The names *first_argument_type*, *second_argument_type*, etc., come from the standard base classes *unary_function* and *binary_function*.

Like the binders, the negaters are most conveniently used indirectly through their helper functions. For example, we can express the binary predicate ''not less than'' and use it to find the first corresponding pair of elements whose first element is greater than or equal to its second:

```
void f(vector<int>& vi, list<int>& li) // revised example from §18.4.2
{
    // ...
    p1 = mismatch(vi.begin(), vi.end(), li.begin(), not2(less<int>()));
    // ...
}
```

That is, *p1* identifies the first pair of elements for which the predicate *not less than* failed.

Predicates deal with Boolean conditions, so there are no equivalents to the bitwise operators |, &, ^, and ~.

Naturally, binders, adapters, and negaters are useful in combination. For example:

```
extern "C" int strcmp(const char*, const char*);    // from <cstdlib>

void f(list<char*>& ls)    // use pointer to function
{
    typedef typename list<char*>::const_iterator LI;
    LI p = find_if(ls.begin(), ls.end(), not1(bind2nd(ptr_fun(strcmp), "funny")));
}
```

This finds an element of the list *ls* that contains the C-style string *"funny"*. The negater is needed because *strcmp()* returns *0* when strings compare equal.

## 18.5 Nonmodifying Sequence Algorithms [algo.nonmodifying]

Nonmodifying sequence algorithms are the basic means for finding something in a sequence without writing a loop. In addition, they allow us to find out things about elements. These algorithms can take const-iterators (§19.2.1) and – with the excetion of *for_each()* – should not be used to invoke operations that modify the elements of the sequence.

### 18.5.1 For_each [algo.foreach]

We use a library to benefit from the work of others. Using a library function, class, algorithm, etc., saves the work of inventing, designing, writing, debugging, and documenting something. Using the standard library also makes the resulting code easier to read for others who are familiar with that library, but who would have to spend time and effort understanding home-brewed code.

A key benefit of the standard library algorithms is that they save the programmer from writing explicit loops. Loops can be tedious and error-prone. The *for_each()* algorithm is the simplest

algorithm in the sense that it does nothing but eliminate an explicit loop. It simply calls its operator argument for a sequence:

```
template<class In, class Op> Op for_each(In first, In last, Op f)
{
    while (first != last) f(*first++);
    return f;
}
```

What functions would people want to call this way? If you want to accumulate information from the elements, consider ***accumulate***() (§22.6). If you want to find something in a sequence, consider ***find***() and ***find_if***() (§18.5.2). If you change or remove elements, consider ***replace***() (§18.6.4) or ***remove***() (§18.6.5). In general, before using ***for_each***(), consider if there is a more specialized algorithm that would do more for you.

The result of ***for_each***() is the function or function object passed as its third argument. As shown in the ***Sum*** example (§18.4), this allows information to be passed back to a caller.

One common use of ***for_each***() is to extract information from elements of a sequence. For example, consider collecting the names of any of a number of ***Club***s:

```
void extract(const list<Club>& lc, list<Person*>& off) // place the officers from 'lc' on 'off'
{
    for_each(lc.begin(), lc.end(), Extract_officers(off));
}
```

In parallel to the examples from §18.4 and §18.4.2, we define a function class that extracts the desired information. In this case, the names to be extracted are found in ***list<Person*>***s in our ***list<Club>***. Consequently, ***Extract_officers*** needs to copy the officers from a ***Club***'s ***officers*** list to our list:

```
class Extract_officers {
    list<Person*>& lst;
public:
    explicit Extract_officers(list<Person*>& x) : lst(x) { }

    void operator()(const Club& c)
        { copy(c.officers.begin(), c.officers.end(), back_inserter(lst)); }
};
```

We can now print out the names, again using ***for_each***():

```
void extract_and_print(const list<Club>& lc)
{
    list<Person*> off;
    extract(lc, off);
    for_each(off.begin(), off.end(), Print_name(cout));
}
```

Writing ***Print_name*** is left as an exercise (§18.13[4]).

The ***for_each***() algorithm is classified as nonmodifying because it doesn't explicitly modify a sequence. However, if applied to a non-***const*** sequence ***for_each***()'s operation (its third argument) may change the elements of the sequence. For an example, see ***delete_ptr***() in §18.6.2.

### 18.5.2 The Find Family [algo.find]

The *find*( ) algorithms look through a sequence or a pair of sequences to find a value or a match on a predicate. The simple versions of *find*( ) look for a value or for a match with a predicate:

```
template<class In, class T> In find(In first, In last, const T& val);

template<class In, class Pred> In find_if(In first, In last, Pred p);
```

The algorithms *find*( ) and *find_if*( ) return an iterator to the first element that matches a value and a predicate, respectively. In fact, *find*( ) can be understood as the version of *find_if*( ) with the predicate ==. Why aren't they both called *find*( )? The reason is that function overloading cannot always distinguish calls of two template functions with the same number of arguments. Consider:

```
bool pred(int);

void f(vector<bool(*f)(int)>& v1, vector<int>& v2)
{
    find(v1.begin(),v1.end(),pred);        // find 'pred'
    find_if(v2.begin(),v2.end(),pred);     // find int for which pred() returns true
}
```

If *find*( ) and *find_if*( ) had had the same name, surprising ambiguities would have resulted. In general, the *_if* suffix is used to indicate that an algorithm takes a predicate.

The *find_first_of*( ) algorithm finds the first element of a sequence that has a match in a second sequence:

```
template<class For, class For2>
    For find_first_of(For first, For last, For2 first2, For2 last2);

template<class For, class For2, class BinPred>
    For find_first_of(For first, For last, For2 first2, For2 last2, BinPred p);
```

For example:

```
int x[] = { 1,3,4 };
int y[] = { 0,2,3,4,5};

void f()
{
    int* p = find_first_of(x,x+3,y,y+5);       // p = &x[1]
    int* q = find_first_of(p+1,x+3,y,y+5);     // q = &x[2]
}
```

The pointer *p* will point to *x*[*1*] because *3* is the first element of *x* with a match in *y*. Similarly, *q* will point to *x*[*2*].

The *adjacent_find*( ) algorithm finds a pair of adjacent matching values:

```
template<class For> For adjacent_find(For first, For last);

template<class For, class BinPred> For adjacent_find(For first, For last, BinPred p);
```

The return value is an iterator to the first matching element. For example:

```
void f(vector<string>& text)
{
    vector<string>::iterator p = adjacent_find(text.begin(), text.end(), "the");
    if (p != text.end()) {
        // I duplicated "the" again!
    }
}
```

### 18.5.3 Count [algo.count]

The *count*() and *count_if*() algorithms count occurrences of a value in a sequence:

```
template<class In, class T>
    iterator_traits<In>::difference_type count(In first, In last, const T& val);

template<class In, class Pred>
    iterator_traits<In>::difference_type count_if(In first, In last, Pred p);
```

The return type of *count*() is interesting. Consider an obvious and somewhat simple-minded version of *count*():

```
template<class In, class T> int count(In first, In last, const T& val)
{
    int res = 0;
    while (first != last) if (*first++ == val) ++res;
    return res;
}
```

The problem is that an *int* might not be the right type for the result. On a machine with small *int*s, there might be too many elements in the sequence for *count*() to fit in an *int*. Conversely, a high-performance implementation on a specialized machine might prefer to keep the count in a *short*.

Clearly, the number of elements in the sequence cannot be larger than the maximum difference between its iterators (§19.2.1). Consequently, the first idea for a solution to this problem is to define the return type as

```
typename In::difference_type
```

However, a standard algorithm should be applicable to built-in arrays as well as to standard containers. For example:

```
void f(const char* p, int size)
{
    int n = count(p,p+size,´e´);    // count the number of occurrences of the letter 'e'
}
```

Unfortunately, *int**::*difference_type* is not valid C++. This problem is solved by partial specialization of an *iterator_traits* (§19.2.2).

### 18.5.4 Equal and Mismatch [algo.equal]

The *equal*( ) and *mismatch*( ) algorithms compare two sequences:

```
template<class In, class In2> bool equal(In first, In last, In2 first2);

template<class In, class In2, class BinPred>
    bool equal(In first, In last, In2 first2, BinPred p);

template<class In, class In2> pair<In, In2> mismatch(In first, In last, In2 first2);

template<class In, class In2, class BinPred>
    pair<In, In2> mismatch(In first, In last, In2 first2, BinPred p);
```

The *equal*( ) algorithm simply tells whether all corresponding pairs of elements of two sequences compare equal; *mismatch*( ) looks for the first pair of elements that compares unequal and returns iterators to those elements. No end is specified for the second sequence; that is, there is no *last2*. Instead, it is assumed that there are at least as many elements in the second sequence as in the first and *first2*+(*last*-*first*) is used as *last2*. This technique is used throughout the standard library, where pairs of sequences are used for operations on pairs of elements.

As shown in §18.5.1, these algorithms are even more useful than they appear at first glance because the user can supply predicates defining what it means to be equal and to match.

Note that the sequences need not be of the same type. For example:

```
void f(list<int>& li, vector<double>& vd)
{
    bool b = equal(li.begin(), li.end(), vd.begin());
}
```

All that is required is that the elements be acceptable as operands of the predicate.

The two versions of *mismatch*( ) differ only in their use of predicates. In fact, we could implement them as one function with a default template argument:

```
template<class In, class In2, class BinPred>
pair<In, In2> mismatch(In first, In last, In2 first2,
                       BinPred p = equal_to<In::value_type, In2::value_type>())// §18.4.2.1
{
    while (first != last && p(*first, *first2)) {
        ++first;
        ++first2;
    }
    return pair<In, In2>(first, first2);
}
```

The difference between having two functions and having one with a default argument can be observed by someone taking pointers to functions. However, thinking of many of the variants of the standard algorithms as simply ''the version with the default predicate'' roughly halves the number of template functions that need to be remembered.

### 18.5.5 Search [algo.search]

The *search*( ), *search_n*( ), and *find_end*( ) algorithms find one sequence as a subsequence in another:

```
template<class For, class For2>
    For search(For first, For last, For2 first2, For2 last2);

template<class For, class For2, class BinPred>
    For search(For first, For last, For2 first2, For2 last2, BinPred p);

template<class For, class For2>
    For find_end(For first, For last, For2 first2, For2 last2);

template<class For, class For2, class BinPred>
    For find_end(For first, For last, For2 first2, For2 last2, BinPred p);

template<class For, class Size, class T>
    For search_n(For first, For last, Size n, const T& val);

template<class For, class Size, class T, class BinPred>
    For search_n(For first, For last, Size n, const T& val, BinPred p);
```

The *search*( ) algorithm looks for its second sequence as a subsequence of its first. If that second sequence is found, an iterator for the first matching element in the first sequence is returned. The end of sequence (*last*) is returned to represent ''not found.'' Thus, the return value is always in the [*first*,*last*] sequence. For example:

```
string quote("Why waste time learning, when ignorance is instantaneous?");

bool in_quote(const string& s)
{
    char* p = search(quote.begin(), quote.end(), s.begin(), s.end()); // find s in quote
    return p!=quote.end();
}

void g()
{
    bool b1 = in_quote("learning");     // b1 = true
    bool b2 = in_quote("lemming");      // b2 = false
}
```

Thus, *search*( ) is an operation for finding a substring generalized to all sequences. This implies that *search*( ) is a very useful algorithm.

The *find_end*( ) algorithm looks for its second input sequence as a subsequence of its first input sequence. If that second sequence is found, *find_end*( ) returns an iterator pointing to the last match in its first input. In other words, *find_end*( ) is *search*( ) ''backwards.'' It finds the last occurrence of its second input sequence in its first input sequence, rather than the first occurrence of its second sequence.

The *search_n*( ) algorithm finds a sequence of at least *n* matches for its *value* argument in the sequence. It returns an iterator to the first element of the sequence of *n* matches.

## 18.6 Modifying Sequence Algorithms [algo.modifying]

If you want to change a sequence, you can explicitly iterate through it. You can then modify values. Wherever possible, however, we prefer to avoid this kind of programming in favor of simpler and more systematic styles of programming. The alternative is algorithms that traverse sequences performing specific tasks. The nonmodifying algorithms (§18.5) serve this need when we just read from the sequence. The modifying sequence algorithms are provided to do the most common forms of updates. Some update a sequence, while others produce a new sequence based on information found during a traversal.

Standard algorithms work on data structures through iterators. This implies that inserting a new element into a container or deleting one is not easy. For example, given only an iterator, how can we find the container from which to remove the element pointed to? Unless special iterators are used (e.g., inserters, §3.8, §19.2.4), operations through iterators do not change the size of a container. Instead of inserting and deleting elements, the algorithms change the values of elements, swap elements, and copy elements. Even *remove*() operates by overwriting the elements to be removed (§18.6.5). In general, the fundamental modifying operations produce outputs that are modified copies of their inputs. The algorithms that appear to modify a sequence are variants that copy within a sequence.

### 18.6.1 Copy [algo.copy]

Copying is the simplest way to produce one sequence from another. The definitions of the basic copy operations are trivial:

```
template<class In, class Out> Out copy(In first, In last, Out res)
{
    while (first != last) *res++ = *first++;
    return res;
}

template<class Bi, class Bi2> Bi2 copy_backward(Bi first, Bi last, Bi2 res)
{
    while (first != last) *--res = *--last;
    return res;
}
```

The target of a copy algorithm need not be a container. Anything that can be described by an output iterator (§19.2.6) will do. For example:

```
void f(list<Club>& lc, ostream& os)
{
    copy(lc.begin(), lc.end(), ostream_iterator<Club>(os));
}
```

To read a sequence, we need a sequence describing where to begin and where to end. To write, we need only an iterator describing where to write to. However, we must take care not to write beyond the end of the target. One way to ensure that we don't do this is to use an inserter (§19.2.4) to grow the target as needed. For example:

```
void f(vector<char>& vs)
{
    vector<char> v;

    copy(vs.begin(),vs.end(),v.begin());          // might overwrite end of v
    copy(vs.begin(),vs.end(),back_inserter(v)); // add elements from vs to end of v
}
```

The input sequence and the output sequence may overlap. We use *copy()* when the sequences do not overlap or if the end of the output sequence is in the input sequence. We use *copy_backward()* when the beginning of the output sequence is in the input sequence. In that way, no element is overwritten until after it has been copied. See also §18.13[13].

Naturally, to copy something backwards we need a bidirectional iterator (§19.2.1) for both the input and the output sequences. For example:

```
void f(vector<char>& vc)
{
    vector<char> v(vc.size());

    copy_backward(vc.begin(),vc.end(),output_iterator<char>(cout)); // error

    copy_backward(vc.begin(),vc.end(),v.end());                     // ok
    copy(v.begin(),v.end(),ostream_iterator<char>(os));

}
```

Often, we want to copy only elements that fulfill some criterion. Unfortunately, *copy_if()* was somehow dropped from the set of algorithms provided by the standard library (mea culpa). On the other hand, it is trivial to define:

```
template<class In, class Out, class Pred> Out copy_if(In first, In last, Out res, Pred p)
{
    while (first != last) {
        if (p(*first)) *res++ = *first;
        ++first;
    }
    return res;
}
```

Now if we want to print elements with a value larger than *n*, we can do it like this:

```
void f(list<int>&ld, int n, ostream& os)
{
    copy_if(ld.begin(),ld.end(),ostream_iterator<int>(os),bind2nd(greater<int>(),n));
}
```

See also *remove_copy_if()* (§18.6.5).

### 18.6.2 Transform [algo.transform]

Somewhat confusingly, *transform()* doesn't necessarily change its input. Instead, it produces an output that is a transformation of its input based on a user-supplied operation:

```
template<class In, class Out, class Op>
Out transform(In first, In last, Out res, Op op)
{
    while (first != last) *res++ = op(*first++);
    return res;
}

template<class In, class In2, class Out, class BinOp>
Out transform(In first, In last, In2 first2, Out res, BinOp op)
{
    while (first != last) *res++ = op(*first++, *first2++);
    return res;
}
```

The ***transform()*** that reads a single sequence to produce its output is rather similar to ***copy()***. Instead of writing its element, it writes the result of its operation on that element. Thus, we could have defined ***copy()*** as ***transform()*** with an operation that returns its argument:

```
template<class T> T identity(const T& x) { return x; }

template<class In, class Out> Out copy(In first, In last, Out res)
{
    return transform(first, last, res, identity);
}
```

Another way to view ***transform()*** is as a variant of ***for_each*** that explicitly produces output. For example, we can produce a list of name ***string***s from a list of ***Club***s using ***transform()***:

```
string nameof(const Club& c)    // extract name string
{
    return c.name;
}

void f(list<Club>& lc)
{
    transform(lc.begin(), lc.end(), ostream_iterator<string>(cout), nameof);
}
```

One reason ***transform()*** is called ''transform'' is that the result of the operation is often written back to where the argument came from. As an example, consider deleting the objects pointed to by a set of pointers:

```
template<class T> T* delete_ptr(T* p) { delete p; return 0; }

void purge(deque<Shape*>& s)
{
    transform(s.begin(), s.end(), s.begin(), delete_ptr);
    // ...
}
```

The ***transform()*** algorithm always produces an output sequence. Here, I directed the result back to the input sequence so that ***delete_ptr(p)*** has the effect ***p=delete_ptr(p)***. This was why I chose to return ***0*** from ***delete_ptr()***.

The *transform*( ) algorithm that takes two sequences allows people to combine information from two sources. For example, an animation may have a routine that updates the position of a list of shapes by applying a translation:

```
Shape* move_shape(Shape* s, Point p)    // *s += p
{
    s->move_to(s->center()+p);
    return s;
}

void update_positions(list<Shape*>& ls, vector<Point>& oper)
{
    // invoke operation on corresponding object:
    transform(ls.begin(), ls.end(), oper.begin(), ls.begin(), move_shape);
}
```

I didn't really want to produce a return value from *move_shape*( ). However, *transform*( ) insists on assigning the result of its operation, so I let *move_shape*( ) return its first operand so that I could write it back to where it came from.

Sometimes, we do not have the freedom to do that. For example, an operation that I didn't write and don't want to modify might not return a value. Sometimes, the input sequence is *const*. In such cases, we might define a two-sequence *for_each*( ) to match the two-sequence *transform*( ):

```
template<class In, class In2, class BinOp>
BinOp for_each(In first, In last, In2 first2, BinOp op)
{
    while (first != last) op(*first++, *first2++);
    return op;
}

void update_positions(list<Shape*>& ls, vector<Point>& oper)
{
    for_each(ls.begin(), ls.end(), oper.begin(), move_shape);
}
```

At other times, it can be useful to have an output iterator that doesn't actually write anything (§19.6[2]).

There are no standard library algorithms that read three or more sequences. Such algorithms are easily written, though. Alternatively, you can use *transform*( ) repeatedly.

### 18.6.3 Unique [algo.unique]

Whenever information is collected, duplication can occur. The *unique*( ) and *unique_copy*( ) algorithms eliminate adjacent duplicate values:

```
template<class For> For unique(For first, For last);
template<class For, class BinPred> For unique(For first, For last, BinPred p);

template<class In, class Out> Out unique_copy(In first, In last, Out res);
template<class In, class Out, class BinPred>
    Out unique_copy(In first, In last, Out res, BinPred p);
```

The ***unique***( ) algorithm eliminates adjacent duplicates from a sequence, ***unique_copy***( ) makes a copy without duplicates. For example:

```
void f(list<string>& ls, vector<string>& vs)
{
    ls.sort(); // list sort (§17.2.2.1)
    unique_copy(ls.begin(),ls.end(),back_inserter(vs));
}
```

This copies *ls* to *vs*, eliminating duplicates in the process. The ***sort***( ) is needed to get equal strings adjacent.

Like other standard algorithms, ***unique***( ) operates on iterators. It has no way of knowing the type of container these iterators point into, so it cannot modify that container. It can only modify the values of the elements. This implies that ***unique***( ) does not eliminate duplicates from its input sequence in the way we naively might expect. Rather, it moves unique elements towards the front (head) of a sequence and returns an iterator to the end of the subsequence of unique elements:

```
template <class For> For unique(For first, For last)
{
    first = adjacent_find(first,last);      // §18.5.2
    return unique_copy(first,last,first);
}
```

The elements after the unique subsequence are left unchanged. Therefore, this does not eliminate duplicates in a vector:

```
void f(vector<string>& vs)      // warning: bad code!
{
    sort(vs.begin(),vs.end());          // sort vector
    unique(vs.begin(),vs.end());        // eliminate duplicates (no it doesn't!)
}
```

In fact, by moving the last elements of a sequence forward to eliminate duplicates, ***unique***( ) can introduce new duplicates. For example:

```
int main()
{
    char v[] = "abbcccde";

    char* p = unique(v,v+strlen(v));
    cout << v << ' ' << p-v << '\n';
}
```

produced

```
abcdecde 5
```

That is, *p* points to the second *c*.

Algorithms that might have removed elements (but can't) generally come in two forms: the ''plain'' version that reorders elements in a way similar to ***unique***( ) and a version that produces a new sequence in a way similar to ***unique_copy***( ). The ***_copy*** suffix is used to distinguish these two kinds of algorithms.

To eliminate duplicates from a container, we must explicitly shrink it:

```
template<class C> void eliminate_duplicates(C& c)
{
    sort(c.begin(),c.end());                                  // sort
    typename C::iterator p = unique(c.begin(),c.end());       // compact
    c.erase(p,c.end());                                       // shrink
}
```

Note that *eliminate_duplicates*() would make no sense for a built-in array, yet *unique*() can still be applied to arrays.

An example of *unique_copy*() can be found in §3.8.3.

#### 18.6.3.1 Sorting Criteria [algo.criteria]

To eliminate all duplicates, the input sequences must be sorted (§18.7.1). Both *unique*() and *unique_copy*() use == as the default criterion for comparison and allow the user to supply alternative criteria. For instance, we might modify the example from §18.5.1 to eliminate duplicate names. After extracting the names of the *Club* officers, we were left with a *list<Person*>* called *off* (§18.5.1). We could eliminate duplicates like this:

```
eliminate_duplicates(off);
```

However, this relies on sorting pointers and assumes that each pointer uniquely identifies a person. In general, we would have to examine the *Person* records to determine whether we would consider them equal. We might write:

```
bool operator==(const Person& x, const Person& y) // equality for object
{
    // compare x and y for equality
}

bool operator<(const Person& x, const Person& y)  // less than for object
{
    // compare x and y for order
}

bool Person_eq(const Person* x, const Person* y)  // equality through pointer
{
    return *x == *y;
}

bool Person_lt(const Person* x, const Person* y)  // less than through pointer
{
    return *x < *y;
}
```

```
void extract_and_print(const list<Club>& lc)
{
    list<Person*> off;
    extract(lc, off);
    sort(off.begin(), off.end(), Person_lt);
    list<Club>::iterator p = unique(off.begin(), off.end(), Person_eq);
    for_each(off.begin(), p, Print_name(cout));
}
```

It is wise to make sure that the criterion used to sort matches the one used to eliminate duplicates. The default meanings of < and == for pointers are rarely useful as comparison criteria for the objects pointed to.

### 18.6.4 Replace [algo.replace]

The ***replace*( )** algorithms traverse a sequence, replacing values by other values as specified. They follow the patterns outlined by ***find*/*find_if*** and ***unique*/*unique_copy***, thus yielding four variants in all. Again, the code is simple enough to be illustrative:

```
template<class For, class T>
void replace(For first, For last, const T& val, const T& new_val)
{
    while (first != last) {
        if (*first == val) *first = new_val;
        ++first;
    }
}

template<class For, class Pred, class T>
void replace_if(For first, For last, Pred p, const T& new_val)
{
    while (first != last) {
        if (p(*first)) *first = new_val;
        ++first;
    }
}

template<class In, class Out, class T>
Out replace_copy(In first, In last, Out res, const T& val, const T& new_val)
{
    while (first != last) {
        *res++ = (*first == val) ? new_val : *first;
        ++first;
    }
    return res;
}
```

```
template<class In, class Out, class Pred, class T>
Out replace_copy_if(In first, In last, Out res, Pred p, const T& new_val)
{
    while (first != last) {
        *res++ = p(*first) ? new_val : *first;
        ++first;
    }
    return res;
}
```

We might want to go through a list of ***string***s, replacing the usual English transliteration of the name of my home town Aarhus with its proper name Århus:

```
void f(list<string>& towns)
{
    replace(towns.begin(), towns.end(), "Aarhus", "Århus");
}
```

This relies on an extended character set (§C.3.3).

### 18.6.5 Remove [algo.remove]

The ***remove***( ) algorithms remove elements from a sequence based on a value or a predicate:

```
template<class For, class T> For remove(For first, For last, const T& val);

template<class For, class Pred> For remove_if(For first, For last, Pred p);

template<class In, class Out, class T>
    Out remove_copy(In first, In last, Out res, const T& val);

template<class In, class Out, class Pred>
    Out remove_copy_if(In first, In last, Out res, Pred p);
```

Assuming that a ***Club*** has an address, we could produce a list of ***Club***s located in Copenhagen:

```
class located_in {
    string town;
public:
    located_in(const string& ss) :town(ss) { }
    bool operator()(const Club& c) const { return c.town == town; }
};

void f(list<Club>& lc)
{
    remove_copy_if(lc.begin(), lc.end(),
                   output_iterator<Club>(cout), not1(located_in("København")));
}
```

Thus, ***remove_copy_if***( ) is ***copy_if***( ) (§18.6.1) with the inverse condition. That is, an element is placed on the output by ***remove_copy_if***( ) if the element does not match the predicate.

The ''plain'' ***remove***( ) compacts non-matching elements at the beginning of the sequence and returns an iterator for the end of the compacted sequence (see also §18.6.3).

### 18.6.6 Fill and Generate [algo.fill]

The *fill*( ) and *generate*( ) algorithms exist to systematically assign values to sequences:

```
template<class For, class T> void fill(For first, For last, const T& val);
template<class Out, class Size, class T> void fill_n(Out res, Size n, const T& val);

template<class For, class Gen> void generate(For first, For last, Gen g);
template<class Out, class Size, class Gen> void generate_n(Out res, Size n, Gen g);
```

The *fill*( ) algorithm assigns a specified value; the *generate*( ) algorithm assigns values obtained by calling its function argument repeatedly. Thus, *fill*( ) is simply the special case of *generate*( ) in which the generator function returns the same value repeatedly. The *_n* versions assign to the first *n* elements of the sequence.

For example, using the random-number generators *Randint* and *Urand* from §22.7:

```
int v1[900];
int v2[900];
vector v3;

void f()
{
    fill(v1,&v1[900],99);                 // set all elements of v1 to 99
    generate(v2,&v2[900],Randint);        // set to random values (§22.7)

    // output 200 random integers in the interval [0..99]:
    generate_n(ostream_iterator<int>(cout),200,Urand(100));

    fill_n(back_inserter(v3),20,99);      // add 20 elements with the value 99 to v3
}
```

The *generate*( ) and *fill*( ) functions assign rather than initialize. If you need to manipulate raw storage, say to turn a region of memory into objects of well-defined type and state, you must use an algorithm like *uninitialized_fill*( ) from <*memory*> (§19.4.4) rather than algorithms from <*algorithm*>.

### 18.6.7 Reverse and Rotate [algo.reverse]

Occasionally, we need to reorder the elements of a sequence:

```
template<class Bi> void reverse(Bi first, Bi last);
template<class Bi, class Out> Out reverse_copy(Bi first, Bi last, Out res);

template<class For> void rotate(For first, For middle, For last);
template<class For, class Out> Out rotate_copy(For first, For middle, For last, Out res);

template<class Ran> void random_shuffle(Ran first, Ran last);
template<class Ran, class Gen> void random_shuffle(Ran first, Ran last, Gen& g);
```

The *reverse*( ) algorithm reverses the order of the elements so that the first element becomes the last, etc. The *reverse_copy*( ) algorithm produces a copy of its input in reverse order.

The *rotate*( ) algorithm considers its [*first*,*last*[ sequence a circle and rotates its elements until its former *middle* element is placed where its *first* element used to be. That is, the element in

position *first+i* moves to position *first+(i+(last-middle))%(last-first)*. The % (modulo) is what makes the rotation cyclic rather than simply a shift to the left. For example:

```
void f()
{
    string v[] = { "Frog", "and", "Peach" };

    reverse(v,v+3);          // Peach and Frog
    rotate(v,v+1,v+3);       // and Frog Peach
}
```

The *rotate_copy()* algorithm produces a copy of its input in rotated order.

By default, *random_shuffle()* shuffles its sequence using a uniform distribution random-number generator. That is, it chooses a permutation of the elements of the sequence in such a way that each permutation has the same chance of being chosen. If you want a different distribution or simply a better random-number generator, you can supply one. For example, using the *Urand* generator from §22.7 we might shuffle a deck of cards like this:

```
void f(deque<Card>& dc)
{
    random_shuffle(dc.begin(),dc.end(),Urand(52));
    // ...
}
```

The movement of elements done by *rotate()*, etc., is done using *swap()* (§18.6.8).

### 18.6.8 Swap [algo.swap]

To do anything at all interesting with elements in a container, we need to move them around. Such movement is best expressed – that is, expressed most simply and most efficiently – as *swap()*s:

```
template<class T> void swap(T& a, T& b)
{
    T tmp = a;
    a = b;
    b = tmp;
}

template<class For, class For2> void iter_swap(For x, For2 y);

template<class For, class For2> For2 swap_ranges(For first, For last, For2 first2)
{
    while (first != last) iter_swap(first++, first2++);
    return first2;
}
```

To swap elements, you need a temporary. There are clever tricks to eliminate that need in specialized cases, but they are best avoided in favor of the simple and obvious. The *swap()* algorithm is specialized for important types for which it matters (§16.3.9, §13.5.2).

The *iter_swap()* algorithm swaps the elements pointed to by its iterator arguments.

The *swap_ranges* algorithm swaps elements in its two input ranges.

## 18.7 Sorted Sequences [algo.sorted]

Once we have collected some data, we often want to sort it. Once the sequence is sorted, our options for manipulating the data in a convenient manner increase significantly.

To sort a sequence, we need a way of comparing elements. This is done using a binary predicate (§18.4.2). The default comparison is *less* (§18.4.2), which in turn uses < by default.

### 18.7.1 Sorting [algo.sort]

The *sort*() algorithms require random-access iterators (§19.2.1). That is, they work best for *vector*s (§16.3) and similar containers:

```
template<class Ran> void sort(Ran first, Ran last);
template<class Ran, class Cmp> void sort(Ran first, Ran last, Cmp cmp);

template<class Ran> void stable_sort(Ran first, Ran last);
template<class Ran, class Cmp> void stable_sort(Ran first, Ran last, Cmp cmp);
```

The standard *list* (§17.2.2) does not provide random-access iterators, so *list*s should be sorted using the specific *list* operations (§17.2.2.1).

The basic *sort*() is efficient – on average *N*log*(*N*) – but its worst-case performance is poor – *O*(*N*N*). Fortunately, the worst case is rare. If guaranteed worst-case behavior is important or a stable sort is required, *stable_sort*() should be used; that is, an *N*log*(*N*)*log*(*N*) algorithm that improves towards *N*log*(*N*) when the system has sufficient extra memory. The relative order of elements that compare equal is preserved by *stable_sort*() but not by *sort*().

Sometimes, only the first elements of a sorted sequence are needed. In that case, it makes sense to sort the sequence only as far as is needed to get the first part in order. That is a partial sort:

```
template<class Ran> void partial_sort(Ran first, Ran middle, Ran last);
template<class Ran, class Cmp>
    void partial_sort(Ran first, Ran middle, Ran last, Cmp cmp);

template<class In, class Ran>
    Ran partial_sort_copy(In first, In last, Ran first2, Ran last2);
template<class In, class Ran, class Cmp>
    Ran partial_sort_copy(In first, In last, Ran first2, Ran last2, Cmp cmp);
```

The plain *partial_sort*() algorithms put the elements in the range *first* to *middle* in order. The *partial_sort_copy*() algorithms produce *N* elements, where *N* is the lower of the number of elements in the output sequence and the number of elements in the input sequence. We need to specify both the start and the end of the result sequence because that's what determines how many elements we need to sort. For example:

```
class Compare_copies_sold {
public:
    int operator()(const Book& b1, const Book& b2) const
        { return b1.copies_sold()<b2.copies_sold(); }
};
```

```
void f(const vector<Book>& sales)    // find the top ten books
{
    vector<Book> bestsellers(10);
    partial_sort_copy(sales.begin(), sales.end(),
                      bestsellers.begin(), bestsellers.end(), Compare_copies_sold());
    copy(bestsellers.begin(), bestsellers.end(), ostream_iterator<Book>(cout));
}
```

Because the target of *partial_sort_copy()* must be a random-access iterator, we cannot sort directly to *cout*.

Finally, algorithms are provided to sort only as far as is necessary to get the *N*th element to its proper place with no element comparing less than the *Nth* element placed after it in the sequence:

```
template<class Ran> void nth_element(Ran first, Ran nth, Ran last);
template<class Ran, class Cmp> void nth_element(Ran first, Ran nth, Ran last, Cmp cmp);
```

This algorithm is particularly useful for people – such as economists, sociologists, and teachers – who need to look for medians, percentiles, etc.

### 18.7.2 Binary Search [algo.bsearch]

A sequential search such as *find()* (§18.5.2) is terribly inefficient for large sequences, but it is about the best we can do without sorting or hashing (§17.6). Once a sequence is sorted, however, we can use a binary search to determine whether a value is in a sequence:

```
template<class For, class T> bool binary_search(For first, For last, const T& val);

template<class For, class T, class Cmp>
    bool binary_search(For first, For last, const T& value, Cmp cmp);
```

For example:

```
void f(list<int>& c)
{
    if (binary_search(c.begin(), c.end(), 7)) {    // is 7 in c?
        // ...
    }
    // ...
}
```

A *binary_search()* returns a *bool* indicating whether a value was present. As with *find()*, we often also want to know where the elements with that value are in that sequence. However, there can be many elements with a given value in a sequence, and we often need to find either the first or all such elements. Consequently, algorithms are provided for finding a range of equal elements, *equal_range()*, and algorithms for finding the *lower_bound()* and *upper_bound()* of that range:

```
template<class For, class T> For lower_bound(For first, For last, const T& val);
template<class For, class T, class Cmp>
    For lower_bound(For first, For last, const T& val, Cmp cmp);
```

```
template<class For, class T> For upper_bound(For first, For last, const T& val);
template<class For, class T, class Cmp>
    For upper_bound(For first, For last, const T& val, Cmp cmp);

template<class For, class T> pair<For, For> equal_range(For first, For last, const T& val);
template<class For, class T, class Cmp>
    pair<For, For> equal_range(For first, For last, const T& val, Cmp cmp);
```

These algorithms correspond to the operations on ***multimap***s (§17.4.2). We can think of ***lower_bound***( ) as a fast ***find***( ) and ***find_if***( ) for sorted sequences. For example:

```
void g(vector<int>& c)
{
    typedef vector<int>::iterator VI;

    VI p = find(c.begin(),c.end(),7);          // probably slow: O(N); c needn't be sorted
    VI q = lower_bound(c.begin(),c.end(),7);   // probably fast: O(log(N)); c must be sorted
    // ...
}
```

If ***lower_bound***(***first***, ***last***, ***k***) doesn't find ***k***, it returns an iterator to the first element with a key greater than ***k***, or ***last*** if no such greater element exists. This way of reporting failure is also used by ***upper_bound***( ) and ***equal_range***( ). This means that we can use these algorithms to determine where to insert a new element into a sorted sequence so that the sequence remains sorted.

### 18.7.3 Merge [algo.merge]

Given two sorted sequences, we can merge them into a new sorted sequence using ***merge***( ) or merge two parts of a sequence using ***inplace_merge***( ):

```
template<class In, class In2, class Out>
    Out merge(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out merge(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class Bi> void inplace_merge(Bi first, Bi middle, Bi last);
template<class Bi, class Cmp> void inplace_merge(Bi first, Bi middle, Bi last, Cmp cmp);
```

Note that these merge algorithms differ from ***list***'s merge (§17.2.2.1) by ***not*** removing elements from their input sequences. Instead, elements are copied.

For elements that compare equal, elements from the first range will always precede elements from the second.

The ***inplace_merge***( ) algorithm is primarily useful when you have a sequence that can be sorted by more than one criterion. For example, you might have a ***vector*** of fish sorted by species (for example, cod, haddock, and herring). If the elements of each species are sorted by weight, you can get the whole vector sorted by weight by applying ***inplace_merge***( ) to merge the information for the different species (§18.13[20]).

### 18.7.4 Partitions [algo.partition]

To partition a sequence is to place every element that satisfies a predicate before every element that doesn't. The standard library provides a *stable_partition*( ), which maintains relative order among the elements that do and do not satisfy the predicate. In addition, the library offers *partition*( ) which doesn't maintain relative order, but which runs a bit faster when memory is limited:

```
template<class Bi, class Pred> Bi partition(Bi first, Bi last, Pred p);
template<class Bi, class Pred> Bi stable_partition(Bi first, Bi last, Pred p);
```

You can think of a partition as a kind of sort with a very simple sorting criterion. For example:

```
void f(list<Club>& lc)
{
    list<Club>::iterator p = partition(lc.begin(), lc.end(), located_in("København"));
    // ...
}
```

This ''sorts'' the *list* so that *Club*s in Copenhagen comes first. The return value (here *p*) points either to the first element that doesn't satisfy the predicate or to the end.

### 18.7.5 Set Operations on Sequences [algo.set]

A sequence can be considered a set. Looked upon that way, it makes sense to provide set operations such as union and intersection for sequences. However, such operations are horribly inefficient unless the sequences are sorted, so the standard library provides set operations for sorted sequences only. In particular, the set operations work well for *set*s (§17.4.3) and *multiset*s (§17.4.4), both of which are sorted anyway.

If these set algorithms are applied to sequences that are not sorted, the resulting sequences will not conform to the usual set-theoretical rules. These algorithms do not change their input sequences, and their output sequences are ordered.

The *includes*( ) algorithm tests whether every member of the first sequence is also a member of the second:

```
template<class In, class In2>
    bool includes(In first, In last, In2 first2, In2 last2);
template<class In, class In2, class Cmp>
    bool includes(In first, In last, In2 first2, In2 last2, Cmp cmp);
```

The *set_union*( ) and *set_intersection*( ) produce their obvious outputs as sorted sequences:

```
template<class In, class In2, class Out>
    Out set_union(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out set_union(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class In, class In2, class Out>
    Out set_intersection(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out set_intersection(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);
```

The *set_difference*( ) algorithm produces a sequence of elements that are members of its first, but

not its second, input sequence. The *set_symmetric_difference*( ) algorithm produces a sequence of elements that are members of either, but not of both, of its input sequences:

```
template<class In, class In2, class Out>
    Out set_difference(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out set_difference(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class In, class In2, class Out>
    Out set_symmetric_difference(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out set_symmetric_difference(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);
```

For example:

```
char v1[] = "abcd";
char v2[] = "cdef";

void f(char v3[])
{
    set_difference(v1,v1+4,v2,v2+4,v3);                 // v3 = "ab"
    set_symmetric_difference(v1,v1+4,v2,v2+4,v3);       // v3 = "abef"
}
```

## 18.8 Heaps [algo.heap]

The word *heap* means different things in different contexts. When discussing algorithms, ''heap'' often refers to a way of organizing a sequence such that it has a first element that is the element with the highest value. Addition of an element (using *push_heap*( )) and removal of an element (using *pop_heap*( )) are reasonably fast, with a worst-case performance of $O(log(N))$, where $N$ is the number of elements in the sequence. Sorting (using *sort_heap*( )) has a worst-case performance of $O(N*log(N))$. A heap is implemented by this set of functions:

```
template<class Ran> void push_heap(Ran first, Ran last);
template<class Ran, class Cmp> void push_heap(Ran first, Ran last, Cmp cmp);

template<class Ran> void pop_heap(Ran first, Ran last);
template<class Ran, class Cmp> void pop_heap(Ran first, Ran last, Cmp cmp);

template<class Ran> void make_heap(Ran first, Ran last);      // turn sequence into heap
template<class Ran, class Cmp> void make_heap(Ran first, Ran last, Cmp cmp);

template<class Ran> void sort_heap(Ran first, Ran last);      // turn heap into sequence
template<class Ran, class Cmp> void sort_heap(Ran first, Ran last, Cmp cmp);
```

The style of the heap algorithms is odd. A more natural way of presenting their functionality would be to provide an adapter class with four operations. Doing that would yield something like a *priority_queue* (§17.3.3). In fact, a *priority_queue* is almost certainly implemented using a heap.

The value pushed by *push_heap*(*first*,*last*) is *(*last*-1). The assumption is that [*first*,*last*-1[ is already a heap, so *push_heap*( ) extends the sequence to [*first*,*last*[ by including the next element. Thus, you can build a heap from an existing sequence by a series of

*push_heap*() operations. Conversely, *pop_heap*(*first*,*last*) removes the first element of the heap by swapping it with the last element (*(*last*-1)) and making [*first*,*last*-1[ into a heap.

## 18.9 Min and Max [algo.min]

The algorithms described here select a value based on a comparison. It is obviously useful to be able to find the maximum and minimum of two values:

```
template<class T> const T& max(const T& a, const T& b)
{
    return (a<b) ? b : a;
}

template<class T, class Cmp> const T& max(const T& a, const T& b, Cmp cmp)
{
    return (cmp(a,b)) ? b : a;
}

template<class T> const T& min(const T& a, const T& b);

template<class T, class Cmp> const T& min(const T& a, const T& b, Cmp cmp);
```

The *max*() and *min*() operations can be generalized to apply to sequences in the obvious manner:

```
template<class For> For max_element(For first, For last);
template<class For, class Cmp> For max_element(For first, For last, Cmp cmp);

template<class For> For min_element(For first, For last);
template<class For, class Cmp> For min_element(For first, For last, Cmp cmp);
```

Finally, lexicographical ordering is easily generalized from strings of characters to sequences of values of a type with comparison:

```
template<class In, class In2>
bool lexicographical_compare(In first, In last, In2 first2, In2 last2);

template<class In, class In2, class Cmp>
bool lexicographical_compare(In first, In last, In2 first2, In2 last2, Cmp cmp)
{
    while (first != last && first2 != last2) {
        if (cmp(*first,*first2)) return true;
        if (cmp(*first2++,*first++)) return false;
    }
    return first == last && first2 != last2;
}
```

This is very similar to the function presented for general strings in (§13.4.1). However, *lexicographical_compare*() compares sequences in general and not just strings. It also returns a ***bool*** rather than the more useful ***int***. The result is ***true*** (only) if the first sequence compares < the second. In particular, the result is ***false*** when the sequences compare equal.

C-style strings and ***string***s are sequences, so *lexicographical_compare*() can be used as a string compare function. For example:

```
char v1[] = "yes";
char v2[] = "no";
string s1 = "Yes";
string s2 = "No";

void f()
{
    bool b1 = lexicographical_compare(v1,v1+strlen(v1),v2,v2+strlen(v2));
    bool b2 = lexicographical_compare(s1.begin(),s1.end(),s2.begin(),s2.end());

    bool b3 = lexicographical_compare(v1,v1+strlen(v1),s1.begin(),s1.end());
    bool b4 = lexicographical_compare(s1.begin(),s1.end(),v1,v1+strlen(v1),Nocase);
}
```

The sequences need not be of the same type – all we need is to compare their elements – and the comparison criterion can be supplied. This makes ***lexicographical_compare()*** more general and potentially a bit slower than ***string***'s compare. See also §20.3.8.

## 18.10 Permutations [algo.perm]

Given a sequence of four elements, we can order them in 4*3*2 ways. Each of these orderings is called a *permutation*. For example, from the four characters ***abcd*** we can produce 24 permutations:

```
abcd abdc acbd acdb adbc adcb bacd badc
bcad bcda bdac bdca cabd cadb cbad cbda
cdab cdba dabc dacb dbac dbca dcab dcba
```

The ***next_permutation()*** and ***prev_permutation()*** functions deliver such permutations of a sequence:

```
template<class Bi> bool next_permutation(Bi first, Bi last);
template<class Bi, class Cmp> bool next_permutation(Bi first, Bi last, Cmp cmp);

template<class Bi> bool prev_permutation(Bi first, Bi last);
template<class Bi, class Cmp> bool prev_permutation(Bi first, Bi last, Cmp cmp);
```

The permutations of ***abcd*** were produced like this:

```
int main()
{
    char v[] = "abcd";
    cout << v << '\t';
    while(next_permutation(v,v+4)) cout << v << '\t';
}
```

The permutations are produced in lexicographical order (§18.9). The return value of ***next_permutation()*** indicates whether a next permutation actually exists. If not, ***false*** is returned and the sequence is the permutation in which the elements are in lexicographical order.

## 18.11 C-Style Algorithms [algo.c]

From the C standard library, the C++ standard library inherited a few algorithms dealing with C-style strings (§20.4.1), plus a quicksort and a binary search, both limited to arrays.

The *qsort*( ) and *bsearch*( ) functions are presented in <*cstdlib*> and <*stdlib.h*>. They each operate on an array of *n* elements of size *elem_size* using a less-than comparison function passed as a pointer to function. The elements must be of a type without a user-defined copy constructor, copy assignment, or destructor:

```
typedef int(*__cmp)(const void*, const void*);      // typedef for presentation only

void qsort(void* p, size_t n, size_t elem_size, __cmp);                    // sort p
void* bsearch(const void* key, void* p, size_t n, size_t elem_size, __cmp);// find key in p
```

The use of *qsort*( ) is described in §7.7.

These algorithms are provided solely for C compatibility; *sort*( ) (§18.7.1) and *search*( ) (§18.5.5) are more general and should also be more efficient.

## 18.12 Advice [algo.advice]

[1] Prefer algorithms to loops; §18.5.1.
[2] When writing a loop, consider whether it could be expressed as a general algorithm; §18.2.
[3] Regularly review the set of algorithms to see if a new application has become obvious; §18.2.
[4] Be sure that a pair of iterator arguments really do specify a sequence; §18.3.1.
[5] Design so that the most frequently-used operations are simple and safe; §18.3, §18.3.1.
[6] Express tests in a form that allows them to be used as predicates; §18.4.2.
[7] Remember that predicates are functions and objects, not types; §18.4.2.
[8] You can use binders to make unary predicates out of binary predicates; §18.4.4.1.
[9] Use *mem_fun*( ) and *mem_fun_ref*( ) to apply algorithms on containers; §18.4.4.2.
[10] Use *ptr_fun*( ) when you need to bind an argument of a function; §18.4.4.3.
[11] Remember that *strcmp*( ) differs from == by returning *0* to indicate ''equal;'' §18.4.4.4.
[12] Use *for_each*( ) and *transform*( ) only when there is no more-specific algorithm for a task; §18.5.1.
[13] Use predicates to apply algorithms using a variety of comparison and equality criteria; §18.4.2.1, §18.6.3.1.
[14] Use predicates and other function objects so as to use standard algorithms with a wider range of meanings; §18.4.2.
[15] The default == and < on pointers are rarely adequate for standard algorithms; §18.6.3.1.
[16] Algorithms do not directly add or subtract elements from their argument sequences; §18.6.
[17] Be sure that the less-than and equality predicates used on a sequence match; §18.6.3.1.
[18] Sometimes, sorted sequences can be used to increase efficiency and elegance; §18.7.
[19] Use *qsort*( ) and *bsearch*( ) for compatibility only; §18.11.

## 18.13 Exercises [algo.exercises]

The solutions to several exercises for this chapter can be found by looking at the source text of an implementation of the standard library. Do yourself a favor: try to find your own solutions before looking to see how your library implementer approached the problems.

1. (*2) Learn *O*( ) notation. Give a realistic example in which an *O*(*N*N*) algorithm is faster than an *O*(*N*) algorithm for some *N>10*.
2. (*2) Implement and test the four *mem_fun*( ) and *mem_fun_ref*( ) functions (§18.4.4.2).
3. (*1) Write an algorithm *match*( ) that is like *mismatch*( ), except that it returns iterators to the first corresponding pair that matches the predicate.
4. (*1.5) Implement and test *Print_name* from §18.5.1.
5. (*1) Sort a *list* using only standard library algorithms.
6. (*2.5) Define versions of *iseq*( ) (§18.3.1) for built-in arrays, *istream*, and iterator pairs. Define a suitable set of overloads for the nonmodifying standard algorithms (§18.5) for *Iseq*s. Discuss how best to avoid ambiguities and an explosion in the number of template functions.
7. (*2) Define an *oseq*( ) to complement *iseq*( ). The output sequence given as the argument to *oseq*( ) should be replaced by the output produced by an algorithm using it. Define a suitable set of overloads for at least three standard algorithms of your choice.
8. (*1.5) Produce a *vector* of squares of numbers 1 through 100. Print a table of squares. Take the square root of the elements of that *vector* and print the resulting vector.
9. (*2) Write a set of functional objects that do bitwise logical operations on their operands. Test these objects on vectors of *char*, *int*, and *bitset<67>*.
10. (*1) Write a *binder3*( ) that binds the second and third arguments of a three-argument function to produce a unary predicate. Give an example where *binder3*( ) is a useful function.
11. (*1.5) Write a small program that that removes adjacent repeated words from from a file file. Hint: The program should remove a *that*, a *from*, and a *file* from the previous statement.
12. (*2.5) Define a format for records of references to papers and books kept in a file. Write a program that can write out records from the file identified by year of publication, name of author, keyword in title, or name of publisher. The user should be able to request that the output be sorted according to similar criteria.
13. (*2) Implement a *move*( ) algorithm in the style of *copy*( ) in such a way that the input and output sequences can overlap. Be reasonably efficient when given random-access iterators as arguments.
14. (*1.5) Produce all anagrams of the word *food*. That is, all four-letter combinations of the letters *f*, *o*, *o*, and *d*. Generalize this program to take a word as input and produce anagrams of that word.
15. (*1.5) Write a program that produces anagrams of sentences; that is, a program that produces all permutations of the words in the sentences (rather than permutations of the letters in the words).
16. (*1.5) Implement *find_if*( ) (§18.5.2) and then implement *find*( ) using *find_if*( ). Find a way of doing this so that the two functions do not need different names.
17. (*2) Implement *search*( ) (§18.5.5). Provide an optimized version for random-access iterators.
18. (*2) Take a sort algorithm (such as *sort*( ) from your standard library or the Shell sort from §13.5.2) and insert code so that it prints out the sequence being sorted after each swap of elements.

19. (∗2) There is no ***sort*( )** for bidirectional iterators. The conjecture is that copying to a vector and then sorting is faster than sorting a sequence using bidirectional iterators. Implement a general sort for bidirectional iterators and test the conjecture.
20. (∗2.5) Imagine that you keep records for a group of sports fishermen. For each catch, keep a record of species, length, weight, date of catch, name of fisherman, etc. Sort and print the records according to a variety of criteria. Hint: ***inplace_merge*( )**.
21. (∗2) Create lists of students taking Math, English, French, and Biology. Pick about 20 names for each class out of a set of 40 names. List students who take both Math and English. List students who take French but not Biology or Math. List students who do not take a science course. List students who take French and Math but neither English nor Biology.
22. (∗1.5) Write a ***remove*( )** function that actually removes elements from a container.