```
void f(map<string,int>& m)
{
        auto p = find_if(m.begin(),m.end(),Greater_than{42});
        // ...
}
```

Here, `Greater_than` is a function object (§5.5) holding the value (`42`) to be compared against:

```
struct Greater_than {
        int val;
        Greater_than(int v) : val{v} { }
        bool operator()(const pair<string,int>& r) { return r.second>val; }
};
```

Alternatively, we could use a lambda expression (§5.5):

```
auto p = find_if(m.begin(), m.end(), [](const pair<string,int>& r) { return r.second>42; });
```

A predicate should not modify the elements to which it is applied.

10.6 Algorithm Overview

A general definition of an algorithm is "a finite set of rules which gives a sequence of operations for solving a specific set of problems [and] has five important features: Finiteness ... Definiteness ... Input ... Output ... Effectiveness" [Knuth,1968,§1.1]. In the context of the C++ standard library, an algorithm is a function template operating on sequences of elements.

The standard library provides dozens of algorithms. The algorithms are defined in namespace `std` and presented in the `<algorithm>` header. These standard-library algorithms all take sequences as inputs. A half-open sequence from `b` to `e` is referred to as [`b`:`e`). Here are a few examples:

Selected Standard Algorithms	
`p=find(b,e,x)`	`p` is the first `p` in [`b`:`e`) so that `*p==x`
`p=find_if(b,e,f)`	`p` is the first `p` in [`b`:`e`) so that `f(*p)==true`
`n=count(b,e,x)`	`n` is the number of elements `*q` in [`b`:`e`) so that `*q==x`
`n=count_if(b,e,f)`	`n` is the number of elements `*q` in [`b`:`e`) so that `f(*q,x)`
`replace(b,e,v,v2)`	Replace elements `*q` in [`b`:`e`) so that `*q==v` by `v2`
`replace_if(b,e,f,v2)`	Replace elements `*q` in [`b`:`e`) so that `f(*q)` by `v2`
`p=copy(b,e,out)`	Copy [`b`:`e`) to [`out`:`p`)
`p=copy_if(b,e,out,f)`	Copy elements `*q` from [`b`:`e`) so that `f(*q)` to [`out`:`p`)
`p=move(b,e,out)`	Move [`b`:`e`) to [`out`:`p`)
`p=unique_copy(b,e,out)`	Copy [`b`:`e`) to [`out`:`p`); don't copy adjacent duplicates
`sort(b,e)`	Sort elements of [`b`:`e`) using `<` as the sorting criterion
`sort(b,e,f)`	Sort elements of [`b`:`e`) using `f` as the sorting criterion
`(p1,p2)=equal_range(b,e,v)`	[`p1`:`p2`) is the subsequence of the sorted sequence [`b`:`e`) with the value `v`; basically a binary search for `v`
`p=merge(b,e,b2,e2,out)`	Merge two sorted sequences [`b`:`e`) and [`b2`:`e2`) into [`out`:`p`)