3

A Tour of the Standard Library

Why waste time learning
when ignorance is instantaneous?
– Hobbes

Standard libraries — output — strings — input — vectors — range checking — lists — maps — container overview — algorithms — iterators — I/O iterators — traversals and predicates — algorithms using member functions — algorithm overview — complex numbers — vector arithmetic— standard library overview — advice.

3.1 Introduction

No significant program is written in just a bare programming language. First, a set of supporting libraries are developed. These then form the basis for further work.

Continuing Chapter 2, this chapter gives a quick tour of key library facilities to give you an idea what can be done using C++ and its standard library. Useful library types, such as ***string***, ***vector***, ***list***, and ***map***, are presented as well as the most common ways of using them. Doing this allows me to give better examples and to set better exercises in the following chapters. As in Chapter 2, 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 an understanding of the simplest uses of the most useful library facilities. A more detailed introduction to the standard library is given in §16.1.2.

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, often referred to as GUIs or window systems, for interaction between a user and a program. Similarly, most application development environments provide ‘‘foundation libraries’’ that support corporate or industrial ‘‘standard’’ development and/or execution environments. 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, but that is left to exercises.

3.2 Hello, world!

The minimal C++ program is

```
int main() { }
```

It defines a function called **main**, which takes no arguments and does nothing.

Every C++ program must have a function named **main()**. The program starts by executing that function. The **int** value returned by **main()**, if any, is the program's return value to ''the system.'' If no value is returned, the system will receive a value indicating successful completion. A nonzero value from **main()** indicates failure.

Typically, a program produces some output. Here is a program that writes out **Hello, world!**:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

The line **#include <iostream>** instructs the compiler to *include* the declarations of the standard stream I/O facilities as found in **iostream**. Without these declarations, the expression

```
std::cout << "Hello, world!\n"
```

would make no sense. The operator **<<** (''put to'') writes its second argument onto its first. In this case, the string literal **"Hello, world!\n"** is written onto the standard output stream **std::cout**. A string literal is a sequence of characters surrounded by double quotes. In a string literal, the backslash character **** followed by another character denotes a single special character. In this case, **\n** is the newline character, so that the characters written are **Hello, world!** followed by a newline.

3.3 The Standard Library Namespace

The standard library is defined in a namespace (§2.4, §8.2) called **std**. That is why I wrote **std::cout** rather than plain **cout**. I was being explicit about using the **standard cout**, rather than some other **cout**.

Every standard library facility is provided through some standard header similar to **<iostream>**. For example:

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

This makes the standard **string** and **list** available. To use them, the **std::** prefix can be used:

```
std::string s = "Four legs Good; two legs Baaad!";
std::list<std::string> slogans;
```

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 §3.7.5, §3.8.6, and §16.1.2). In addition, you must either use the **std::** prefix or make every name from **std** global (§8.2.3). For example:

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

string s = "Ignorance is bliss!";   // ok: string is std::string
```

It is generally in poor taste to dump every name from a namespace into the global namespace. However, to keep short the program fragments used to illustrate language and library features, I omit repetitive **#include**s and **std::** qualifications. In this book, I use the standard library almost exclusively, so if a name from the standard library is used, it either is a use of what the standard offers or part of an explanation of how the standard facility might be defined.

3.4 Output

The iostream library defines output for every built-in type. Further, it is easy to define output of a user-defined type. By default, values output to **cout** are converted to a sequence of characters. For example,

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

will place the character **1** followed by the character **0** on the standard output stream. So will

```
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';
}
```

If **i** has the value **10**, the output will be

```
the value of i is 10
```

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

```
void k()
{
    cout << 'a';
    cout << 'b';
    cout << 'c';
}
```

will output *abc*.

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 is equivalent to `h()`. Streams are explained in more detail in Chapter 21.

3.5 Strings

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

```
string s1 = "Hello";
string s2 = "world";

void m1()
{
    string s3 = s1 + ", " + s2 + "!\n";

    cout << s3;
}
```

Here, *s3* is initialized to the character sequence

Hello, world!

followed by a newline. Addition of strings means concatenation. You can add strings, string literals, and characters to a string.

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 concise and likely to be more efficiently implemented.

Naturally, ***string***s 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 standard library string class is described in Chapter 20. 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"
}
```

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.

3.5.1 C-Style Strings

A C-style string is a zero-terminated array of characters (§5.2.2). As shown, we can easily enter a C-style string into a ***string***. To call functions that take C-style strings, we need to be able to extract the value of a ***string*** in the form of a C-style string. The ***c_str()*** function does that (§20.3.7). For example, we can print the ***name*** using the C output function ***printf()*** (§21.8) like this:

```
void f()
{
    printf("name: %s\n",name.c_str());
}
```

3.6 Input

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
}
```

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*.

Here is an example that performs inch-to-centimeter and centimeter-to-inch conversions. You input a number followed by a character indicating the unit: centimeters or inches. The program then outputs the corresponding value in the other unit:

```
int main()
{
    const double factor = 2.54;      // 1 inch equals 2.54 cm
    double x, in, cm;
    char ch = 0;

    cout << "enter length: ";

    cin >> x;          // read a floating-point number
    cin >> ch;         // read a suffix

    switch (ch) {
    case 'i':          // inch
        in = x;
        cm = x*factor;
        break;
    case 'c':          // cm
        in = x/factor;
        cm = x;
        break;
    default:
        in = cm = 0;
        break;
    }

    cout << in << " in = " << cm << " cm\n";
}
```

The *switch-statement* tests a value against a set of constants. The *break-statement*s are used to exit

the *switch-statement*. The case constants must be distinct. If the value tested does not match any of them, the ***default*** is chosen. The programmer need not provide a ***default***.

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 (§5.2.2) 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 using the ***getline()*** function. For example:

```
int main()
{
    string str;

    cout << "Please enter your name\n";
    getline(cin,str);
    cout << "Hello, " << str << "!\n";
}
```

With this program, the input

```
Eric Bloodaxe
```

yields the desired output:

```
Hello, Eric Bloodaxe!
```

The standard strings have the nice property of expanding to hold what you put in them, so if you enter a couple of megabytes of semicolons, the program will echo pages of semicolons back at you – unless your machine or operating system runs out of some critical resource first.

3.7 Containers

Much computing involves creating collections of various forms of objects 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's most useful 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.

3.7.1 Vector

For many C programmers, a built-in array of (name,number) pairs would seem to be a suitable starting point:

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

Entry phone_book[1000];

void print_entry(int i)    // simple use
{
    cout << phone_book[i].name << ´ ´ << phone_book[i].number << ´\n´;
}
```

However, a built-in array has a fixed size. If we choose a large size, we waste space; if we choose a smaller size, the array will overflow. In either case, we will have to write low-level memory-management code. The standard library provides a ***vector*** (§16.3) that takes care of that:

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

void print_entry(int i)    // simple use, exactly as for array
{
    cout << phone_book[i].name << ´ ´ << phone_book[i].number << ´\n´;
}

void add_entries(int n) // increase size by n
{
    phone_book.resize(phone_book.size()+n);
}
```

The ***vector*** member function ***size()*** gives the number of elements.

Note the use of parentheses in the definition of ***phone_book***. We made a single object of type ***vector<Entry>*** and supplied its initial size as an initializer. This is very different from declaring a built-in array:

```
vector<Entry> book(1000);      // vector of 1000 elements
vector<Entry> books[1000];     // 1000 empty vectors
```

If you make the mistake of using `[]` where you meant `()` when declaring a ***vector***, your compiler will almost certainly catch the mistake and issue an error message when you try to use the ***vector***.

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 should be used.

3.7.2 Range Checking

The standard library ***vector*** does not provide range checking by default (§16.3.3). For example:

```
void f()
{
    int i = phone_book[1001].number;    // 1001 is out of range
    // ...
}
```

The initialization is likely to place some random value in ***i*** rather than giving an error. This is undesirable, so I will use a simple range-checking adaptation of ***vector***, called ***Vec***, in the following chapters. A ***Vec*** is like a ***vector***, except that it throws an exception of type ***out_of_range*** (presented in ***<stdexcept>***) if a subscript is out of range.

Techniques for implementing types such as ***Vec*** and for using exceptions effectively are discussed in §11.12, §8.3, Chapter 14, and Appendix E. However, the definition here is sufficient for the examples in this book:

```
template<class T> class Vec : public vector<T> {
public:
    Vec() : vector<T>() { }
    Vec(int s) : vector<T>(s) { }

    T& operator[](int i) { return at(i); }              // range-checked
    const T& operator[](int i) const { return at(i); }  // range-checked
};
```

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 (§16.3.3). If necessary, it is possible to prevent access to the ***vector<T>*** base; see §15.3.2.

Returning to the problem of keeping names and telephone numbers, we can now use a ***Vec*** to ensure that out-of-range accesses are caught. For example:

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

```
void print_entry(int i)      // simple use, exactly as for vector
{
      cout << phone_book[i].name << ´ ´ << phone_book[i].number << ´\n´;
}
```

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

```
void f()
{
      try {
            for (int i = 0; i<10000; i++) print_entry(i);
      }
      catch (out_of_range) {
            cout << "range error\n";
      }
}
```

The exception will be thrown, and then caught, when *phone_book[i]* is tried with *i==1000*.
If the user doesn't catch this kind of 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 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* (§21.2.1).

3.7.3 List

Insertion and deletion of phone book entries could be common. Therefore, a list could be more appropriate than a vector for representing a simple phone book. For example:

```
list<Entry> phone_book;
```

When we use a 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 §3.8:

```
void print_entry(const string& s)
{
      typedef list<Entry>::const_iterator LI;
```

```
        for (LI i = phone_book.begin(); i != phone_book.end(); ++i) {
                const Entry& e = *i; // reference used as shorthand
                if (s == e.name) {
                        cout << e.name << ´ ´ << e.number << ´\n´;
                        return;
                }
        }
}
```

The search for ***s*** starts at the beginning of the list and proceeds until either ***s*** is found or the end is reached. 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 (§16.3.2). Given an iterator ***i***, ***++i*** advances ***i*** to refer to the next element. Given an iterator ***i***, the element it refers to is ****i***.

A user need not know the exact type of the iterator for a standard container. That iterator type is part of the definition of the container and can be referred to by name. When we don't need to modify an element of the container, ***const_iterator*** is the type we want. Otherwise, we use the plain ***iterator*** type (§16.3.1).

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

```
void f(const Entry& e, list<Entry>::iterator i, list<Entry>::iterator p)
{
        phone_book.push_front(e);       // add at beginning
        phone_book.push_back(e);        // add at end
        phone_book.insert(i,e);         // add before the element referred to by 'i'

        phone_book.erase(p);            // remove the element referred to by 'p'
}
```

For a more complete description of ***insert***() and ***erase***(), see §16.3.6.

3.7.4 Map

Writing code to look up a name in a list of (name,number) pairs is really quite tedious. In addition, a linear search is quite inefficient for all but the shortest lists. Other data structures directly support insertion, deletion, and searching based on values. In particular, the standard library provides the ***map*** type (§17.4.1). A ***map*** is a container of pairs of values. For example:

```
map<string,int> phone_book;
```

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

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:

```
void print_entry(const string& s)
{
        if (int i = phone_book[s]) cout << s << ´ ´ << i << ´\n´;
}
```

If no match was found for the key ***s***, a default value is returned from the ***phone_book***. The default value for an integer type in a ***map*** is ***0***. Here, I assume that ***0*** isn't a valid telephone number.

3.7.5 Standard Containers

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 a ***vector*** is cheap and easy. On the other hand, inserting an element between two elements tends to be expensive. A ***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.

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-sized vector (§16.3)
list<T>	A doubly-linked list (§17.2.2)
queue<T>	A queue (§17.3.2)
stack<T>	A stack (§17.3.1)
deque<T>	A double-ended queue (§17.2.3)
priority_queue<T>	A queue sorted by value (§17.3.3)
set<T>	A set (§17.4.3)
multiset<T>	A set in which a value can occur many times (§17.4.4)
map<key,val>	An associative array (§17.4.1)
multimap<key,val>	A map in which a key can occur many times (§17.4.2)

The standard containers are presented in §16.2, §16.3, and Chapter 17. The containers are defined in namespace ***std*** and presented in headers ***<vector>***, ***<list>***, ***<map>***, etc. (§16.2).

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. In general, basic operations apply to every kind of container. For example, ***push_back()*** can be used (reasonably efficiently) to add elements to the end of a ***vector*** as well as for a ***list***, and every container has a ***size()*** member function that returns its number of elements.

This notational and semantic uniformity enables programmers to provide new container types to be used like the standard ones. The range-checked vector, ***Vec*** (§3.7.2), is an example of that. Chapter 17 demonstrates how a ***hash_map*** can be added to the framework. The uniformity of container interfaces also allows us to specify algorithms independently of individual container types.

3.8 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. 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, we can define `==` and `<` for ***Entry***, sort a ***vector*** of ***Entry***s, and place a copy of each unique ***vector*** element on a ***list***:

```
bool operator==(const Entry& a, const Entry& b) { return a.name==b.name; }
bool operator<(const Entry& a, const Entry& b) { return a.name<b.name; }
```

```
void f(vector<Entry>& ve, list<Entry>& le)
{
    sort(ve.begin(),ve.end());
    unique_copy(ve.begin(),ve.end(),le.begin());
}
```

The standard algorithms are described in Chapter 18. They are expressed in terms of sequences of elements (§2.7.2). 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 also be overwritten.

If we wanted to add the new elements to the end of a container, we could have written:

```
void f(vector<Entry>& ve, list<Entry>& le)
{
    sort(ve.begin(),ve.end());
    unique_copy(ve.begin(),ve.end(),back_inserter(le));    // append to le
}
```

A ***back_inserter***( ) adds elements at the end of a container, extending the container to make room for them (§19.2.4). Thus, the standard containers plus ***back_inserter***( )s eliminate the need to use error-prone, explicit C-style memory management using ***realloc***( ) (§16.3.5). Forgetting to use a ***back_inserter***( ) when appending can lead to errors. For example:

```
void f(vector<Entry>& ve, list<Entry>& le)
{
    copy(ve.begin(),ve.end(),le);            // error: le not an iterator
    copy(ve.begin(),ve.end(),le.end());      // bad: writes beyond the end
    copy(ve.begin(),ve.end(),le.begin());    // overwrite elements
}
```

3.8.1 Use of Iterators

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. Using ***find***, we can count the number of occurrences of a character in a ***string***:

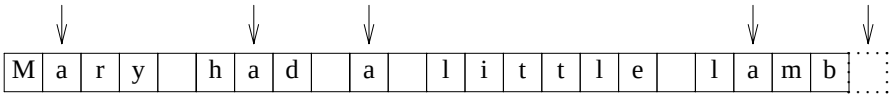
```
int count(const string& s, char c)    // count occurrences of c in s
{
    int n = 0;
    string::const_iterator i = find(s.begin(),s.end(),c);
    while (i != s.end()) {
        ++n;
        i = find(i+1,s.end(),c);
    }
    return n;
}
```

The ***find*** algorithm returns an iterator to the first occurrence of a value in a sequence or the one-past-the-end iterator. Consider what happens for a simple call of ***count***:

```
void f()
{
    string m = "Mary had a little lamb";
    int a_count = count(m,´a´);
}
```

The first call to ***find***() finds the ´***a***´ in ***Mary***. Thus, the iterator points to that character and not to ***s.end***(), so we enter the loop. In the loop, we start the search at ***i+1***; that is, we start one past where we found the ´***a***´. We then loop finding the other three ´***a***´s. That done, ***find***() reaches the end and returns ***s.end***() so that the condition ***i!=s.end***() fails and we exit the loop.

That call of ***count***() could be graphically represented like this:



The arrows indicate the initial, intermediate, and final values of the iterator ***i***.

Naturally, the ***find*** algorithm will work equivalently on every standard container. Consequently, we could generalize the ***count***() function in the same way:

```
template<class C, class T> int count(const C& v, T val)
{
    typename C::const_iterator i = find(v.begin(),v.end(),val);  // "typename"; see §C.13.5
    int n = 0;
    while (i != v.end()) {
        ++n;
        ++i; // skip past the element we just found
        i = find(i,v.end(),val);
    }
    return n;
}
```

This works, so (using ***complex*** from <complex> ; §3.9.1, §22.5) we can say:

```
void f(list< complex<double> >& lc, vector<string>& vs, string s)
{
    int i1 = count(lc,complex<double>(1,3));
    int i2 = count(vs,"Diogenes");
    int i3 = count(s,´x´);
}
```

However, we don't have to define a ***count*** template. Counting occurrences of an element is so generally useful that the standard library provides that algorithm. To be fully general, the standard library ***count*** takes a sequence as its argument, rather than a container, so we would say:

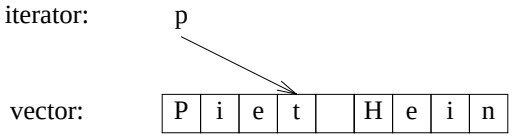
```
void f(list< complex<double> >& lc, vector<string>& vs, string s)
{
    int i1 = count(lc.begin(),lc.end(),complex<complex>(1,3));
    int i2 = count(vs.begin(),vs.end(),"Diogenes");
    int i3 = count(s.begin(),s.end(),'x');
}
```

The use of a sequence allows us to use ***count*** for a built-in array and also to count parts of a container. For example:

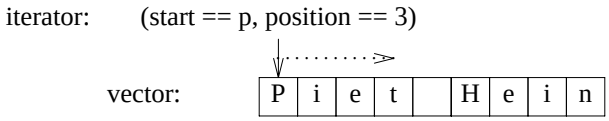
```
void g(char cs[], int sz)
{
    int i1 = count(&cs[0],&cs[sz],'z');      // 'z's in array
    int i2 = count(&cs[0],&cs[sz/2],'z');    // 'z's in first half of array
}
```

3.8.2 Iterator Types

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 is most likely an ordinary pointer because a pointer is quite a reasonable way of referring to an element of a ***vector***:

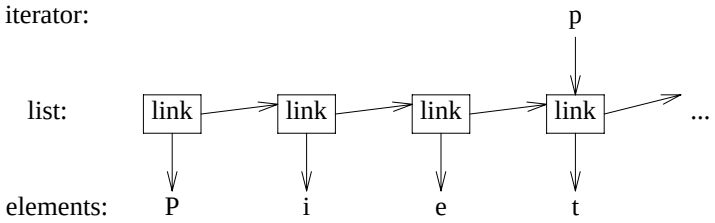


Alternatively, a ***vector*** iterator could be implemented as a pointer to the ***vector*** plus an index:



Using such an iterator would allow range checking (§19.3).

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 (§19.2.1). 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>***. I rarely have to worry about the details of how that type is defined.

3.8.3 Iterators and I/O

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. This way wouldn't be my first choice for that simple task, but the utility of treating output as a write-only container will soon be obvious – if it isn't already.

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);
```

Because input iterators invariably appear in pairs representing a sequence, we must provide an

istream_iterator to indicate the end of input. This is the default ***istream_iterator***:

```
istream_iterator<string> eos;
```

We could now read ***Hello, world!*** from input and write it out again like this:

```
int main()
{
    string s1 = *ii;
    ++ii;
    string s2 = *ii;

    cout << s1 << ' ' << s2 << '\n';
}
```

Actually, ***istream_iterator***s and ***ostream_iterator***s are not meant to be used directly. Instead, they are typically 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.c_str());              // input stream (c_str(); see §3.5.1 and §20.3.7)
    istream_iterator<string> ii(is);        // input iterator for stream
    istream_iterator<string> eos;           // input sentinel

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

    ofstream os(to.c_str());                // output stream
    ostream_iterator<string> oo(os,"\n");   // output iterator for stream

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

    return !is.eof() || !os;                // return error state (§3.2, §21.3.3)
}
```

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.

3.8.4 Traversals and Predicates

Iterators allow us to write loops to iterate through a sequence. However, writing loops can be tedious, so the standard library provides ways for a function to be called for each element of a sequence.

Consider writing a program that reads words from input and records the frequency of their occurrence. The obvious representation of the strings and their associated frequencies is a ***map***:

```
map<string,int> histogram;
```

The obvious action to be taken for each string to record its frequency is:

```
void record(const string& s)
{
    histogram[s]++;      // record frequency of ''s''
}
```

Once the input has been read, we would like to output the data we have gathered. The ***map*** consists of a sequence of (string,int) pairs. Consequently, we would like to call

```
void print(const pair<const string,int>& r)
{
    cout << r.first << ' ' << r.second << '\n';
}
```

for each element in the map (the first element of a ***pair*** is called ***first***, and the second element is called ***second***). The first element of the ***pair*** is a ***const string*** rather than a plain ***string*** because all ***map*** keys are constants.

Thus, the main program becomes:

```
int main()
{
    istream_iterator<string> ii(cin);
    istream_iterator<string> eos;

    for_each(ii,eos,record);
    for_each(histogram.begin(),histogram.end(),print);
}
```

Note that we don't need to sort the ***map*** to get the output in order. A ***map*** keeps its elements ordered so that an iteration traverses the ***map*** in (increasing) order.

Many programming tasks involve looking for something in a container rather than simply doing something to every element. For example, the ***find*** algorithm (§18.5.2) provides a convenient way of looking for a specific value. A more general variant of this idea looks for an element that fulfills a specific requirement. 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***:

```
bool gt_42(const pair<const string,int>& r)
{
    return r.second>42;
}

void f(map<string,int>& m)
{
    typedef map<string,int>::const_iterator MI;
    MI i = find_if(m.begin(),m.end(),gt_42);
    // ...
}
```

Alternatively, we could count the number of words with a frequency higher than 42:

```
void g(const map<string,int>& m)
{
    int c42 = count_if(m.begin(),m.end(),gt_42);
    // ...
}
```

A function, such as ***gt_42()***, that is used to control the algorithm is called a *predicate*. A predicate is called for each element and returns a Boolean value, which the algorithm uses to perform its intended action. For example, ***find_if()*** searches until its predicate returns ***true*** to indicate that an element of interest has been found. Similarly, ***count_if()*** counts the number of times its predicate is ***true***.

The standard library provides a few useful predicates and some templates that are useful for creating more (§18.4.2).

3.8.5 Algorithms Using Member Functions

Many algorithms apply a function to elements of a sequence. For example, in §3.8.4

```
for_each(ii,eos,record);
```

calls ***record()*** for each string read from input.

Often, we deal with containers of pointers and we really would like to call a member function of the object pointed to, rather than a global function on the pointer. For example, we might want to call the member function ***Shape::draw()*** for each element of a ***list<Shape*>***. To handle this specific example, we simply write a nonmember function that invokes the member function. For example:

```
void draw(Shape* p)
{
    p->draw();
}

void f(list<Shape*>& sh)
{
    for_each(sh.begin(),sh.end(),draw);
}
```

By generalizing this technique, we can write the example like this:

```
void g(list<Shape*>& sh)
{
    for_each(sh.begin(),sh.end(),mem_fun(&Shape::draw));
}
```

The standard library ***mem_fun()*** template (§18.4.4.2) takes a pointer to a member function (§15.5) as its argument and produces something that can be called for a pointer to the member's class. The result of ***mem_fun(&Shape::draw)*** takes a ***Shape**** argument and returns whatever ***Shape::draw()*** returns.

The ***mem_fun()*** mechanism is important because it allows the standard algorithms to be used for containers of polymorphic objects.

3.8.6 Standard Library Algorithms

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 set of templates 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. Here are a few I have found particularly useful:

Selected Standard Algorithms	
for_each()	Invoke function for each element (§18.5.1)
find()	Find first occurrence of arguments (§18.5.2)
find_if()	Find first match of predicate (§18.5.2)
count()	Count occurrences of element (§18.5.3)
count_if()	Count matches of predicate (§18.5.3)
replace()	Replace element with new value (§18.6.4)
replace_if()	Replace element that matches predicate with new value (§18.6.4)
copy()	Copy elements (§18.6.1)
unique_copy()	Copy elements that are not duplicates (§18.6.1)
sort()	Sort elements (§18.7.1)
equal_range()	Find all elements with equivalent values (§18.7.2)
merge()	Merge sorted sequences (§18.7.3)

These algorithms, and many more (see Chapter 18), can be applied to elements of containers, ***string***s, and built-in arrays.

3.9 Math

Like C, C++ wasn't designed primarily with numerical computation in mind. However, a lot of numerical work is done in C++, and the standard library reflects that.

3.9.1 Complex Numbers

The standard library supports a family of complex number types along the lines of the ***complex*** class described in §2.5.2. To support complex numbers where the scalars are single-precision, floating-point numbers (***float***s), double precision numbers (***double***s), etc., the standard library ***complex*** is a template:

```
template<class scalar> class complex {
public:
    complex(scalar re, scalar im);
    // ...
};
```

The usual arithmetic operations and the most common mathematical functions are supported for complex numbers. For example:

```
// standard exponentiation function from <complex>:
template<class C> complex<C> pow(const complex<C>&, int);

void f(complex<float> fl, complex<double> db)
{
    complex<long double> ld = fl+sqrt(db);
    db += fl*3;
    fl = pow(1/fl,2);
    // ...
}
```

For more details, see §22.5.

3.9.2 Vector Arithmetic

The ***vector*** described in §3.7.1 was designed to be a general mechanism for holding values, to be flexible, and to fit into the architecture of containers, iterators, and algorithms. However, it does not support mathematical vector operations. Adding such operations to ***vector*** would be easy, but its generality and flexibility precludes optimizations that are often considered essential for serious numerical work. Consequently, the standard library provides a vector, called ***valarray***, that is less general and more amenable to optimization for numerical computation:

```
template<class T> class valarray {
    // ...
    T& operator[](size_t);
    // ...
};
```

The type ***size_t*** is the unsigned integer type that the implementation uses for array indices.

The usual arithmetic operations and the most common mathematical functions are supported for ***valarray***s. For example:

```
// standard absolute value function from <valarray>:
template<class T> valarray<T> abs(const valarray<T>&);

void f(valarray<double>& a1, valarray<double>& a2)
{
    valarray<double> a = a1*3.14+a2/a1;
    a2 += a1*3.14;
    a = abs(a);
    double d = a2[7];
    // ...
}
```

For more details, see §22.4.

3.9.3 Basic Numeric Support

Naturally, the standard library contains the most common mathematical functions – such as ***log***(), ***pow***(), and ***cos***() – for floating-point types; see §22.3. In addition, classes that describe the properties of built-in types – such as the maximum exponent of a ***float*** – are provided; see §22.2.

3.10 Standard Library Facilities

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 §16.1.3.
[2] The C standard library (with very minor modifications to minimize violations of the type system); see §16.1.2.
[3] Strings and I/O streams (with support for international character sets and localization); see Chapter 20 and Chapter 21.
[4] A framework of containers (such as ***vector***, ***list***, and ***map***) and algorithms using containers (such as general traversals, sorts, and merges); see Chapter 16, Chapter 17, Chapter 18, and Chapter 19.
[5] Support for numerical computation (complex numbers plus vectors with arithmetic operations, BLAS-like and generalized slices, and semantics designed to ease optimization); see Chapter 22.

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. Essentially, the C++ standard library provides the most common fundamental data structures together with the fundamental algorithms used on them.

Every algorithm works with every container without the use of conversions. This framework, conventionally called the STL [Stepanov,1994], is extensible in the sense that users can easily provide containers and algorithms in addition to the ones provided as part of the standard and have these work directly with the standard containers and algorithms.

3.11 Advice

[1] Don't reinvent the wheel; use libraries.
[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 to 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; §3.3.
[6] Remember that standard library facilities are defined in namespace ***std***; §3.3.
[7] Use ***string*** rather than ***char****; §3.5, §3.6.
[8] If in doubt use a range-checked vector (such as ***Vec***); §3.7.2.
[9] Prefer ***vector<T>***, ***list<T>***, and ***map<key,value>*** to ***T***[]; §3.7.1, §3.7.3, §3.7.4.
[10] When adding elements to a container, use ***push_back***() or ***back_inserter***(); §3.7.3, §3.8.
[11] Use ***push_back***() on a ***vector*** rather than ***realloc***() on an array; §3.8.
[12] Catch common exceptions in ***main***(); §3.7.2.