# 12

# Containers

*It was new. It was singular. It was simple.*
*It must succeed!*
*– H. Nelson*

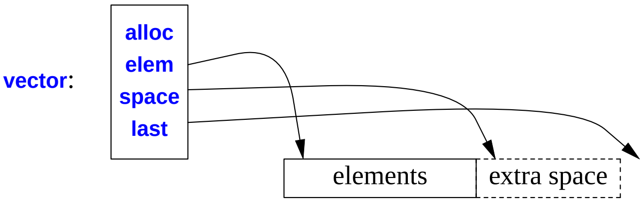
## 12.1 Introduction

Most computing involves creating collections of values and then manipulating such collections. Reading characters into a **string** and printing out the **string** is a simple example. A class with the main purpose of holding objects is commonly called a *container*. Providing suitable containers for a given task and supporting them with useful fundamental operations are important steps in the construction of any program.

To illustrate the standard-library containers, consider a simple program for keeping names and telephone numbers. This is the kind of program for which different approaches appear ‘‘simple and obvious’’ to people of different backgrounds. The **Entry** class from §11.5 can be used to hold a simple phone book entry. Here, we deliberately ignore many real-world complexities, such as the fact that many phone numbers do not have a simple representation as a 32-bit **int**.

## 12.2 vector

The most useful standard-library container is vector. A vector is a sequence of elements of a given type. The elements are stored contiguously in memory. A typical implementation of vector (§5.2.2, §6.2) will consist of a handle holding pointers to the first element, one-past-the-last element, and one-past-the-last allocated space (§13.1) (or the equivalent information represented as a pointer plus offsets):



In addition, it holds an allocator (here, alloc), from which the vector can acquire memory for its elements. The default allocator uses new and delete to acquire and release memory (§12.7). Using a slightly advanced implementation technique, we can avoid storing any data for simple allocators in a vector object.

We can initialize a vector with a set of values of its element type:

```
vector<Entry> phone_book = {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

Elements can be accessed through subscripting. So, assuming that we have defined << for Entry, we can write:

```
void print_book(const vector<Entry>& book)
{
        for (int i = 0; i!=book.size(); ++i)
                cout << book[i] << '\n';
}
```

As usual, indexing starts at 0 so that book[0] holds the entry for David Hume. The vector member function size() gives the number of elements.

The elements of a vector constitute a range, so we can use a range-for loop (§1.7):

```
void print_book(const vector<Entry>& book)
{
        for (const auto& x : book)        // for "auto" see §1.4
                cout << x << '\n';
}
```

When we define a vector, we give it an initial size (initial number of elements):

```
vector<int> v1 = {1, 2, 3, 4};          // size is 4
vector<string> v2;                       // size is 0
vector<Shape*> v3(23);                   // size is 23; initial element value: nullptr
vector<double> v4(32,9.9);               // size is 32; initial element value: 9.9
```

An explicit size is enclosed in ordinary parentheses, for example, (23), and by default, the elements are initialized to the element type's default value (e.g., nullptr for pointers and 0 for numbers). If you don't want the default value, you can specify one as a second argument (e.g., 9.9 for the 32 elements of v4).

The initial size can be changed. One of the most useful operations on a vector is push_back(), which adds a new element at the end of a vector, increasing its size by one. For example, assuming that we have defined >> for Entry, we can write:

```
void input()
{
        for (Entry e; cin>>e; )
                phone_book.push_back(e);
}
```

This reads Entrys from the standard input into phone_book until either the end-of-input (e.g., the end of a file) is reached or the input operation encounters a format error.

The standard-library vector is implemented so that growing a vector by repeated push_back()s is efficient. To show how, consider an elaboration of the simple Vector from Chapter 5 and Chapter 7 using the representation indicated in the diagram above:

```
template<typename T>
class Vector {
        allocator<T> alloc;  // standard-library allocator of space for Ts
        T* elem;             // pointer to first element
        T* space;            // pointer to first unused (and uninitialized) slot
        T* last;             // pointer to last slot
public:
        // ...
        int size() const { return space−elem; }           // number of elements
        int capacity() const { return last−elem; }        // number of slots available for elements
        // ...
        void reserve(int newsz);              // increase capacity() to newsz
        // ...
        void push_back(const T& t);           // copy t into Vector
        void push_back(T&& t);                // move t into Vector
};
```

The standard-library vector has members capacity(), reserve(), and push_back(). The reserve() is used by users of vector and other vector members to make room for more elements. It may have to allocate new memory and when it does, it moves the elements to the new allocation. When reserve() moves elements to a new and larger allocation, any pointers to those elements will now point to the wrong location; they have become *invalidated* and should not be used.

Given capacity() and reserve(), implementing push_back() is trivial:

```
template<typename T>
void Vector<T>::push_back(const T& t)
{
        if (capacity()<=size())                    // make sure we have space for t
                reserve(size()==0?8:2*size());   // double the capacity
        construct_at(space,t);                     // initialize *space to t ("place t at space")
        ++space;
}
```

Now allocation and relocation of elements happens only infrequently. I used to use reserve() to try to improve performance, but that turned out to be a waste of effort: the heuristic used by vector is on average better than my guesses, so now I only explicitly use reserve() to avoid reallocation of elements when I want to use pointers to elements.

A vector can be copied in assignments and initializations. For example:

```
vector<Entry> book2 = phone_book;
```

Copying and moving vectors are implemented by constructors and assignment operators as described in §6.2. Assigning a vector involves copying its elements. Thus, after the initialization of book2, book2 and phone_book hold separate copies of every Entry in the phone book. When a vector holds many elements, such innocent-looking assignments and initializations can be expensive. Where copying is undesirable, references or pointers (§1.7) or move operations (§6.2.2) should be used.

The standard-library vector is very flexible and efficient. Use it as your default container; that is, use it unless you have a solid reason to use some other container. If you avoid vector because of vague concerns about "efficiency," measure. Our intuition is most fallible in matters of the performance of container uses.

### 12.2.1 Elements

Like all standard-library containers, vector is a container of elements of some type T, that is, a vector<T>. Just about any type qualifies as an element type: built-in numeric types (such as char, int, and double), user-defined types (such as string, Entry, list<int>, and Matrix<double,2>), and pointers (such as const char*, Shape*, and double*). When you insert a new element, its value is copied into the container. For example, when you put an integer with the value 7 into a container, the resulting element really has the value 7. The element is not a reference or a pointer to some object containing 7. This makes for nice, compact containers with fast access. For people who care about memory sizes and run-time performance this is critical.

If you have a class hierarchy (§5.5) that relies on virtual functions to get polymorphic behavior, do not store objects directly in a container. Instead store a pointer (or a smart pointer; §15.2.1). For example:

```
vector<Shape> vs;                      // No, don't - there is no room for a Circle or a Smiley (§5.5)
vector<Shape*> vps;                    // better, but see §5.5.3 (don't leak)
vector<unique_ptr<Shape>> vups;        // OK
```

### 12.2.2 Range Checking

The standard-library vector does not guarantee range checking. For example:

```
void silly(vector<Entry>& book)
{
        int i = book[book.size()].number;           // book.size() is out of range
        // ...
}
```

That initialization is likely to place some random value in i rather than giving an error. This is undesirable, and out-of-range errors are a common problem. Consequently, I often use a simple range-checking adaptation of vector:

```
template<typename T>
struct Vec : std::vector<T> {
        using vector<T>::vector;                    // use the constructors from vector (under the name Vec)

        T& operator[](int i) { return vector<T>::at(i); }                   // range check
        const T& operator[](int i) const { return vector<T>::at(i); }      // range check const objects; §5.2.1

        auto begin() { return Checked_iter<vector<T>>{*this}; }          // see §13.1
        auto end() { return Checked_iter<vector<T>>{*this, vector<T>::end()}; }
};
```

Vec inherits everything from vector except for the subscript operations that it redefines to do range checking. The at() operation is a vector subscript operation that throws an exception of type out_of_range if its argument is out of the vector's range (§4.2).

For Vec, an out-of-range access will throw an exception that the user can catch. For example:

```
void checked(Vec<Entry>& book)
{
        try {
                book[book.size()] = {"Joe",999999};       // will throw an exception
                // ...
        }
        catch (out_of_range&) {
                cerr << "range error\n";
        }
}
```

The exception will be thrown, and then caught (§4.2). If the user doesn't catch an exception, the program will terminate in a well-defined manner rather than proceeding or failing in an undefined manner. One way to minimize surprises from uncaught exceptions is to use a main() with a try-block as its body. For example:

```
int main()
try {
        // your code
}
```

```
catch (out_of_range&) {
        cerr << "range error\n";
}
catch (...) {
        cerr << "unknown exception thrown\n";
}
```

This provides default exception handlers so that if we fail to catch some exception, an error message is printed on the standard error-diagnostic output stream cerr (§11.2).

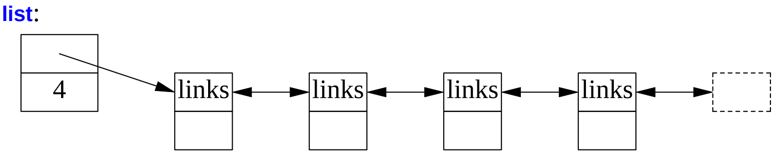
Why doesn't the standard guarantee range checking? Many performance-critical applications use vectors and checking all subscripting implies a cost on the order of 10%. Obviously, that cost can vary dramatically depending on hardware, optimizers, and an application's use of subscripting. However, experience shows that such overhead can lead people to prefer the far more unsafe built-in arrays. Even the mere fear of such overhead can lead to disuse. At least vector is easily range checked at debug time and we can build checked versions on top of the unchecked default.

A range-for avoids range errors at no cost by implicitly accessing all elements in the range. As long as their arguments are valid, the standard-library algorithms do the same to ensure the absence of range errors.

If you use vector::at() directly in your code, you don't need my Vec workaround. Furthermore, some standard libraries have range-checked vector implementations that offer more complete checking than Vec.

## 12.3 list

The standard library offers a doubly-linked list called list:



We use a list for sequences where we want to insert and delete elements without moving other elements. Insertion and deletion of phone book entries could be common, so a list could be appropriate for representing a simple phone book. For example:

```
list<Entry> phone_book = {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

When we use a linked list, we tend not to access elements using subscripting the way we commonly do for vectors. Instead, we might search the list looking for an element with a given value. To do this, we take advantage of the fact that a list is a sequence as described in Chapter 13:

```
int get_number(const string& s)
{
        for (const auto& x : phone_book)
                if (x.name==s)
                        return x.number;
        return 0;   // use 0 to represent "number not found"
}
```

The search for s starts at the beginning of the list and proceeds until s is found or the end of phone_book is reached.

Sometimes, we need to identify an element in a list. For example, we may want to delete an element or insert a new element before it. To do that we use an *iterator*: a list iterator identifies an element of a list and can be used to iterate through a list (hence its name). Every standard-library container provides the functions begin() and end(), which return an iterator to the first and to one-past-the-last element, respectively (§13.1). Using iterators explicitly, we can – less elegantly – write the get_number() function like this:

```
int get_number(const string& s)
{
        for (auto p = phone_book.begin(); p!=phone_book.end(); ++p)
                if (p−>name==s)
                        return p−>number;
        return 0;   // use 0 to represent "number not found"
}
```

In fact, this is roughly the way the terser and less error-prone range-for loop is implemented by the compiler. Given an iterator p, *p is the element to which it refers, ++p advances p to refer to the next element, and when p refers to a class with a member m, then p−>m is equivalent to (*p).m.

Adding elements to a list and removing elements from a list is easy:

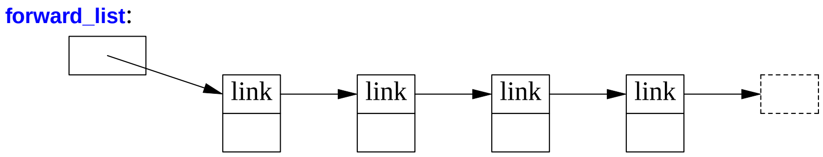
```
void f(const Entry& ee, list<Entry>::iterator p, list<Entry>::iterator q)
{
        phone_book.insert(p,ee);        // add ee before the element referred to by p
        phone_book.erase(q);            // remove the element referred to by q
}
```

For a list, insert(p,elem) inserts an element with a copy of the value elem before the element pointed to by p. Here, p may be an iterator pointing one-beyond-the-end of the list. Conversely, erase(p) removes the element pointed to by p and destroys it.

These list examples could be written identically using vector and (surprisingly, unless you understand machine architecture) often perform better with a vector than with a list. When all we want is a sequence of elements, we have a choice between using a vector and a list. Unless you have a reason not to, use a vector. A vector performs better for traversal (e.g., find() and count()) and for sorting and searching (e.g., sort() and equal_range(); §13.5, §15.3.3).

## 12.4 forward_list

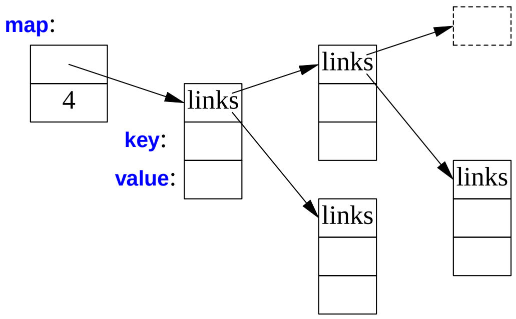
The standard library also offers a singly-linked list called forward_list:



A forward_list differs from a (doubly-linked) list by only allowing forward iteration. The point of that is to save space. There is no need to keep a predecessor pointer in each link and the size of an empty forward_list is just one pointer. A forward_list doesn't even keep its number of elements. If you need the number of elements, count. If you can't afford to count, you probably shouldn't use a forward_list.

## 12.5 map

Writing code to look up a name in a list of *(name,number)* pairs is quite tedious. In addition, a linear search is inefficient for all but the shortest lists. The standard library offers a balanced binary search tree (usually a red-black tree) called map:



In other contexts, a map is known as an associative array or a dictionary.

The standard-library map is a container of pairs of values optimized for lookup and insertion. We can use the same initializer as for vector and list (§12.2, §12.3):

```
map<string,int> phone_book {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

When indexed by a value of its first type (called the *key*), a map returns the corresponding value of the second type (called the *value* or the *mapped type*). For example:

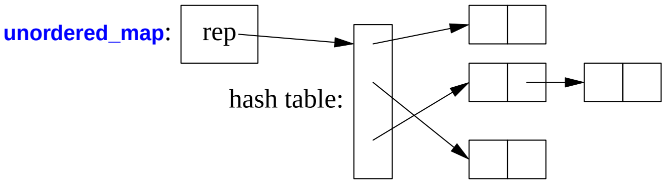
```
int get_number(const string& s)
{
        return phone_book[s];
}
```

In other words, subscripting a map is essentially the lookup we called get_number(). If a key isn't found, it is entered into the map with a default value for its value. The default value for an integer type is 0 and that just happens to be a reasonable value to represent an invalid telephone number.

If we wanted to avoid entering invalid numbers into our phone book, we could use find() and insert() (§12.8) instead of [ ].

## 12.6 unordered_map

The cost of a map lookup is O(log(n)) where n is the number of elements in the map. That's pretty good. For example, for a map with 1,000,000 elements, we perform only about 20 comparisons and indirections to find an element. However, in many cases, we can do better by using a hashed lookup rather than a comparison using an ordering function, such as <. The standard-library hashed containers are referred to as "unordered" because they don't require an ordering function:



For example, we can use an unordered_map from <unordered_map> for our phone book:

```
unordered_map<string,int> phone_book {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

Like for a map, we can subscript an unordered_map:

```
int get_number(const string& s)
{
        return phone_book[s];
}
```

The standard library provides a default hash function for strings as well as for other built-in and standard-library types. If necessary, we can provide our own. Possibly, the most common need for a custom hash function comes when we want an unordered container of one of our own types. A hash function is often implemented as a function object (§7.3.2). For example:

```
struct Record {
        string name;
        int product_code;
        // ...
};

struct Rhash {          // a hash function for Record
        size_t operator()(const Record& r) const
        {
                return hash<string>()(r.name) ^ hash<int>()(r.product_code);
        }
};

unordered_set<Record,Rhash> my_set; // set of Records using Rhash for lookup
```

Designing good hash functions is an art and often requires knowledge of the data to which it will be applied. Creating a new hash function by combining existing hash functions using exclusive-or (^) is simple and often very effective. However, be careful to ensure that every value that takes part in the hash really helps to distinguish the values. For example, unless you can have several names for the same product code (or several product codes for the same name), combining the two hashes provides no benefits.

We can avoid explicitly passing the hash operation by defining it as a specialization of the standard-library hash:

```
namespace std {              // make a hash function for Record

        template<> struct hash<Record> {
                using argument_type = Record;
                using result_type = size_t;

                result_type operator()(const Record& r) const
                {
                        return hash<string>()(r.name) ^ hash<int>()(r.product_code);
                }
        };
}
```

Note the differences between a map and an unordered_map:

- A map requires an ordering function (the default is <) and yields an ordered sequence.
- A unordered_map requires an equality function (the default is ==); it does not maintain an order among its elements.

Given a good hash function, an unordered_map is much faster than a map for large containers. However, the worst-case behavior of an unordered_map with a poor hash function is far worse than that of a map.

## 12.7 Allocators

By default, standard-library containers allocate space using **new**. Operators **new** and **delete** provide a general free store (also called dynamic memory or heap) that can hold objects of arbitrary size and user-controlled lifetime. This implies time and space overheads that can be eliminated in many special cases. Therefore, the standard-library containers offer the opportunity to install allocators with specific semantics where needed. This has been used to address a wide variety of concerns related to performance (e.g., pool allocators), security (allocators that clean-up memory as part of deletion), per-thread allocation, and non-uniform memory architectures (allocating in specific memories with pointer types to match). This is not the place to discuss these important, but very specialized and often advanced techniques. However, I will give one example motivated by a real-world problem for which a pool allocator was the solution.

An important, long-running system used an event queue (see §18.4) using **vector**s as events that were passed as **shared_ptr**s. That way, the last user of an event implicitly deleted it:

```
struct Event {
      vector<int> data = vector<int>(512);
};

list<shared_ptr<Event>> q;

void producer()
{
      for (int n = 0; n!=LOTS; ++n) {
            lock_guard lk {m};              // m is a mutex; see §18.3
            q.push_back(make_shared<Event>());
            cv.notify_one();                // cv is a condition_variable; see §18.4
      }
}
```

From a logical point of view this worked nicely. It is logically simple, so the code is robust and maintainable. Unfortunately, this led to massive fragmentation. After 100,000 events had been passed among 16 producers and 4 consumers, more than 6GB of memory had been consumed.

The traditional solution to fragmentation problems is to rewrite the code to use a pool allocator. A pool allocator is an allocator that manages objects of a single fixed size and allocates space for many objects at a time, rather than using individual allocations. Fortunately, C++ offers direct support for that. The pool allocator is defined in the **pmr** (“polymorphic memory resource”) sub-namespace of **std**:

```
pmr::synchronized_pool_resource pool;           // make a pool

struct Event {
      vector<int> data = vector<int>{512,&pool};  // let Events use the pool
};

list<shared_ptr<Event>> q {&pool};              // let q use the pool
```

```
void producer()
{
        for (int n = 0; n!=LOTS; ++n) {
                scoped_lock lk {m};         // m is a mutex (§18.3)
                q.push_back(allocate_shared<Event,pmr::polymorphic_allocator<Event>>{&pool});
                cv.notify_one();
        }
}
```

Now, after 100,000 events had been passed among 16 producers and 4 consumers, less than 3MB of memory had been consumed. That's about a 2000-fold improvement! Naturally, the amount of memory actually in use (as opposed to memory wasted to fragmentation) is unchanged. After eliminating fragmentation, memory use was stable over time so the system could run for months.

Techniques like this have been applied with good effects from the earliest days of C++, but generally they required code to be rewritten to use specialized containers. Now, the standard containers optionally take allocator arguments. The default is for the containers to use new and delete. Other polymorphic memory resources include

- unsynchronized_polymorphic_resource; like polymorphic_resource but can only be used by one thread.
- monotonic_polymorphic_resource; a fast allocator that releases its memory only upon its destruction and can only be used by one thread.

A polymorphic resource must be derived from memory_resource and define members allocate(), deallocate(), and is_equal(). The idea is for users to build their own resources to tune code.

## 12.8 Container Overview

The standard library provides some of the most general and useful container types to allow the programmer to select a container that best serves the needs of an application:

| Standard Container Summary | |
|---|---|
| vector<T> | A variable-size vector (§12.2) |
| list<T> | A doubly-linked list (§12.3) |
| forward_list<T> | A singly-linked list |
| deque<T> | A double-ended queue |
| map<K,V> | An associative array (§12.5) |
| multimap<K,V> | A map in which a key can occur many times |
| unordered_map<K,V> | A map using a hashed lookup (§12.6) |
| unordered_multimap<K,V> | A multimap using a hashed lookup |
| set<T> | A set (a map with just a key and no value) |
| multiset<T> | A set in which a value can occur many times |
| unordered_set<T> | A set using a hashed lookup |
| unordered_multiset<T> | A multiset using a hashed lookup |

The unordered containers are optimized for lookup with a key (often a string); in other words, they are hash tables.

The containers are defined in namespace std and presented in headers <vector>, <list>, <map>, etc. (§9.3.4). In addition, the standard library provides container adaptors queue<T>, stack<T>, and priority_queue<T>. Look them up if you need them. The standard library also provides more specialized container-like types, such as array<T,N> (§15.3.1) and bitset<N> (§15.3.2).

The standard containers and their basic operations are designed to be similar from a notational point of view. Furthermore, the meanings of the operations are equivalent for the various containers. Basic operations apply to every kind of container for which they make sense and can be efficiently implemented:

**Standard Container Operations (partial)**

| | |
|---|---|
| value_type | The type of an element |
| p=c.begin() | p points to first element of c; also cbegin() for an iterator to const |
| p=c.end() | p points to one-past-the-last element of c; also cend() for an iterator to const |
| k=c.size() | k is the number of elements in c |
| c.empty() | Is c empty? |
| k=c.capacity() | k is the number of elements that c can hold without a new allocation |
| c.reserve(k) | Increase the capacity to k; if k<=c.capacity(), c.reserve(k) does nothing |
| c.resize(k) | Make the number of elements k; added elements have the default value value_type{} |
| c[k] | The kth element of c; zero-based; no range guaranteed checking |
| c.at(k) | The kth element of c; if out of range, throw out_of_range |
| c.push_back(x) | Add x at the end of c; increase the size of c by one |
| c.emplace_back(a) | Add value_type{a} at the end of c; increase the size of c by one |
| q=c.insert(p,x) | Add x before p in c |
| q=c.erase(p) | Remove element at p from c |
| c=c2 | Assignment: copy all elements from c2 to get c==c2 |
| b=(c==c2) | Equality of all elements of c and c2; b==true if equal |
| x=(c<=>c2) | Lexicographical order of c and c2: x<0 if c is less than c2, x==0 if equal, and 0<x if greater than. !=, <, <=, >, and >= are generated from <=> |

This notational and semantic uniformity enables programmers to provide new container types that can be used in a very similar manner to the standard ones. The range-checked vector, Vector (§4.3, Chapter 5), is an example of that. The uniformity of container interfaces allows us to specify algorithms independently of individual container types. However, each has strengths and weaknesses. For example, subscripting and traversing a vector is cheap and easy. On the other hand, vector elements are moved to different locations when we insert or remove elements; list has exactly the opposite properties. Please note that a vector is usually more efficient than a list for short sequences of small elements (even for insert() and erase()). I recommend the standard-library vector as the default type for sequences of elements: you need a reason to choose another.

Consider the singly-linked list, forward_list, a container optimized for the empty sequence (§12.3). An empty forward_list occupies just one word, whereas an empty vector occupies three. Empty sequences, and sequences with only an element or two, are surprisingly common and useful.

An emplace operation, such as **emplace_back()** takes arguments for an element's constructor and builds the object in a newly allocated space in the container, rather than copying an object into the container. For example, for a **vector<pair<int,string>>** we could write:

```
v.push_back(pair{1,"copy or move"});     // make a pair and move it into v
v.emplace_back(1,"build in place");      // build a pair in v
```

For simple examples like this, optimizations can result in equivalent performance for both calls.

## 12.9 Advice

[1] An STL container defines a sequence; §12.2.
[2] STL containers are resource handles; §12.2, §12.3, §12.5, §12.6.
[3] Use **vector** as your default container; §12.2, §12.8; [CG: SL.con.2].
[4] For simple traversals of a container, use a range-**for** loop or a begin/end pair of iterators; §12.2, §12.3.
[5] Use **reserve()** to avoid invalidating pointers and iterators to elements; §12.2.
[6] Don't assume performance benefits from **reserve()** without measurement; §12.2.
[7] Use **push_back()** or **resize()** on a container rather than **realloc()** on an array; §12.2.
[8] Don't use iterators into a resized **vector**; §12.2 [CG: ES.65].
[9] Do not assume that **[ ]** range checks; §12.2.
[10] Use **at()** when you need guaranteed range checks; §12.2; [CG: SL.con.3].
[11] Use range-**for** and standard-library algorithms for cost-free avoidance of range errors; §12.2.2.
[12] Elements are copied into a container; §12.2.1.
[13] To preserve polymorphic behavior of elements, store pointers (built-in or user-defined); §12.2.1.
[14] Insertion operations, such as **insert()** and **push_back()**, are often surprisingly efficient on a **vector**; §12.3.
[15] Use **forward_list** for sequences that are usually empty; §12.8.
[16] When it comes to performance, don't trust your intuition: measure; §12.2.
[17] A **map** is usually implemented as a red-black tree; §12.5.
[18] An **unordered_map** is a hash table; §12.6.
[19] Pass a container by reference and return a container by value; §12.2.
[20] For a container, use the **()**-initializer syntax for sizes and the **{}**-initializer syntax for sequences of elements; §5.2.3, §12.2.
[21] Prefer compact and contiguous data structures; §12.3.
[22] A **list** is relatively expensive to traverse; §12.3.
[23] Use unordered containers if you need fast lookup for large amounts of data; §12.6.
[24] Use ordered containers (e.g., **map** and **set**) if you need to iterate over their elements in order; §12.5.
[25] Use unordered containers (e.g., **unordered_map**) for element types with no natural order (i.e., no reasonable **<**); §12.5.
[26] Use associative containers (e.g., **map** and **list**) when you need pointers to elements to be stable as the size of the container changes; §12.8.

[27] Experiment to check that you have an acceptable hash function; §12.6.

[28] A hash function obtained by combining standard hash functions for elements using the exclusive-or operator (^) is often good; §12.6.

[29] Know your standard-library containers and prefer them to handcrafted data structures; §12.8.

[30] If your application is suffering performance problems related to memory, minimize free store use and/or consider using a specialized allocator; §12.7.