

A Tour of C++: Containers and Algorithms

*Why waste time learning
when ignorance is instantaneous?
– Hobbes*

- Libraries
 - Standard Library Overview; The Standard-library Headers and Namespace
- Strings
- Stream I/O
 - Output; Input; [string](#) I/O; I/O of User-defined Types
- Containers
 - [vector](#); [list](#); [map](#); [unordered_map](#); Container Overview
- Algorithms
 - Use of Iterators; Iterator Types; Stream Iterators; Predicates; Algorithm Overview; Container Algorithms
- Advice

4.1 Libraries [tour3.lib]

No significant program is written in just a bare programming language. First, a set of supporting libraries is developed. These then form the basis for further work. Most programs are tedious to write in the bare language, whereas just about any task can be rendered simple by the use of good libraries.

Continuing from Chapter 2 and Chapter 3, this chapter and the next give a quick tour of key standard-library facilities. The assumption is that you have programmed before. If not, please consider reading a textbook, such as *Programming: Principles and Practice using C++* [Stroustrup, 2009], before continuing here. Even if you have programmed

before, the libraries you used or the applications you wrote may be very different from the style of C++ presented here. If you find this “lightning tour” confusing, another approach could be to skip to the more systematic and bottom up language presentation starting in Chapter 6. Similarly, a more systematic description of the standard library starts in Chapter 30.

I very briefly present useful standard-library types, such as `string`, `ostream`, `vector`, `map` (this chapter), `unique_ptr`, `thread`, `regex`, and `complex` (Chapter 5), as well as the most common ways of using them. Doing this allows me to give better examples in the following chapters. As in Chapter 2 and Chapter 3, you are strongly encouraged not to be distracted or discouraged by an incomplete understanding of details. The purpose of this chapter is to give you a taste of what is to come and to convey a basic understanding of the most useful library facilities.

The standard library facilities described in this book are part of every complete C++ implementation. In addition to the standard C++ library, most implementations offer “graphical user interface” systems (GUIs), Web interfaces, database interfaces, etc. Similarly, most application development environments provide “foundation libraries” for corporate or industrial “standard” development and/or execution environments. Here, I do not describe such systems and libraries. The intent is to provide a self-contained description of C++ as defined by the standard and to keep the examples portable, except where specifically noted. Naturally, a programmer is encouraged to explore the more extensive facilities available on most systems.

4.1.1 Standard-library Overview [tour3.post]

The facilities provided by the standard library can be classified like this:

- [1] Basic run-time language support (e.g., for allocation and run-time type information); see §30.3.
- [2] The C standard library (with very minor modifications to minimize violations of the type system); see Chapter 41.
- [3] Strings and I/O streams (with support for international character sets and localization); see Chapter 35, Chapter 37, and Chapter 38. I/O streams is an extensible framework to which users can add their own streams, buffering strategies, and character sets.
- [4] A framework of containers (such as `vector`, `list`, and `map`) and algorithms (such as `find()`, `sort()`, and `merge()`); see §4.4, §4.5, Chapter 31, Chapter 32, and Chapter 33. This framework, conventionally called the STL [Stepanov,1994], is extensible so that users can easily add their own containers and algorithms.
- [5] Support for numerical computation (such as standard mathematical functions, complex numbers, vectors with arithmetic operations, and random number generators); see §3.2.1.1 and Chapter 39.
- [6] Support for regular expression matching; see §5.5 and Chapter 36.
- [7] Support for concurrent processing, including `threads` and `locks`; see §5.3 and Chapter 40. The concurrency support is foundational so that users can add support for new models of concurrency as libraries.

- [8] Utilities to support template metaprogramming (e.g., type traits; §5.4.2, §28.2.4, §34.7), STL-style generic programming (e.g., `pair`; §5.4.3, §34.2.4.1), and general programming (e.g., `clock`; §5.4.1, §34.6).
- [9] “Smart pointers” for resource management (e.g., `unique_ptr` and `shared_ptr`; §5.2.1, §34.3) and an interface to garbage collectors (§34.8).
- [10] Special-purpose containers, such as `array` (§34.2.1), `bitset` (§34.2.2), and `tuple` (§34.2.4.2).

The main criterion for including a class in the library was that it would somehow be used by almost every C++ programmer (both novices and experts), that it could be provided in a general form that did not add significant overhead compared to a simpler version of the same facility, and that simple uses should be easy to learn (relative to the inherent complexity of the task performed). Essentially, the C++ standard library provides the most common fundamental data structures together with the fundamental algorithms used on them.

4.1.2 The Standard-library Headers and Namespace [tour3.name]

Every standard library facility is provided through some standard header. For example:

```
#include<string>
#include<list>
```

This makes the standard `string` and `list` available.

The standard library is defined in a namespace (§2.4.2, §14.3.1) called `std`. To use standard library facilities, the `std::` prefix can be used:

```
std::string s {"Four legs Good; two legs Baaad!"};
std::list<std::string> slogans {"War is peace", "Freedom is Slavery", "Ignorance is Strength"};
```

For simplicity, I will rarely use the `std::` prefix explicitly in examples. Neither will I always `#include` the necessary headers explicitly. To compile and run the program fragments here, you must `#include` the appropriate headers (as listed in §4.4.5, §4.5.5, and §30.2) and make the names they declare accessible. For example:

```
#include<string>           // make the standard string facilities accessible
using namespace std;       // make std names available without std:: prefix

string s {"C++ is a general-purpose programming language"}; // ok: string is std::string
```

It is generally in poor taste to dump every name from a namespace into the global namespace. However, in this book, I use the standard library almost exclusively and it is good to know what it offers.

Here is a table of selected standard-library headers, all supplying declarations in namespace `std`:

Selected Standard Library Headers			
<algorithm>	copy(), find(), sort()	§32.2	§iso.25
<array>	array	§34.2.1	§iso.23.3.2
<cmath>	sqrt(), pow()	§39.3	§iso.26.8
<complex>	complex, sqrt(), pow()	§39.4	§iso.26.8
<fstream>	fstream, ifstream, ofstream	§37.2.1	§iso.27.9.1
<future>	future, promise	§5.3.5	§iso.30.6
<iostream>	istream, ostream, cin, cout	§37.1	§iso.27.4
<iterator>	back_inserter(), reverse_iterator, begin()	§33.2	§iso.24.3
<limits>	numeric_limits	§39.2	§iso.18.3
<list>	list	§31.4.2	§iso.23.3.5
<map>	map, multimap	§31.4.3	§iso.23.4.4
<memory>	unique_ptr, shared_ptr, allocator	§5.2.1	§iso.20.6
<mutex>	mutex, timed_mutex, recursive_mutex	§40.6.1	§iso.30.4
<regex>	regex, smatch	Chapter 36	§iso.28.8
<set>	set, multiset	§31.4.3	§iso.23.4.6
<sstream>	istringstream, ostringstream	§37.2.2	§iso.27.8
<string>	string, basic_string	Chapter 35	§iso.21.3
<thread>	thread	§5.3.1	§iso.30.3
<unordered_map>	unordered_map, unordered_multimap	§31.4.3.2	§iso.23.5.4
<utility>	move(), swap(), pair	§34.9	§iso.20.1
<valarray>	valarray, slice, gslice	§39.5	§iso.26.6
<vector>	vector	§31.2	§iso.23.3.6

This listing is far from complete, see §30.2 for more information.

4.2 Strings [tour3.string]

The standard library provides a `string` type to complement the string literals. The `string` type provides a variety of useful string operations, such as concatenation. For example:

```
string compose(const string& name, const string& domain)
{
    return name + '@' + domain;
}

auto addr = compose("dmr", "bell-labs.com");
```

Here, `addr` is initialized to the character sequence `dmr@bell-labs.com`. “Addition” of strings means concatenation. You can concatenate a `string`, a string literal, a C-style string, or a character to a `string`. The standard `string` has a move constructor so returning even long `strings` by value is efficient (§3.3.2).

In many applications, the most common form of concatenation is adding something to the end of a `string`. This is directly supported by the `+=` operation. For example:

```
void m2(string& s1, string& s2)
{
    s1 = s1 + '\n';    // append newline
    s2 += '\n';        // append newline
}
```

The two ways of adding to the end of a `string` are semantically equivalent, but I prefer the latter because it is more explicit about what it does, more concise, and possibly more efficiently implemented.

A `string` is mutable. In addition to `=` and `+=`, subscripting (using `[]`) and substring operations are supported. The standard-library `string` is described in Chapter 35. Among other useful features, it provides the ability to manipulate substrings. For example:

```
string name = "Niels Stroustrup";

void m3()
{
    string s = name.substr(6,10);    // s = "Stroustrup"
    name.replace(0,5,"nicholas");    // name becomes "nicholas Stroustrup"
    name[0] = 'N';                  // name becomes "Nicholas Stroustrup"
}
```

The `substr()` operation returns a `string` that is a copy of the substring indicated by its arguments. The first argument is an index into the `string` (a position), and the second argument is the length of the desired substring. Since indexing starts from 0, `s` gets the value `Stroustrup`.

The `replace()` operation replaces a substring with a value. In this case, the substring starting at 0 with length 5 is `Niels`; it is replaced by `Nicholas`. Thus, the final value of `name` is `Nicholas Stroustrup`. Note that the replacement string need not be the same size as the substring that it is replacing.

Naturally, `strings` can be compared against each other and against string literals. For example:

```
string incantation;

void respond(const string& answer)
{
    if (answer == incantation) {
        // perform magic
    }
    else if (answer == "yes") {
        // ...
    }
    // ...
}
```

The `string` library is described in Chapter 35. The most common techniques for implementing `string` are presented in the `String` example (§19.3).

4.3 Stream I/O [tour3.streams]

The standard library provides formatted character input and output through the `iostream` library. The input operations are typed and extensible to handle user-defined types. This section is a very brief introduction to the use of `istream`; Chapter 37 is a reasonably complete description of the `istream` library facilities.

Other forms of user interaction, such as graphical I/O, are handled through libraries that are not part of the ISO standard and therefore not described here.

4.3.1 Output [tour3 ostream]

The I/O stream library defines output for every built-in type. Further, it is easy to define output of a user-defined type (§4.3.4). The operator `<<` (“put to”) is used as an output operator on objects of type `ostream`; `cout` is the standard output stream and `cerr` is the standard stream for reporting errors. By default, values written to `cout` are converted to a sequence of characters. For example, to output the decimal number `10`, we can write:

```
void f()
{
    cout << 10;
}
```

This places the character `1` followed by the character `0` on the standard output stream.

Equivalently, we could write:

```
void g()
{
    int i {10};
    cout << i;
}
```

Output of different types can be combined in the obvious way:

```
void h(int i)
{
    cout << "the value of i is ";
    cout << i;
    cout << '\n';
}
```

For `h(10)`, the output will be

the value of i is 10

People soon tire of repeating the name of the output stream when outputting several related items. Fortunately, the result of an output expression can itself be used for further output. For example:

```
void h2(int i)
{
    cout << "the value of i is " << i << '\n';
}
```

This `h2()` produces the same output as `h()`.

A character constant is a character enclosed in single quotes. Note that a character is output as a character rather than as a numerical value. For example:

```
void k()
{
    int b = 'b';      // note: char implicitly converted to int
    char c = 'c';
    cout << 'a' << b << c;
}
```

The integer value of the character `'b'` is `98` (in the ASCII encoding used on the C++ implementation that I used), so this will output `a98c`.

4.3.2 Input [tour3.istream]

The standard library offers `istream`s for input. Like `ostream`s, `istream`s deal with character string representations of built-in types and can easily be extended to cope with user-defined types.

The operator `>>` (“get from”) is used as an input operator; `cin` is the standard input stream. The type of the right-hand operand of `>>` determines what input is accepted and what is the target of the input operation. For example:

```
void f()
{
    int i;
    cin >> i;    // read an integer into i

    double d;
    cin >> d;    // read a double-precision floating-point number into d
}
```

This reads a number, such as `1234`, from the standard input into the integer variable `i` and a floating-point number, such as `12.34e5`, into the double-precision floating-point variable `d`.

4.3.3 string I/O [tour3.stringio]

Often, we want to read a sequence of characters. A convenient way of doing that is to read into a `string`. For example:

```
int main()
{
    string str;
    cout << "Please enter your name\n";
    cin >> str;
    cout << "Hello, " << str << "\n";
}
```

If you type in `Eric` the response is

`Hello, Eric!`

By default, a whitespace character (§7.3.2), such as a space, terminates the read, so if you enter `Eric Bloodaxe` pretending to be the ill-fated king of York, the response is still

`Hello, Eric!`

You can read a whole line (including the terminating newline character) using the `getline()` function. For example:

```
int main()
{
    cout << "Please enter your name\n";
    string str;
    getline(cin, str);
    cout << "Hello, " << str << "\n";
}
```

With this program, the input `Eric Bloodaxe` yields the desired output:

`Hello, Eric Bloodaxe!`

The newline that terminated the line is discarded, so `cin` is ready for the next input line.

The standard strings have the nice property of expanding to hold what you put in them; you don't have to precalculate a maximum size. So, if you enter a couple of megabytes of semicolons, the program will echo pages of semicolons back at you.

4.3.4 I/O of User-defined Types [tour3.udtio]

In addition to the I/O of built-in types and standard `strings`, the `iostream` library allows programmers to define I/O for their own types. For example, consider a simple type `Entry` that we might use to represent entries in a telephone book:

```
struct Entry {
    string name;
    int number;
};
```

We can define a simple output operator to write an `Entry` using a `{"name", number}` format similar to the one we use for initialization in code:

```
ostream& operator<<(ostream& os, const Entry& e)
{
    return os << "{" << e.name << ", " << e.number << "}";
}
```

A user-defined output operator takes its output stream (by reference) as its first argument and returns it as its result. See §37.4.2 for details.

The corresponding input operator is more complicated because it has to check for correct formatting and deal with errors:

```

istream& operator>>(istream& is, Entry& e)
    // read { "name" , number } pair. Note: formatted with { " " , and }
{
    char c, c2;
    if (is>>c && c=='{' && is>>c2 && c2=='"') { // start with a { "
        string name; // the default value of a string is the empty string: ""
        while (is.get(c) && c!="") // anything before a " is part of the name
            name+=c;

        if (is>>c && c==',') {
            int number = 0;
            if (is>>number>>c && c=='}') { // read the number and a }
                e = {name,number}; // assign to the entry
                return is;
            }
        }
    }
    is.setf(ios_base::failbit); // register the failure in the stream
    return is;
}

```

An input operation returns a reference to its `istream` which can be used to test if the operation succeeded. For example, when used as a condition `cin>>c` means, did we succeed at reading from `cin` into `c`?

The `is>>c` skips whitespace by default, but `is.get(c)` does not so that this `Entry`-input operator ignores (skips) whitespace outside the name string, but not within it. For example:

```

{ "John Marwood Cleese" , 123456 }
{ "Michael Edward Palin",987654}

```

We can read such a pair of values from input into an `Entry` like this:

```

for (Entry ee; cin>>ee; ) // read from cin into ee
    cout << ee << '\n'; // write ee to cout

```

See §37.4.1 for more technical details and techniques for writing input operators for user-defined types. See §5.5 and Chapter 36 for a more systematic technique for recognizing patterns in streams of characters (regular expression matching).

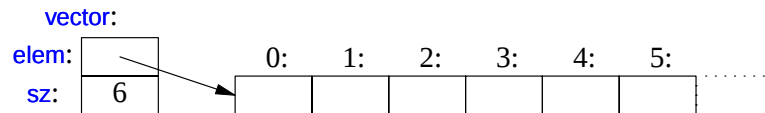
4.4 Containers [tour3.stl]

Much 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 §4.3.4 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`.

4.4.1 `vector` [tour3.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:



The `Vector` examples in §3.2.3 and §3.4 give an idea of the implementation of `vector` and §13.6 and §31.2 provide an exhaustive discussion.

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:

```
void print_book(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` (obviously) constitute a range, so we can use the simpler range-for loop (§2.2.5):

```
void print_book(vector<Entry>& book)
{
    for (const auto& x : book) // for "auto" see §2.2.2
        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, e.g., (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 1. For example:

```
for (Entry e; cin>>e;)
    phone_book.push_back(e);
```

This reads `Entry`s 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.

A `vector` is a single object that can be assigned. For example:

```
void f(vector<Entry>& v)
{
    vector<Entry> v2 = phone_book;
    v = v2;
    // ...
}
```

Assigning a `vector` involves copying its elements. Thus, after the initialization and assignment in `f()`, `v` and `v2` each holds a separate copy of every `Entry` in the phone book. When a `vector` holds many elements, such innocent-looking assignments and initializations can be prohibitively expensive. Where copying is undesirable, references or pointers (§7.2; §7.7) or move operations (§3.3.2; §17.5.2) should be used.

4.4.1.1 Elements [tour3.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.

4.4.1.2 Range Checking [tour3.range]

The standard library `vector` does not guarantee range checking (§31.2.2). For example:

```
vector<Entry> phone_book(1000);

int i = phone_book[2001].number;    // 2001 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>
class Vec : public std::vector<T> {
public:
    using vector<T>::vector;    // use the constructors from vector
                                // (under the name Vec); see §20.3.5.1

    T& operator[](int i) { return vector<T>::at(i); }    // range-checked
    const T& operator[](int i) const { return vector<T>::at(i); }    // range-checked
                                // for const objects; §3.2.1.1
};
```

`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 (§2.4.3.1, §31.2.2).

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

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

The exception will be thrown, and then caught (§2.4.3.1; Chapter 13). 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:

```

int main()
try {
    // your code
}
catch (out_of_range) {
    cerr << "range error\n";
}
catch (...) {
    cerr << "unknown exception thrown\n";
}

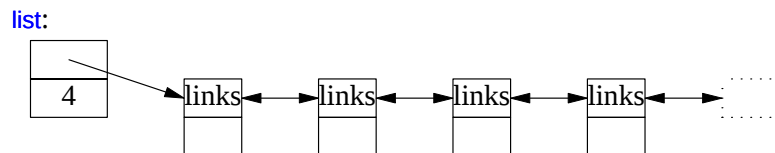
```

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` (§37.1).

Some implementations save you the bother of defining `Vec` (or equivalent) by providing a range-checked version of `vector` (e.g., as a compiler option).

4.4.2 `list` [tour3.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 non-zero value. To do this, we take advantage of the fact that a `list` is a sequence as described in §4.5:

```

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 either `s` is found or the

end is reached.

Sometimes, we need to identify an element in a `list`. For example, we may want to delete it or insert a new entry 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 (§4.5; §33.1.1). 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:

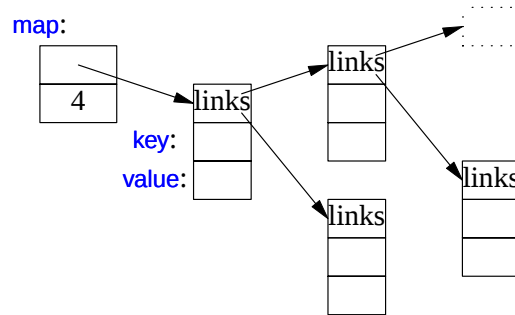
```
void f(const Entry& ee, list<Entry>::iterator p, list<Entry>::iterator q)
{
    phone_book.insert(p,ee); // add ee before the element referred to by p
    phone_book.erase(q);    // remove the element referred to by q
}
```

For a more complete description of `insert()` and `erase()`, see §31.3.7.

Note that these `list` examples could be written identically using `vector` and (surprisingly, unless you understand machine architecture) perform better with a small `vector` than with a small `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 operations (e.g., `sort()` and `binary_search()`).

4.4.3 `map` [tour3.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 search tree called `map`:



In other contexts, a **map** is known as an associative array or a dictionary. It is implemented as a balanced binary tree.

The standard-library **map** (§31.4.3) is a container of pairs of values optimized for lookup. For example:

```
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:

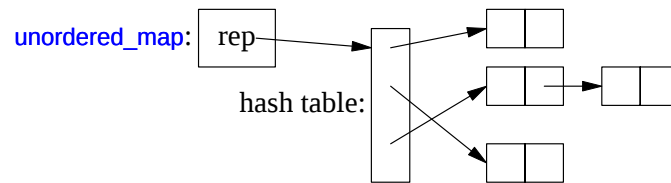
```
int get_number(const string& s)
{
    return phone_book[s];
}
```

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`; the value I just happened to choose represents an invalid telephone number.

If we wanted to avoid entering invalid numbers into our phone book, we could use `find()` and `insert()` instead of `[]` (§31.4.3.1).

4.4.4 **Unordered_map** [tour3.unorderedmap]

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 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>` to implement 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 `unordered_map` provides a default hash function for `strings`. If necessary, you can provide your own (§31.4.3.4).

4.4.5 Container Overview [tour3.stdcontainer]

A `map`, a `list`, and a `vector` can each be used to represent a phone book. 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 when we insert or remove elements; `list` has exactly the opposite properties. A `map` resembles a `list` of (key,value) pairs except that it is optimized for finding values based on keys. 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.

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	
<code>vector<T></code>	A variable-sized vector (§31.2)
<code>list<T></code>	A doubly-linked list (§31.4.2)
<code>forward_list<T></code>	A singly-linked list (§31.4.2)
<code>set<T></code>	A set (§31.4.3)
<code>multiset<T></code>	A set in which a value can occur many times (§31.4.3)
<code>map<K,V></code>	An associative array (§31.4.3)
<code>multimap<K,V></code>	A map in which a key can occur many times (§31.4.3)
<code>unordered_map<K,V></code>	A map using a hashed lookup (§31.4.3.2)
<code>unordered_multimap<K,V></code>	A multimap using a hashed lookup (§31.4.3.2)
<code>unordered_set<T></code>	A set using a hashed lookup (§31.4.3.2)
<code>unordered_multiset<T></code>	A multiset using a hashed lookup (§31.4.3.2)

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

The standard containers are described in §31.2. The containers are defined in namespace `std` and presented in headers `<vector>`, `<list>`, `<map>`, etc. (§4.1.2, §30.2). In addition, the standard library provides container adapters `queue<T>` (§31.5.2), `stack<T>` (§31.5.1), `deque<T>` (§31.2), and `priority_queue<T>` (§31.5.3). The standard library also provides more specialized container-like types, such as a fixed-sized array `array<T,N>` (§34.2.1) and `bitset<N>` (§34.2.2).

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. For example,

- `begin()` and `end()` give iterators to the first and one-beyond-last elements, respectively
- `push_back()` can be used (efficiently) to add elements to the end of a `vector` as well as for a `list`
- `size()` returns the number of elements.

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` (§2.3.2, §2.4.3.1), is an example of that. The uniformity of container interfaces also allows us to specify algorithms independently of individual container types.

4.5 Algorithms [tour3.algorithms]

A data structure, such as a list or a vector, is not very useful on its own. To use one, we need operations for basic access such as adding and removing elements (as is provided for `list` and `vector`). Furthermore, we rarely just store objects in a container. We sort them, print them, extract subsets, remove elements, search for objects, etc. Consequently, the standard library provides the most common algorithms for containers in addition to providing the most common container types. For example, the following sorts a `vector` and places a copy of each unique `vector` element on a `list`:

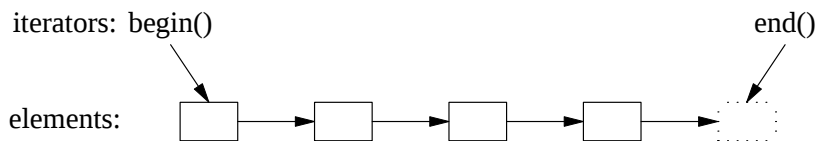
```

bool operator<(const Entry& x, const Entry& y) // less than
{
    return x.name<y.name;           // order Entries by their Names
}

void f(vector<Entry>& vec, list<Entry>& lst)
{
    sort(vec.begin(),vec.end());           // use < for order
    unique_copy(vec.begin(),vec.end(),lst.begin()); // don't copy adjacent equal elements
}

```

The standard algorithms are described in Chapter 32. They are expressed in terms of sequences of elements. A *sequence* is represented by a pair of iterators specifying the first element and the one-beyond-the-last element:



In the example, `sort()` sorts the sequence from `ve.begin()` to `ve.end()` – which just happens to be all the elements of a `vector`. For writing, you need only to specify the first element to be written. If more than one element is written, the elements following that initial element will be overwritten. Thus, to avoid errors, `lst` must have at least as many elements as there are unique values in `vec`.

If we wanted to place the unique elements in a new container, we could have written:

```

list<Entry> f(vector<Entry>& vec)
{
    list<Entry> res;
    sort(vec.begin(),vec.end());
    unique_copy(vec.begin(),vec.end(),back_inserter(res)); // append to res
    return res;
}

```

A `back_inserter()` adds elements at the end of a container, extending the container to make room for them (§33.2.2). Thus, the standard containers plus `back_inserter()`s eliminate the need to use error-prone, explicit C-style memory management using `realloc()` (§31.5.1). The standard-library `list` has a move constructor (§3.3.2, §17.5.2) that makes returning `res` by value efficient (even for `lists` of thousands of elements).

If you find the pair-of-iterators style of code, such as `sort(ve.begin(),ve.end())` tedious, you can define container version of the algorithms and write `sort(ve)` (§4.5.6).

4.5.1 Use of Iterators [tour3.iteruse]

When you first encounter a container, a few iterators referring to useful elements can be obtained; `begin()` and `end()` are the best examples of this. In addition, many algorithms return iterators. For example, the standard algorithm `find` looks for a value in a sequence and returns an iterator to the element found:

```
bool has_c(const string& s, char c)      // does s contain the character c?
{
    auto p = find(s.begin(),s.end(),c);
    if (p!=s.end())
        return true;
    else
        return false;
}
```

Note that `find` returns `end()` to indicate “not found.” An equivalent, shorter, definition of `has_c()` is:

```
bool has_c(const string& s, char c)      // does s contain the character c?
{
    return find(s.begin(),s.end(),c)!=s.end();
}
```

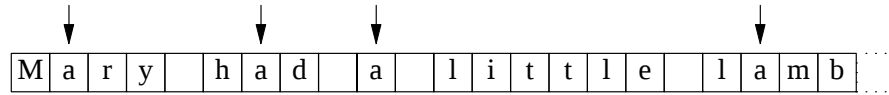
A more interesting exercise would be to find the location of all occurrences of a character in a string. We can return the set of occurrences as a `vector` of `string` iterators. Assuming that we would like to modify the locations found, we pass a non-const string:

```
vector<string::iterator> find_all(string& s, char c)    // find all occurrences of c in s
{
    vector<string::iterator> res;
    for (auto p = s.begin(); p!=s.end(); ++p)
        if (*p==c)
            res.push_back(p);
    return res;
}
```

We iterate through the string using a conventional loop, moving the iterator `p` forward one element at a time using `++` and looking at the elements using the dereference operator `*`. We could test `find_all()` like this:

```
void test()
{
    string m {"Mary had a little lamb"};
    for (auto p : find_all(m,'a'))
        if (*p!='a')
            cerr << "a bug!\n";
}
```

That call of `find_all()` could be graphically represented like this:



The arrows indicate the values of the result `vector`.

Iterators and standard algorithms will work equivalently on every standard container for which their use makes sense. Consequently, we could generalize `find_all()`:

```
template<typename C, typename V>
vector<typename C::iterator> find_all(C& c, V v)    // find all occurrences of v in c
{
    vector<typename C::iterator> res;
    for (auto p = c.begin(); p!=c.end(); ++p)
        if (*p==v)
            res.push_back(p);
    return res;
}
```

The “`typename`” is needed to inform the compiler that `C`’s `iterator` is supposed to be a type and not a value of some type, say, the integer 7. We can hide this implementation detail by introducing a type alias (§3.4.5) for `iterator`:

```
template<typename T>
using Iterator<T> = typename T::iterator;

template<typename C, typename V>
vector<Iterator<C>> find_all(C& c, V v)    // find all occurrences of v in c
{
    vector<Iterator<C>> res;
    for (auto p = c.begin(); p!=c.end(); ++p)
        if (*p==v)
            res.push_back(p);
    return res;
}
```

We can now write:

```
void test()
{
    string m {"Mary had a little lamb"};
    for (auto p : find_all(m,'a'))    // p is a string::iterator
        if (*p!='a')
            cerr << "string bug!\n";

    list<double> ld {1.1, 2.2, 3.3, 1.1};
    for (auto p : find_all(ld,1.1))
        if (*p!=1.1)
            cerr << "list bug!\n";
}
```

```

vector<string> vs { "red", "blue", "green", "green", "orange", "green" };
for (auto p : find_all(vs,"green"))
    if (*p!="green")
        cerr << "vector bug!\n";

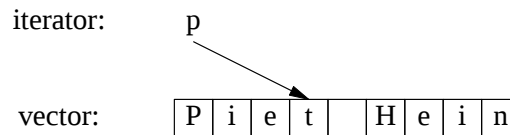
for (auto p : find_all(vs,"green"))
    *p = "vert";
// ...
}

```

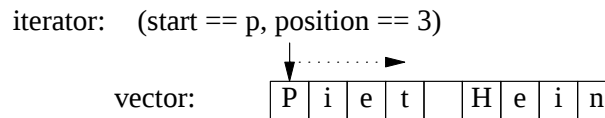
Iterators are used to separate algorithms and containers. An algorithm operates on its data through iterators and knows nothing about the container in which the elements are stored. Conversely, a container knows nothing about the algorithms operating on its elements; all it does is to supply iterators upon request (e.g., `begin()` and `end()`). The result is very general and flexible software.

4.5.2 Iterator Types [tour3.iter]

What are iterators really? Any particular iterator is an object of some type. There are, however, many different iterator types, because an iterator needs to hold the information necessary for doing its job for a particular container type. These iterator types can be as different as the containers and the specialized needs they serve. For example, a `vector`'s iterator could be an ordinary pointer, because a pointer is quite a reasonable way of referring to an element of a `vector`:

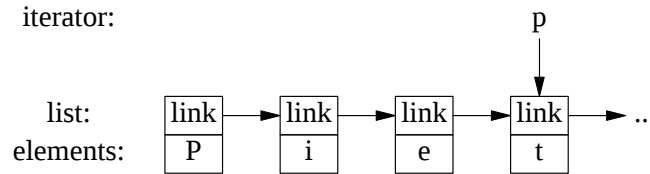


Alternatively, a `vector` iterator could be implemented as a pointer to the `vector` plus an index:



Using such an iterator would allow range checking.

A `list` iterator must be something more complicated than a simple pointer to an element because an element of a `list` in general does not know where the next element of that `list` is. Thus, a `list` iterator might be a pointer to a link:



What is common for all iterators is their semantics and the naming of their operations. For example, applying `++` to any iterator yields an iterator that refers to the next element. Similarly, `*` yields the element to which the iterator refers. In fact, any object that obeys a few simple rules like these is an iterator (§33.1.4). Furthermore, users rarely need to know the type of a specific iterator; each container “knows” its iterator types and makes them available under the conventional names `iterator` and `const_iterator`. For example, `list<Entry>::iterator` is the general iterator type for `list<Entry>`. We rarely have to worry about the details of how that type is defined.

4.5.3 Stream Iterators [tour3.ioiterators]

Iterators are a general and useful concept for dealing with sequences of elements in containers. However, containers are not the only place where we find sequences of elements. For example, an input stream produces a sequence of values and we write a sequence of values to an output stream. Consequently, the notion of iterators can be usefully applied to input and output.

To make an `ostream_iterator`, we need to specify which stream will be used and the type of objects written to it. For example, we can define an iterator that refers to the standard output stream, `cout`:

```
ostream_iterator<string> oo {cout};
```

The effect of assigning to `*oo` is to write the assigned value to `cout`. For example:

```
int main()
{
    *oo = "Hello, ";    // meaning cout<<"Hello, "
    ++oo;
    *oo = "world!\n";  // meaning cout<<"world!\n"
}
```

This is yet another way of writing the canonical message to standard output. The `++oo` is done to mimic writing into an array through a pointer.

Similarly, an `istream_iterator` is something that allows us to treat an input stream as a read-only container. Again, we must specify the stream to be used and the type of values expected:

```
istream_iterator<string> ii {cin};
```

Input iterators are used in pairs representing a sequence, so we must provide an `istream_iterator` to indicate the end of input. This is the default `istream_iterator`:

```
istream_iterator<string> eos {};
```

Typically, `istream_iterator`s and `ostream_iterator`s are not used directly. Instead, they are provided as arguments to algorithms. For example, we can write a simple program to read a file, sort the words read, eliminate duplicates, and write the result to another file:

```
int main()
{
    string from, to;
    cin >> from >> to;                // get source and target file names

    ifstream is {from};                // input stream for file "from"
    istream_iterator<string> ii {is};   // input iterator for stream
    istream_iterator<string> eos {};    // input sentinel

    ofstream os{to};                   // output stream for file "to"
    ostream_iterator<string> oo {os, "\n"}; // output iterator for stream

    vector<string> b {ii, eos};         // b is a vector initialized from input [ii: eos)
    sort(b.begin(), b.end());           // sort the buffer

    unique_copy(b.begin(), b.end(), oo); // copy buffer to output, discard replicated values

    return !is.eof() || !os;           // return error state (§2.2.1, §37.3)
}
```

An `ifstream` is an `istream` that can be attached to a file, and an `ofstream` is an `ostream` that can be attached to a file. The `ostream_iterator`'s second argument is used to delimit output values.

Actually, this program is longer than it needs to be. We read the strings into a `vector`, then we `sort()` them, and then we write them out eliminating duplicates. A more elegant solution is not to store duplicates at all. This can be done by keeping the `strings` in a `set`, which does not keep duplicates and keeps its elements in order (§31.4.3). That way, we could replace the two lines using a `vector` with one using a `set` and replace `unique_copy()` with the simpler `copy()`:

```
set<string> b {ii, eos};               // collect strings from input
copy(b.begin(), b.end(), oo);         // copy buffer to output
```

We used the names `ii`, `eos`, and `oo` only once after their definition, so we could further reduce the size of the program:

```
int main()
{
    string from, to;
    cin >> from >> to;                // get source and target file names

    ifstream is {from};                // input stream for file "from"
    ofstream os {to};                  // output stream for file "to"
```

```

    set<string> b {istream_iterator<string>{is},istream_iterator<string>{}}; // read input
    copy(b.begin(),b.end(),ostream_iterator<string>{os,"n"});           // copy to output

    return !is.eof() || !os;           // return error state (§2.2.1, §37.3)
}

```

It is a matter of taste and experience whether or not this last simplification improves readability. If your tastes lean toward the very terse, you can further eliminate the name `os`.

4.5.4 Predicates [tour3.predicates]

In the examples above, the algorithms have simply “built in” the action to be done for each element of a sequence. However, we often want to make that action a parameter to the algorithm. For example, the `find` algorithm (§32.3) provides a convenient way of looking for a specific value. A more general variant looks for an element that fulfills a specified requirement, a *predicate* (§3.4.2). For example, we might want to search a `map` for the first value larger than `42`. A `map` allows us to access its elements as a sequence of (key,value) pairs, so we can search a `map<string,int>`’s sequence for a `pair<const string,int>` where the `int` is greater than `42`:

```

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

```

Here, `Greater_than` is a function object (§3.4.3) holding the value (`42`) to be compared against:

```

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

```

Alternatively, we could use a lambda expression (§3.4.3):

```

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

```

4.5.5 Algorithm Overview [tour3.algolist]

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

The standard library provides dozens of algorithms. The algorithms are defined in namespace `std` and presented in the `<algorithm>` header. These standard-library algorithms

all take sequences as inputs (§4.5). A half-open sequence from **b** to **e** is referred to as **[b:e)**. Here are a few I have found particularly useful:

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

These algorithms, and many more (see Chapter 32), can be applied to elements of containers, *strings*, and built-in arrays.

4.5.6 Container Algorithms [tour3.container-algo]

A sequence is defined by a pair of iterators **[begin:end)**. This is general and flexible, but most often, we apply an algorithm to a sequence that is the contents of a container. For example:

```
sort(v.begin(),v.end());
```

Why don't we just say `sort(v)`? We can easily provide that shorthand:

```
namespace Estd {
    using namespace std;

    template<class C>
    void sort(C& c)
    {
        sort(c.begin(),c.end());
    }
}
```

```

template<class C, class Pred>
void sort(C& c, Pred p)
{
    sort(c.begin(),c.end(),p);
}

// ...
}

```

I put the container versions of `sort()` (and other algorithms) into their own namespace `Estd` (“extended `std`”) to avoid interfering with other programmers’ uses of and extensions to `std`.

4.6 Advice [tour3.advice]

- [1] Don’t reinvent the wheel; use libraries; §4.1.
- [2] Don’t believe in magic; understand what your libraries do, how they do it, and at what cost they do it.
- [3] When you have a choice, prefer the standard library over other libraries.
- [4] Do not think that the standard library is ideal for everything.
- [5] Remember to `#include` the headers for the facilities you use; §4.1.2.
- [6] Remember that standard library facilities are defined in namespace `std`; §4.1.2.
- [7] Prefer `strings` over C-style strings (a `char*`; §2.2.5) §4.2, §4.3.2.
- [8] `iostreams` are type sensitive, type safe, and extensible; §4.3.
- [9] Prefer `vector<T>`, `map<K,T>`, and `unordered_map<K,T>` over `T[]`; §4.4.
- [10] Know your standard containers and their tradeoffs; §4.4.
- [11] Use `vector` as your default container; §4.4.1.
- [12] Prefer compact data structures; §4.4.1.1.
- [13] If in doubt, use a range-checked vector (such as `Vec`); §4.4.1.2.
- [14] Use `push_back()` or `back_inserter()` to add elements to a container; §4.4.1, §4.5.
- [15] Use `push_back()` on a `vector` rather than `realloc()` on an array; §4.5.
- [16] Catch common exceptions in `main()`; §4.4.1.2.
- [17] Know your standard algorithms and prefer them over handwritten loops; §4.5.5.
- [18] If iterator use get tedious, define container algorithms; §4.5.6.