17

Standard Containers

Now is a good time to put your work
on a firm theoretical foundation.
– Sam Morgan

Standard containers — container and operation summaries — efficiency — representation — element requirements — sequences — ***vector*** — ***list*** — ***deque*** — adapters — ***stack*** — ***queue*** — ***priority_queue*** — associative containers — ***map*** — comparisons — ***multimap*** — ***set*** — ***multiset*** — ‘‘almost containers’’ — ***bitset*** — arrays — hash tables — implementing a ***hash_map*** — advice — exercises.

17.1 Standard Containers [cont.intro]

The standard library defines two kinds of containers: sequences and associative containers. The sequences are all much like ***vector*** (§16.3). Except where otherwise stated, the member types and functions mentioned for ***vector*** can also be used for any other container and produce the same effect. In addition, associative containers provide element access based on keys (§3.7.4).

Built-in arrays (§5.2), ***string***s (Chapter 20), ***valarray***s (§22.4), and ***bitset***s (§17.5.3) hold elements and can therefore be considered containers. However, these types are not fully-developed standard containers. If they were, that would interfere with their primary purpose. For example, a built-in array cannot both hold its own size and remain layout-compatible with C arrays.

A key idea for the standard containers is that they should be logically interchangeable wherever reasonable. The user can then choose between them based on efficiency concerns and the need for specialized operations. For example, if lookup based on a key is common, a ***map*** (§17.4.1) can be used. On the other hand, if general list operations dominate, a ***list*** (§17.2.2) can be used. If many additions and removals of elements occur at the ends of the container, a ***deque*** (double-ended queue, §17.2.3), a ***stack*** (§17.3.1), or a ***queue*** (§17.3.2) should be considered. In addition, a user can design additional containers to fit into the framework provided by the standard containers

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

(§17.6). By default, a *vector* (§16.3) should be used; it will be implemented to perform well over a wide range of uses.

The idea of treating different kinds of containers – and more generally all kinds of information sources – in uniform ways leads to the notion of generic programming (§2.7.2, §3.8). The standard library provides many generic algorithms to support this idea (Chapter 18). Such algorithms can save the programmer from having to deal directly with details of individual containers.

17.1.1 Operations Summary [cont.operations]

This section lists the common and almost common members of the standard containers. For more details, read your standard headers (<*vector*>, <*list*>, <*map*>, etc.; §16.1.2).

Member Types (§16.3.1)	
value_type	Type of element.
allocator_type	Type of memory manager.
size_type	Type of subscripts, element counts, etc.
difference_type	Type of difference between iterators.
iterator	Behaves like *value_type**.
const_iterator	Behaves like *const value_type**.
reverse_iterator	View container in reverse order; like *value_type**.
const_reverse_iterator	View container in reverse order; like *const value_type**.
reference	Behaves like *value_type&*.
const_reference	Behaves like *const value_type&*.
key_type	Type of key (for associative containers only).
mapped_type	Type of *mapped_value* (for associative containers only).
key_compare	Type of comparison criterion (for associative containers only).

A container can be viewed as a sequence either in the order defined by the container's *iterator* or in reverse order. For an associative container, the order is based on the container's comparison criterion (by default <):

Iterators (§16.3.2)	
begin()	Points to first element.
end()	Points to one-past-last element.
rbegin()	Points to first element of reverse sequence.
rend()	Points to one-past-last element of reverse sequence.

Some elements can be accessed directly:

Element Access (§16.3.3)	
front()	First element.
back()	Last element.
[]	Subscripting, unchecked access (not for list).
at()	Subscripting, checked access (not for list).

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

Most containers provide efficient operations at the end (back) of their sequence of elements. In addition, lists and deques provide the equivalent operations on the start (front) of their sequences:

Stack and Queue Operations (§16.3.5, §17.2.2.2)	
push_back()	Add to end.
pop_back()	Remove last element.
push_front()	Add new first element (for list and deque only).
pop_front()	Remove first element (for list and deque only).

Containers provide list operations:

List Operations (§16.3.6)	
insert(p,x)	Add *x* before *p*.
insert(p,n,x)	Add *n* copies of *x* before *p*.
insert(p,first,last)	Add elements from [*first*:*last*[before *p*.
erase(p)	Remove element at *p*.
erase(first,last)	Erase [*first*:*last*[.
clear()	Erase all elements.

All containers provide operations related to the number of elements and a few other operations:

Other Operations (§16.3.8, §16.3.9, §16.3.10)	
size()	Number of elements.
empty()	Is the container empty?
max_size()	Size of the largest possible container.
capacity()	Space allocated for *vector* (for vector only).
reserve()	Reserve space for future expansion (for vector only).
resize()	Change size of container (for vector, list, and deque only).
swap()	Swap elements of two containers.
get_allocator()	Get a copy of the container's allocator.
==	Is the content of two containers the same?
!=	Is the content of two containers different?
<	Is one container lexicographically before another?

Containers provide a variety of constructors and assignment operations:

Constructors, etc. (§16.3.4)	
container()	Empty container.
container(n)	*n* elements default value (not for associative containers).
container(n,x)	*n* copies of *x* (not for associative containers).
container(first,last)	Initial elements from [*first*:*last*[.
container(x)	Copy constructor; initial elements from container *x*.
~container()	Destroy the container and all of its elements.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

Assignments (§16.3.4)	
operator=(x)	Copy assignment; elements from container *x*.
assign(n,x)	Assign *n* copies of *x* (not for associative containers).
assign(first,last)	Assign from [*first*:*last*[.

Associative containers provide lookup based on keys:

Associative Operations (§17.4.1)	
operator[](k)	Access the element with key *k* (for containers with unique keys).
find(k)	Find the element with key *k*.
lower_bound(k)	Find the first element with key *k*.
upper_bound(k)	Find the first element with key greater than *k*.
equal_range(k)	Find the *lower_bound* and *upper_bound* of elements with key *k*.
key_comp()	Copy of the key comparison object.
value_comp()	Copy of the *mapped_value* comparison object.

In addition to these common operations, most containers provide a few specialized operations.

17.1.2 Container Summary [cont.summary]

The standard containers can be summarized like this:

Standard Container Operations					
	[]	List Operations	Front Operations	Back (Stack) Operations	Iterators
	§16.3.3	§16.3.6	§17.2.2.2	§16.3.5	§19.2.1
	§17.4.1.3	§20.3.9	§20.3.9	§20.3.12	
vector	const	O(n)+		const+	Ran
list		const	const	const	Bi
deque	const	O(n)	const	const	Ran
stack				const+	
queue			const	const+	
priority_queue			O(log(n))	O(log(n))	
map	O(log(n))	O(log(n))+			Bi
multimap		O(log(n))+			Bi
set		O(log(n))+			Bi
multiset		O(log(n))+			Bi
string	const	O(n)+	O(n)+	const+	Ran
array	const				Ran
valarray	const				Ran
bitset	const				

In the *iterators* column, ***Ran*** means random-access iterator and ***Bi*** means bidirectional iterator; the operations for a bidirectional operator are a subset of those of a random-access iterator (§19.2.1).

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

Other entries are measures of the efficiency of the operations. A ***const*** entry means the operation takes an amount of time that does not depend on the number of elements in the container. Another conventional notation for *constant time* is $O(1)$. An $O(n)$ entry means the entry takes time proportional to the number of elements involved. A + suffix indicates that occasionally a significant extra cost is incurred. For example, inserting an element into a ***list*** has a fixed cost (so it is listed as ***const***), whereas the same operation on a ***vector*** involves moving the elements following the insertion point (so it is listed as $O(n)$). Occasionally, all elements must be relocated (so I added a +). The ''big O'' notation is conventional. I added the + for the benefit of programmers who care about predictability in addition to average performance. A conventional term for $O(n)$+ is *amortized linear time*.

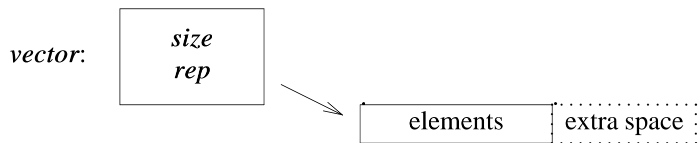
Naturally, if a constant is large it can dwarf a small cost proportional to the number of elements. However, for large data structures ***const*** tends to mean ''cheap,'' $O(n)$ to mean ''expensive,'' and $O(log(n))$ to mean ''fairly cheap.'' For even moderately large values of ***n***, $O(log(n))$ is closer to constant time than to $O(n)$. People who care about cost must take a closer look. In particular, they must understand what elements are counted to get the ***n***. No basic operation is ''very expensive,'' that is, $O(n*n)$ or worse.

Except for ***string***, the measures of costs listed here reflect requirements in the standard. The ***string*** estimates are my assumptions.

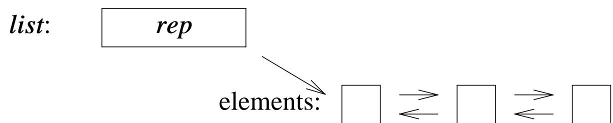
These measures of complexity and cost are upper bounds. The measures exist to give users some guidance as to what they can expect from implementations. Naturally, implementers will try to do better in important cases.

17.1.3 Representation [cont.rep]

The standard doesn't prescribe a particular representation for each standard container. Instead, the standard specifies the container interfaces and some complexity requirements. Implementers will choose appropriate and often cleverly optimized implementations to meet the general requirements. A container will almost certainly be represented by a data structure holding the elements accessed through a handle holding size and capacity information. For a ***vector***, the element data structure is most likely an array:

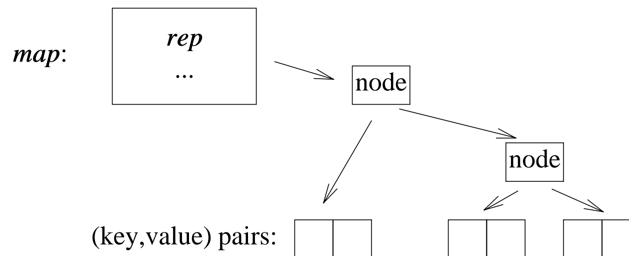


Similarly, a ***list*** is most likely represented by a set of links pointing to the elements:

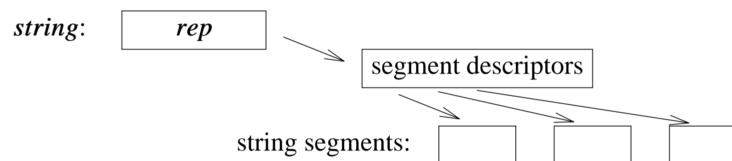


The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

A *map* is most likely implemented as a (balanced) tree of nodes pointing to (key,value) pairs:



A *string* might be implemented as outlined in §11.12 or maybe as a sequence of arrays holding a few characters each:



17.1.4 Element Requirements [cont.elem]

Elements in a container are copies of the objects inserted. Thus, to be an element of a container, an object must be of a type that allows the container implementation to copy it. The container may copy elements using a copy constructor or an assignment; in either case, the result of the copy must be an equivalent object. This roughly means that any test for equality that you can devise on the value of the objects must deem the copy equal to the original. In other words, copying an element must work much like an ordinary copy of built-in types (including pointers). For example,

```
X& X::operator=(const X& a)   // proper assignment operator
{
        // copy all of a's members to *this
        return *this;
}
```

makes *X* acceptable as an element type for a standard container, but

```
void Y::operator=(const Y& a) // improper assignment operator
{
        // zero out all of a's members
}
```

renders *Y* unsuitable because *Y*'s assignment has neither the conventional return type nor the conventional semantics.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

Some violations of the rules for standard containers can be detected by a compiler, but others cannot and might then cause unexpected behavior. For example, a copy operation that throws an exception might leave a partially copied element behind. It could even leave the container itself in a state that could cause trouble later. Such copy operations are themselves bad design (§14.4.6.1).

When copying elements isn't right, the alternative is to put pointers to objects into containers instead of the objects themselves. The most obvious example is polymorphic types (§2.5.4, §12.2.6). For example, we use *vector<Shape*>* rather than *vector<Shape>* to preserve polymorphic behavior.

17.1.4.1 Comparisons [cont.comp]

Associative containers require that their elements can be ordered. So do many operations that can be applied to containers (for example *sort*()). By default, the < operator is used to define the order. If < is not suitable, the programmer must provide an alternative (§17.4.1.5, §18.4.2). The ordering criterion must define a *strict weak ordering*. Informally, this means that both less-than and equality must be transitive. That is, for an ordering criterion *cmp*:

[1] *cmp*(*x*,*x*) is *false*.
[2] If *cmp*(*x*,*y*) and *cmp*(*y*,*z*), then *cmp*(*x*,*z*).
[3] Define *equiv*(*x*,*y*) to be !(*cmp*(*x*,*y*)||*cmp*(*y*,*x*)). If *equiv*(*x*,*y*) and *equiv*(*y*,*z*), then *equiv*(*x*,*z*).

Consider:

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

The first version uses < and the second uses a user-supplied comparison *cmp*. For example, we might decide to sort *fruit* using a comparison that isn't case-sensitive. We do that by defining a function object (§11.9, §18.4) that does the comparison when invoked for a pair of *string*s:

```
class Nocase {          // case-insensitive string compare
public:
        bool operator()(const string&, const string&) const;
};

bool Nocase::operator()(const string& x, const string& y) const
        // return true if x is lexicographically less than y, not taking case into account
{
        string::const_iterator p = x.begin();
        string::const_iterator q = y.begin();

        while (p!=x.end() && q!=y.end() && toupper(*p)==toupper(*q)) {
                ++p;
                ++q;
        }
        if (p == x.end()) return q != y.end();
        return toupper(*p) < toupper(*q);
}
```

We can call *sort*() using that comparison criterion. For example, given:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
fruit:
      apple  pear  Apple  Pear  lemon
```

Sorting using *sort(fruit.begin(),fruit.end(),Nocase())* would yield:

```
fruit:
      Apple  apple  lemon  Pear  pear
```

whereas plain *sort(fruit.begin(),fruit.end())* would give:

```
fruit:
      Apple  Pear  apple  lemon  pear
```

assuming a character set in which uppercase letters precede lowercase letters.

Beware that < on C-style strings (that is, *char**) does not define lexicographical order (§13.5.2). Thus, associative containers will not work as most people would expect them to when C-style strings are used as keys. To make them work properly, a less-than operation that compares based on lexicographical order must be used. For example:

```
struct Cstring_less {
      bool operator()(const char* p, const char* q) const { return strcmp(p,q)<0; }
};

map<char*,int,Cstring_less> m;       // map that uses strcmp() to compare const char* keys
```

17.1.4.2 Other Relational Operators [cont.relops]

By default, containers and algorithms use < when they need to do a less-than comparison. When the default isn't right, a programmer can supply a comparison criterion. However, no mechanism is provided for also passing an equality test. Instead, when a programmer supplies a comparison ***cmp***, equality is tested using two comparisons. For example:

```
if (x == y) // not done where the user supplied a comparison

if (!cmp(x,y) && !cmp(y,x)) // done where the user supplied a comparison cmp
```

This saves us from having to add an equality parameter to every associative container and most algorithms. It may look expensive, but the library doesn't check for equality very often, and in 50% of the cases, only a single call of *cmp()* is needed.

Using an equivalence relationship defined by less-than (by default <) rather than equality (by default ==) also has practical uses. For example, associative containers (§17.4) compare keys using an equivalence test *!(cmp(x,y)||cmp(y,x))*. This implies that equivalent keys need not be equal. For example, a ***multimap*** (§17.4.2) that uses case-insensitive comparison as its comparison criteria will consider the strings ***Last***, ***last***, ***lAst***, ***laSt***, and ***lasT*** equivalent, even though == for strings deems them different. This allows us to ignore differences we consider insignificant when sorting.

Given < and ==, we can easily construct the rest of the usual comparisons. The standard library defines them in the namespace *std::rel_ops* and presents them in <***utility***>:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
template<class T> bool rel_ops::operator!=(const T& x, const T& y) { return !(x==y); }
template<class T> bool rel_ops::operator>(const T& x, const T& y) { return y<x; }
template<class T> bool rel_ops::operator<=(const T& x, const T& y) { return !(y<x); }
template<class T> bool rel_ops::operator>=(const T& x, const T& y) { return !(x<y); }
```

Placing these operations in *rel_ops* ensures that they are easy to use when needed, yet they don't get created implicitly unless extracted from that namespace:

```
void f()
{
    using namespace std;
    // !=, >, etc., not generated by default
}

void g()
{
    using namespace std;
    using namespace std::rel_ops;
    // !=, >, etc., generated by default
}
```

The `!=`, etc., operations are not defined directly in *std* because they are not always needed and sometimes their definition would interfere with user code. For example, if I were writing a generalized math library, I would want *my* relational operators and not the standard library versions.

17.2 Sequences [cont.seq]

Sequences follow the pattern described for *vector* (§16.3). The fundamental sequences provided by the standard library are:

```
vector    list    deque
```

From these,

```
stack    queue    priority_queue
```

are created by providing suitable interfaces. These sequences are called *container adapters*, *sequence adapters*, or simply *adapters* (§17.3).

17.2.1 Vector [cont.vector]

The standard *vector* is described in detail in §16.3. The facilities for reserving space (§16.3.8) are unique to *vector*. By default, subscripting using `[]` is not range checked. If a check is needed, use *at*() (§16.3.3), a checked vector (§3.7.1), or a checked iterator (§19.3). A *vector* provides random-access iterators (§19.2.1).

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

17.2.2 List [cont.list]

A ***list*** is a sequence optimized for insertion and deletion of elements. Compared to ***vector*** (and ***deque***; §17.2.3), subscripting would be painfully slow, so subscripting is not provided for ***list***. Consequently, ***list*** provides bidirectional iterators (§19.2.1) rather than random-access iterators. This implies that a ***list*** will typically be implemented using some form of a doubly-linked list (see §17.8[16]).

A ***list*** provides all of the member types and operations offered by ***vector*** (§16.3), with the exceptions of subscripting, ***capacity***(), and ***reserve***():

```
template <class T, class A = allocator<T> > class std::list {
public:
        // types and operations like vector's, except [], at(), capacity(), and reserve()
        // ...
};
```

17.2.2.1 Splice, Sort, and Merge [cont.splice]

In addition to the general sequence operations, ***list*** provides several operations specially suited for list manipulation:

```
template <class T, class A = allocator<T> > class list {
public:
        // ...
        // list-specific operations:

        void splice(iterator pos, list& x);                    // move all elements from x to before
                                                               // pos in this list without copying.
        void splice(iterator pos, list& x, iterator p);        // move *p from x to before
                                                               // pos in this list without copying.
        void splice(iterator pos, list& x, iterator first, iterator last);

        void merge(list&);          // merge sorted lists
        template <class Cmp> void merge(list&, Cmp);

        void sort();
        template <class Cmp> void sort(Cmp);

        // ...
};
```

These ***list*** operations are all *stable*; that is, they preserve the relative order of elements that have equivalent values.

The ***fruit*** examples from §16.3.6 work with ***fruit*** defined to be a ***list***. In addition, we can extract elements from one list and insert them into another by a single ''splice'' operation. Given:

```
fruit:
        apple   pear

citrus:
        orange   grapefruit   lemon
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

we can splice the ***orange*** from ***citrus*** into ***fruit*** like this:

```
list<string>::iterator p = find_if(fruit.begin(),fruit.end(),initial('p'));
fruit.splice(p,citrus,citrus.begin());
```

The effect is to remove the first element from ***citrus*** (***citrus.begin()***) and place it just before the first element of ***fruit*** with the initial letter ***p***, thereby giving:

```
fruit:
    apple   orange   pear
citrus:
    grapefruit   lemon
```

Note that ***splice()*** doesn't copy elements the way ***insert()*** does (§16.3.6). It simply modifies the ***list*** data structures that refer to the element.

In addition to splicing individual elements and ranges, we can ***splice()*** all elements of a ***list***:

```
fruit.splice(fruit.begin(),citrus);
```

This yields:

```
fruit:
    grapefruit   lemon   apple   orange   pear
citrus:
    <empty>
```

Each version of ***splice()*** takes as its second argument the ***list*** from which elements are taken. This allows elements to be removed from their original ***list***. An iterator alone wouldn't allow that because there is no general way to determine the container holding an element given only an iterator to that element (§18.6).

Naturally, an iterator argument must be a valid iterator for the ***list*** into which it is supposed to point. That is, it must point to an element of that ***list*** or be the ***list***'s ***end()***. If not, the result is undefined and possibly disastrous. For example:

```
list<string>::iterator p = find_if(fruit.begin(),fruit.end(),initial('p'));
fruit.splice(p,citrus,citrus.begin());       // ok
fruit.splice(p,citrus,fruit.begin());        // error: fruit.begin() doesn't point into citrus
citrus.splice(p,fruit,fruit.begin());        // error: p doesn't point into citrus
```

The first ***splice()*** is ok even though ***citrus*** is empty.

A ***merge()*** combines two sorted lists by removing the elements from one ***list*** and entering them into the other while preserving order. For example,

```
f1:
    apple   quince   pear
f2:
    lemon   grapefruit   orange   lime
```

can be sorted and merged like this:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
f1.sort();
f2.sort();
f1.merge(f2);
```

This yields:

```
f1:
      apple   grapefruit   lemon   lime   orange   pear   quince
f2:
      <empty>
```

If one of the lists being merged is not sorted, *merge*( ) will still produce a list containing the union of elements of the two lists. However, there are no guarantees made about the order of the result.

Like *splice*( ), *merge*( ) refrains from copying elements. Instead, it removes elements from the source list and splices them into the target list. After an *x.merge*(*y*), the *y* list is empty.

17.2.2.2 Front Operations [cont.front]

Operations that refer to the first element of a *list* are provided to complement the operations referring to the last element provided by every sequence (§16.3.6):

```
template <class T, class A = allocator<T> > class list {
public:
      // ...
      // element access:

      reference front();                  // reference to first element
      const_reference front() const;

      void push_front(const T&);          // add new first element
      void pop_front();                   // remove first element

      // ...
};
```

The first element of a container is called its *front*. For a *list*, front operations are as efficient and convenient as back operations (§16.3.5). When there is a choice, back operations should be preferred over front operations. Code written using back operations can be used for a *vector* as well as for a *list*. So if there is a chance that the code written using a *list* will ever evolve into a generic algorithm applicable to a variety of containers, it is best to prefer the more widely available back operations. This is a special case of the rule that to achieve maximal flexibility, it is usually wise to use the minimal set of operations to do a task (§17.1.4.1).

17.2.2.3 Other Operations [cont.list.etc]

Insertion and removal of elements are particularly efficient for *list*s. This, of course, leads people to prefer *list*s when these operations are frequent. That, in turn, makes it worthwhile to support common ways of removing elements directly:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
template <class T, class A = allocator<T> > class list {
public:
        // ...

        void remove(const T& val);
        template <class Pred> void remove_if(Pred p);

        void unique();                                      // remove duplicates using ==
        template <class BinPred> void unique(BinPred b);   // remove duplicates using b

        void reverse();                                     // reverse order of elements
};
```

For example, given

```
fruit:
        apple   orange   grapefruit   lemon   orange   lime   pear   quince
```

we can remove all elements with the value `"orange"` like this:

```
fruit.remove("orange");
```

yielding:

```
fruit:
        apple   grapefruit   lemon   lime   pear   quince
```

Often, it is more interesting to remove all elements that meet some criterion rather than simply all elements with a given value. The *remove_if*() operation does that. For example,

```
fruit.remove_if(initial('l'));
```

removes every element with the initial *'l'* from *fruit* giving:

```
fruit:
        apple   grapefruit   pear   quince
```

A common reason for removing elements is to eliminate duplicates. The *unique*() operation is provided for that. For example:

```
fruit.sort();
fruit.unique();
```

The reason for sorting is that *unique* removes only duplicates that appear consecutively. For example, had fruit contained:

```
        apple   pear   apple   apple   pear
```

a simple *fruit.unique*() would have produced

```
        apple   pear   apple   pear
```

whereas sorting first gives:

```
        apple   pear
```

If only certain duplicates should be eliminated, we can provide a predicate to specify which

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

duplicates we want to remove. For example, we might define a binary predicate (§18.4.2) *initial2(x)* to compare *string*s that have the initial *x* but yield *false* for every *string* that doesn't. Given:

```
pear   pear   apple   apple
```

we can remove consecutive duplicates of every *fruit* with the initial *p* by a call

```
fruit.unique(initial2('p'));
```

This would give

```
pear   apple   apple
```

As noted in §16.3.2, we sometimes want to view a container in reverse order. For a *list*, it is possible to reverse the elements so that the first becomes the last, etc., without copying the elements. The *reverse()* operation is provided to do that. Given:

```
fruit:
        banana  cherry   lime   strawberry
```

fruit.reverse() produces:

```
fruit:
        strawberry  lime   cherry  banana
```

An element that is removed from a list is destroyed. However, note that destroying a pointer does not imply that the object it points to is *delete*d. If you want a container of pointers that *delete*s elements pointed to when the pointer is removed from the container or the container is destroyed, you must write one yourself (§17.8[13]).

17.2.3 Deque [cont.deque]

A *deque* (it rhymes with check) is a double-ended queue. That is, a *deque* is a sequence optimized so that operations at both ends are about as efficient as for a *list*, whereas subscripting approaches the efficiency of a *vector*:

```
template <class T, class A = allocator<T> > class std::deque {
        // types and operations like vector (§16.3.3, §16.3.5, §16.3.6)
        // plus front operations (§17.2.2.2) like list
};
```

Insertion and deletion of elements ''in the middle'' have *vector*-like (in)efficiencies rather than *list*-like efficiencies. Consequently, a *deque* is used where additions and deletions take place ''at the ends.'' For example, we might use a *deque* to model a section of a railroad or to represent a deck of cards in a game:

```
deque<car> siding_no_3;
deque<Card> bonus;
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

17.3 Sequence Adapters [cont.adapters]

The *vector*, *list*, and *deque* sequences cannot be built from each other without loss of efficiency. On the other hand, *stack*s and *queue*s can be elegantly and efficiently implemented using those three basic sequences. Therefore, *stack* and *queue* are defined not as separate containers, but as adaptors of basic containers.

A container adapter provides a restricted interface to a container. In particular, adapters do not provide iterators; they are intended to be used only through their specialized interfaces.

The techniques used to create a container adapter from a container are generally useful for non-intrusively adapting the interface of a class to the needs of its users.

17.3.1 Stack [cont.stack]

The *stack* container adapter is defined in <*stack*>. It is so simple that the best way to describe it is to present an implementation:

```
template <class T, class C = deque<T> > class std::stack {
protected:
      C c;
public:
      typedef typename C::value_type value_type;
      typedef typename C::size_type size_type;
      typedef C container_type;

      explicit stack(const C& a = C()) : c(a) { }

      bool empty() const { return c.empty(); }
      size_type size() const { return c.size(); }

      value_type& top() { return c.back(); }
      const value_type& top() const { return c.back(); }

      void push(const value_type& x) { c.push_back(x); }
      void pop() { c.pop_back(); }
};
```

That is, a *stack* is simply an interface to a container of the type passed to it as a template argument. All *stack* does is to eliminate the non-stack operations on its container from the interface and give *back*(), *push_back*(), and *pop_back*() their conventional names: *top*(), *push*(), and *pop*().

By default, a *stack* makes a *deque* to hold its elements, but any sequence that provides *back*(), *push_back*(), and *pop_back*() can be used. For example:

```
stack<char> s1;                  // uses a deque<char> to store elements of type char
stack< int,vector<int> > s2;     // uses a vector<int> to store elements of type int
```

It is possible to supply an existing container to initialize a stack. For example:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
void print_backwards(vector<int>& v)
{
      stack<int> state(v);   // initialize state from v
      while (state.size()) {
            cout << state.top();
            state.pop();
      }
}
```

However, the elements of a container argument are copied, so supplying an existing container can be expensive.

Elements are added to a ***stack*** using ***push_back***`()` on the container that is used to store the elements. Consequently, a ***stack*** cannot overflow as long as there is memory available on the machine for the container to acquire (using its allocator; see §19.4).

On the other hand, a ***stack*** can underflow:

```
void f()
{
      stack<int> s;
      s.push(2);
      if (s.empty()) {                          // underflow is preventable
            // don't pop
      }
      else {                                    // but not impossible
            s.pop();   // fine: s.size() becomes 0
            s.pop();   // undefined effect, probably bad
      }
}
```

Note that one does not ***pop***`()` an element to use it. Instead, the ***top***`()` is accessed and then ***pop***`()`'d when it is no longer needed. This is not too inconvenient, and it is more efficient when the ***pop***`()` isn't necessary:

```
void f(stack<char>& s)
{
      if (s.top()=='c') s.pop();     // remove optional initial 'c'
      // ...
}
```

Unlike fully developed containers, ***stack*** (like other container adapters) doesn't have an allocator template parameter. Instead, the ***stack*** and its users rely on the allocator from the container used to implement the ***stack***.

17.3.2 Queue [cont.queue]

Defined in <***queue***>, a ***queue*** is an interface to a container that allows the insertion of elements at the ***back***`()` and the extraction of elements at the ***front***`()`:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
template <class T, class C = deque<T> > class std::queue {
protected:
      C c;
public:
      typedef typename C::value_type value_type;
      typedef typename C::size_type size_type;
      typedef C container_type;

      explicit queue(const C& a = C()) : c(a) { }

      bool empty() const { return c.empty(); }
      size_type size() const { return c.size(); }

      value_type& front() { return c.front(); }
      const value_type& front() const { return c.front(); }

      value_type& back() { return c.back(); }
      const value_type& back() const { return c.back(); }

      void push(const value_type& x) { c.push_back(x); }
      void pop() { c.pop_front(); }
};
```

By default, a *queue* makes a *deque* to hold its elements, but any sequence that provides *front*(), *back*(), *push_back*(), and *pop_front*() can be used. Because a *vector* does not provide *pop_front*(), a *vector* cannot be used as the underlying container for a queue.

Queues seem to pop up somewhere in every system. One might define a server for a simple message-based system like this:

```
struct Message {
      // ...
};

void server(queue<Message>& q)
{
      while(!q.empty()) {
            Message& m = q.front();   // get hold of message
            m.service();               // call function to serve request
            q.pop();                   // destroy message
      }
}
```

Messages would be put on the *queue* using *push*().

If the requester and the server are running in different processes or threads, some form of synchronization of the queue access would be necessary. For example:

```
void server2(queue<Message>& q, Lock& lck)
{
      while(!q.empty()) {
            Message m;
            {     LockPtr h(lck);            // hold lock only while extracting message (see §14.4.7)
                  if (q.empty()) return;     // somebody else got the message
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
                m = q.front();
                q.pop();
            }
            m.service();                    // call function to serve request
        }
    }
```

There is no standard definition of concurrency or locking in C++ or in the world in general. Have a look to see what your system has to offer and how to access it from C++ (§17.8[8]).

17.3.3 Priority Queue [cont.pqueue]

A *priority_queue* is a queue in which each element is given a priority that controls the order in which the elements get to be *top()*:

```
template <class T, class C = vector<T>, class Cmp = less<typename C::value_type> >
class std::priority_queue {
protected:
    C c;
    Cmp cmp;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;

    explicit priority_queue(const Cmp& a1 = Cmp(), const C& a2 = C())
        : c(a2), cmp(a1) { }
    template <class In>
    priority_queue(In first, In last, const Cmp& = Cmp(), const C& = C());

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }

    const value_type& top() const { return c.front(); }

    void push(const value_type&);
    void pop();
};
```

The declaration of *priority_queue* is found in *<queue>*.

By default, the *priority_queue* simply compares elements using the < operator and *pop()* returns the largest element:

```
struct Message {
    int priority;
    bool operator<(const Message& x) const { return priority < x.priority; }
    // ...
};
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
void server(priority_queue<Message>& q, Lock& lck)
{
    while(!q.empty()) {
        Message m;
        {   LockPtr h(lck);             // hold lock only while extracting message (see §14.4.7)
            if (q.empty()) return;      // somebody else got the message
            m = q.top();
            q.pop();
        }
        m.service();    // call function to serve request
    }
}
```

This example differs from the *queue* example (§17.3.2) in that *messages* with higher priority will get served first. The order in which elements with equal priority come to the head of the queue is not defined. Two elements are considered of equal priority if neither has higher priority than the other (§17.4.1.5).

An alternative to < for comparison can be provided as a template argument. For example, we could sort strings in a case-insensitive manner by placing them in

```
priority_queue<string,Nocase> pq;    // use Nocase::operator()() for comparisons (§17.1.4.1)
```

using *pq.push()* and then retrieving them using *pq.top()* and *pq.pop()*.

Objects defined by templates given different template arguments are of different types (§13.6.3.1). For example:

```
void f(priority_queue<string>& pq1)
{
    pq = pq1;   // error: type mismatch
}
```

We can supply a comparison criterion without affecting the type of a *priority_queue* by providing a comparison object of the appropriate type as a constructor argument. For example:

```
struct String_cmp {   // type used to express comparison criteria at run time
    String_cmp(int n = 0);      // use comparison criteria n
    // ...
};

void g(priority_queue<string,String_cmp>& pq)
{
    priority_queue<string> pq2(String_cmp(nocase));
    pq = pq2;   // ok: pq and pq2 are of the same type, pq now also uses String_cmp(nocase)
}
```

Keeping elements in order isn't free, but it needn't be expensive either. One useful way of implementing a *priority_queue* is to use a tree structure to keep track of the relative positions of elements. This gives an *O(log(n))* cost of both *push()* and *pop()*.

By default, a *priority_queue* makes a *vector* to hold its elements, but any sequence that provides *front()*, *push_back()*, *pop_back()*, and random iterators can be used. A *priority_queue* is most likely implemented using a *heap* (§18.8).

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

17.4 Associative Containers [cont.assoc]

An *associative array* is one of the most useful general, user-defined types. In fact, it is often a built-in type in languages primarily concerned with text processing and symbolic processing. An associative array, often called a *map* and sometimes called a *dictionary*, keeps pairs of values. Given one value, called the *key*, we can access the other, called the *mapped value*. An associative array can be thought of as an array for which the index need not be an integer:

```
template<class K, class V> class Assoc {
public:
      V& operator[](const K&);  // return a reference to the V corresponding to K
      // ...
};
```

Thus, a key of type ***K*** names a mapped value of type ***V***.

Associative containers are a generalization of the notion of an associative array. The ***map*** is a traditional associative array, where a single value is associated with each unique key. A ***multimap*** is an associative array that allows duplicate elements for a given key, and ***set*** and ***multiset*** can be seen as degenerate associative arrays in which no value is associated with a key.

17.4.1 Map [cont.map]

A ***map*** is a sequence of (key,value) pairs that provides for fast retrieval based on the key. At most one value is held for each key; in other words, each key in a ***map*** is unique. A ***map*** provides bidirectional iterators (§19.2.1).

The ***map*** requires that a less-than operation exist for its key types (§17.1.4.1) and keeps its elements sorted so that iteration over a ***map*** occurs in order. For elements for which there is no obvious order or when there is no need to keep the container sorted, we might consider using a ***hash_map*** (§17.6).

17.4.1.1 Types [cont.map.types]

A ***map*** has the usual container member types (§16.3.1) plus a few relating to its specific function:

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key,T> > >
class std::map {
public:
      // types:

      typedef Key key_type;
      typedef T mapped_type;

      typedef pair<const Key, T> value_type;

      typedef Cmp key_compare;
      typedef A allocator_type;

      typedef typename A::reference reference;
      typedef typename A::const_reference const_reference;
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
        typedef implementation_defined1 iterator;
        typedef implementation_defined2 const_iterator;

        typedef typename A::size_type size_type;
        typedef typename A::difference_type difference_type;

        typedef std::reverse_iterator<iterator> reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
        // ...
};
```

Note that the *value_type* of a *map* is a (key,value) *pair*. The type of the mapped values is referred to as the *mapped_type*. Thus, a *map* is a sequence of *pair<const Key, mapped_type>* elements.

As usual, the actual iterator types are implementation-defined. Since a *map* most likely is implemented using some form of a tree, these iterators usually provide some form of tree traversal.

The reverse iterators are constructed from the standard *reverse_iterator* templates (§19.2.5).

17.4.1.2 Iterators and Pairs [cont.map.iter]

A *map* provides the usual set of functions that return iterators (§16.3.2):

```
template <class Key, class T, class Cmp = less<Key>,
        class A = allocator< pair<const Key,T> > > class map {
public:
        // ...
        // iterators:

        iterator begin();
        const_iterator begin() const;

        iterator end();
        const_iterator end() const;

        reverse_iterator rbegin();
        const_reverse_iterator rbegin() const;

        reverse_iterator rend();
        const_reverse_iterator rend() const;

        // ...
};
```

Iteration over a *map* is simply an iteration over a sequence of *pair<const Key, mapped_type>* elements. For example, we might print out the entries of a phone book like this:

```
void f(map<string,number>& phone_book)
{
        typedef map<string,number>::const_iterator CI;
        for (CI p = phone_book.begin(); p!=phone_book.end(); ++p)
                cout << p->first << '\t' << p->second << '\n';
}
```

A *map* iterator presents the elements in ascending order of its keys (§17.4.1.5). Therefore, the *phone_book* entries will be output in lexicographical order.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

We refer to the first element of any *pair* as *first* and the second as *second* independently of what types they actually are:

```
template <class T1, class T2> struct std::pair {
	typedef T1 first_type;
	typedef T2 second_type;

	T1 first;
	T2 second;

	pair() :first(T1()), second(T2()) { }
	pair(const T1& x, const T2& y) :first(x), second(y) { }
	template<class U, class V>
		pair(const pair<U, V>& p) :first(p.first), second(p.second) { }
};
```

The last constructor exists to allow conversions in the initializer (§13.6.2). For example:

```
pair<int,double> f(char c, int i)
{
	return pair<int,double>(c,i);	// conversions required
}
```

In a *map*, the key is the first element of the pair and the mapped value is the second.

The usefulness of *pair* is not limited to the implementation of *map*, so it is a standard library class in its own right. The definition of *pair* is found in *<utility>*. A function to make it convenient to create *pair*s is also provided:

```
template <class T1, class T2> pair<T1,T2> std::make_pair(T1 t1, T2 t2)
{
	return pair<T1,T2>(t1,t2);
}
```

A *pair* is by default initialized to the default values of its element types. In particular, this implies that elements of built-in types are initialized to *0* (§5.1.1) and *string*s are initialized to the empty string (§20.3.4). A type without a default constructor can be an element of a *pair* only provided the pair is explicitly initialized.

17.4.1.3 Subscripting [cont.map.element]

The characteristic *map* operation is the associative lookup provided by the subscript operator:

```
template <class Key, class T, class Cmp = less<Key>,
		class A = allocator< pair<const Key,T> > >
class map {
public:
	// ...

	mapped_type& operator[](const key_type& k);	// access element with key k

	// ...
};
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

The subscript operator performs a lookup on the key given as an index and returns the corresponding value. If the key isn't found, an element with the key and the default value of the ***mapped_type*** is inserted into the ***map***. For example:

```
void f()
{
    map<string,int> m;   // map starting out empty
    int x = m["Henry"]; // create new entry for "Henry", initialize to 0, return 0
    m["Harry"] = 7;     // create new entry for "Harry", initialize to 0, and assign 7
    int y = m["Henry"]; // return the value from "Henry"'s entry
    m["Harry"] = 9;     // change the value from "Harry"'s entry to 9
}
```

As a slightly more realistic example, consider a program that calculates sums of items presented as input in the form of (item-name,value) pairs such as

```
nail 100 hammer 2 saw 3 saw 4 hammer 7 nail 1000 nail 250
```

and also calculates the sum for each item. The main work can be done while reading the (item-name,value) pairs into a ***map***:

```
void readitems(map<string,int>& m)
{
    string word;
    int val = 0;
    while (cin >> word >> val) m[word] += val;
}
```

The subscript operation ***m[word]*** identifies the appropriate ***(string,int)*** pair and returns a reference to its ***int*** part. This code takes advantage of the fact that a new element gets its ***int*** value set to ***0*** by default.

A ***map*** constructed by ***readitems()*** can then be output using a conventional loop:

```
int main()
{
    map<string,int> tbl;
    readitems(tbl);

    int total = 0;
    typedef map<string,int>::const_iterator CI;

    for (CI p = tbl.begin(); p!=tbl.end(); ++p) {
        total += p->second;
        cout << p->first << '\t' << p->second << '\n';
    }

    cout << "----------------\ntotal\t" << total << '\n';

    return !cin;
}
```

Given the input above, the output is:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
hammer  9
nail    1350
saw     7
----------------
total   1366
```

Note that the items are printed in lexical order (§17.4.1, §17.4.1.5).

A subscripting operation must find the key in the *map*. This, of course, is not as cheap as subscripting an array with an integer. The cost is *O(log(size_of_map))*, which is acceptable for many applications. For applications for which this is too expensive, a hashed container is often the answer (§17.6).

Subscripting a *map* adds a default element when the key is not found. Therefore, there is no version of *operator[]()* for *const maps*. Furthermore, subscripting can be used only if the *mapped_type* (value type) has a default value. If the programmer simply wants to see if a key is present, the *find()* operation (§17.4.1.6) can be used to locate a *key* without modifying the *map*.

17.4.1.4 Constructors [cont.map.ctor]

A *map* provides the usual complement of constructors, etc. (§16.3.4) :

```
template <class Key, class T, class Cmp =less<Key>,
        class A =allocator<pair<const Key,T> > >
class map {
public:
    // ...
    // construct/copy/destroy:

    explicit map(const Cmp& = Cmp(), const A& = A());
    template <class In> map(In first, In last, const Cmp& = Cmp(), const A& = A());
    map(const map&);

    ~map();

    map& operator=(const map&);

    // ...
};
```

Copying a container implies allocating space for its elements and making copies of each element (§16.3.4). This can be very expensive and should be done only when necessary. Consequently, containers such as *maps* tend to be passed by reference.

The member template constructor takes a sequence of *pair<const Key,T>*s described by a pair input iterator *In*. It *insert()*s (§17.4.1.7) the elements from the sequence into the *map*.

17.4.1.5 Comparisons [cont.map.comp]

To find an element in a *map* given a key, the *map* operations must compare keys. Also, iterators traverse a *map* in order of increasing key values, so insertion will typically also compare keys (to place an element into a tree structure representing the *map*).

By default, the comparison used for keys is < (less than), but an alternative can be provided as a

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

template parameter or as a constructor argument (see §17.3.3). The comparison given is a comparison of keys, but the *value_type* of a *map* is a (key,value) pair. Consequently, *value_comp*() is provided to compare such pairs using the key comparison function:

```
template <class Key, class T, class Cmp = less<Key>,
        class A = allocator< pair<const Key,T> > >
class map {
public:
    // ...

    typedef Cmp key_compare;

    class value_compare : public binary_function<value_type,value_type,bool> {
    friend class map;
    protected:
        Cmp cmp;
        value_compare(Cmp c) : cmp(c) {}
    public:
        bool operator()(const T& x, const T& y) const { return cmp(x.first, y.first); }
    };

    key_compare key_comp() const;
    value_compare value_comp() const;

    // ...
};
```

For example:

```
map<string,int> m1;
map<string,int,Nocase> m2;                    // specify comparison type (§17.1.4.1)
map<string,int,String_cmp> m3;                // specify comparison type (§17.1.4.1)
map<string,int> m4(String_cmp(literary));     // pass comparison object
```

The *key_comp*() and *value_comp*() member functions make it possible to query a *map* for the kind of comparisons used for keys and values. This is usually done to supply the same comparison criterion to some other container or algorithm. For example:

```
void f(map<string,int>& m)
{
    map<string,int> mm;                       // compare using < by default
    map<string,int> mmm(m.key_comp());        // compare the way m does
    // ...
}
```

See §17.1.4.1 for an example of how to define a particular comparison and §18.4 for an explanation of function objects in general.

17.4.1.6 Map Operations [cont.map.map]

The crucial idea for *map*s and indeed for all associative containers is to gain information based on a key. Several specialized operations are provided for that:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key,T> > >
class map {
public:
      // ...
      // map operations:

      iterator find(const key_type& k);                    // find element with key k
      const_iterator find(const key_type& k) const;

      size_type count(const key_type& k) const;            // find number of elements with key k

      iterator lower_bound(const key_type& k);             // find first element with key k
      const_iterator lower_bound(const key_type& k) const;
      iterator upper_bound(const key_type& k);             // find first element with key greater than k
      const_iterator upper_bound(const key_type& k) const;

      pair<iterator,iterator> equal_range(const key_type& k);
      pair<const_iterator,const_iterator> equal_range(const key_type& k) const;

      // ...
};
```

A ***m.find(k)*** operation simply yields an iterator to an element with the key ***k***. If there is no such element, the iterator returned is ***m.end()***. For a container with unique keys, such as ***map*** and ***set***, the resulting iterator will point to the unique element with the key ***k***. For a container with non-unique keys, such as ***multimap*** and ***multiset***, the resulting iterator will point to the first element that has that key. For example:

```
void f(map<string,int>& m)
{
      map<string,int>::iterator p = m.find("Gold");
      if (p!=m.end()) {                                  // if "Gold" was found
            // ...
      }
      else if (m.find("Silver")!=m.end()) {    // look for "Silver"
            // ...
      }
      // ...
}
```

For a ***multimap*** (§17.4.2), finding the first match is rarely as useful as finding all matches; ***m.lower_bound(k)*** and ***m.upper_bound(k)*** give the beginning and the end of the subsequence of elements of ***m*** with the key ***k***. As usual, the end of a sequence is an iterator to the one-past-the-last element of the sequence. For example:

```
void f(multimap<string,int>& m)
{
      multimap<string,int>::iterator lb = m.lower_bound("Gold");
      multimap<string,int>::iterator ub = m.upper_bound("Gold");
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
        for (multimap<string,int>::iterator p = lb; p!=ub; ++p) {
                // ...
        }
}
```

Finding the upper bound and lower bound by two separate operations is neither elegant nor efficient. Consequently, the operation *equal_range*() is provided to deliver both. For example:

```
void f(multimap<string,int>& m)
{
        typedef multimap<string,int>::iterator MI;
        pair<MI,MI> g = m.equal_range("Gold");

        for (MI p = g.first; p!=g.second; ++p) {
                // ...
        }
}
```

If *lower_bound*(*k*) doesn't find *k*, it returns an iterator to the first element that has a key greater than *k*, or *end*() if no such greater element exists. This way of reporting failure is also used by *upper_bound*() and *equal_range*().

17.4.1.7 List Operations [cont.map.modifier]

The conventional way of entering a value into an associative array is simply to assign to it using subscripting. For example:

```
phone_book["Order department"] = 8226339;
```

This will make sure that the Order department has the desired entry in the *phone_book* independently of whether it had a prior entry. It is also possible to *insert*() entries directly and to remove entries using *erase*():

```
template <class Key, class T, class Cmp = less<Key>,
                class A = allocator< pair<const Key,T> > >
class map {
public:
        // ...
        // list operations:

        pair<iterator, bool> insert(const value_type& val);      // insert (key,value) pair
        iterator insert(iterator pos, const value_type& val);   // pos is just a hint
        template <class In> void insert(In first, In last);     // insert elements from sequence

        void erase(iterator pos);                        // erase the element pointed to
        size_type erase(const key_type& k);              // erase element with key k (if present)
        void erase(iterator first, iterator last);       // erase range
        void clear();

        // ...
};
```

The operation *m.insert*(*val*) attempts to add a (*Key*,*T*) pair *val* to *m*. Since *map*s rely on

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

unique keys, insertion takes place only if there is not already an element in the *m* with that key. The return value of *m.insert(val)* is a *pair<iterator,bool>*. The *bool* is *true* if *val* was actually inserted. The iterator refers to the element of *m* holding the key *k*. For example:

```
void f(map<string,int>& m)
{
    pair<string,int> p99("Paul",99);

    pair<map<string,int>::iterator,bool> p = m.insert(p99);
    if (p.second) {
        // "Paul" was inserted
    }
    else {
        // "Paul" was there already
    }
    map<string,int>::iterator i = p.first;    // points to m["Paul"]
    // ...
}
```

Usually, we do not care whether a key is newly inserted or was present in the *map* before the *insert()*. When we are interested, it is often because we want to register the fact that a value is in a *map* somewhere else (outside the *map*). The other two versions of *insert()* do not return an indication of whether a value was actually inserted.

Specifying a position, *insert(pos,val)*, is simply a hint to the implementation to start the search for the key *val* at *pos*. If the hint is good, significant performance improvements can result. If the hint is bad, you'd have done better without it both notationally and efficiency-wise. For example:

```
void f(map<string,int>& m)
{
    m["Dilbert"] = 3;    // neat, possibly less efficient
    m.insert(m.begin(),make_pair(const string("Dogbert"),99));    // ugly
}
```

In fact, `[]` is little more than a convenient notation for *insert()*. The result of *m[k]* is equivalent to the result of `(*(m.insert(make_pair(k,V())).first)).second`, where *V()* is the default value for the mapped type. When you understand that equivalence, you probably understand associative containers.

Because `[]` always uses *V()*, you cannot use subscripting on a *map* with a value type that does not have a default value. This is an unfortunate limitation of the standard associative containers. However, the requirement of a default value is not a fundamental property of associative containers (see §17.6.2).

You can erase elements specified by a key. For example:

```
void f(map<string,int>& m)
{
    int count = phone_book.erase("Ratbert");
    // ...
}
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

The integer returned is the number of erased elements. In particular, *count* is *0* if there was no element with the key "*Ratbert*" to erase. For a *multimap* or *multiset*, the value can be larger than *1*. Alternatively, one can erase an element given an iterator pointing to it or a range of elements given a sequence. For example:

```
void g(map<string,int>& m)
{
    m.erase(m.find("Catbert"));
    m.erase(m.find("Alice"),m.find("Wally"));
}
```

Naturally, it is faster to erase an element for which you already have an iterator than to first find the element given its key and then erase it. After *erase*(), the iterator cannot be used again because the element to which it pointed is no longer there. Erasing *end*() is harmless.

17.4.1.8 Other Functions [cont.map.etc]

Finally, a *map* provides the usual functions dealing with the number of elements and a specialized *swap*():

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key,T> > >
class map {
public:
    // ...
    // capacity:

    size_type size() const;          // number of elements
    size_type max_size() const;      // size of largest possible map
    bool empty() const { return size()==0; }

    void swap(map&);
};
```

As usual, a value returned by *size*() or *max_size*() is a number of elements.

In addition, *map* provides ==, !=, <, >, <=, >=, and *swap*() as nonmember functions:

```
template <class Key, class T, class Cmp, class A>
bool operator==(const map<Key,T,Cmp,A>&, const map<Key,T,Cmp,A>&);

// similarly !=, <, >, <=, and >=

template <class Key, class T, class Cmp, class A>
void swap(map<Key,T,Cmp,A>&, map<Key,T,Cmp,A>&);
```

Why would anyone want to compare two *map*s? When we specifically compare two *map*s, we usually want to know not just if the *map*s differ, but also how they differ if they do. In such cases, we don't use == or !=. However, by providing ==, <, and *swap*() for every container, we make it possible to write algorithms that can be applied to every container. For example, these functions allow us to *sort*() a *vector* of *map*s and to have a *set* of *map*s.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

17.4.2 Multimap [cont.multimap]

A ***multimap*** is like a ***map***, except that it allows duplicate keys:

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key,T> > >
class std::multimap {
public:
      // like map, except:

      iterator insert(const value_type&);    // returns iterator, not pair

      // no subscript operator []
};
```

For example (using ***Cstring_less*** from §17.1.4.1 to compare C-style strings):

```
void f(map<char*,int,Cstring_less>& m, multimap<char*,int,Cstring_less>& mm)
{
      m.insert(make_pair("x",4));
      m.insert(make_pair("x",5)); // no effect: there already is an entry for "x" (§17.4.1.7)
      // now m["x"] == 4

      mm.insert(make_pair("x",4));
      mm.insert(make_pair("x",5));
      // mm now holds both ("x",4) and ("x",5)
}
```

This implies that ***multimap*** cannot support subscripting by key values in the way ***map*** does. The ***equal_range***( ), ***lower_bound***( ), and ***upper_bound***( ) operations (§17.4.1.6) are the primary means of accessing multiple values with the same key.

Naturally, where several values can exist for a single key, a ***multimap*** is preferred over a ***map***. That happens far more often than people first think when they hear about ***multimap***. In some ways, a ***multimap*** is even cleaner and more elegant than a ***map***.

Because a person can easily have several phone numbers, a phone book is a good example of a ***multimap***. I might print my phone numbers like this:

```
void print_numbers(const multimap<string,int>& phone_book)
{
      typedef multimap<string,int>::const_iterator I;
      pair<I,I> b = phone_book.equal_range("Stroustrup");
      for (I i = b.first; i != b.second; ++i) cout << i->second << '\n';
}
```

For a ***multimap***, the argument to ***insert***( ) is always inserted. Consequently, the ***multimap::insert***( ) returns an iterator rather than a ***pair<iterator,bool>*** like ***map*** does. For uniformity, the library could have provided the general form of ***insert***( ) for both ***map*** and ***multimap*** even though the ***bool*** would have been redundant for a ***multimap***. Yet another design alternative would have been to provide a simple ***insert***( ) that didn't return a ***bool*** in either case and then supply users of ***map*** with some other way of figuring out whether a key was newly inserted. This is a case in which different interface design ideas clash.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

17.4.3 Set [cont.set]

A *set* can be seen as a *map* (§17.4.1), where the values are irrelevant, so we keep track of only the keys. This leads to only minor changes to the user interface:

```
template <class Key, class Cmp = less<Key>, class A = allocator<Key> >
class std::set {
public:
        // like map except:

        typedef Key value_type;          // the key itself is the value
        typedef Cmp value_compare;
        // no subscript operator []
};
```

Defining *value_type* as the *key_type* type is a trick to allow code that uses *map*s and *set*s to be identical in many cases.

Note that *set* relies on a comparison operation (by default <) rather than equality (==). This implies that equivalence of elements is defined by inequality (§17.1.4.1) and that iteration through a *set* has a well-defined order.

Like *map*, *set* provides ==, !=, <, >, <=, >=, and *swap*().

17.4.4 Multiset [cont.multiset]

A *multiset* is a *set* that allows duplicate keys:

```
template <class Key, class T, class Cmp = less<Key>, class A = allocator<Key> >
class std::multiset {
public:
        // like set, except:
        iterator insert(const value_type&);    // returns iterator, not pair
};
```

The *equal_range*(), *lower_bound*(), and *upper_bound*() operations (§17.4.1.6) are the primary means of accessing multiple occurrences of a key.

17.5 Almost Containers [cont.etc]

Built-in arrays (§5.2), *string*s (Chapter 20), *valarray*s (§22.4), and *bitset*s (§17.5.3) hold elements and can therefore be considered containers for many purposes. However, each lacks some aspect or other of the standard container interface, so these ‘‘almost containers’’ are not completely interchangeable with fully developed containers such as *vector* and *list*.

17.5.1 String [cont.string]

A *basic_string* provides subscripting, random-access iterators, and most of the notational conveniences of a container (Chapter 20). However, *basic_string* does not provide as wide a selection of types as elements. It also is optimized for use as a string of characters and is typically used in ways that differ significantly from a container.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

17.5.2 Valarray [cont.valarray]

A *valarray* (§22.4) is a vector for optimized numeric computation. Consequently, a *valarray* doesn't attempt to be a general container. A *valarray* provides many useful numeric operations. However, of the standard container operations (§17.1.1), it offers only *size*() and a subscript operator (§22.4.2). A pointer to an element of a *valarray* is a random-access iterator (§19.2.1).

17.5.3 Bitset [cont.bitset]

Often, aspects of a system, such as the state of an input stream (§21.3.3), are represented as a set of flags indicating binary conditions such as good/bad, true/false, and on/off. C++ supports the notion of small sets of flags efficiently through bitwise operations on integers (§6.2.4). These operations include & (and), | (or), ^ (exclusive or), << (shift left), and >> (shift right). Class *bitset<N>* generalizes this notion and offers greater convenience by providing operations on a set of *N* bits indexed from *0* through *N−1*, where *N* is known at compile time. For sets of bits that don't fit into a *long int*, using a *bitset* is much more convenient than using integers directly. For smaller sets, there may be an efficiency tradeoff. If you want to name the bits, rather than numbering them, using a *set* (§17.4.3), an enumeration (§4.8), or a bitfield (§C.8.1) are alternatives.

A *bitset<N>* is an array of *N* bits. A *bitset* differs from a *vector<bool>* (§16.3.11) by being of fixed size, from *set* (§17.4.3) by having its bits indexed by integers rather than associatively by value, and from both *vector<bool>* and *set* by providing operations to manipulate the bits.

It is not possible to address a single bit directly using a built-in pointer (§5.1). Consequently, *bitset* provides a reference-to-bit type. This is actually a generally useful technique for addressing objects for which a built-in pointer for some reason is unsuitable:

```
template<size_t N> class std::bitset {
public:

    class reference {           // reference to a single bit:
        friend class bitset;
        reference();
    public:                                     // b[i] refers to the (i+1)'th bit:
        ~reference();
        reference& operator=(bool x);               // for b[i] = x;
        reference& operator=(const reference&);     // for b[i] = b[j];
        bool operator~() const;                     // return ~b[i]
        operator bool() const;                      // for x = b[i];
        reference& flip();                          // b[i].flip();
    };

    // ...
};
```

The *bitset* template is defined in namespace *std* and presented in *<bitset>*.

For historical reasons, *bitset* differs somewhat in style from other standard library classes. For example, if an index (also known as a *bit position*) is out of range, an *out_of_range* exception is thrown. No iterators are provided. Bit positions are numbered right to left in the same way bits often are in a word, so the value of *b[i]* is *pow(i,2)*. Thus, a bitset can be thought of as an *N*-bit binary number:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

position:	9	8	7	6	5	4	3	2	1	0
bitset<10>:	1	1	1	1	0	1	1	1	0	1

17.5.3.1 Constructors [cont.bitset.ctor]

A ***bitset*** can be constructed with default values, from the bits in an ***unsigned long int***, or from a ***string***:

```
template<size_t N> class bitset {
public:
      // ...
      // constructors:

      bitset();                         // N zero-bits
      bitset(unsigned long val);        // bits from val

      template<class Ch, class Tr, class A>                  // Tr is a character trait (§20.2)
      explicit bitset(const basic_string<Ch,Tr,A>& str,      // bits from string str
            basic_string<Ch,Tr,A>::size_type pos = 0,
            basic_string<Ch,Tr,A>::size_type n = basic_string<Ch,Tr,A>::npos);

      // ...
};
```

The default value of a bit is ***0***. When an ***unsigned long int*** argument is supplied, each bit in the integer is used to initialize the corresponding bit in the bitset (if any). A ***basic_string*** (Chapter 20) argument does the same, except that the character ***´0´*** gives the bitvalue ***0***, the character ***´1´*** gives the bitvalue ***1***, and other characters cause an ***invalid_argument*** exception to be thrown. By default, a complete string is used for initialization. However, in the style of a ***basic_string*** constructor (§20.3.4), a user can specify that only the range of characters from ***pos*** to the end of the string or to ***pos+n*** are to be used. For example:

```
void f()
{
      bitset<10> b1;   // all 0

      bitset<16> b2 = 0xaaaa;                        // 1010101010101010
      bitset<32> b3 = 0xaaaa;                        // 00000000000000001010101010101010

      bitset<10> b4("1010101010");                   // 1010101010
      bitset<10> b5("10110111011110",4);             // 0111011110
      bitset<10> b6("10110111011110",2,8);           // 0011011101

      bitset<10> b7("n0g00d");                       // invalid_argument thrown
      bitset<10> b8 = "n0g00d";                      // error: no char* to bitset conversion
}
```

A key idea in the design of ***bitset*** is that an optimized implementation can be provided for bitsets that fit in a single word. The interface reflects this assumption.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

17.5.3.2 Bit Manipulation Operations [cont.bitset.oper]

A *bitset* provides the operators for accessing individual bits and for manipulating all bits in the set:

```
template<size_t N> class std::bitset {
public:
        // ...
        // bitset operations:

        reference operator[](size_t pos);              // b[i]

        bitset& operator&=(const bitset& s);           // and
        bitset& operator|=(const bitset& s);           // or
        bitset& operator^=(const bitset& s);           // exclusive or

        bitset& operator<<=(size_t n);                 // logical left shift (fill with zeros)
        bitset& operator>>=(size_t n);                 // logical right shift (fill with zeros)

        bitset& set();                                 // set every bit to 1
        bitset& set(size_t pos, int val = 1);          // b[pos]=val

        bitset& reset();                               // set every bit to 0
        bitset& reset(size_t pos);                     // b[pos]=0

        bitset& flip();                                // change the value of every bit
        bitset& flip(size_t pos);                      // change the value of b[pos]

        bitset operator~() const { return bitset<N>(*this).flip(); }     // make complement set
        bitset operator<<(size_t n) const { return bitset<N>(*this)<<=n; }  // make shifted set
        bitset operator>>(size_t n) const { return bitset<N>(*this)>>=n; }  // make shifted set

        // ...
};
```

The subscript operator throws *out_of_range* if the subscript is out of range. There is no unchecked subscript operation.

The *bitset&* returned by these operations is **this*. An operator returning a *bitset* (rather than a *bitset&*) makes a copy of **this*, applies its operation to that copy, and returns the result. In particular, >> and << really are shift operations rather than I/O operations. The output operator for a *bitset* is a << that takes an *ostream* and a *bitset* (§17.5.3.3).

When bits are shifted, a logical (rather than cyclic) shift is used. That implies that some bits ‘‘fall off the end’’ and that some positions get the default value 0. Note that because *size_t* is an unsigned type, it is not possible to shift by a negative number. It does, however, imply that *b<<-1* shifts by a very large positive value, thus leaving every bit of the *bitset b* with the value *0*. Your compiler should warn against this.

17.5.3.3 Other Operations [cont.bitset.etc]

A *bitset* also supports common operations such as *size*(), ==, I/O , etc.:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
template<size_t N> class bitset {
public:
	// ...

	unsigned long to_ulong() const;

	template <class Ch, class Tr, class A> basic_string<Ch,Tr,A> to_string() const;

	size_t count() const;			// number of bits with value 1
	size_t size() const { return N; }	// number of bits

	bool operator==(const bitset& s) const;
	bool operator!=(const bitset& s) const;

	bool test(size_t pos) const;		// true if b[pos] is 1
	bool any() const;			// true if any bit is 1
	bool none() const;			// true if no bit is 1
};
```

The operations *to_ulong()* and *to_string()* provide the inverse operations to the constructors. To avoid nonobvious conversions, named operations were preferred over conversion operations. If the value of the *bitset* has so many significant bits that it cannot be represented as an *unsigned long*, *to_ulong()* throws *overflow_error*.

The *to_string()* operation produces a string of the desired type holding a sequence of *'0'* and *'1'* characters; *basic_string* is the template used to implement strings (Chapter 20). We could use *to_string* to write out the binary representation of an *int*:

```
void binary(int i)
{
	bitset<8*sizeof(int)> b = i;	// assume 8-bit byte (see also §22.2)
	cout << b.template to_string<char>() << '\n';
}
```

Unfortunately, invoking an explicitly qualified member template requires a rather elaborate and rare syntax (§C.13.6).

In addition to the member functions, *bitset* provides binary & (and), | (or), ^ (exclusive or), and the usual I/O operators:

```
template<size_t N> bitset<N>& std::operator&(const bitset<N>&, const bitset<N>&);
template<size_t N> bitset<N>& std::operator|(const bitset<N>&, const bitset<N>&);
template<size_t N> bitset<N>& std::operator^(const bitset<N>&, const bitset<N>&);

template <class charT, class Tr, size_t N>
basic_istream<charT,Tr>& std::operator>>(basic_istream<charT,Tr>&, bitset<N>&);
template <class charT, class Tr, size_t N>
basic_ostream<charT,Tr>& std::operator<<(basic_ostream<charT,Tr>&, const bitset<N>&);
```

We can therefore write out a bitset without first converting it to a string. For example:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
void binary(int i)
{
      bitset<8*sizeof(int)> b = i;       // assume 8-bit byte (see also §22.2)
      cout << b << '\n';
}
```

This prints the bits represented as *1*s and *0*s left-to-right, with the most significant bit leftmost.

17.5.4 Built-In Arrays [cont.array]

A built-in array supplies subscripting and random-access iterators in the form of ordinary pointers (§2.7.2). However, an array doesn't know its own size, so users must keep track of that size. In general, an array doesn't provide the standard member operations and types.

It is possible, and sometimes useful, to provide an ordinary array in a guise that provides the notational convenience of a standard container without changing its low-level nature:

```
template<class T, int max> struct c_array {
      typedef T value_type;

      typedef T* iterator;
      typedef const T* const_iterator;

      typedef T& reference;
      typedef const T& const_reference;

      T v[max];
      operator T*() { return v; }

      reference operator[](size_t i) { return v[i]; }
      const_reference operator[](size_t i) const { return v[i]; }

      iterator begin() { return v; }
      const_iterator begin() const { return v; }

      iterator end() { return v+max; }
      const_iterator end() const { return v+max; }

      ptrdiff_t size() const { return max; }
};
```

The *c_array* template is not part of the standard library. It is presented here as a simple example of how to fit a ''foreign'' container into the standard container framework. It can be used with standard algorithms (Chapter 18) using *begin*(), *end*(), etc. It can be allocated on the stack without any indirect use of dynamic memory. Also, it can be passed to a C-style function that expects a pointer. For example:

```
void f(int* p, int sz);      // C-style

void g()
{
      c_array<int,10> a;
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
    f(a,a.size());                                           // C-style use
    c_array<int,10>::iterator p = find(a.begin(),a.end(),777);  // C++/STL style use
    // ...
}
```

17.6 Defining a New Container [cont.hash]

The standard containers provide a framework to which a user can add. Here, I show how to provide a container in such a way that it can be used interchangeably with the standard containers wherever reasonable. The implementation is meant to be realistic, but it is not optimal. The interface is chosen to be very close to that of existing, widely-available, and high-quality implementations of the notion of a *hash_map*. Use the *hash_map* provided here to study the general issues. Then, use a supported *hash_map* for production use.

17.6.1 Hash_map [cont.hash.map]

A *map* is an associative container that accepts almost any type as its element type. It does that by relying only on a less-than operation for comparing elements (§17.4.1.5). However, if we know more about a key type we can often reduce the time needed to find an element by providing a hash function and implementing a container as a hash table.

A hash function is a function that quickly maps a value to an index in such a way that two distinct values rarely end up with the same index. Basically, a hash table is implemented by placing a value at its index, unless another value is already placed there, and ‘‘nearby’’ if one is. Finding an element placed at its index is fast, and finding one ‘‘nearby’’ is not slow, provided equality testing is reasonably fast. Consequently, it is not uncommon for a *hash_map* to provide five to ten times faster lookup than a *map* for larger containers, where the speed of lookup matters most. On the other hand, a *hash_map* with an ill-chosen hash function can be much slower than a *map*.

There are many ways of implementing a hash table. The interface of *hash_map* is designed to differ from that of the standard associative containers only where necessary to gain performance through hashing. The most fundamental difference between a *map* and a *hash_map* is that a *map* requires a < for its element type, while a *hash_map* requires an == and a hash function. Thus, a *hash_map* must differ from a *map* in the non-default ways of creating one. For example:

```
map<string,int> m1;                     // compare strings using <
map<string,int,Nocase> m2;              // compare strings using Nocase() (§17.1.4.1)

hash_map<string,int> hm1;               // hash using Hash<string>() (§17.6.2.3), compare using ==
hash_map<string,int,hfct> hm2;          // hash using hfct(), compare using ==
hash_map<string,int,hfct,eql> hm3;      // hash using hfct(), compare using eql
```

A container using hashed lookup is implemented using one or more tables. In addition to holding its elements, the container needs to keep track of which values have been associated with each hashed value (‘‘index’’ in the prior explanation); this is done using a ‘‘hash table.’’ Most hash table implementations seriously degrade in performance if that table gets ‘‘too full,’’ say 75% full. Consequently, the *hash_map* defined next is automatically resized when it gets too full. However, resizing can be expensive, so it is useful to be able to specify an initial size.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

Thus, a first approximation of a ***hash_map*** looks like this:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<T> >
class hash_map {
    // like map, except:

    typedef H Hasher;
    typedef EQ key_equal;

    hash_map(const T& dv =T(), size_type n =101, const H& hf =H(), const EQ& =EQ());
    template<class In> hash_map(In first, In last,
        const T& dv =T(), size_type n =101, const H& hf =H(), const EQ& =EQ());
};
```

Basically, this is the ***map*** interface (§17.4.1.4), with < replaced by == and a hash function.

The uses of a ***map*** in this book so far (§3.7.4, §6.1, §17.4.1) can be converted to use a ***hash_map*** simply by changing the name ***map*** to ***hash_map***. Often, a change between a ***map*** and a ***hash_map*** can be eased by using ***typedef***. For example:

```
typedef hash_map<string,record> Map;
Map dictionary;
```

The ***typedef*** is also useful to further hide the actual type of the dictionary from its users.

Though not strictly correct, I think of the tradeoff between a ***map*** and a ***hash_map*** as simply a space/time tradeoff. If efficiency isn't an issue, it isn't worth wasting time choosing between them: either will do well. For large and heavily used tables, ***hash_map*** has a definite speed advantage and should be used unless space is a premium. Even then, I might consider other ways of saving space before choosing a ''plain'' ***map***. Actual measurement is essential to avoid optimizing the wrong code.

The key to efficient hashing is the quality of the hash function. If a good hash function isn't available, a ***map*** can easily outperform a ***hash_map***. Hashing based on a C-style string, a ***string***, or an integer is usually very effective. However, it is worth remembering that the effectiveness of a hash function critically depends on the actual values being hashed (§17.8[35]). A ***hash_map*** must be used where < is not defined or is unsuitable for the intended key. Conversely, a hash function does not define an ordering the way < does, so a ***map*** must be used when it is important to keep the elements sorted.

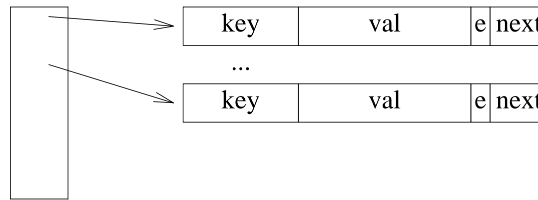
Like ***map***, ***hash_map*** provides ***find***() to allow a programmer to determine whether a key has been inserted.

17.6.2 Representation and Construction [cont.hash.rep]

Many different implementations of a ***hash_map*** are possible. Here, I use one that is reasonably fast and whose most important operations are fairly simple. The key operations are the constructors, the lookup (operator [ ]), the resize operation, and the operation removing an element (***erase***()).

The simple implementation chosen here relies on a hash table that is a ***vector*** of pointers to entries. Each ***Entry*** holds a ***key***, a ***value***, a pointer to the next ***Entry*** (if any) with the same hash value, and an ***erased*** bit :

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.



Expressed as declarations, it looks like this:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<T> >
class hash_map {
    // ...
private:                // representation
    struct Entry {
        key_type key;
        mapped_type val;
        Entry* next;    // hash overflow link
        bool erased;
        Entry(key_type k, mapped_type v, Entry* n)
            : key(k), val(v), next(n), erased(false) { }
    };

    vector<Entry> v;    // the actual entries
    vector<Entry*> b;   // the hash table: pointers into v

    // ...
};
```

Note the *erased* bit. The way several values with the same hash value are handled here makes it hard to remove an element. So instead of actually removing an element when *erase*() is called, I simply mark the element *erased* and ignore it until the table is resized.

In addition to the main data structure, a *hash_map* needs a few pieces of administrative data. Naturally, each constructor needs to set up all of this. For example:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<T> >
class hash_map {
    // ...

    hash_map(const T& dv =T(), size_type n =101, const H& h =H(), const EQ& e =EQ())
        : default_value(dv), b(n), no_of_erased(0), hash(h), eq(e)
    {
        set_load();                         // defaults
        v.reserve(max_load*b.size());       // reserve space for growth
    }

    void set_load(float m = 0.7, float g = 1.6) { max_load = m; grow = g; }

    // ...
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
private:
    float max_load;             // keep v.size()<=b.size()*max_load
    float grow;                 // when necessary, resize(bucket_count()*grow)

    size_type no_of_erased;     // number of entries in v occupied by erased elements

    Hasher hash;                // hash function
    key_equal eq;               // equality

    const T default_value;      // default value used by []
};
```

The standard associative containers require that a mapped type have a default value (§17.4.1.7). This restriction is not logically necessary and can be inconvenient. Making the default value an argument allows us to write:

```
hash_map<string,Number> phone_book1;                  // default: Number()
hash_map<string,Number> phone_book2(Number(411));     // default: Number(411)
```

17.6.2.1 Lookup [cont.hash.lookup]

Finally, we can provide the crucial lookup operations:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<T> >
class hash_map {
    // ...
    mapped_type& operator[](const key_type& k);

    iterator find(const key_type&);
    const_iterator find(const key_type&) const;
    // ...
};
```

To find a *value*, *operator*[]() uses a hash function to find an index in the hash table for the *key*. It then searches through the entries until it finds a matching *key*. The *value* in that *Entry* is the one we are seeking. If it is not found, a default value is entered:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<T> >
mapped_type& hash_map::operator[](const key_type& k)
{
    size_type i = hash(k)%b.size();                    // hash

    for(Entry* p = b[i]; p; p = p->next)   // search among entries hashed to i
        if (eq(k,p->key)) {                // found
            if (p->erased) {               // re-insert
                p->erased = false;
                no_of_erased--;
                return p->val = default_value;
            }
            return p->val;
        }
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
        // not found:

        if (b.size()*max_load < v.size()) {  // if ''too full''
              resize(b.size()*grow);          // grow
              return operator[](k);           // rehash
        }

        v.push_back(Entry(k,default_value,b[i]));     // add Entry
        b[i] = &v.back();                             // point to new element

        return b[i]->val;
}
```

Unlike *map*, *hash_map* doesn't rely on an equality test synthesized from a less-than operation (§17.1.4.1). This is because of the call of *eq*() in the loop that looks through elements with the same hash value. This loop is crucial to the performance of the lookup, and for common and obvious key types such as *string* and C-style strings, the overhead of an extra comparison could be significant.

I could have used a *set<Entry>* to represent the set of values that have the same hash value. However, if we have a good hash function (*hash*()) and an appropriately-sized hash table (*b*), most such sets will have exactly one element. Consequently, I linked the elements of that set together using the *next* field of *Entry* (§17.8[27]).

Note that *b* keeps pointers to elements of *v* and that elements are added to *v*. In general, *push_back*() can cause reallocation and thus invalidate pointers to elements (§16.3.5). However, in this case constructors (§17.6.2) and *resize*() carefully *reserve*() enough space so that no unexpected reallocation happens.

17.6.2.2 Erase and Rehash [cont.hash.erase]

Hashed lookup becomes inefficient when the table gets too full. To lower the chance of that happening, the table is automatically *resize*()d by the subscript operator. The *set_load*() (§17.6.2) provides a way of controlling when and how resizing happens. Other functions are provided to allow a programmer to observe the state of a *hash_map*:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<T> >
class hash_map {
      // ...

      void resize(size_type n);                  // make the size of the hash table n

      void erase(iterator position);             // erase the element pointed to

      size_type size() const { return v.size()-no_of_erased; }  // number of elements

      size_type bucket_count() const { return b.size(); }       // size of hash table

      Hasher hash_fun() const { return hash; }                   // hash function used
      key_equal key_eq() const { return eq; }                    // equality used
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
    // ...
};
```

The *resize*() operation is essential, reasonably simple, and potentially expensive:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<T> >
void hash_map::resize(size_type s)
{
    if (s <= b.size()) return;
    b.resize(s);    // add s-b.size() pointers
    b.clear();
    v.reserve(s*max_load);    // if v needs to reallocate, let it happen now

    if (no_of_erased) {    // really remove erased elements
        for (size_type i = v.size()-1; 0<=i; i--)
            if (v[i].erased) {
                v.erase(&v[i]);
                if (--no_of_erased == 0) break;
            }
    }

    for (size_type i = 0; i<v.size(); i++) {             // rehash:
        size_type ii = hash(v[i].key)%b.size();          // hash
        v[i].next = b[ii];                               // link
        b[ii] = &v[i];
    }
}
```

If necessary, a user can ''manually'' call *resize*() to ensure that the cost is incurred at a predictable time. I have found a *resize*() operation important in some applications, but it is not fundamental to the notion of hash tables. Some implementation strategies don't need it.

All of the real work is done elsewhere (and only if a *hash_map* is resized), so *erase*() is trivial:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<T> >
void hash_map::erase(iterator p)    // erase the element pointed to
{
    if (p->erased == false) no_of_erased++;
    p->erased = true;
}
```

17.6.2.3 Hashing [cont.hasher]

To complete *hash_map*::*operator*[]() , we need to define *hash*() and *eq*() . For reasons that will become clear in §18.4, a hash function is best defined as *operator*()() for a function object:

```
template <class T> struct Hash : unary_function<T, size_t> {
    size_t operator()(const T& key) const;
};
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

A good hash function takes a key and returns an integer so that different keys yield different integers with high probability. Choosing a good hash function is an art. However, exclusive-or'ing the bits of the key's representation into an integer is often acceptable:

```
template <class T> size_t Hash<T>::operator()(const T& key) const
{
      size_t res = 0;

      size_t len = sizeof(T);
      const char* p = reinterpret_cast<const char*>(&key);

      while (len--) res = (res<<1)^*p++; // use bytes of key's representation
      return res;
}
```

The use of ***reinterpret_cast*** (§6.2.7) is a good indication that something unsavory is going on and that we can do better in cases when we know more about the object being hashed. In particular, if an object contains a pointer, if the object is large, or if the alignment requirements on members have left unused space (''holes'') in the representation, we can usually do better (see §17.8[29]).

A C-style string is a pointer (to the characters), and a ***string*** contains a pointer. Consequently, specializations are in order:

```
size_t Hash<char*>::operator()(const char* key) const
{
      size_t res = 0;

      while (*key) res = (res<<1)^*key++;        // use int value of characters
      return res;
}

template <class C>
size_t Hash< basic_string<C> >::operator()(const basic_string<C>& key) const
{
      size_t res = 0;

      typedef basic_string<C>::const_iterator CI;
      CI p = key.begin();
      CI end = key.end();

      while (p!=end) res = (res<<1)^*p++;        // use int value of characters
      return res;
}
```

An implementation of ***hash_map*** will include hash functions for at least integer and string keys. For more adventurous key types, the user may have to help out with suitable specializations. Experimentation supported by good measurement is essential when choosing a hash function. Intuition tends to work poorly in this area.

To complete the ***hash_map***, we need to define the iterators and a minor host of trivial functions; this is left as an exercise (§17.8[34]).

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

17.6.3 Other Hashed Associative Containers [cont.hash.other]

For consistency and completeness, the ***hash_map*** should have matching ***hash_set***, ***hash_multimap***, and ***hash_multiset***. Their definitions are obvious from those of ***hash_map***, ***map***, ***multimap***, ***set***, and ***multiset***, so I leave these as an exercise (§17.8[34]). Good public domain and commercial implementations of these hashed associative containers are available. For real programs, these should be preferred to locally concocted versions, such as mine.

17.7 Advice [cont.advice]

[1] By default, use ***vector*** when you need a container; §17.1.
[2] Know the cost (complexity, big-O measure) of every operation you use frequently; §17.1.2.
[3] The interface, implementation, and representation of a container are distinct concepts. Don't confuse them; §17.1.3.
[4] You can sort and search according to a variety of criteria; §17.1.4.1.
[5] Do not use a C-style string as a key unless you supply a suitable comparison criterion; §17.1.4.1.
[6] You can define a comparison criteria so that equivalent, yet different, key values map to the same key; §17.1.4.1.
[7] Prefer operations on the end of a sequence (***back***-operations) when inserting and deleting elements; §17.1.4.1.
[8] Use ***list*** when you need to do many insertions and deletions from the front or the middle of a container; §17.2.2.
[9] Use ***map*** or ***multimap*** when you primarily access elements by key; §17.4.1.
[10] Use the minimal set of operations to gain maximum flexibility; §17.1.1
[11] Prefer a ***map*** to a ***hash_map*** if the elements need to be kept in order; §17.6.1.
[12] Prefer a ***hash_map*** to a ***map*** when speed of lookup is essential; §17.6.1.
[13] Prefer a ***hash_map*** to a ***map*** if no less-than operation can be defined for the elements; §17.6.1.
[14] Use ***find*()** when you need to check if a key is in an associative container; §17.4.1.6.
[15] ***Use equal_range*()** to find all elements of a given key in an associative container; §17.4.1.6.
[16] Use ***multimap*** when several values need to be kept for a single key; §17.4.2.
[17] Use ***set*** or ***multiset*** when the key itself is the only value you need to keep; §17.4.3.

17.8 Exercises [cont.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. Then, look at your implementation's version of the containers and their operations.

1. (*2.5) Understand the ***O*()** notation (§17.1.2). Do some measurements of operations on standard containers to determine the constant factors involved.
2. (*2) Many phone numbers don't fit into a ***long***. Write a ***phone_number*** type and a class that provides a set of useful operations on a container of ***phone_numbers***.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

3. (∗2) Write a program that lists the distinct words in a file in alphabetical order. Make two versions: one in which a word is simply a whitespace-separated sequence of characters and one in which a word is a sequence of letters separated by any sequence of non-letters.
4. (∗2.5) Implement a simple solitaire card game.
5. (∗1.5) Implement a simple test of whether a word is a palindrome (that is, if its representation is symmetric; examples are ***ada***, ***otto***, and ***tut***). Implement a simple test of whether an integer is a palindrome. Implement a simple test of a whether sentence is a palindrome. Generalize.
6. (∗1.5) Define a queue using (only) two ***stack***s.
7. (∗1.5) Define a stack similar to ***stack*** (§17.3.1), except that it doesn't copy its underlying container and that it allows iteration over its elements.
8. (∗3) Your computer will have support for concurrent activities through the concept of a thread, task, or process. Figure out how that is done. The concurrency mechanism will have a concept of locking to prevent two tasks accessing the same memory simultaneously. Use the machine's locking mechanism to implement a lock class.
9. (∗2.5) Read a sequence of dates such as ***Dec85***, ***Dec50***, ***Jan76***, etc., from input and then output them so that later dates come first. The format of a date is a three-letter month followed by a two-digit year. Assume that all the years are from the same century.
10. (∗2.5) Generalize the input format for dates to allow dates such as ***Dec1985***, ***12/3/1990***, ***(Dec,30,1950)***, ***3/6/2001***, etc. Modify exercise §17.8[9] to cope with the new formats.
11. (∗1.5) Use a ***bitset*** to print the binary values of some numbers, including ***0***, ***1***, ***−1***, ***18***, ***−18***, and the largest positive ***int***.
12. (∗1.5) Use ***bitset*** to represent which students in a class were present on a given day. Read the ***bitset***s for a series of 12 days and determine who was present every day. Determine which students were present at least 8 days.
13. (∗1.5) Write a ***List*** of pointers that ***delete***s the objects pointed to when it itself is destroyed or if the element is removed from the ***List***.
14. (∗1.5) Given a ***stack*** object, print its elements in order (without changing the value of the stack).
15. (∗2.5) Complete ***hash_map*** (§17.6.1). This involves implementing ***find()*** and ***equal_range()*** and devising a way of testing the completed template. Test ***hash_map*** with at least one key type for which the default hash function would be unsuitable.
16. (∗2.5) Implement and test a list in the style of the standard ***list***.
17. (∗2) Sometimes, the space overhead of a ***list*** can be a problem. Write and test a singly-linked list in the style of a standard container.
18. (∗2.5) Implement a list that is like a standard ***list***, except that it supports subscripting. Compare the cost of subscripting for a variety of lists to the cost of subscripting a ***vector*** of the same length.
19. (∗2) Implement a template function that merges two containers.
20. (∗1.5) Given a C-style string, determine whether it is a palindrome. Determine whether an initial sequence of at least three words in the string is a palindrome.
21. (∗2) Read a sequence of ***(name,value)*** pairs and produce a sorted list of ***(name,total,mean,median)*** 4-tuples. Print that list.
22. (∗2.5) Determine the space overhead of each of the standard containers on your implementation.
23. (∗3.5) Consider what would be a reasonable implementation strategy for a ***hash_map*** that needed to use minimal space. Consider what would be a reasonable implementation strategy for

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

a ***hash_map*** that needed to use minimal lookup time. In each case, consider what operations you might omit so as to get closer to the ideal (no space overhead and no lookup overhead, respectively). Hint: There is an enormous literature on hash tables.

24. (∗2) Devise a strategy for dealing with overflow in ***hash_map*** (different values hashing to the same hash value) that makes ***equal_range()*** trivial to implement.
25. (∗2.5) Estimate the space overhead of a ***hash_map*** and then measure it. Compare the estimate to the measurements. Compare the space overhead of your ***hash_map*** and your implementation's ***map***.
26. (∗2.5) Profile your ***hash_map*** to see where the time is spent. Do the same for your implementation's ***map*** and a widely-distributed ***hash_map***.
27. (∗2.5) Implement a ***hash_map*** based on a ***vector<map<K,V>*>*** so that each ***map*** holds all keys that have the same hash value.
28. (∗3) Implement a ***hash_map*** using Splay trees (see D. Sleator and R. E. Tarjan: *Self-Adjusting Binary Search Trees*, JACM, Vol. 32. 1985).
29. (∗2) Given a data structure describing a string-like entity:

```
struct St {
      int size;
      char type_indicator;
      char* buf;              // point to size characters
      st(const char* p);      // allocate and fill buf
};
```

   Create 1000 ***St***s and use them as keys for a ***hash_map***. Devise a program to measure the performance of the ***hash_map***. Write a hash function (a ***Hash***; §17.6.2.3) specifically for ***St*** keys.

30. (∗2) Give at least four different ways of removing the ***erased*** elements from a ***hash_map***. You should use a standard library algorithm (§3.8, Chapter 18) to avoid an explicit loop.
31. (∗3) Implement a ***hash_map*** that erases elements immediately.
32. (∗2) The hash function presented in §17.6.2.3 doesn't always consider all of the representation of a key. When will part of a representation be ignored? Write a hash function that always considers all of the representations of a key. Give an example of when it might be wise to ignore part of a key and write a hash function that computes its value based only on the part of a key considered relevant.
33. (∗2.5) The code of hash functions tends to be similar: a loop gets more data and then hashes it. Define a ***Hash*** (§17.6.2.3) that gets its data by repeatedly calling a function that a user can define on a per-type basis. For example:

```
size_t res = 0;
while (size_t v = hash(key)) res = (res<<3)^v;
```

   Here, a user can define ***hash(K)*** for each type ***K*** that needs to be hashed.

34. (∗3) Given some implementation of ***hash_map***, implement ***hash_multimap***, ***hash_set***, and ***hash_multiset***.
35. (∗2.5) Write a hash function intended to map uniformly distributed ***int*** values into hash values intended for a table size of about 1024. Given that function, devise a set of 1024 key values, all of which map to the same value.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.