2

A Tour of C++: The Basics

The first thing we do, let´s
kill all the language lawyers.
– Henry VI, part II

- Introduction
- The Basics
  Hello, World!; Types, Variables, and Arithmetic; Constants; Tests and Loops; Pointers, Arrays, and Loops
- User-defined Types
  Structures; Classes; Enumerations
- Modularity
  Separate Compilation; Namespaces; Error Handling
- Postscript
- Advice

2.1 Introduction [tour1.intro]

The aim of this chapter and the next three is to give you an idea of what C++ is, without going into a lot of details. This chapter informally presents the notation of C++, C++'s model of memory and computation, and the basic mechanisms for organizing code into a program. These are the language facilities supporting the styles most often seen in C and sometimes called procedural programming. Chapter 3 follows up by presenting C++'s abstraction mechanisms. Chapter 4 and Chapter 5 give examples of 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],

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

before continuing here. Even if you have programmed before, the language 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, skip to the more systematic presentation starting in Chapter 6.

This chapter and the next three save us from a strictly bottom-up presentation of language and library facilities by enabling the use of a rich set of facilities even in early chapters. For example, loops are not discussed in detail until Chapter 10, but they will be used in obvious ways long before that. Similarly, the detailed description of classes, templates, free-store use and the standard library are spread over many chapters, but standard-library types, such as vector, string, complex, map, unique_pointer, and ostream, are used freely where needed to improve code examples.

As an analogy, think of a short sightseeing tour of a city, such as Copenhagen or New York. In just a few hours, you are given a quick peek at the major attractions, told a few background stories, and usually given some suggestions about what to see next. You do *not* know the city after such a tour. You do *not* understand all you have seen and heard. To really know a city, you have to live in it, often for years. However, with a bit of luck, you will have gained a bit of an overview, a notion of what might be special about the city, and ideas of what might be of interest to you. After the tour, the real exploration can begin.

2.2 The Basics [tour1.basics]

C++ is a compiled language. That means that for a program to run, its source text has to be processed by a compiler, producing object files, which are combined by a linker yielding an executable program. A C++ program typically consists of many source code files (typically simply called "source files"). An executable program is created for a specific hardware/system combination; it is not portable, say, from a Mac to a Windows PC. When we talk about portability of C++ programs, we usually mean portability of source code; that is, the source code can be successfully compiled and run on a variety of systems.

The ISO C++ standard defines two kinds of entities:

- *Core language features*, such as built-in types (e.g., char and int) and loops (e.g., for-statements and while-statements)
- *Standard-library components*, such as containers (e.g., vector and map) and I/O operations (e.g., << and getline())

The standard-library components are mostly perfectly ordinary C++ code that happens to be provided by every C++ implementation. That is, the C++ standard library can be implemented in C++ itself (and is with very minor uses of machine code for things such as thread context switching). This implies that C++ is sufficiently expressive and efficient for the most demanding systems programming tasks.

C++ is a statically typed language. That is, the type of every entity (e.g., object, value, name, and expression) must be known to the compiler at its point of use. The type of an object determines the set of operations applicable to it.

D R A F T

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

2.2.1 Hello, World! [tour1.hello]

The minimal C++ program is

```
int main() { }            // the minimal C++ program
```

This defines a function called main, which takes no arguments and does nothing (§15.4).

Curly braces, { }, express grouping in C++. Here, they indicate the start and end of the function body. The double slash, //, begins a comment that extends to the end of the line. A comment is for the human reader; the compiler ignores comments.

Every C++ program must have exactly one global 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. Not every operating system and execution environment make use of that return value: Linux/Unix-based environments often do, but Windows-based environments rarely do.

Typically, a program produces some output. Here is a program that writes 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.

The std:: specifies that the name cout is to be found in the standard-library namespace (§2.4.2, Chapter 14).

Essentially all executable code is placed in functions and called directly or indirectly from main(). For example:

```
#include <iostream>

double square(double x)    // square a double precision floating-point number
{
        return x*x;
}
```

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
void print_square(double x)
{
      std::cout << "the square of " << x << " is " << square(x) << '\n';
}

int main()
{
      print_square(1.234);  // print: the square of 1.234 is 1.52276
      print_square(5.555);  // print: the square of 5.555 is 30.858
}
```

A ‘‘return type’’ void indicates that a function does not return a value.

2.2.2 Types, Variables, and Arithmetic [tour1.var]

Every name and every expression has a type that determines the operations that may be performed on it. For example, the declaration

```
int inch;
```

specifies that inch is of type int; that is, inch is an integer variable.

A *declaration* is a statement that introduces a name into the program. It specifies a type for the named entity:

- A *type* defines a set of possible values and a set of operations (for an object).
- An *object* is some memory that holds a value of some type.
- A *value* is a set of bits interpreted according to a type.
- A *variable* is a named object.

C++ offers a variety of fundamental types, which correspond directly to hardware facilities. For example:

```
bool        // Boolean, possible values are true and false
char        // character, for example, 'a', 'z', and '9'
int         // integer, for example, 1, 42, and 1066
double      // double-precision floating-point number, for example, 3.14 and 299793.0
```

Each of the fundamental types has a fixed size that determines the range of values that can be stored in them (for integers) or the precision and range of those values (for floating point numbers). A char variable is of the natural size to hold a character on a given machine (typically an 8-bit byte), and the sizes of other types are quoted in multiples of the size of a char. The size of a type is implementation defined (i.e., it can vary among different machines) and can be obtained by the sizeof operator; for example sizeof(char) equals 1 and sizeof(int) is often 4. We can represent sizes graphically:

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

bool: [1 byte]

char: [1 byte]

int: [4 bytes]

double: [8 bytes]

The arithmetic operators can be used for appropriate combinations of these types:

```
x+y      // plus
+x       // unary plus
x−y      // minus
−x       // unary minus
x∗y      // multiply
x/y      // divide
x%y      // remainder (modulus) for integers
```

So can the comparison operators:

```
x==y     // equal
x!=y     // not equal
x<y      // less than
x>y      // greater than
x<=y     // less than or equal
x>=y     // greater than or equal
```

In assignments and in arithmetic operations, C++ performs all meaningful conversions (§10.5.3) between the basic types so that they can be mixed freely:

```
void some_function() // function that doesn't return a value
{
     double d = 2.2;  // initialize floating-point number
     int i = 7;       // initialize integer
     d = d+i;         // assign sum to d
     i = d∗i;         // assign product to i (truncating the double to an int)
}
```

Note that = is the assignment operator and == tests equality.

C++ offers a variety of notations for expressing initialization, such as the = used above, and a universal form based on curly brace delimited initializer lists:

```
double d1 = 2.3;
double d2 {2.3};
```

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
complex<double> z = 1;          // a complex number with double-precision floating-point scalars
complex<double> z2 {d1,d2};
complex<double> z3 = {1,2};     // the = is optional with { ... }

vector<int> v {1,2,3,4,5,6};    // a vector of ints
```

The = form is traditional and dates back to C, but if in doubt, use the general {}-list form (§6.3.5.2). If nothing else, it saves you from conversions that lose information (narrowing conversions; §10.5):

```
int i1 = 7.2;     // i1 becomes 7
int i2 {7.2};     // error: floating-point to integer conversion
```

A constant (§2.2.3) cannot be left uninitialized and a variable should only be left uninitialized in extremely rare circumstances. Don't introduce a name until you have a suitable value for it. User-defined types (such as, string, vector, Matrix, and Orc) can be defined to be implicitly initialized (§3.2.1.1).

When defining a variable, you don't actually need to state its type explicitly when it can be deduced from the initializer:

```
auto b = true;        // a bool
auto ch = 'x';        // a char
auto i = 123;         // an int
auto d = 1.2;         // a double
auto z = sqrt(y);     // z has the type of whatever sqrt(y) returns
```

With auto, we use the = syntax because there is no type conversion involved that might cause problems (§6.3.6.2).

We use auto where we don't have a specific reason to mention the type explicitly. "Specific reasons" include

- The definition is in a large scope where we want to make the type clearly visible to readers of our code
- We want to be explicit about a variable's range or precision (e.g., long double rather than double).

Using auto, we avoid redundancy and typing long type names. This is especially important in generic programming where the exact type of an object can be hard for the programmer to know and the type names can be quite long (§4.5.1).

In addition to the conventional arithmetic and logical operators (§10.3), C++ offers more specific operations for modifying a variable:

```
x+=y     // x = x+y
++x      // increment: x = x+1
x−=y     // x = x-y
−−x      // decrement: x = x-1
x∗=y     // scaling: x = x*y
x/=y     // scaling: x = x/y
x%=y     // x = x%y
```

These operators are concise, convenient, and very frequently used.

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

2.2.3 Constants [tour1.const]

C++ supports two notions of immutability (§7.5):

- const: meaning roughly ‘‘I promise not to change this value’’ (§7.5). This is used primarily to specify interfaces, so that data can be passed to functions without fear of it being modified. The compiler enforces the promise made by const.
- constexpr: meaning roughly ‘‘to be evaluated at compile time’’ (§10.4). This is used primarily to specify constants, to allow placement of data in memory where it is unlikely to be corrupted, and for performance.

For example:

```
double sum(const vector<double>&);              // sum will not modify its argument
const int dmv = 17;                              // dmv is a named constant
constexpr double max1 = 1.4*square(dmv);         // OK if square(17) is a constant expression
const double max2 = 1.4*square(dmv);             // OK, may be evaluated at run time
vector<double> v { 1.2, 3.4, 4.5 };              // v is not a constant
const double s1 = sum(v);                        // OK: evaluated at run time
constexpr double s2 = sum(v);                    // error: sum(v) not constant expression
```

For a function to be useful in a *constant expression*, that is, in an expression that will be evaluated by the compiler, it must be defined constexpr. For example,

```
constexpr double square(double x) { return x*x; }
```

To be constexpr a function must be rather simple: just a return statement computing a value. A constexpr function can be used for non-constant arguments, but when that is done the result is not a constant expression. We allow a constexpr function to be called with non-constant-expression arguments in contexts that do not require constant expressions, so that we don’t have to define essentially the same function twice: once for constant expressions and once for variables.

In a few places, constant expressions are required by language rules (e.g., array bounds (§2.2.5, §7.3), case labels (§2.2.4, §9.4.2), and some template arguments (§25.2). In other cases, compile-time evaluation is important for performance. Independently of performance issues, the notion of immutability (of an object with an unchangeable state) is an important design concern (§10.4).

2.2.4 Tests and Loops [tour1.loop]

C++ provides a conventional set of statements for expressing selection and looping. For example, here is a simple function that prompts the user and returns a Boolean indicating the response:

```
bool accept()
{
     cout << "Do you want to proceed (y or n)?\n";      // write question

     char answer = 0;
     cin >> answer;                                       // read answer
```

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
        if (answer == 'y') return true;
        return false;
}
```

To match the << output operator (“put to”), the >> operator (“get from”) is used for input; cin is the standard input stream. The type of the right-hand operand of >> determines what input is accepted and its right-hand operand is the target of the input operation. The \n character at the end of the output string represents a newline (§2.2.1).

The example could be slightly improved by taking an n (for “no”) answer into account:

```
bool accept2()
{
        cout << "Do you want to proceed (y or n)?\n";     // write question

        char answer = 0;
        cin >> answer;                                      // read answer

        switch (answer) {
        case 'y':
                return true;
        case 'n':
                return false;
        default:
                cout << "I'll take that for a no.\n";
                return false;
        }
}
```

A switch-statement tests a value against a set of constants. The case constants must be distinct, and if the value tested does not match any of them, the default is chosen. The programmer need not provide a default. If there is no default, no action is taken if the value doesn't match any case constant.

Few programs are written without loops. In this case, we might like to give the user a few tries:

```
bool accept3()
{
        int tries = 1;
        while (tries<4) {
                cout << "Do you want to proceed (y or n)?\n";     // write question
                char answer = 0;
                cin >> answer;                                      // read answer
```

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
        switch (answer) {
        case 'y':
                return true;
        case 'n':
                return false;
        default:
                cout << "Sorry, I don't understand that.\n";
                ++tries;      // increment
        }
    }
    cout << "I'll take that for a no.\n";
    return false;
}
```

The while-statement executes until its condition becomes false.

2.2.5 Pointers, Arrays and Loops [tour1.ptr]

An array of elements of type char can be declared like this:

```
char v[6];          // array of 6 characters
```

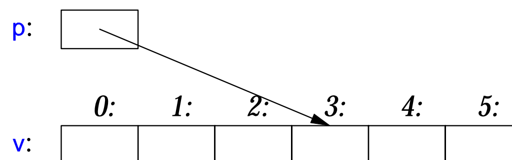
Similarly, a pointer can be declared like this:

```
char∗ p;            // pointer to character
```

In declarations, [] means ‘‘array of’’ and ∗ means ‘‘pointer to.’’ All arrays have 0 as their lower bound, so v has six elements, v[0] to v[5]. The size of an array must be a constant expression (§2.2.3). A pointer variable can hold the address of an object of the appropriate type:

```
char∗ p = &v[3]; // p points to v's fourth element
int x = ∗p;         // read the value of what p points to
```

In an expression, prefix unary ∗ means ‘‘contents of’’ and prefix unary & means ‘‘address of.’’ We can represent the result of that initialized definition graphically:



Consider copying ten elements from one array to another:

```
void copy_fct()
{
    int v1[10];
    int v2[10] = {0,1,2,3,4,5,6,7,8,9};
```

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
        for (auto i=0; i<10; ++i)
                v1[i]=v2[i];
        // ...
}
```

This for-statement can be read as "set i to zero; while i is less than 10, copy the ith element and increment i." When applied to an integer variable, the increment operator ++ simply adds 1. C++ also offers a simpler for-statement, called a range-for-statement, for loops that traverse a sequence in the simplest way:

```
void print()
{
        int v[] = {0,1,2,3,4,5,6,7,8,9};

        for (auto x : v)                // for each x in v
                cout << x << '\n';

        for (auto x : {10,21,32,43,54,65})
                cout << x << '\n';
        // ...
}
```

This range-for-statement can be read as "for every element of v, from the first to the last, place a copy in x and print it." Note that we don't have to specify an array bound when we initialize it with a list. The range-for-statement can be used for any sequence of elements (§3.4.1).

If we didn't want to copy the values from v into the variable x, but rather just have x refer to an element, we could write:

```
void increment()
{
        int v[] = {0,1,2,3,4,5,6,7,8,9};

        for (auto& x : v)
                ++x;
        // ...
}
```

In a declaration, the unary suffix & means "reference to." A reference is similar to a pointer, except that you don't need to use a prefix * to get to the value referred to by the reference. When used in declarations, operators (such as * and []) are called *declarator operators*:

```
T a[n];     // T[n]: array of n Ts (§7.3)
T* p;       // T*: pointer to T (§7.2)
T& r;       // T&: reference to T   (§7.7)
T f(A);     // T(A): function taking an argument of type A returning a result of type T (§2.2.1)
```

We try to ensure that a pointer always points to an object, so that dereferencing it is valid. When we don't have an object to point to or if we need to represent the notion of "no object available" (e.g. for an end of a list), we give the pointer the value nullptr ("the null

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

pointer''). There is only one nullptr shared by all pointer types:

```
double* pd = nullptr;
Link<Record>* lst = nullptr;
```

It is often wise to check whether a pointer argument actually points to something:

```
int count_x(char* p, char x)      // count the number of occurrences of x in p[]
        // p is assumed to point to a zero-terminated array of char (or to nothing)
{
        if (p==nullptr) return 0;

        int count = 0;
        for (; *p!=0; ++p)
                if (*p==x)
                        ++count;
        return count;
}
```

Note how we can move a pointer to point to the next element of an array using ++ and that we can leave out the initializer in a for-statement if we don't need it.

The definition of count_x() assumes that the char* is a *C-style string*; that is, that the pointer points to a zero-terminated array of char.

In older code, 0 or NULL is typically used instead of nullptr (§7.2.2). However, using nullptr eliminates potential confusion between integers (such as 0 or NULL) and pointers, (such as nullptr).

2.3 User-defined Types [tour1.udt]

We call the types that can be built out of the fundamental types (§2.2.2), the const modifier (§2.2.3), and the declarator operators (§2.2.5) *built-in types*. C++'s set of built-in types and operations is rich, but deliberately low-level. They directly and efficiently reflect the capabilities of conventional computer hardware. However, they don't provide the programmer with high-level facilities to conveniently write advanced applications. Instead, C++ augments the built-in types and operations with a sophisticated set of *abstraction mechanisms* out of which programmers can build such high-level facilities. The C++ abstraction mechanisms are primarily designed to let programmers design and implement their own types, with suitable representations and operations, and for programmers to simply and elegantly use such types. Types built out of the built-in types using C++'s abstraction mechanisms are called *user-defined types*. They are referred to as classes and enumerations. Most of this book is devoted to the design, implementation, and use of user-defined types. The rest of this chapter presents the simplest and most fundamental facilities for that. Chapter 3 is a more complete description of the abstraction mechanisms and the programming styles they support. Chapter 4 and Chapter 5 present an overview of the standard library, and since the standard library mainly consists of user-defined types, they provide examples of what can be built using the language facilities and programming techniques presented in Chapter 2 and Chapter 3.

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

2.3.1 Structures [tour1.struct]

The first step in building a new type is often to organize the elements it needs into a data structure, a struct:

```
struct Vector {
	int sz;          // number of elements
	double* elem;    // pointer to elements
};
```

This Vector consists of an int and a double*. A variable of type Vector can be defined like this:

```
Vector v;
```

However, by itself that is not of much use because the elem pointer doesn't point to any elements. To be useful, we must give v some elements to point to. For example, we can construct a Vector like this:

```
void vector_init(Vector& v, int s)
{
	v.elem = new double[s];     // allocate an array of s doubles
	v.sz = s;
}
```

That is, v's elem member gets a pointer produced by the new operator and v's size member gets the number of elements. The & in Vector& indicates that we pass v by non-const reference; that way, vector_init() can modify the vector passed to it.

The new operator allocates memory from an area called "the free store" (also known as "dynamic memory" and "heap"; §11.2).

A simple use of Vector looks like this:

```
double read_and_sum(int s)
{
	Vector v;
	vector_init(v,s);          // allocate s elements for v
	for (int i=0; i<s; ++i)
		cin>>v.elem[i];        // read into elements

	double sum = 0;
	for (int i=0; i<s; ++i)
		sum+=v.elem[i];        // take the sum of the elements
	return sum;
}
```

There is a long way to go before our Vector is as elegant and flexible as the standard-library vector. In particular, a user of Vector has to know every detail of Vector's representation. The rest of this chapter and the next gradually improve Vector as an example of language features and techniques, Chapter 4 presents the standard-library vector, which contains all the nice improvements and more, and Chapter 31 presents the complete vector in the context of other standard-library facilities.

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

I use vector and other standard-library components as examples
- to illustrate language features and design techniques
- to help you learn and use the standard-library components.

Don't re-invent standard-library components, such as vector and string, use them.

We use . (dot) to access struct members through a name (and through a reference) and -> to access struct members through a pointer. For example:

```
void f(Vector v, Vector& rv, Vector* pv)
{
    int i1 = v.sz;
    int i2 = rv.sz;
    int i3 = (*pv).sz;      // * means "contents of" (§2.2.5)
    int i4 = pv->sz;
    int i5 = (&v)->sz;      // & means "address of" (§2.2.5)
}
```

2.3.2 Classes [tour1.class]

Having the data specified separately from the operations on it has advantages, such as the ability to use the data in arbitrary ways. However, a tighter connection between the representation and the operations is needed for a user-defined type to have all the properties expected of a "real type." In particular, we often want to keep the representation inaccessible to users, so as to ease use, guarantee consistent use of the data, and allow us to later improve the representation. To do that we have to distinguish between the interface to a type (to be used by all) and its implementation (which has access to the otherwise inaccessible data). The language mechanism for that is called a *class*. A class is defined to have a set of *members*, which can be data, function, or type members. The interface is defined by the public members of a class and private members are accessible only through that interface. For example:

```
class Vector {
public:
    Vector(int s) :elem{new double[s]}, sz{s} { } // construct a Vector
    double& operator[](int i) { return elem[i]; }   // element access: subscripting
    int size() { return sz; }
private:
    double* elem;   // pointer to the elements
    int sz;         // the number of elements
};
```

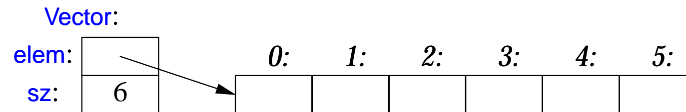
Given that, we can define a variable of our new type Vector:

```
Vector v(6);     // a Vector with six elements
```

We can illustrate a Vector object graphically:

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.



Basically, the Vector object is a "handle" containing a pointer to the elements (elem) plus the number of elements (sz). The number of elements (6 in the example) can vary from Vector object to Vector object, and a Vector object can have a different number of elements at different times. However, the Vector object itself is always the same size. This is the basic technique for handling varying amounts of information in C++: a fixed-sized handle referring to a variable amount of data "elsewhere" (e.g., on the free store allocated by new; §11.2). How to design and use such objects is the main topic of Chapter 3.

Here, the representation of a Vector (the members elem and sz) is accessible only through the interface provided by the public members: Vector(), operator[](), and size(). For example:

```
double read_and_sum(int s)
{
        Vector v(s);                                   // make a vector of s elements
        for (int i=0; i<v.size(); ++i) cin>>v[i];      // read into elements

        double sum = 0;
        for (int i=0; i<v.size(); ++i) sum+=v[i];      // take the sum of the elements
        return sum;
}
```

A "function" with the same name as its class is called a *constructor*; that is, a function used to construct objects of a class. So, the constructor, Vector(), replaces vector_init() from §2.3.1. Unlike an ordinary function, a constructor is guaranteed to be used to initialize objects of its class. Thus, defining a constructor eliminates the problem of uninitialized variables for a class.

Vector(int) defines how objects of type Vector are constructed. In particular, it states that it needs an integer to do that. That integer is used as the number of elements. The constructor initializes the Vector members using a member initializer list:

```
:elem{new double[s]}, sz{s}
```

That is, we first initialize elem with a pointer to s elements of type double obtained from the free store. Then, we initialize sz to s.

Access to elements are provided by a subscript function, called operator[]. It returns a reference to the appropriate element.

The size() function is supplied to give users the number of elements.

Obviously, error handling is completely missing, but we'll return to that in §2.4.3. Similarly, we did not provide a mechanism to "give back" the array of doubles acquired by new; §3.2.1.2 shows how to use a destructor to elegantly do that.

D R A F T

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

2.3.3 Enumerations [tour1.enum]

In addition to classes, C++ supports a simple form of user-defined type for which we can enumerate the values:

```
enum class Color { red, blue, green };
enum class Traffic_light { green, yellow, red };

Color col = Color::red;
Traffic_light light = Traffic_light::red;
```

Note that enumerators (e.g., red) are in the scope of their enum class, so that they can be used repeatedly in different enum classes without confusion. For example, Color::red is Color's red which is different from Traffic_light::red.

Enumerations are used to represent small sets of integer values. They are used to make code more readable and less error-prone than it would have been had the symbolic (and mnemonic) enumerator names not been used.

An enumeration is a user-defined type so we can define operators for it:

```
Traffic_light& operator++(Traffic_light& t)     // prefix increment: ++
{
        switch (t) {
        case Traffic_light::green:    return t=Traffic_light::yellow;
        case Traffic_light::yellow:   return t=Traffic_light::red;
        case Traffic_light::red:      return t=Traffic_light::green;
        }
}

Traffic_light next = ++light;          // next becomes Traffic_light::green
```

By default, an enum class has only assignment, initialization, and comparisons (e.g., == and <; §2.2.2) defined. The class after the enum specifies that an enumeration is strongly typed and that its enumerators are scoped. Being separate types, enum classes help prevent accidental misuses of constants. In particular, we cannot mix Traffic_light and Color values:

```
Color x = red;                    // error: which red?
Color y = Traffic_light::red;     // error: that red is not a Color
Color z = Color::red;             // OK
```

Similarly, we cannot implicitly mix Color and integer values:

```
int i = Color::red;               // error: Color::red is not an int
Color c = 2;                      // error: 2 is not a Color
```

If you don't want to explicitly qualify enumerator names and want enumerator values to be ints (without the need for an explicit conversion), you can remove the class from enum class to get a "plain enum" (§8.4.2).

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

2.4 Modularity [tour1.module]

A C++ program consists of many separately-developed parts, such as functions (§2.2.1, Chapter 12), user-defined types (§2.3, §3.2, Chapter 16), class hierarchies (§3.2.5, Chapter 20), and templates (§3.4, Chapter 23). The key to managing this is to clearly define the interactions among those parts. The first and most important distinction is between the interface to a part and its implementation. At the language level, C++ represents interfaces by declarations. A *declaration* specifies all that's needed to use a function or a type. For example:

```
double sqrt(double);  // the square root function takes a double and returns a double

class Vector {
public:
        Vector(int s);
        double& operator[](int i);
        int size();
private:
        double* elem;  // elem points to an array of sz doubles
        int sz;
};
```

The key point here is that the function bodies, the function *definitions*, are "elsewhere." For this example, we might like for the representation of Vector to be "elsewhere" also, but we will deal with that later (abstract types; §3.2.3). The definition of sqrt() will look like this:

```
double sqrt(double d)        // definition of sqrt()
{
        // ... algorithm as found in math textbook ...
}
```

For Vector, we need to define all three functions:

```
Vector::Vector(int s)                    // definition of the constructor
        :elem{new double[s]}, sz{s}
{
}

double& Vector::operator[](int i)  // definition of subscripting
{
        return elem[i];
}

int Vector::size()                       // definition of size()
{
        return sz;
}
```

The Vector functions are the ones we define, but the sqrt() is part of the standard library. However, that makes no difference: a library is simply some "other code we happen to be

D R A F T

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

using'' written using same language facilities as we use.

2.4.1 Separate Compilation [tour1.comp]

C++ supports a notion of separate compilation where user code sees only declarations of types and functions used. The definitions of those types and functions are in separate source files and compiled separately. This can be used to organize a program into a set of semi-independent fragments.

Typically, we place the declarations that specify the interface to a module in a file with a name indicating its intended use. For example:

```
// Vector.h:

class Vector {
public:
        Vector(int s);
        double& operator[](int i);
        int size();
private:
        double∗ elem;   // elem points to an array of sz doubles
        int sz;
};
```

This declaration would be placed in a file Vector.h, and users will *include* that file, called a *header file*, like this to access that interface:

```
// user.cpp:

#include "Vector.h"     // get the interface
#include "math.h"       // get the interface including sqrt()

double f(Vector& v)
{
        double sum = 0;
        for (int i=0; i<v.size(); ++i)
                sum+=sqrt(v[i]);
        return sum;
}
```

To help the compiler ensure consistency, the .cpp file providing the implementation of Vector will also include the .h file providing its interface:

```
// Vector.cpp:

#include "Vector.h"     // get the interface

Vector::Vector(int s) :elem{new double[s]}, sz{s} { }
```

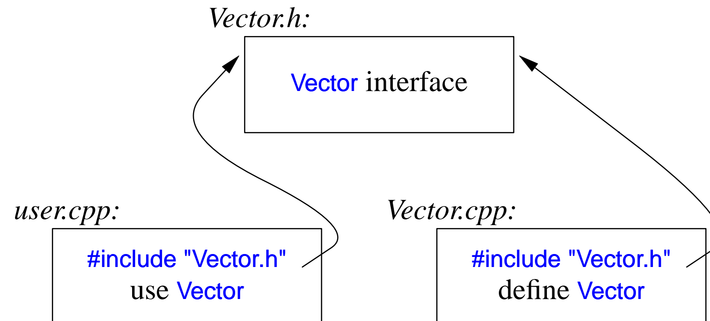
D R A F T

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
double& Vector::operator[](int i) { return elem[i]; }

int Vector::size() { return sz; }
```

The code in user.cpp and Vector.cpp shares the Vector interface information presented in Vector.h, but the two files are otherwise independent and can be separately compiled. Graphically, the program fragments can be represented like this:



Strictly speaking, using separate compilation isn't a language issue; it is an issue of how best to take advantage of a particular language implementation. However, it is of great practical importance. The best approach is to maximize modularity, represent that modularity logically through language features, and then exploit the modularity physically through files for effective separate compilation (Chapter 14, Chapter 15).

2.4.2 Namespaces [tour1.namespace]

In addition to functions (§2.2.1, Chapter 12), classes (Chapter 16), and enumerations (§2.3.3, §8.4), C++ offers *namespaces* (Chapter 14) as a mechanism for expressing that some declarations belong together and that their names shouldn't clash with other names. For example, I might want to experiment with my own complex number type (§3.2.1.1, §18.3, §39.4):

```
namespace My_code {
	class complex { /* ... */ };
	complex sqrt(complex);
	// ...
	int main();
}

int My_code::main()
{
	complex z {1,2};
	auto z2 = sqrt(z);
	std::cout << '{' << z2.real() << ',' << z2.imag() << "}\n";
	// ...
};
```

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
int main()
{
      return My_code::main();
}
```

By putting my code into the namespace My_code, I make sure that my names do not conflict with the standard-library names in namespace std (§4.1.2). The precaution is wise, because the standard library does provide support for complex arithmetic (§3.2.1.1, §39.4).

The simplest way to access a name in another namespace is to qualify it with the namespace name (e.g., std::cout and My_code::main). The "real main()" is defined in the global namespace; that is, not local to a defined namespace, class, function, etc. To gain access to all the names in the standard-library namespace, we can use a using-directive (§14.2.3):

```
using namespace std;
```

Namespaces are primarily used to organize larger program components, such as libraries. They simplify the composition of a program out of separately developed parts.

2.4.3 Error Handling [tour1.error]

Error handling is a large and complex topic with concerns and ramifications that go far beyond language facilities into programming techniques and tools. However, C++ provides a few features to help. The major tool is the type system itself. Instead of painstakingly building up our applications from the built-in types (e.g. char, int, and double) and statements (e.g., if, while, and for), we build more types that are appropriate for our application (e.g. string, map, and regex) and algorithms (e.g., sort(), find_if(), and draw_all()). Such higher level constructs simplify our programming, limit our opportunities for mistakes (e.g., you are unlikely to try to apply a tree traversal to a dialog box) and increase the compiler's chances to catch such errors. The majority of C++ constructs are dedicated to the design and implementation of elegant and efficient abstractions (e.g., user-defined types and algorithms using them). One effect of this modularity and abstraction (in particular, the use of libraries) is that the point where a run-time error can be detected is separated from the point where it can be handled. As programs grow, and especially when libraries are used extensively, standards for handling errors become important.

2.4.3.1 Exceptions [tour1.exception]

Consider again the Vector example. What *ought* to be done when we try to access an element that is out of range for the vector from §2.3.2?

- The writer of Vector doesn't know what the user would like to have done in this case (the writer of Vector typically doesn't even know in which program the vector will be running).
- The user of Vector cannot consistently detect the problem (if the user could, the out-of-range access wouldn't happen in the first place).

The solution is for the Vector implementer to detect the attempted out-of-range access and then tell the user about it. The user can then take appropriate action. For example,

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

Vector::operator[]() can detect an attempted out-of-range access and throw an out_of_range exception:

```
double& Vector::operator[](int i)
{
      if (i<0 || size()<=i) throw out_of_range{"Vector::operator[]"};
      return elem[i];
}
```

The throw transfers control to a handler for exceptions of type out_of_range in some function that directly or indirectly called Vector::operator[](). To do that, the implementation will unwind the function call stack as needed to get back to the context of that caller (§13.5.1). For example:

```
void f(Vector& v)
{
      // ...
      try { // exceptions here are handled by the handler defined below

            v[v.size()] = 7;    // try to access beyond the end of v
      }
      catch (out_of_range) {       // oops: out_of_range error
            // ... handle range error ...
      }
      // ...
}
```

We put code for which we are interested in handling exceptions into a try-block. That attempted assignment to v[v.size()] will fail. Therefore, the catch-clause providing a handler for out_of_range will be entered. The out_of_range type is defined in the standard library and is in fact used by some standard-library container access functions.

Use of the exception-handling mechanisms can make error handling simpler, more systematic, and more readable. See Chapter 13 for further discussion, details, and examples.

2.4.3.2 Invariants [tour1.invariant]

The use of exceptions to signal out-of-range access is an example of a function checking its argument and refusing to operate because a basic assumption, a *precondition*, didn't hold. Had we formally specified Vector's subscript operator, we would have said something like "the index must be in the [0:size()) range," and that was in fact what we tested in our operator[](). Whenever we define a function, we should consider what its preconditions are and if feasible test them (see §12.4, §13.4).

However, operator[]() operates on objects of type Vector and nothing it does makes any sense unless the members of Vector have "reasonable" values. In particular, we did say "*elem points to an array of sz doubles*" but we only said that in a comment. Such a statement of what is assumed to be true for a class is called a class invariant, or simply an *invariant*. It is the job of a constructor to establish the invariant for its class (so that the member functions can rely on it) and for the member functions to make sure that the

D R A F T

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

invariant holds when they exit. Unfortunately, our Vector constructor only partially did its job. It properly initialized the Vector members, but it failed to check that the arguments passed to it made sense. Consider:

```
Vector v(−27);
```

This is likely to cause chaos. Here is a more appropriate definition:

```
Vector::Vector(int s)
{
        if (s<0) throw length_error{};
        elem = new double[s];
        sz = s;
}
```

I use the standard-library exception length_error to report a non-positive number of elements because some standard-library operations use that exception to report problems of this kind. If operator new can't find memory to allocate, it throws a std::bad_alloc. We can now write

```
void test()
{
        try {
                Vector v(−27);
        }
        catch (std::length_error) {
                // handle negative size
        }
        catch (std::bad_alloc) {
                // handle memory exhaustion
        }
}
```

You can define your own classes to be used as exceptions and have them carry arbitrary information from a point where an error is detected to a point where it can be handled (§13.5).

Often, a function has no way of completing its assigned task after an exception is thrown. Then, ''handling'' an exception simply means doing some minimal local cleanup and rethrowing the exception (§13.5.2.1).

The notion of invariants is central to the design of classes and preconditions serve a similar role in the design of functions:

- It helps us to understand precisely what we want.
- It forces us to be specific; that gives us better chance of getting our code correct (after debugging and testing).

More concretely, the notion of invariants underlies C++'s notions of resource management supported by constructors (§2.3.2) and destructors (§3.2.1.2, §5.2). See also §13.4, §16.3.1, and §17.2.

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

2.4.3.3 Static Assertions [tour1.assert]

Exceptions report errors found at run time. If an error can be found at compile time, it is usually preferable to do so. That's what much of the type system and the facilities for specifying the interfaces to user-defined types are for. However, we can also perform simple checks on other properties that are known at compile time and report failures as compiler error messages. For example:

```
static_assert(4<=sizeof(int), "integers are too small"); // check integer size
```

This will write integers are too small if 4<=sizeof(int) does not hold; that is, if an int on this system does not have at least 4 bytes. Such statements of expectations are called *assertions*.

The static_assert mechanism can be used for anything that can be expressed in terms of constant expressions (§2.2.3, §10.4). For example:

```
constexpr double C = 299792.458;          // km/s

void f(double speed)
{
        const double local_max = 160*60*60;           // 160 km/h

        static_assert(speed<C,"can't go that fast");   // error: speed must be a constant
        static_assert(local_max<C,"can't go that fast"); // OK

        // ...
}
```

In general, static_assert(A,S) prints S as a compiler error message if A is not true.

The most important uses of static_cast come when we to make assertions about types used as parameters in generic programming (§5.4.2, §24.3).

2.5 Postscript [tour1.postscript]

The topics covered in this chapter roughly correspond to the contents of Part II (Chapters 5-15). Those are the parts of C++ that underlie all programming techniques and styles supported by C++. Experienced C and C++ programmers, please note that this foundation does not closely correspond to the C or C++98 subsets of C++ (that is, C++11).

2.6 Advice [tour1.advice]

[1] Don't panic! All will become clear in time; §2.1.
[2] You don't have to know every detail of C++ to write good programs; §1.3.1.
[3] Focus on programming techniques, not on language features; §2.1.

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

3

A Tour of C++: Abstraction Mechanisms

Don´t Panic!
– Douglas Adams

- Introduction
- Classes
  Concrete Types; Initializing Containers; Abstract Types; Virtual Functions; Class Hierarchies
- Copy and Move
  Copying Containers; Moving Containers; Preventing Copy and Move
- Templates
  Parameterized Types; Function Templates; Function Objects; Variadic Templates; Aliases
- Advice

3.1 Introduction [tour2.intro]

This chapter aims to give you an idea of C++'s support for abstraction and resource management without going into a lot of detail. This chapter informally presents ways of defining and using new types (*user-defined types*). In particular, it presents the basic properties, implementation techniques, and language facilities used for *concrete classes*, *abstract classes*, and *class hierarchies*. Templates are introduced as a mechanism for parameterizing types and algorithms with (other) types and algorithms. Computations on user-defined and built-in types are represented as functions, sometimes generalized to *template functions* and *function objects*. These are the language facilities supporting the programming styles known as *object-oriented programming* and *generic programming*. The next two chapters follow up by presenting examples of standard-library facilities and their use.

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

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 language 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, skip to the more systematic presentation starting in Chapter 6.

3.2 Classes [tour2.class]

The central language feature of C++ is the *class*. A class is a user-defined type provided to represent a concept in the code of a program. Whenever our design for a program has a useful concept, idea, entity, etc., we try to represent it as a class in the program so that the idea is there in the code, rather than just in our head, in a design document, or in some comments. A program built out of a well chosen set of classes is far easier to understand and get right than one that builds everything directly in terms of the built-in types. In particular, classes are often what libraries offer.

Essentially all language facilities beyond the fundamental types, operators, and statements exist to help define better classes or to use them more conveniently. By "better," I mean more correct, easier to maintain, more efficient, more elegant, easier to use, easier to read, and easier to reason about. Most programming techniques rely on the design and implementation of specific kinds of classes. The needs and tastes of programmers vary immensely. Consequently, the support for classes is extensive. Here, we will just consider the basic support for three important kinds of classes:

- concrete classes (§3.2.1)
- abstract classes (§3.2.3)
- classes in class hierarchies (§3.2.5)

An astounding number of useful classes turn out to be of these three kinds. Even more can be seen as simple variants of these are implemented using combinations of the techniques used for these.

3.2.1 Concrete Types [tour2.concrete]

The basic idea of *concrete classes* is that they behave "just like built-in types." For example, a complex number type and an infinite-precision integer are much like built-in `int`, except of course that they have their own semantics and sets of operations. Similarly, a `vector` and a `string` are much like built-in arrays, except that they are better behaved (§4.2, §4.3.3, §4.4.1).

The defining characteristic of a concrete type is that its representation is part of its definition. That allows implementations to be optimally efficient in time and space. In particular, it allows us to

- place objects of concrete types on the stack, in statically allocated memory, and in other objects.
- refer to objects directly (and not just through pointers or references).

D R A F T

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

- initialize objects immediately and completely (e.g., by using constructors; §2.3.2).
- copy objects (§3.3).

The representation can be private (as it is for Vector; §2.3.2) and accessible only through the member functions, but it is present. Therefore, if the representation changes in any significant way, a user must recompile. This is the price to pay for having concrete types behave exactly like built-in types. For types that don't change often, and where local variables provide much-needed clarity and efficiency, this is acceptable and often ideal. To increase flexibility, a concrete type can keep major parts of its representation on the free store and access them through the part stored in the class object itself. That's the way vector and string are implemented; they can be considered resource handles with carefully crafted interfaces.

3.2.1.1 An Arithmetic Type [tour2.complex]

The "classical user-defined arithmetic type" is complex:

```
class complex {
        double re, im;    // representation: two doubles
public:
        complex(double r, double i) :re{r}, im{i} {}  // construct complex from two scalars
        complex(double r) :re{r}, im{0} {}            // construct complex from one scalar
        complex() :re{0}, im{0} {}                    // default complex: {0,0}

        double real() const { return re; }
        void real(double d) { re=d; }
        double imag() const { return im; }
        void imag(double d) { im=i; }

        complex operator+=(complex z) { return {re+=z.re, im+=z.im}; }  // add to re and im
                                                                        // and return the result
        complex operator−=(complex z) { return {re−=z.re, im−=z.im}; }
        complex operator*=(complex); // defined out-of-class somewhere
        complex operator/=(complex); // defined out-of-class somewhere
};
```

This is a slightly simplified version of the standard library complex (§3.2.1.1, §39.4). The class definition itself contains only the operations requiring access to the representation. Complex has a simple and conventional representation (which for practical reasons has to be compatible with what Fortran provided 50 years ago) and a lot of conventional operators. In addition to the logical demands, complex must also be efficient or it will remain unused. This implies that simple operations must be inlined. That is, simple operations (such as constructors, +, and imag()) must be implemented without function calls in the generated machine code. Functions defined in a class are inlined by default. An industrial strength complex (like the standard library one) would be carefully implemented to do appropriate inlining.

A constructor that can be invoked without an argument is called a *default constructor*. Thus, complex() is complex's default constructor. By defining a default constructor you

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

eliminate the possibility of unintialized variables of that type.

Note the const specifier on the functions returning the real and imaginary parts. There, const is used to indicate that a function may not modify the object for which it was invoked.

Other useful operations can be defined separately from the class definition:

```
complex operator+(complex a, complex b) { return a+=b; }
complex operator−(complex a, complex b) { return a−=b; }
complex operator−(complex a) { return {−a.real(), −a.imag()}; } // unary minus
complex operator∗(complex a, complex b) { return a∗=b; }
complex operator/(complex a, complex b) { return a/=b; }

bool operator==(complex a, complex b)    // equal
{
      return a.real()==b.real() && a.imag()==b.imag();
}

bool operator!=(complex a, complex b)    // not equal
{
      return !(a==b);
}

complex sqrt(complex);

// ...
```

Class complex can be used like this:

```
void f(complex z)
{
      complex a {2.3};
      complex b {1/a};
      complex c {a+z∗complex{1,2.3}};
      // ...
      if (c != b) c = −(b/a)+2∗b;
}
```

The compiler converts operators involving complex numbers into appropriate function calls. For example, c!=b means operator!=(c,b) and 1/a means operator/(complex{1},a).

User-defined operators (“overloaded operators”) should be used cautiously and conventionally. The syntax is fixed by the language, so you can’t define a unary /. Also, it is not possible to change the meaning of an operator for built-in types, so you can’t re-define + to subtract ints.

3.2.1.2 A Container [tour2.container]

A *container* is an object holding a collection of elements, so we call a type like Vector a container because it is the type of container objects. As defined in §2.3.2, Vector isn’t an unreasonable container of doubles: it is simple to understand, establishes a useful invariant (§2.4.3.2), provides range-checked access (§2.4.3.1), and provides size() to allow us to

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

iterate over its elements. However, it does have a fatal flaw: it allocates elements using new, but never deallocates them. That's not a good idea because although C++ defines an interface for a garbage collector (§34.8), it is not guaranteed that one is available or will run to make unused memory available for new objects. In some environments you can't use a collector and sometimes you prefer more detailed control of construction and destruction (§13.6.4) for logical or performance reasons. We need a mechanism to ensure that the memory allocated by the constructor is deallocated; that mechanism is a *destructor*:

```
class Vector {
private:
      double* elem;   // elem points to an array of sz doubles
      int sz;
public:
      Vector(int s) :elem{new double[s]}, sz{s}      // constructor: acquire resources
      {
            for (int i=0;i<s; ++i) elem[i]=0;       // initialize elements
      }
      ~Vector() { delete[] elem; }                   // destructor: release resources

      double& operator[](int i);
      int size() const;
};
```

The name of a destructor is the complement operator, ~, followed by the name of the class; it is the complement of a constructor. The constructor allocates some memory on the free store (also called the *heap* or *dynamic store*) using the new operator. The destructor cleans up by freeing that memory using the delete operator. This is all done without intervention by users of Vector. The users simply create and use Vectors much as they would variables of built-in types. For example:

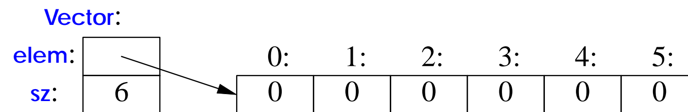
```
void fct(int x)
{
      Vector v(x);
      // use v
      {
            Vector v2(2*x);
            // use v and v2
      } // v2 is destroyed here
      // use v
} // v is destroyed here
```

The Vector obeys the same rules for naming, scope, allocation, lifetime, etc., as does a built-in type, such as int and char. For details on how to control the lifetime of an object, see §6.4.

The constructor/destructor combination is the basis of many elegant techniques and is in particular the basis for most C++ general resource management techniques (§5.2, §13.3). Consider a graphical illustration of a Vector:

D R A F T

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.



The constructor allocates the elements and initializes the Vector members appropriately. The destructor deallocates the elements. This *handle-to-data model* is very commonly used to manage data that can vary in size during the lifetime of an object. The technique of acquiring resources in a constructor and releasing them in the destructor technique, known as *Resource Acquisition Is Initialization* or *RAII*, allows us to eliminate "naked new operations;" that is, to avoid allocations in general code and keep them buried inside the implementation of well-behaved abstractions. Similarly "naked delete operations" should be avoided. Avoiding naked new and naked delete makes code far less error-prone and far easier to keep free of resource leaks (§5.2).

3.2.2 Initializing Containers [tour2.initializer_list]

A container exists to hold elements, so obviously we need convenient ways of getting elements into a container. We can handle that by creating a Vector with an appropriate number of elements and then assign to them, but typically other ways are more elegant. Here, I mention two favorites:

- *initializer-list constructor*: initialize with a list of elements
- push_back(): add a new element at the end (at the back of the sequence)

These can be declared like this:

```
class Vector {
      // ...
      Vector(std::initializer_list<double>);    // initialize with a list
      // ...
      void push_back(double);                   // add element at end increasing the size by one
      // ...
};
```

The push_back() is particularly useful for input of arbitrary numbers of elements. For example:

```
Vector read(istream& is)
{
      Vector v;
      for (double d; is>>d;) v.push_back(d);
      return v;
}
```

The input loop is terminated by an end-of-file or a formatting error. Until that happens, each number read is added to the Vector so that at the end, v's size is the number of elements read. I used a for-statement rather than the more conventional while-statement to keep the scope of d limited to the loop. The implementation of push_back() is discussed in §13.6.4.3. The way to provide Vector with a move constructor, so that returning a

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

potentially huge amount of data from read() is cheap, is explained in §3.3.2.

The std::initializer_list used to define the initializer-list constructor is a standard library type known to the compiler: When we use a { }-list, such as {1,2,3,4}, the compiler will create an object of type initializer_list to give to the program. So, we can write:

```
Vector v1 = {1,2,3,4,5};              // v1 has 5 elements
Vector v2 = { 1.23, 3.45, 6.7, 8 };   // v2 has 4 elements
```

Vector's initializer-list constructor might be defined like this:

```
Vector::Vector(std::initializer_list<double> lst)     // initialize with a list
     :elem{new double[lst.size()]}, sz{lst.size()}
{
     copy(lst.begin(),lst.end(),elem);        // copy from lst into elem
}
```

3.2.3 Abstract Types [tour2.abstract]

Types such as complex and Vector are called *concrete types* because their representation is part of their definition. In that, they resemble built-in types. In contrast, an *abstract type* is a type that completely insulates a user from implementation details. To do that, we must decouple the interface from the representation and give up genuine local variables. Since we don't know anything about the representation of an abstract type (not even its size) we must allocate objects on the free store (§3.2.1.2, §11.2) and access them through references or pointers (§2.2.5, §7.2, §7.7).

First, we define the interface of a class Container which we will design as a more abstract version of our Vector:

```
class Container {
public:
     virtual double& operator[](int) = 0;     // pure virtual function
     virtual int size() const = 0;            // const member function (§3.2.1.1)
     virtual ~Container() {}                  // destructor (§3.2.1.2)
};
```

This class is a pure interface to specific containers defined later. The word virtual means "may be redefined later in a class derived from this one." A class derived from Container provides an implementation for the Container interface. The curious =0 syntax says the function is *pure virtual*; that is, some class derived from Container *must* define the function. Thus, it is not possible to define an object that is just a Container; a Container can only serve as the interface to a class that implements its operator[]() and size() functions. A class with a pure virtual function is called an *abstract class*.

This Container can be used like this:

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
void use(Container& c)
{
      const int sz = c.size();

      for (int i=0; i<sz; ++i)
            cout << c[i] << '\n';
}
```

Note how use() uses the Container interface in complete ignorance of implementation details. It uses size() and [] without any idea of exactly which type provides their implementation. A class that provides the interface to a variety of other classes is often called a *polymorphic type* (§20.3.2).

As is common for abstract classes, Container does not have a constructor. After all, it does not have any data to initialize. On the other hand, Container does have a destructor and that destructor is virtual. Again, that is common for abstract classes because they tend to be manipulated through references or pointers and someone destroying a Container through a pointer has no idea what resources are owned by its implementation; see also §3.2.5.

Not surprisingly, the implementation could consist of everything from the concrete class Vector:

```
class Vector_container : public Container { // Vector_container implements Container
      Vector v;
public:
      Vector_container(int s) : v(s) { }  // Vector of s elements
      ~Vector_container() {}

      double& operator[](int i) { return v[i]; }
      int size() const { return v.size(); }
};
```

The ":public" can be read as "is derived from" or "is a subtype of." Class Vector_container is said to be *derived* from class Container, and class Container is said to be a *base* of class Vector_container. An alternative terminology calls Vector_container and Container *subclass* and *superclass*, respectively. The derived class is said to inherit members from its base class, so the use of base and derived classes is commonly referred to as *inheritance*.

The members operator[]() and size() are said to *override* the corresponding members in the base class Container (§20.3.2). The destructor (~Vector_container()) overrides the base class destructor (~Container()). Note that the member destructor (~Vector()) is implicitly invoked by its class' destructor (~Vector_container()).

For a function like use(Container&) to use a Container in complete ignorance of implementation details, some other function will have to make an object on which it can operate. For example:

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
void g()
{
        Vector_container vc(200);
        // fill vc
        use(vc);
}
```

Since use() doesn't know about Vector_containers but only knows the Container interface, it will work just as well for a different implementation of a Container. For example:

```
class List_container : public Container {      // List_container implements Container
        std::list<double> ld;           // (standard library) list of doubles (§4.4.2)
public:
        List_container() { }            // empty List
        List_container(initializer_list<double> il) : ld{il} { }
        ~List_container() {}

        double& operator[](int i);
        int size() const { return ld.size(); }

};

double& List_container::operator[](int i)
{
        for (auto& x : ld) {
                if (i==0) return x;
                --i;
        }
        throw out_of_range("List container");
}
```

Here, the representation is a standard-library list<double>. Usually, I would not implement a container with a subscript operation using a list, because performance of list subscripting is atrocious compared to vector subscripting. However, here I just wanted to show an implementation that is radically different from the usual one.

A function can create a List_container and have use() use it:

```
void h()
{
        List_container lc = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        use(lc);
}
```

The key point is that use(Container&) has no idea if its argument is a Vector_container, a List_container, or some other kind of container; it doesn't need to know. It can use any kind of Container. It knows only the interface defined by Container. Consequently, use(Container&) needn't be recompiled if the implementation of List_container changes or a brand new class derived from Container is used.

The flip side of this flexibility is that objects must be manipulated through pointers or references (§3.3).

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

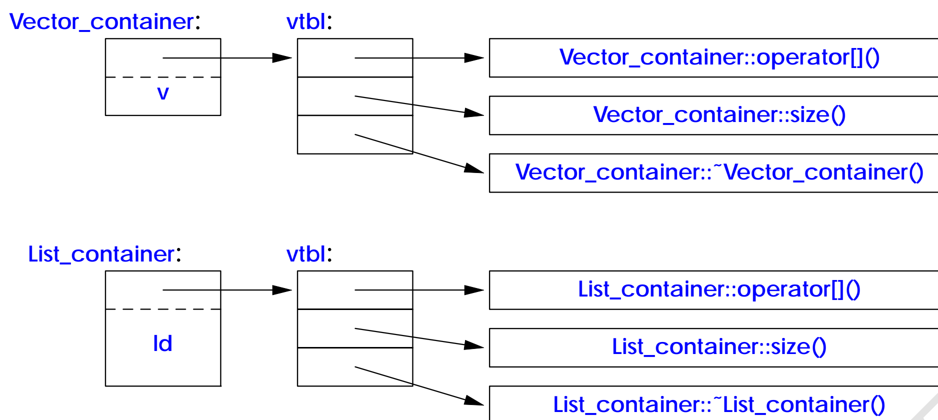
3.2.4 Virtual Functions [tour2.virtual]

Consider again the use of Container:

```
void use(Container& c)
{
      const int sz = c.size();

      for (int i=0; i<sz; ++i)
            cout << c[i] << '\n';
}
```

How is the call c[i] in use() resolved to the right operator[]()? When use() is called from h(), List_container::operator[]() must be called. When use() is called from g(), Vector_container::operator[]() must be called. To achieve this resolution, a Container object must contain information to allow it to select the right function to call at run-time. The usual implementation technique is for the compiler to convert the name of a virtual function into an index into a table of pointers to functions. That table is usually called *the virtual function table* or simply, the vtbl. Each class with virtual functions has its own vtbl identifying its virtual functions. This can be represented graphically like this:



The functions in the vtbl allow the object to be used correctly even when the size of the object and the layout of its data are unknown to the caller. The implementation of the caller needs only to know the location of the pointer to the vtbl in a Container and the index used for each virtual function. This virtual call mechanism can be made almost as efficient as the "normal function call" mechanism (within 25%). Its space overhead is one pointer in each object of a class with virtual functions plus one vtbl for each such class.

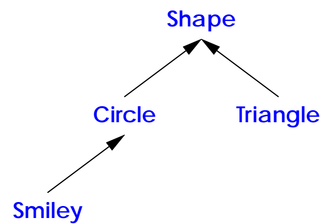
3.2.5 Class Hierarchies [tour2.hier]

The Container example is a very simple example of a class hierarchy. A *class hierarchy* is a set of classes ordered in a lattice created by derivation (e.g. : public). We use class hierarchies to represent concepts that have hierarchical relationships, such as "a fire engine is

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

a kind of a truck which is a kind of a vehicle'' and ''a smiley face is a kind of a circle which is a kind of a shape.'' Huge hierarchies, with hundreds of classes, that are both deep and wide are common. As a semi-realistic classic example, let's consider shapes on a screen:



The arrows represent inheritance relationships. For example, class Circle is derived from class Shape. To represent that simple diagram in code, we must first specify a class that defines the general properties of all shapes:

```
class Shape {
public:
        virtual Point center() const =0;    // pure virtual
        virtual void move(Point to) =0;

        virtual void draw() const = 0;      // draw on current "Canvas"
        virtual void rotate(int angle) = 0;

        virtual ˜Shape() {}
        // ...
};
```

Naturally, this interface is an abstract class: as far as representation is concerned, *nothing* (except the location of the pointer to the vtbl) is common for every Shape. Given this definition, we can write general functions manipulating vectors of pointers to shapes:

```
void rotate_all(vector<Shape*>& v, int angle) // rotate v's elements by angle degrees
{
        for (auto p : v)
                p->rotate(angle);
}
```

To define a particular shape, we must say that it is a Shape and specify its particular properties (including its virtual functions):

D R A F T

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
class Circle : public Shape {
private:
     Point x;     // center
     int r;       // radius
public:
     Circle(Point p, int rr);          // constructor

     Point center() const { return x; }
     void move(Point to) { x=to; }

     void draw() const;
     void rotate(int) {}               // nice simple algorithm
};
```

So far, the Shape and Circle example provides nothing new compared to the Container and Vector_container example, but we can build further:

```
class Smiley : public Circle {       // use the circle as base for a face
private:
     vector<Shape*> eyes;            // usually two eyes
     Shape* mouth;
public:
     Smiley(Point p, int r) : Circle{p,r}, mouth{nullptr} { }
     // ...
     ~Smiley()
     {
          delete mouth;
          for(auto p : eyes) delete p;
     }

     void move(Point to);

     void draw() const;
     void rotate(int);

     void add_eye(Shape* s) { eyes.push_back(s); }
     void set_mouth(Shape* s);
     virtual void wink(int i);       // wink eye number i
};
```

The push_back() member function adds its argument to the vector (here, eyes), increasing that vector's size by one.

We can now define Smiley::draw() using calls to Smiley's base and member draw()s:

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
void Smiley::draw()
{
      Circle::draw();
      for (auto p : eyes) p−>draw();
      mouth−>draw();
}
```

Note the way that Smiley keeps its eyes in a standard library vector and deletes them in its destructor. Shape's destructor is virtual and Smiley's destructor overrides it. A virtual destructor is essential for an abstract class because an object of a derived class may be deleted through a pointer to a base class. Then, the virtual function call mechanism ensures that the proper destructor is called. That destructor then implicitly invokes the destructors of its bases and members.

In this simplified example, it is the programmer's task to place the eyes and mouth appropriately within the circle representing the face.

We can add data members, operations, or both as we define a new class by derivation. This gives great flexibility with corresponding opportunities for confusion and poor design. See Chapter 21. A class hierarchy offers two kinds of benefits:

- *Interface inheritance*: An object of a derived class can be used wherever an object of a base class is required. That is, the base class acts as an interface for the derived class. The Container and Shape classes are examples. Such classes are often abstract classes.
- *Implementation inheritance*: A base class provides functions or data that simplifies the implementation of derived classes. Smiley's use of Circle's constructor and of Circle::draw() are examples. Such base classes often have data members and constructors.

Concrete classes – especially classes with small representations – are much like built-in types: we define them as local variables, access them using their names, copy them around, etc. Classes in class hierarchies are different: we tend to allocate them on the free store using new and we access them through pointers or references. For example, consider a function that reads data describing shapes from an input stream and constructs the appropriate Shape objects:

```
enum class Kind { circle, triangle, smiley };

Shape∗ read_shape(istream& is)      // read shape descriptions from input stream is
{
      // read shape header from is and find its Kind k

      switch (k) {
      case Kind::circle:
            // read circle data {Point,int} into p and r
            return new Circle{p,r};
      case Kind::triangle:
            // read triangle data {Point,Point,Point} into p1, p2, and p3
            return new Triangle{p1,p2,p3};
```

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
        case Kind::smiley:
                // read smiley data {Point,int,Shape,Shape,Shape} into p, r, e1 ,e2, and m
                Smiley* ps = new Smiley{p,r};
                ps->add_eye(e1);
                ps->add_eye(e2);
                ps->set_mouth(m);
                return ps;
        }
}
```

A program may use that shape reader like this:

```
void user()
{
        std::vector<Shape*> v;
        while (cin)
                v.push_back(read_shape(cin));
        draw_all(v);                    // call draw() for each element
        rotate_all(v,45);               // call rorate(45) for each element
        for (auto p : v) delete p;      // remember to delete elements
}
```

Obviously, the example is simplified – especially with respect to error handling – but it vividly illustrates that user() has absolutely no idea of which kinds of shapes it manipulates. The user() code can be compiled once and later used for new Shapes added to the program. Note that there are no pointers to the shapes outside user(), so user is responsible for deallocating them. This is done with the delete operator and relies critically on Shape's virtual destructor. Because that destructor is virtual, delete invokes the destructor for the most derived class. This is crucial because a derived class may have acquired all kinds of resources (such as file handles, locks, and output streams) that need to be released. In this case, a Smiley deletes its eyes and mouth objects.

Experienced programmers will notice that I left open two obvious opportunities for mistakes:

- A user might fail to place the pointer returned by read_shape into a container and also forget to delete it.
- The owner of the container of Shape pointers might forget to delete the objects pointed to.

In that sense, functions returning a pointer to an object allocated on the free store are dangerous. One solution to both problems is to return a standard-library unique_ptr (§5.2.1) rather than a "naked pointer" and store unique_ptrs in the container:

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
unique_ptr<Shape> read_shape(istream& is)     // read shape descriptions from input stream is
{
        // read shape header from is and find its Kind k

        switch (k) {
        case Kind::circle:
                // read circle data {Point,int} into p and r
                return unique_ptr<Shape>{new Circle{p,r}}; // §5.2.1
        // ...
}

void user()
{
        std::vector<unique_ptr<Shape>> v;
        while (cin)
                v.push_back(read_shape(cin));
        draw_all(v);                 // call draw() for each element
        rotate_all(v,45);            // call rorate(45) for each element
} // all Shapes implicitly destroyed
```

Now the object is owned by the unique_ptr which will delete it when needed.

For the unique_ptr version of user() to work, we need versions of draw_all() and rotate_all() that accept vector<unique_ptr<Shape>>s. Writing many such _all() functions could become tedious, so §3.4.3 shows an alternative.

3.3 Copy and Move [tour2.copy]

By default, objects can be copied. This is true for objects of user-defined types as well as for built-in types. The default meaning of copy is memberwise copy: copy each member. For example, using complex from (§3.2.1.1):

```
complex z1 {1,2};
complex z2 {z1};      // copy initialization
complex z3;
z3 = z2;              // copy assignment
```

Now z1, z2, and z3 each have the same value because both the assignment and the initialization copied both members.

When we design a class, we must always consider if and how an object might be copied. For simple concrete types, memberwise copy is often exactly the right semantics for copy. For some sophisticated concrete types, such as Vector, memberwise copy is not the right semantics for copy and for abstract types it almost never is.

3.3.1 Copying Containers [tour2.copy.container]

When a class is a *resource handle*; that is, it is responsible for an object accessed through a pointer, the default memberwise copy is typically a disaster. Memberwise copy would violate the resource handle's invariant (§2.4.3.2). For example, the default copy would

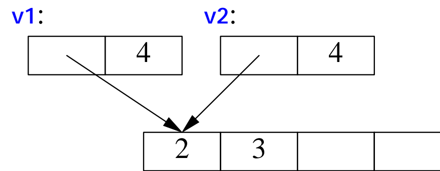
DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

leave a copy of a Vector referring to the same elements as the original:

```
Vector v1(4);
Vector v2 = v1;
v1[0] = 2;          // v2[0] is now also 2!
v2[1] = 3;          // v1[1] is now also 3!
```

Graphically:



Fortunately, the fact that Vector has a destructor is a strong hint that the default (memberwise) copy semantics is wrong and the compiler should at least warn against this example (§17.6). We need to define a better copy semantics. Copying is defined by two functions: a *copy constructor* and a *copy assignment*:

```
class Vector {
private:
        // elem points to an array of sz doubles
        double* elem;
        int sz;
public:
        Vector(int s);                      // constructor: establish invariant, acquire resources
        ~Vector() { delete[] elem; }        // destructor: release resources

        Vector(const Vector& a);                // copy constructor
        Vector& operator=(const Vector& a);     // copy assignment

        double& operator[](int i);
        const double& operator[](int i) const;

        int size() const;
};
```

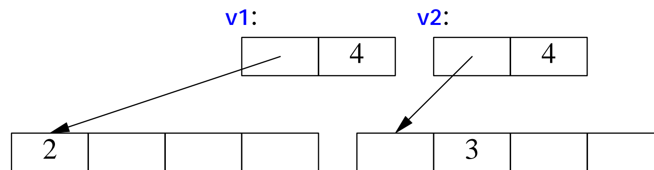
A suitable definition of Vector copy for a container simply copies the elements:

```
Vector::Vector(const Vector& a) // copy constructor
        :sz(a.sz)
{
        elem = new double[sz];
        for (int i=0; i<sz; ++i)
                elem[i] = a.elem[i];
}
```

The result of the v2=v1 example can now be presented as:

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.



Of course, we need a copy assignment in addition to the copy constructor:

```
Vector& Vector::operator=(const Vector& a)          // copy assignment
{
        double* p = new double[a.sz];
        for (int i=0; i<a.sz; ++i)
                p[i] = a.elem[i];
        delete[] elem;           // delete old elements
        elem = p;
        sz = a.sz;
        return *this;
}
```

The name `this` is predefined in every member function and points to the object for which the member function is called.

A copy constructor and a copy assignment for a class `X` are typically declared to take an argument of type `const X&`.

3.3.2 Moving Containers [tour2.copy.move]

We can control copying by defining a copy constructor and a copy assignment, but copying can be costly for large containers. Consider:

```
Vector operator+(const Vector& a, const Vector& b)
{
        if (a.size()!=b.size())
                throw Vector_size_mismatch{};
        Vector res(a.size());
        for (int i=0; i<a.size(); ++i)
                res[i]=a[i]+b[i];
        return res;
}
```

Returning from a `+` involves copying the result out of the local variable `res` and into some place where the caller can access it. We might use this `+` like this:

DRAFT

```
void f(const Vector& x, const Vector& y, const Vector& z)
{
	Vector r;
	// ...
	r = x+y+z;
	// ...
}
```

That would be copying a Vector at least twice (one for each use of the + operator). If a Vector is large, say 10000 doubles, that could be embarrassing. The most embarrassing part is that res is never used again after the copy. We didn't really want a copy, we just wanted to get the result out of a function: we wanted to *move* a Vector rather than to *copy* it. Fortunately, we can state that intent:

```
class Vector {
private:
	// elem points to an array of sz doubles
	double* elem;
	int sz;
public:
	// ...

	Vector(const Vector& a);               // copy constructor
	Vector& operator=(const Vector& a);    // copy assignment

	Vector(Vector&& a);                    // move constructor
	Vector& operator=(Vector&& a);         // move assignment

	// ...
};
```

Given that, the compiler will choose the *move constructor* to implement the transfer of the return value out of the function. This means that the r=x+y+z will involve no copying of Vectors. Instead, Vectors are just moved.

As is typical, Vector's move constructor is trivial to define:

```
Vector::Vector(Vector&& a)
{
	elem = a.elem;          // "grab the elements" from a
	sz = a.sz;
	a.elem = nullptr;              // now a has no elements
	a.sz = 0;
}
```

The && means "rvalue reference" and is a reference to which we can bind an rvalue (§6.4.1). The word "rvalue" is intended to complement "lvalue," which roughly means "something that can appear on the left hand of an assignment." So an rvalue is – to a first approximation – a value that you can't assign to, such as an integer returned by a function call, and an rvalue reference is a reference to something that nobody else can assign to. Note that a move constructor does *not* take a const argument: after all, a move constructor

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

is supposed to remove the value from its argument. A *move assignment* is defined similarly.

A move operation is applied when an rvalue reference is used as an initializer or as the rght-hand side of an assignment.

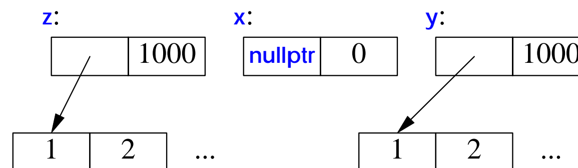
After a move, an object should be in a state that allows a destructor to be run. Typically, we should also allow assignment to a moved-from object (§17.5, §17.6.2).

In cases where the programmer knows that a value will not be used again, but the compiler can't be expected to be smart enough to figure that out, the programmer can be specific:

```
Vector f()
{
        Vector x(1000);
        Vector y(1000);
        Vector z(1000);
        // ...
        z = x;                  // we get a copy
        y = std::move(x);       // we get a move
        // ...
        return z;               // we get a move
};
```

The standard-library function move() returns an rvalue reference to its argument.

Just before the return we have:



By the time z is destroyed, it too has been moved from (by the return) so that, like x, it holds no elements.

3.3.3 Resource Management [tour2.copy.resource]

By defining constructors, copy operations, move operations, and a destructors, a programmer can provide complete control of the lifetime of a contained resource (such as the elements of a container). In particular, a move constructor allows an object to move simply and cheaply from one scope to another. That way, we can move objects that we cannot or would not want to copy out of a scope. Consider a standard-library thread representing a concurrent activity (§5.3.1) and a Vector of a million doubles. We can't copy the former and don't want to copy the latter.

D R A F T

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
std::vector<thread> my_threads;

Vector init()
{
    thread t {heartbeat};                    // run heartbeat concurrently (on its own thread)
    my_threads.push_back(move(t));           // move t into my_threads

    Vector<double> vec;
    // ... fill vec ...
    return vec;                              // move res out of run()
}

auto v = init();    // start heartbeat and initialize v
```

This makes resource handles, such as Vector and thread an alternative to using pointers in many cases. In fact, the standard-library "smart pointers" such as unique_ptr, are themselves such resource handles (§5.2.1).

I used the standard-library vector because we don't get to parameterize Vector with an element type until §3.4.1.

3.3.4 Preventing Copy and Move [tour2.copy.hier]

Using the default copy or move for a class in a hierarchy is typically a disaster: Given only a pointer to a base, we simply don't know what members the derived class has (§3.3.3), so we can't know how to copy them. So, the best thing to do is usually to *delete* the default copy and move operations; that is, to eliminate to default definitions of those two operations:

```
class Shape {
public:
    Shape(const Shape&) =delete;                // no copy operations
    Shape& operator=(const Shape&) =delete;

    Shape(Shape&&) =delete;                     // no move operations
    Shape& operator=(Shape&&) =delete;

    ~Shape();
    // ...
};
```

Now an attempt to copy a Shape will be caught by the compiler. If you need to copy an object in a class hierarchy, write some kind of clone function (§22.2.4).

In case you forgot to delete a copy or move operation, no harm is done. A move operation is *not* implicitly generated for a class where the user has explicitly declared a destructor. Furthermore, the generation of copy operations are deprecated in this case (§42.2.3). This can be a good reason to explicitly define a destructor even where the compiler would have implicitly provided one (§17.2.3).

D R A F T

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

A base class in a class hierarchy is just one example of an object we wouldn't want to copy. A resource handle generally can't be copied just by copying its members (§5.2, §17.2.2).

3.4 Templates [tour2.generic]

Someone who wants a vector is unlikely always to want a vector of doubles. A vector is a general concept, independent of the notion of a floating-point number. Consequently, the element type of a vector ought to be represented independently. A *template* is a class or a function that we parameterize with a set of types or values. We use templates to represent concepts that are best understood as something very general from which we can generate specific types and functions by specifying arguments, such as the element type double.

3.4.1 Parameterized Types [tour2.containers]

We can generalize our vector-of-doubles type to a vector-of-anything type by making it a template and replacing the specific type double with a parameter. For example:

```
template<typename T>
class Vector {
private:
        T* elem;    // elem points to an array of sz elements of type T
        int sz;
public:
        Vector(int s);                  // constructor: establish invariant, acquire resources
        ~Vector() { delete[] elem; }    // destructor: release resources

        // copy and move operations

        T& operator[](int i);
        const T& operator[](int i) const;
        int size() const { return sz; }
};
```

The template<typename T> prefix makes T a parameter of the declaration it prefixes. It is C++'s version of the mathematical "for all T" or more precisely "for all types T."

The member functions might be defined similarly:

```
template<typename T>
Vector<T>::Vector(int s)
{
        if (s<0) throw Negative_size{};
        elem = new T[s];
        sz = s;
}
```

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
template<typename T>
const T& Vector<T>::operator[](int i) const
{
        if (i<0 || size()<=i) throw out_of_range{"Vector::operator[]"};
        return elem[i];
}
```

Given these definitions, we can define Vectors like this:

```
Vector<char> vc(200);          // vector of 200 characters
Vector<string> vs(17);         // vector of 17 integers
Vector<list<int>> vli(45);     // vector of 45 lists of integers
```

The >> in Vector<list>> terminates the nested template arguments; it is not a misplaced input operator. It is not (as in C++98) necessary to place a space between the two >s.

We can use Vectors like this:

```
void f(const Vector<string>& vs)        // Vector of some strings
{
        for (int i = 0; i<vs.size(); ++i))
                cout << vs[i] << '\n';
}
```

If we also want to use the range-for loop for our Vector, we must define suitable begin() and end():

```
template<typename T>
T* begin(Vector<T>& x)
{
        return &x[0];           // pointer to first element
}

template<typename T>
T* end(Vector<T>& x)
{
        return x.begin()+x.size();   // pointer to one-past-last element
}
```

Given those, we can write:

```
void f2(const Vector<string>& vs)       // Vector of some strings
{
        for (auto s : vs)
                cout << s << '\n';
}
```

Similarly, we can define lists, vectors, maps (that is, associative arrays), etc., as templates (§4.4, §23.2, Chapter 31).

Templates are a compile-time mechanism, so their use incurs no run-time overhead compared to "hand-written code" (§23.2.2).

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

3.4.2 Function Templates [tour2.algorithms]

Templates have many more uses than simply parameterizing a container with an element type. In particular, they are extensively used for parameterization of both types and algorithms in the standard library (§3.4.1, §3.4.2). For example, we can write a function that calculates the sum of the element values of any container like this:

```
template<typename Container, typename Value>
Value sum(const Container& c, Value v)
{
    for (auto x : c) v+=x;
    return v;
}
```

The Value template argument and the function argument v are there to allow the caller to specify the type and initial value of the accumulator (the variable in which to accumulate the sum):

```
void user(Vector<int>& vi, std::list<double>& ld, std::vector<complex<double>>& vc)
{
    int x = sum(vi,0);                       // the sum of a vector (add ints)
    double d = sum(vi,0.0);                  // the sum of a vector (add doubles)
    double dd = sum(ld,0.0);                 // the sum of a list of doubles
    auto z = sum(vc,complex<double>{});      // the sum of a vector of complex<double>
}
```

The point of adding ints in a double would be to gracefully handle a number larger than the largest int. Note how the types of the template arguments for sum<T,V> are deduced from the function arguments.

This sum() is a simplified version of the standard-library accumulate() (§39.6).

3.4.3 Function Objects [tour2.functionobjects]

One particularly useful kind of template is the *function object* (sometimes called a *functor*), which is used to define objects that can be called like functions. For example:

```
template<typename T>
class Less_than {
    const T& val;    // value to compare against
public:
    Less_than(const T& v) :val(v) { }
    bool operator()(const T& x) const { return x<val; }     // call operator
};
```

The function called operator() implements the ‘‘function call,’’ ‘‘call,’’ or ‘‘application’’ operator ().

We can define named variables of type Less_than for some argument type:

```
Less_than<int> lti {42};              // will compare to 42 (using <)
Less_than<string> lts {"Backus"};     // will compare to "Backus" (using <)
```

We can call such an object, just as we call a function:

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
void fct(int n, const string & s)
{
     bool b1 = lti(n);        // true if n<42
     bool b2 = lts(s);        // true if s<"Backus"
     // ...
}
```

Such function objects are widely used as arguments to algorithms. For example, we can count the occurrences of values for which a predicate returns true:

```
template<typename C, typename P>
int count(const C& c, P pred)
{
     int cnt = 0;
     for (const auto& x : c)
          if (pred(x)) ++cnt;
     return cnt;
}
```

A *predicate* is something that we can invoke to return true or false. For example:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
     cout << "number of values less than " << x
          << ": " << count(vec,Less_than<int>{x})
          << '\n';
     cout << "number of values less than " << s
          << ": " << count(lst,Less_than<string>{s})
          << '\n';
}
```

Here, Less_than<int>{x} constructs an object for which the call operator compares to the int called x; Less_than<string>{s} constructs an object that compares to the string called s. The beauty of these function objects is that they carry the value to be compared against with them. We don't have to write a separate function for each value (and each type) and we don't have to introduce nasty global variables to hold values. Also, for a simple function object like Less_than inlining is simple so that a call of Less_than is far more efficient than an indirect function call. The ability to carry data plus their efficiency makes function objects particularly useful as arguments to algorithms.

Function objects that is used to specify the meaning of key operations of a an general algorithm (such as Less_than for count()) are often referred to a *policy objects*.

We have to define Less_than separately from its use. That could be seen as inconvenient. Consequently, there is a notation for implicitly generating function objects:

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
	cout << "number of values less than " << x
		<< ": " << count(vec,[&](int a){ return a<x; })
		<< '\n';
	cout << "number of values less than " << s
		<< ": " << count(lst,[&](const string& a){ return a<s; })
		<< '\n';
}
```

The notation `[&](int a){ return a<x; }` is called a *lambda expression* (§11.4). It generates a function object exactly like `Less_than<int>`. The `[&]` is a *capture list* specifying that local names used (such as `x`) will be passed by reference. Had we wanted to "capture" only `x`, we could have said so: `[&x]`. Had we wanted to give the generated object a copy of `x`, we could have said so: `[=x]`. Capture nothing is `[]`, capture all local names used by references is `[&]`, and capture all local names used by value is `[=]`.

Using lambdas can be convenient and terse, but also obscure. For non-trivial actions (say, more than a simple expression), I prefer to name the operation so as to more clearly state its purpose and to make it available for use in several places in a program.

In §3.2.5, we noticed the annoyance of having to write many functions to perform operations on elements of `vector`s of pointers and `unique_ptr`s, such as `draw_all()` and `rotate_all()`. Function objects (in particular, lambdas) can help by allowing us to separate the traversal of the container from the specification of what is to be done with each element.

First we need a function that applies an operation to each object pointed to by the elements of a container of pointers:

```
template<class C, class Oper>
void for_all(C& c, Oper op)          // assume that C is a container of pointers
{
	for (auto& x : c) op(*x);    // pass op() a reference to each element pointed to
}
```

Now we can write a version of `user()` from §3.2.5 without writing a set of *_all* functions:

```
void user()
{
	vector<unique_ptr<Shape>> v;
	while (cin)
		v.push_back(read_shape(cin));
	for_all(v,[](Shape& s){ s.draw(); });          // draw_all()
	for_all(v,[](Shape& s){ s.rotate(45); });      // rotate_all(45)
}
```

I pass a reference to `Shape` to a lambda so that the lambdas don't have to care exactly how the objects are stored in the container. In particular, those `for_all()` calls would still work if I changed `v` to a `vector<Shape*>`.

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

3.4.4 Variadic Templates [tour2.variadic]

A template can be defined to accept an arbitrary number of arguments of arbitrary types. Such a template is called a *variadic template*. For example:

```
template<typename T, typename... Tail>
void f(T head, Tail... tail)
{
        g(head);   // do someting to head
        f(tail...);     // try again with tail
}

void f() { } // do nothing
```

The key to implementing a variadic template is to note that when you pass a list of arguments to it, you can separate the first argument from the rest. Here, we do something to the first argument (the `head`) and then recursively call `f()` with the rest of the arguments (the `tail`). The ellipses, `...`, is used to indicate "the rest" of a list. Eventually, of course, the tail will become empty and we need a separate function to deal with that.

We can call this `f()` like this:

```
int main()
{
        cout << "first: ";
        f(1,2.2,"hello");

        cout << "\nsecond: "
        f(0.2,'c',"yuck!",0,1,2);
        cout << "\n";
}
```

This would call `f(1,2.2,"hello")`, which will call `f(2.2,"hello")`, which will call `f("hello")`, which will call `f()`. What might `g()` do? Obviously, in a real program it will do whatever we wanted done to each argument. For example, we could make it write its argument to output:

```
template<typename T>
void g(T x)
{
        cout << x << " ";
}
```

Given that, the output will be:

```
first: 1 2.2 hello
second: 0.2 c yuck! 0 1 2
```

It seems that `f()` is a simple variant of `printf()` printing arbitrary lists or values – implemented in three lines of code plus their surrounding declarations.

The strength of variadic templates (sometimes just called variadics) is that they can accept any arguments you care to give them. The weakness is that the type checking of

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

the interface is a possibly elaborate template program. For details, see §28.6. For examples, see §34.2.4.2 (N-tuples) and Chapter 29 (N-dimensional matrices).

3.4.5 Aliases [tour2.alias]

Surprisingly often, it is useful to introduce a synonym for a type or a template (§6.5). For example, the standard header <cstddef> contains a definition of the alias size_t, maybe:

```
using size_t = unsigned int;
```

The actual type named size_t is implementation dependent, so in another implementation size_t may be an unsigned long. Having the alias size_t allows the programmer to write portable code.

It is very common for a parameterized type to provide an alias for types related to their template arguments. For example:

```
template<typename T>
class Vector {
public:
        using value_type = T;
        // ...
};
```

In fact, every standard library container provides value_type as the name of their value type (§31.3.1). This allows us to write code that will work for every container that follows this convention. For example:

```
template<typename C>
using Element_type = typename C::value_type;

template<typename Container>
void algo(Container& c)
{
        Vector<Element_type<Container>> vec;     // keep results here
        // ...
}
```

The aliasing mechanism can be used to define a new template by binding some or all templates arguments. For example:

```
template<typename Key, typename Value>
class Map {
        // ...
};

template<typename Value>
using String_map = Map<string,Value>;

String_map<int> m;    // m is a Map<string,int>
```

See §23.6.

D R A F T

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

3.5 Advice [tour2.advice]

[1] Express ideas directly in code; §3.2.
[2] Define classes to represent application concepts directly in code; §3.2.
[3] Use concrete classes for simple concepts and performance critical components; §3.2.1.
[4] Avoid ‘‘naked’’ `new` and `delete` operations; §3.2.1.2.
[5] Use resource handles and RAII to manage resources; §3.2.1.2.
[6] Use abstract classes as interfaces when complete separation of interface and implementation is needed; §3.2.3.
[7] Use class hierarchies to represent concepts with an inherent hierarchical structure; §3.2.5.
[8] When designing a class hierarchy, distinguish between implementation inheritance and interface inheritance; §3.2.5.
[9] Control construction, copy, move, and destruction of objects; §3.3.
[10] Use containers, defined as resource handle templates, to hold collections of values of the same type; §3.4.1.
[11] Use function templates to represent general algorithms; §3.4.2.
[12] Use function objects, including lambdas, to represent policies and actions; §3.4.3.
[13] Use type and template aliases to provide a uniform notation for types that may vary among similar types or among implementations; §3.4.5.

DRAFT

The C++ Programming Language, 4th edition ©2012 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

4

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

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

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.

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

[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:

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

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>	istrstream, ostrstream	§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:

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
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).

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

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 iostreams; Chapter 37 is a reasonably complete description of the iostream 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';
}
```

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

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 istreams for input. Like ostreams, istreams 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!
```

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

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:

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
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.

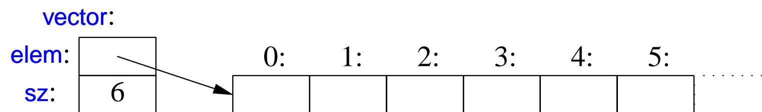
DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

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):

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
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 Entrys from the standard input into phone_book until either the end of input (e.g., the end of a file) is reached or the input operation encounters a format error. The standard-library vector is implemented so that growing a vector by repeated push_back()s is efficient.

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.

D R A F T

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

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:

D R A F T

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

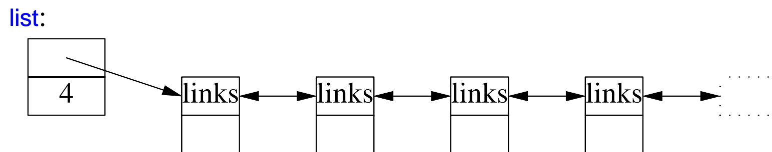
```
int main()
try {
        // your code
}
catch (out_of_range) {
        cerr << "range error\n";
}
catch (...) {
        cerr << "unknown exception thrown\n";
}
```

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

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

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.

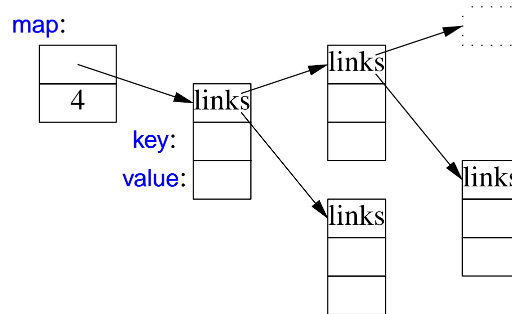
Note that these list examples could be written identically using vector and (surprisingly, unless you understand machine architecture) perform better with a small vector than with a small list. When all we want is a sequence of elements, we have a choice between using a vector and a list. Unless you 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:

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.



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.

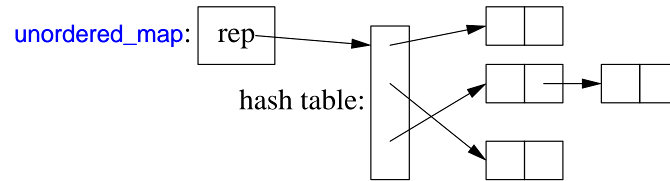
If we wanted to avoid entering invalid numbers into our phone book, we could use find() and insert() instead of [] (§31.4.3.1).

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:

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.



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:

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

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

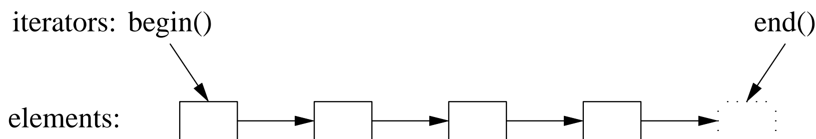
D R A F T

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
bool operator<(const Entry& x, const Entry& y)   // less than
{
      return x.name<y.name;            // order Entrys 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).

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

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;
}
```

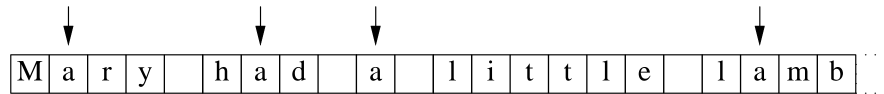
We iterate through the string using a conventional loop, moving the iterator p forward one element at a time using ++ and looking at the elements using the dereference operator *. We could test find_all() like this:

```
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:

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.



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";
```

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

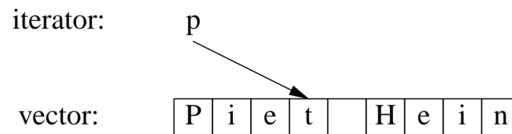
```
        vector<string> vs { "red", "blue", "green", "green", "orange", "green" };
        for (auto p : find_all(vs,"green"))
                if (*p!="green")
                        cerr << "vector bug!\n";

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

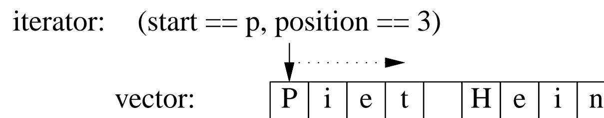
Iterators are used to separate algorithms and containers. An algorithm operates on its data through iterators and knows nothing about the container in which the elements are stored. Conversely, a container knows nothing about the algorithms operating on its elements; all it does is to supply iterators 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:

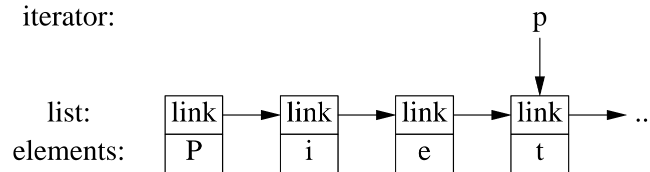


Using such an iterator would allow range checking.

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:

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.



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:

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

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

Typically, istream_iterators and ostream_iterators 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"
```

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
        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

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

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

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());
        }
```

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

```
    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.

DRAFT

The C++ Programming Language, 4th edition ©2013 by Pearson Education, Inc. Reproduced in draft form with the permission of the publisher.

5

A Tour of C++: Concurrency and Utilities

Programming is like sex:
It may give some concrete results,
but that is not why we do it.
– apologies to Richard Feynman

- Introduction
- Resource Management
  unique_ptr and shared_ptr
- Concurrency
  Tasks and threads; Passing Arguments; Returning Results; Sharing Data; Communicating Tasks
- Small Utility Components
  Time; Type Functions; pair and tuple
- Regular Expressions
- Math
  Mathematical Functions and Algorithms; Complex Numbers; Random Numbers; Vector Arithmetic; Numeric Limits
- Advice

5.1 Introduction [tour4.intro]

From an end-user's perspective, the ideal standard library would provide components directly supporting essentially every need. For a given application domain, a huge commercial library can come close to that ideal. However, that is not what the C++ standard library is trying to do. A manageable, universally available, library cannot be everything to everybody. Instead, the C++ standard library aims to provide components that are

D R A F T

useful to most people in most application areas. That is, it aims to serve the intersection of all needs rather than their union. In addition, support for a few widely important application areas, such as mathematical computation and text manipulation, have crept in.

5.2 Resource Management [tour4.resources]

One of the key tasks of any nontrivial program is to manage resources. A resource is something that must be acquired and later (explicitly or implicitly) released. Examples are memory, locks, sockets, thread handles, and file handles. For a long-running program, failing to release a resource in a timely manner (“a leak”) can cause serious performance degradation and possibly even a miserable crash. Even for short programs, a leak can become an embarrassment, say by a resource shortage increasing the run time by orders of magnitude.

The standard library components are designed not to leak resources. To do this, they rely on the basic language support for resource management using constructor/destructor pairs to ensure that a resource doesn’t outlive an object responsible for it. The use of a constructor/destructor pair in Vector to manage the lifetime of its elements is an example (§3.2.1.2) and all standard-library containers are implemented in similar ways. Importantly, this approach interacts correctly with error handling using exceptions. For example, the technique is used for the standard-library lock classes:

```
mutex m;  // used to protect access to shared data
// ...
void f()
{
	lock_guard<mutex> lck {m};	// acquire the mutex m
	// ... manipulate shared data ...
}
```

A thread will not proceed until lck’s constructor has acquired its mutex, m (§5.3.4). The corresponding destructor releases the resource. So, in this example, lock_guard’s destructor releases the mutex when the thread of control leaves f() (through a return, by “falling off the end of the function,” or through an exception throw).

This is an application of the “Resource Acquisition Is Initialization” technique (RAII; §3.2.1.2, §13.3). This technique is fundamental to the idiomatic handling of resources in C++. Containers (such as vector and map), string, and iostream manage their resources (such as file handles and buffers) similarly.

5.2.1 unique_ptr and shared_ptr [tour4.smart]

The examples so far take care of objects defined in a scope, releasing the resources they acquire at the exit from the scope, but what about objects allocated on the free store? In <memory>, the standard library provides two “smart pointers” to help manage objects on the free store:

[1] unique_ptr to represent unique ownership (§34.3.1)

DRAFT

[2] shared_ptr to represent shared ownership (§34.3.2)

The most basic use of one of these ‘‘smart pointers’’ is to prevent memory leaks caused by careless programming:

```
void f(int i, int j)  // X* vs. unique_ptr<X>
{
        X* p = new X;                  // allocate a new X
        unique_ptr<X> sp {new X};// allocate a new X and give its pointer to unique_ptr
        // ...
        if (i<99) throw Z{};           // may throw an exception
        if (j<77) return;              // may return "early"
        p->do_something();             // may throw an exception
        sp->do_something();            // may throw an exception
        // ...
        delete p;                      // destroy *p
}
```

Here, we ‘‘forgot’’ to delete p if i<99 or if j<77. On the other hand, unique_ptr ensures that its object is properly destroyed whichever way we exit f() (by throwing an exception, by executing return, or by ‘‘falling off the end’’).

In this simple case, we could have solved the problem simply by *not* using a pointer and *not* using new:

```
void f(int i, int j)  // use a local variable
{
        X x;
        // ...
}
```

Unfortunately, overuse of new (and of pointers and references) seems to be an increasing problem.

However, when you really need the semantics of pointers, unique_ptr is a very lightweight mechanism with no space or time overhead compared to correct use of a built-in pointer. Its further uses include passing free-store allocated objects in and out of functions:

```
unique_ptr<X> make_X(int i)
        // make an X and immediately give it to a unique_ptr
{
        // check i, etc.
        return unique_ptr<X>{new X{i}};
}
```

A unique_ptr is a handle to an individual object (or an array) in much the same way that a vector is a handle to a sequence of objects. Both control the lifetime of other objects (using RAII) and both rely on move semantics to make return simple and efficient.

The shared_ptr is similar to unique_ptr except that shared_ptrs are copied rather than moved. The shared_ptrs for an object share ownership of an object and that object is destroyed when the last of its shared_ptrs is destroyed. For example:

DRAFT

```
void f(shared_ptr<fstream>);
void g(shared_ptr<fstream>);
void h(shared_ptr<fstream>);

void user(const string& name, ios_base::openmode mode)
{
        shared_ptr<fstream> fp {new fstream(name,mode)};
        if (!*fp) throw No_file{};      // make sure the file was properly opened

        f(fp);
        g(fp);
        h(fp);
        // ...
}
```

Now, the file opened by fp's constructor will be closed by the last function to (explicitly or implicitly) destroy a copy of fp. Note that f(), g(), or h() may spawn a task holding a copy of fp or in some other way store a copy that outlives user(). Thus, shared_ptr provides a form of garbage collection that respects the destructor-based resource management of the memory-managed objects. This is neither cost free nor exorbitantly expensive, but does make the lifetime of the shared object hard to predict. Use shared_ptr only if you actually need shared ownership.

Given unique_ptr and shared_ptr, we can implement a complete "no naked new" policy (§3.2.1.2) for many programs. However, these "smart pointers" are still conceptually pointers and therefore only my second choice for resource management – after containers and other types that manage their resources at a higher conceptual level. In particular, shared_ptrs do not in themselves provide any rules for which of their owners can read and/or write the shared object. Data races (§41.2.4) and other forms of confusion are not addressed simply by eliminating the resource management issues.

Where do we use "smart pointers" (such as unique_ptr) rather than resource handles with operations designed specifically for the resource (such as vector or thread)? Unsurprisingly, the answer is "when we need pointer semantics."

- When we share an object, we need pointers (or references) to refer to the shared object, so shared_ptr becomes the obvious choice (unless there is an obvious single owner).
- When we refer to a polymorphic object, we need a pointer (or a reference) because we don't know the exact type of the object or even its size), so unique_ptr becomes the obvious choice.
- A shared polymorphic object typically requires shared_ptrs.

We do *not* need to use a pointer to return a collection of objects from a function; a container that is a resource handle will do that simply and efficiently (§3.3.2).

DRAFT

5.3 Concurrency [tour4.concurrency]

Concurrency – the execution of several tasks simultaneously – is widely used to improve throughput (by using several processors for a single computation) or to improve responsiveness (by allowing one part of a program to progress while another is waiting for a response). All modern programming languages provide support for this. The support provided by the C++ standard library is a portable and type-safe variant of what has been used in C++ for more than 20 years and is almost universally supported by modern hardware. The standard-library support is primarily aimed at supporting systems-level concurrency rather than directly providing sophisticated higher-level concurrency models; those can be supplied as libraries built using the standard-library facilities.

The standard library directly supports concurrent execution of multiple threads in a single address space. To allow that, C++ provides a suitable memory model (§41.2) and a set of atomic operations (§41.3). However, most users will see concurrency only in terms of the standard library and libraries built on top of that. This section briefly gives examples of the main standard-library concurrency support facilities: threads, mutexes, lock() operations, packaged_tasks, and futures. These features are built directly upon what operating systems offer and do not incur performance penalties compared with those.

5.3.1 Tasks and threads [tour4.thread]

We call a computation that can potentially be executed concurrently with other computations a *task*. A *thread* is the system-level representation of a task in a program. A task to be executed concurrently with other tasks is launched by constructing a std::thread (found in <thread>) with the task as its argument. A task is a function or a function object:

```
void f();                   // function

struct F {                  // function object
      void operator()();    // F's call operator (§3.4.3)
};

void user()
{
      thread t1 {f};        // f() executes in separate thread
      thread t2 {F()};      // F()() executes in separate thread

      t1.join();            // wait for t1
      t2.join();            // wait for t2
}
```

The join()s ensure that we don't exit user() until the threads have completed. To "join" means to "wait for the thread to terminate."

Threads of a program share a single address space. In this, threads differ from processes, which generally do not directly share data. Since threads share an address space, they can communicate through shared objects (§5.3.4). Such communication is typically controlled by locks or other mechanisms to prevent data races (uncontrolled concurrent

DRAFT

access to a variable).

Programming concurrent tasks can be *very* tricky. Consider possible implementations of f and F:

```
void f() { cout << "Hello "; }

struct F {
      void operator()() { cout << "Parallel World!\n"; }
};
```

This is an example of a bad error: Here, f and F() each use the object cout without any form of synchronization. The resulting output would be unpredictable and could vary between different executions of the program because the order of execution of the individual operations in the two tasks is not defined. The program may crash because cout was corrupted or produce ‘‘odd’’ output, such as

```
PaHerallllel o World!
```

When defining tasks of a concurrent program, our aim is to keep tasks completely separate except where they communicate in simple and obvious ways. The simplest way of thinking of a concurrent task is as a function that happens to run concurrently with its caller. For that to work, we just have to pass arguments, get a result back, and make sure that there is no use of shared data in between (no data races).

5.3.2 Passing Arguments [tour4.passing]

Typically, a task needs data to work upon. We can easily pass data (or pointers or references to the data) as arguments. Consider:

```
void f(vector<double>& v);        // function do something with v

struct F {                        // function object: do something with v
      vector<double>& v;
      F(vector<double>& vv) :v{vv} { }
      void operator()();          // application operator; §3.4.3
};

int main()
{
      vector<double> some_vec {1,2,3,4,5,6,7,8,9};
      vector<double> vec2 {10,11,12,13,14};

      thread t1 {f,some_vec};     // f(some_vec) executes in a separate thread
      thread t2 {F{vec2}};        // F(vec2)() executes in a separate thread

      t1.join();
      t2.join();
}
```

Obviously, F{vec2} saves a reference to the argument vector in F. F can now use that array

D R A F T

and hopefully no other task accesses vec2 while F is executing. Passing vec2 by value would eliminate that risk.

The initialization with {f,some_vec} uses a thread variadic template constructor that can accept an arbitrary sequence of arguments (§28.6). The compiler checks that the first argument can be invoked given the following arguments and builds the necessary function object to pass to the thread. Thus, if F::operator()() and f() perform the same algorithm, the handling of the two tasks are roughly equivalent: in both cases, a function object is constructed for the thread to execute.

5.3.3 Returning Results [tour4.results]

In the example in §5.3.2, I pass the arguments by non-const reference. I only do that if I expect the task to modify the value of the data referred to (§7.7). That's a somewhat sneaky, but not uncommon, way of returning a result. A less obscure technique is to pass the input data by const reference and to pass the location of a place to deposit the result as a separate argument:

```
void f(const vector<double>& v, double* res);     // take input from v; place result in *res

class F {
public:
        F(const vector<double>& vv, double* p) :v{vv}, res{p} { }
        void operator()();                       // place result in *res
private:
        const vector<double>& v;                 // source of input
        double* res;                             // target for output
};

int main()
{
        vector<double> some_vec;
        vector<double> vec2;
        // ...

        double res1;
        double res2;

        thread t1 {f,some_vec,&res1};      // f(some_vec,&res1) executes in a separate thread
        thread t2 {F{vec2,&res2}};         // F{vec2,&res2}() executes in a separate thread

        t1.join();
        t2.join();

        cout << res1 << ' ' << res2 << '\n';
}
```

I don't consider returning results through arguments particularly elegant, so I return to this topic in §5.3.5.1.

DRAFT

5.3.4 Sharing Data [tour4.sharing]

Sometimes tasks need to share data. In that case, the access has to be synchronized so that at most one task at a time has access. Experienced programmers will recognize this as a simplification (e.g., there is no problem with many tasks simultaneously reading immutable data), but consider how to ensure that at most one task at a time has access to a given set of objects.

The fundamental element of the solution is a mutex, a ''mutual exclusion object.'' A thread acquires a mutex using a lock() operation:

```
mutex m;  // controlling mutex
int sh;   // shared data

void f()
{
	lock_guard<mutex> lck {m};	// acquire mutex
	sh += 7;			// manipulate shared data
}	// release mutex implicitly
```

The lock_guard's constructor acquires the mutex (through a call m.lock()). If another thread has already acquired the mutex, the thread waits (''blocks'') until the other thread completes its access. Once a thread has completed its access to the shared data, the lock_guard releases the mutex (with a call m.unlock()). The mutual exclusion and locking facilities are found in <mutex>.

The correspondence between the shared data and a mutex is conventional: The programmer simply has to know which mutex is supposed to correspond to which data. Obviously, this is error-prone, and equally obviously we try to make the correspondence clear through various language means. For example:

```
class Record {
public:
	mutex rm;
	// ...
};
```

It doesn't take a genius to guess that for a Record called rec, rec.rm is a mutex that you are supposed to acquire before accessing the other data of rec, though a comment or a better name might have helped a reader.

It is not uncommon to need to simultaneously access several resources to perform some action. This can lead to deadlock. For example, if thread1 acquires mutex1 and then tries to acquire mutex2 while thread2 acquires mutex2 and then tries to acquire mutex1, then neither task will ever proceed further. The standard library offers help in the form of an operation for acquiring several locks simultaneously:

DRAFT

```
void f()
{
        // ...
        lock_guard<mutex> lck1 {m1,defer_lock};   // defer_lock: don't yet try to acquire the mutex
        lock_guard<mutex> lck2 {m2,defer_lock};
        lock_guard<mutex> lck3 {m3,defer_lock};
        // ...
        lock(lck1,lck2,lck3);                       // acquire all three locks
        // ... manipulate shared data ...
} // implicitly release all mutexes
```

This lock() will only proceed after acquiring all its mutex arguments and will never block ("go to sleep") while holding a mutex. The destructors for the individual lock_guards ensure that the mutexes are released when a thread leaves the scope.

Communicating through shared data is pretty low level. In particular, the programmer has to devise ways of knowing what work has and has not been done by various tasks. In that regard, use of shared data is inferior to the notion of call and return. On the other hand, some people are convinced that sharing must be more efficient than copying arguments and returns. That can indeed be so when large amounts of data are involved, but locking and unlocking are relatively expensive operations. On the other hand, modern machines are very good at copying data, especially compact data, such as vector elements. So don't choose shared data for communication because of "efficiency" without thought and preferably not without measurement.

5.3.4.1 Waiting for Events [tour4.condition]

Sometimes, a thread needs to wait for some kind of external event, such as another thread completing a task or a certain amount of time having passed. The simplest "event" is simply time passing. Consider:

```
using namespace std::chrono;    // see §35.2

auto t0 = high_resolution_clock::now();
this_thread::sleep_for(milliseconds{20});
auto t1 = high_resolution_clock::now();
cout << nanoseconds(t1−t0).count() << " nanoseconds passed\n";
```

Note that I didn't even have to launch a thread; by default, this_thread refers to the one and only thread (§42.2.6).

See _tour4.time_ and §35.2 before trying anything more complicated than this with time. The time facilities are found in <chrono>.

The basic support for communicating using external events is provided by condition_variables found in <condition_variable> (§42.3.4). A condition_variable is a mechanism allowing one thread to wait for another. In particular, it allows a thread to wait for some *condition* (often called an *event*) to occur as the result of work done by other threads.

Consider the classical example of two threads communicating by passing messages through a queue. For simplicity, I declare the queue and the mechanism for avoiding race

DRAFT

conditions on that queue global to the producer and consumer:

```
class Message {         // object to be communicated
        // ...
};

queue<Message> mqueue;         // the queue of messages
condition_variable mcond;      // the variable communicating events
mutex mmutex;                  // the locking mechanism
```

The types queue, condition_variable, and mutex are provided by the standard library.

The consumer() reads and processes Messages:

```
void consumer()
{
        while(true) {
                unique_lock<mutex> lck{mmutex};      // acquire mmutex
                mcond.wait(lck);                     // release lck and wait;
                                                     // re-acquire lck upon wakeup
                auto m = mqueue.top();               // get the message
                mqueue.pop();
                lck.unlock();                        // release lck
                // ... process m ...
        }
}
```

Here, I explicitly protect the operations on the queue and on the condition_variable with a unique_lock on the mutex. Waiting on condition_variable releases its lock argument until the wait is over (so that the queue is non-empty) and then reacquires it.

The corresponding producer looks like this:

```
void producer()
{
        while(true) {
                Message m;
                // ... fill the message ...
                unique_lock<mutex> lck {mmutex};     // protect operations
                mqueue.push(m);
                mcond.notify_one();                  // notify
        }                                            // release lock (at end of scope)
}
```

Using condition_variables supports many forms of elegant and efficient sharing, but can be rather tricky (§42.3.4).

5.3.5 Communicating Tasks [tour4.task]

The standard library provides a few facilities to allow programmers to operate at the conceptual level of tasks (work to potentially be done concurrently) rather than directly at the lower level of threads and locks:

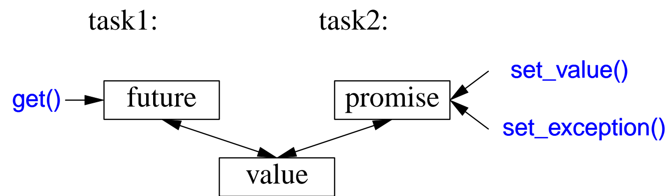
DRAFT

[1] future and promise for returning a value from a task spawned on a separate thread
[2] packaged_task to help launch tasks and connect up the mechanisms for returning a result
[3] async() for launching of a task in a manner very similar to calling a function.

These facilities are found in <future>.

5.3.5.1 future and promise [tour4.future]

The important point about future and promise is that they enable a transfer of a value between two tasks without explicit use of a lock; "the system" implements the transfer efficiently. The basic idea is simple: When a task wants to pass a value to another, it puts the value into a promise. Somehow, the implementation makes that value appear in the corresponding future, from which it can be read (typically by the launcher of the task). We can represent this graphically:



If we have a future<X> called fx, we can get() a value of type X from it:

```
X v = fx.get();    // if necessary, wait for the value to get computed
```

If the value isn't there yet, our thread is blocked until it arrives. If the value couldn't be computed, get() might throw an exception (from the system or transmitted from the task from which we were trying to get() the value).

The main purpose of a promise is to provide simple "put" operations (called set_value() and set_exception()) to match future's get(). The names "future" and "promise" are historical; please don't blame me. They are yet another fertile source of puns.

If you have a promise and need to send a result of type X to a future, you can do one of two things: pass a value or pass an exception. For example:

```
void f(promise<X>& px)    // a task: place the result in px
{
        // ...
        try {
                X res;
                // ... compute a value for res ...
                px.set_value(res);
        }
```

DRAFT

```
        catch (...) {        // oops: couldn't compute res
                // pass the exception to the future's thread:
                px.set_exception(current_exception());
        }
}
```

The current_exception() refers to the caught exception (§30.4.1.2).

To deal with an exception transmitted through a future, the caller of get() must be prepared to catch it somewhere. For example:

```
void g(future<X>& fx)        // a task: get the result from fx
{
        // ...
        try {
                X v = fx.get();    // if necessary, wait for the value to get computed
                // ... use v ...
        }
        catch (...) {        // oops: someone couldn't compute v
                // ... handle error ...
        }
}
```

5.3.5.2 packaged_task [tour4.packaged]

How do we get a future into the task that needs a result and the corresponding promise into the thread that should produce that result? The packaged_task type is provided to simplify setting up tasks connected with futures and promises to be run on threads. A packaged_task provides wrapper code to put the return value or exception from the task into a promise (like the code shown in §5.3.5.1). If you ask it, the packaged_task will give you the corresponding future. For example, we can set up two tasks to each add half of the elements of a vector<double> using the standard-library accumulate() (§3.4.2, §40.6):

```
double accum(double* beg, double * end, double init)
        // compute the sum of [beg:end) starting with the initial value init;
{
        return accumulate(beg,end,init);
}

double comp2(vector<double>& v)
{
        using Task_type = double(double*,double*,double);            // type of task

        packaged_task<Task_type> pt0 {accum};    // package the task (i.e., accum)
        packaged_task<Task_type> pt1 {accum};

        future<double> f0 {pt0.get_future()};    // get hold of pt0's future
        future<double> f1 {pt1.get_future()};    // get hold of pt1's future
```

D-R-A-F-T: *The C++ Programming Language, 4th edition* ©2013 Bjarne Stroustrup.

DRAFT

```
        double∗ first = &v[0];
        thread t1 {move(pt0),first,first+v.size()/2,0};           // start a thread for pt0
        thread t2 {move(pt1),first+v.size()/2,first+v.size(),0};  // start a thread for pt1

        // ...

        return f0.get()+f1.get();          // get the results
}
```

The packaged_task template takes the type of the task as its template argument (here Task_type, an alias for double(double∗,double∗,double)) and the task as its constructor argument (here, accum). The move() operations are needed because a packaged_task cannot be copied.

Please note the absence of explicit mention of locks in this code: we are able to concentrate on tasks to be done, rather than on the mechanisms used to manage their communication. The two tasks will be run on separate threads and thus potentially in parallel.

5.3.5.3 async() [tour4.async]

The line of thinking I have pursued in this chapter is the one I believe to be the simplest yet still among the most powerful: Treat a task as a function that may happen to run concurrently with other tasks. It is far from the only model supported by the C++ standard library, but it serves well for a wide range of needs. More subtle and tricky models, e.g., styles of programming relying on shared memory, can be used as needed.

The standard-library function async() provides a very simple way of executing a task asynchronously:

```
double comp4(vector<double>& v)
        // spawn many tasks if v is large enough
{
        if (v.size()<10000) return accum(v.begin(),v.end(),0.0);

        auto v0 = &v[0];
        auto sz = v.size();

        auto f0 = async(accum,v0,v0+sz/4,0.0);          // first quarter
        auto f1 = async(accum,v0+sz/4,v0+sz/2,0.0);     // second quarter
        auto f2 = async(accum,v0+sz/2,v0+sz∗3/4,0.0);   // third quarter
        auto f3 = async(accum,v0+sz∗3/4,v0+sz,0.0);     // fourth quarter

        return f0.get()+f1.get()+f2.get()+f3.get();     // collect and combine the results
}
```

Basically, async() separates the ‘‘call part’’ of a function call from the ‘‘get the result part,’’ and separates both from the actual execution of the task. Using async(), you don’t have to think about threads and locks. Instead, you think just in terms of tasks that potentially compute their results asynchronously. There is an obvious limitation: Don’t even think of using async() for tasks that share resources needing locking – with async() you don’t even

DRAFT

know how many threads will be used because that's up to async() to decide based on what it knows about the system resources available at the time of a call. For example, async() may check whether any idle cores (processors) are available before deciding how many threads to use.

Please note that async() is not just a mechanism specialized for parallel computation for increased performance. For example, it can also be used to spawn a task for getting information from a user, leaving the "main program" active with something else (§42.4.6).

5.4 Small Utility Components [tour4.utilities]

Not all standard-library components come as part of obviously labeled facilities, such as "containers" or "I/O." This section gives a few examples of small, widely useful components:

- clock and duration for measuring time.
- Type functions, such as iterator_traits and is_arithmetic, for gaining information about types.
- pair and tuple for representing small potentially heterogeneous sets of values.

The point here is that a function or a type need not be complicated or closely tied to a mass of other functions and types to be useful. Such library components mostly act as building blocks for more powerful library facilities, including other components of the standard library.

5.4.1 Time [tour4.clock]

The standard library provides facilities for dealing with time. For example, here is the basic way of timing something:

```
using namespace std::chrono;    // see §35.2

auto t0 = high_resolution_clock::now();
do_work();
auto t1 = high_resolution_clock::now();
cout << duration_cast<milliseconds>(t1−t0).count() << "msec\n";
```

The clock returns a time_point (a point in time). Subtracting two time_points gives a duration (a period of time). Various clocks give their results in various units of time (the clock I used measures nanoseconds), so it is usually a good idea to convert a duration into a known unit. That's what duration_cast does.

The standard-library facilities for dealing with time are found in the subnamespace std::chrono in <chrono> (§35.2).

Don't make statements about "efficiency" of code without first doing time measurements. Guesses about performance are most unreliable.

DRAFT

5.4.2 Type Functions [tour4.typetraits]

A *type function* is a function that is evaluated at compile-time given a type as its argument or returning a type. The standard library provides a variety of type functions to help library implementers and programmers in general to write code that take advantage of aspects of the language, the standard library, and code in general.

For numerical types, numeric_limits from <limits> presents useful information (§5.6.5). For example:

```
constexpr float min = numeric_limits<float>::min();      // smallest positive float (§40.2)
```

Similarly, information about sizes can be extracted by the built-in sizeof operator (§2.2.2). For example:

```
constexpr int szi = sizeof(int);     // the number of bytes in an int
```

Such type functions are part of C++'s mechanisms for compile-time computation that allow tighter type checking and better performance than would otherwise have been possible. Use of such features is often called *metaprogramming* or (when templates are involved) *template metaprogramming* (Chapter 28). Here, I just present two facilities provided by the standard library: iterator_traits (§5.4.2.1) and type predicates (§5.4.2.2).

5.4.2.1 iterator_traits [tour4.iteratortraits]

The standard-library sort() takes a pair of iterators supposed to define a sequence (§4.5). Furthermore, those iterators must offer random access to that sequence, that is, they must be *random-access iterators*. Some containers, such as forward_list, do not offer that. In particular, a forward_list is a singly-linked list so subscripting would be expensive and there is no reasonable way to refer back to a previous element. However, like most containers, forward_list offers *forward iterators* that can be used to traverse the sequence by algorithms and for-statements (§33.1.1).

The standard library provides a mechanism, iterator_traits that allows us to check which kind of iterator is supported. Given that, we can improve the range sort() from §4.5.6 to accept either a vector or a forward_list. For example:

```
void test(vector<string>& v, forward_list<int>& lst)
{
        sort(v);     // sort the vector
        sort(lst);   // sort the singly-linked list
}
```

The techniques needed to make that work are generally useful.

First, I write two helper functions that take an extra argument indicating whether they are to be used for random-access iterators or forward iterators. The version for random-access iterators is trivial:

D R A F T

```
template<typename Ran>          // for random-access iterators
void sort_helper(Ran beg, Ran end, random_access_iterator_tag)
        // we can subscript into [beg:end)
{
        sort(beg,end);   // just sort it

}
```

The version for forward iterators is almost as simple; just copy the list into a vector, sort, and copy back again:

```
template<typename For>          // for forward iterators
void sort_helper(For beg, For end, forward_iterator_tag)
        // we can traverse [beg:end)
{
        vector<decltype(*beg)> v {beg,end};  // initialize a vector from [beg:end)
        sort(v.begin(),v.end());
        copy(v.begin(),v.end(),beg);         // copy the elements back
}
```

The decltype() is a built-in type function that returns the declared type of its argument (§6.3.6.3). Thus, v is a vector<X> where X is the element type of the input sequence.

The real ‘‘type magic’’ is in the selection of helper functions:

```
template<class C)
void sort(C& c)
{
        using Iter = Iterator_type<C>;
        sort_helper(c.begin(),c.end(),Iterator_category<Iter>{});
}
```

Here, I use two type functions: Iterator_type<C> returns the iterator type of C (that is, C::iterator) and then Iterator_category<Iter>{} constructs a ‘‘tag’’ value indicating the kind of iterator provided:

- std::random_access_iterator_tag if C’s iterator supports random access.
- std::forward_iterator_tag if C’s iterator supports forward iteration.

Given that, we can select between the two sorting algorithms at compile time. This technique, called *tag dispatch* is one of several used in the standard library and elsewhere to improve flexibility and performance.

The standard-library support for techniques for using iterators, such as tag dispatch, comes in the form of a simple class template iterator_traits from <iterator> (§33.1.3). This allows simple definitions of the type functions used in sort():

```
template<typename C>
        using Iterator_type = typename C::iterator; // C's iterator type

template<typename Iter>
        using Iterator_category = typename std::iterator_traits<Iter>::iterator_category;    // Iter's category
```

If you don’t want to know what kind of ‘‘compile-time type magic’’ is used to provide the

D R A F T

standard-library features, you are free to ignore facilities such as iterator_traits. But then you can't use the techniques they support to improve your own code.

5.4.2.2 Type Predicates [tour4.typepredicates]

A standard-library type predicate is a simple type function that answers a fundamental question about types. For example:

```
bool b1 = Is_arithmetic<int>();        // yes, int is an arithmetic type
bool b2 = Is_arithmetic<string>();     // no, std::string is not an arithmetic type
```

These predicates are found in <type_traits> and described in §35.4.1. Other examples are is_class, is_pod, is_literal_type, has_virtual_destructor, and is_base_of. They are most useful when we write templates. For example:

```
template<typename Scalar>
class complex {
        Scalar re, im;
public:
        static_assert(Is_arithmetic<Scalar>(), "Sorry, I only support complex of arithmetic types");
        // ...
};
```

To improve readability compared to using the standard library directly, I defined a type function:

```
template<typename T>
constexpr bool Is_arithmetic()
{
        return std::is_arithmetic<T>::value ;
}
```

Older programs use ::value directly instead of (), but I consider that quite ugly and it exposes implementation details.

5.4.3 pair and tuple [tour4.pair]

Often, we need some data that is just data; that is, a collection of values, rather than an object of a class with a well-defined semantics and an invariant for its value (§2.4.3.2, §13.4). In such cases, we could define a simple struct with an appropriate set of appropriately named members. Alternatively, we could let the standard library write the definition for us. For example, the standard-library algorithm equal_range (§32.6.1) returns a pair of iterators specifying a sub-sequence meeting a predicate:

```
template<typename Forward_iterator, typename T, typename Compare>
        pair<Forward_iterator,Forward_iterator>
        equal_range(Forward_iterator first, Forward_iterator last, const T& val, Compare cmp);
```

Given a sorted sequence [first:last), equal_range() will return the pair representing the subsequence that matches the predicate cmp. We can use that to search in a sorted sequence of Records:

D R A F T

D-R-A-F-T: *The C++ Programming Language, 4th edition* ©2013 Bjarne Stroustrup.

```
void f(const vector<Record>& v)
{
        // assume that v is sorted on its "name" field
        auto er = equal_range(v.begin(),v.end(), "Reg",
                                   [](const Record& r1, const Record& r2) { return r1.name==r2.name;}
                                 );
        for (auto p = er.first; p!=er.second; ++p)           // print all equal records
                cout << *p;                                  // assume that << is defined for Record
}
```

The first member of a pair is called first and the second member is called second. This naming is not particularly creative and may look a bit odd at first, but such consistent naming is a boon when we want to write generic code.

The standard-library pair (from <utilty>) is quite frequently used in the standard library and elsewhere. A pair provides operators, such as =, ==, and <, if its elements do. The make_pair() function makes it easy to create a pair without explicitly mentioning its type (§34.2.4.1). For example:

```
void f(vector<string>& v)
{
        auto pp = make_pair(v.begin(),2);      // pp is a pair<vector<string>::iterator,int>
        // ...
}
```

If you need more than two elements (or less), you can use tuple (from <utility>; §34.2.4.2). A tuple is a heterogeneous sequence of elements; for example:

```
tuple<string,int,double> t2("Sild",123, 3.14);      // the type is explicitly specified

auto t = make_tuple(string("Herring"),10, 1.23);  // the type is deduced
                                                   // t is a tuple<string,int,double>

string s = get<0>(t);   // get first element of tuple
int x = get<1>(t);
double d = get<2>(t);
```

The elements of a tuple are numbered (starting with zero), rather than named the way elements of pairs are (first and second). To get compile-time selection of elements, I must unfortunately use the ugly get<1>(t), rather than get(t,1) or t[1] (§28.5.2).

Like pairs, tuples can be assigned and compared if their elements can be.

A pair is common in interfaces because often we want to return more than one value, such as a result and an indicator of the quality of that result. It is less common to need three or more parts to a result, so tuples are more often found in the implementations of generic algorithms.

D R A F T

5.5 Regular Expressions [tour4.regex]

Regular expressions are a powerful tool for text processing. They provide a way to simply and tersely describe patterns in text (e.g., a U.S. ZIP code such as TX 77845, or an ISO-style date, such as 2009–06–07) and to efficiently find such patterns in text. In <regex>, the standard library provides support for regular expressions in the form of the std::regex class and its supporting functions. To give a taste of the style of the regex library, let us define and print a pattern:

```
regex pat (R"(\w{2}\s*\d{5}(–\d{4})?)");       // ZIP code pattern: XXddddd-dddd and variants
cout << "pattern: " << pat << '\n';
```

People who have used regular expressions in just about any language will find \w{2}\s*\d{5}(–\d{4})? familiar. It specifies a pattern starting with two letters \w{2} optionally followed by some space \s* followed by five digits \d{5} and optionally followed by a dash and four digits –\d{4}. If you are not familiar with regular expressions, this may be a good time to learn about them ([Stroustrup 2009], [Maddock,2009], [Friedl,1997]). Regular expressions are summarized in §37.1.1.

To express the pattern, I used a *raw string literal* (§7.3.2.1) starting with a R"(and terminated by)" . This allows backslashes and quotes to be represented in the string without the use of special notation.

The simplest way of using a pattern is to search for it in a stream:

```
int lineno = 0;
for (string line; getline(cin,line);) {          // read into line buffer
     ++lineno;
     smatch matches;                            // matched strings go here
     if (regex_search(line,matches,pat))        // search for pat in line
          cout << lineno << ": " << matches[0] << '\n';
}
```

The regex_search(line,matches,pat) searches the line for anything that matches the regular expression stored in pat and if it finds any matches, it stores them in matches. If no match was found, regex_search(line,matches,pat) returns false. The matches variable is of type smatch. The "s" stands for "sub" and an smatch is a vector of sub-matches. The first element, here matches[0], is the complete match.

For a more complete description see Chapter 37.

5.6 Math [tour4.math]

C++ wasn't designed primarily with numerical computation in mind. However, C++ is heavily used for numerical computation and the standard library reflects that.

5.6.1 Mathematical Functions and Algorithms [tour4.stdmath]

In <cmath>, we find the "usual mathematical functions," such as sqrt(), log(), and sin() for arguments of type float, double, and long double (§40.3). Their complex number versions are found in <complex> (§40.4).

D R A F T

In <numeric> we find a small set of generalized numerical algorithms, such as accumulate(). For example:

```
list<double> lst {1, 2, 3, 4, 5, 6 , 9999.99999};
auto s = accumulate(lst.begin(),lst.end(),0.0);
cout << s << '\n';
```

These algorithms work for every standard-library sequence and can have operations supplied as arguments (§40.6).

5.6.2 Complex Numbers [tour4.complex]

The standard library supports a family of complex number types along the lines of the complex class described in §2.3. To support complex numbers where the scalars are single-precision floating-point numbers (floats), double-precision floating-point numbers (doubles), etc., the standard library complex is a template:

```
template<typename Scalar>
class complex {
public:
        complex(const Scalar& re ={}, const Scalar& im ={});
        // ...
};
```

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

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

The sqrt() and pow() (exponentiation) functions are among the usual mathematical functions defined in <complex>. For more details, see §40.4.

5.6.3 Random Numbers [tour4.random]

Random numbers are useful in many contexts, such as testing, games, simulation, and security. The diversity of application areas is reflected in the wide selection of random number generators provided by the standard library in <random>. A random number generator consists of two parts:

[1] an *engine* that produces a sequence of random or pseudo-random values.
[2] a *distribution* that maps those values into a mathematical distribution in a range.

Examples of distributions are uniform_int_distribution (where all integers produced are equally likely), normal_distribution ("the bell curve"), and exponential_distribution (exponential growth); each for some specified range. For example:

DRAFT

```
using my_engine = default_random_engine;            // type of engine
using my_distribution = uniform_int_distribution<>; // type of distribution

my_engine re {};                          // the default engine
my_distribution one_to_six {1,6};         // distribution that maps to the ints 1..6
auto dice = bind(one_to_six,re);          // make a generator

int x = dice();    // roll the dice: x becomes a value in [1:6]
```

The standard-library function bind() makes a function object that will invoke its first argument (here, one_to_six) given its second argument (here, re) as its argument (§33.5.1). Thus a call dice() is equivalent to a call one_to_six(re).

Thanks to its uncompromising attention to generality and performance one expert has deemed the standard-library random number component "what every random number library wants to be when it grows up." However, it can hardly be deemed "novice friendly." The using statements makes what is being done a bit more obvious. Instead, I could just have written:

```
auto dice = bind(uniform_int_distribution<>{1,6}, default_random_engine{});
```

Which version is the more readable depends entirely on the context and the reader.

For novices (of any background) the fully general interface to the random number library can be a serious obstacle. A simple uniform random number generator is often sufficient to get started. For example:

```
Rand_int rnd {1,10};     // make a random number generator for [1:10]
int x = rnd();           // x is a number in [1:10]
```

So, how could we get that? We have to get something like dice() inside a class Rand_int:

```
class Rand_int {
public:
        Rand_int(int low, int high) :dist{low,high} { }
        int operator()() { return r(); }
private:
        default_random_engine re;
        uniform_int_distribution<> dist;
        auto r = bind(dist,re);
};
```

That definition is still "expert level," but the *use* of Rand_int() is manageable in the first week of a C++ course for novices. For example:

```
int main()
{
        Rand_int rnd {0,9};          // make a uniform random number generator

        vector<int> mn(10);          // make a vector of size 10
        for (int i=0; i!=500; ++i)
                ++mn[rnd()];         // fill mn with the frequencies of numbers [0:9]
```

DRAFT

```
        for (int i = 0; i!=mn.size(); ++i) {   // write out a bar graph
                cout << i << '\t';
                for (int j=0; j!=mn[i]; ++j) cout << '*';
                cout << endl;
        }
}
```

The output is a (reassuringly boring) uniform distribution (with reasonable statistical variation):

```
0       ****************************************************
1       ******************************************
2       *************************************************
3       *************************************************
4       ************************************************
5       *************************************************
6       *****************************************************
7       ************************************************************
8       **************************************************
9       ************************************************
```

There is no standard graphics library for C++, so I use "ASCII graphics." Obviously, there are lots of open source and commercial graphics and GUI libraries for C++, but in this book I'll restrict myself to ISO standard facilities.

For more information about random numbers, see §40.7.

5.6.4 Vector Arithmetic [tour4.valarray]

The vector described in §4.4.1 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 (in <valarray>) a vector-like template, called valarray, that is less general and more amenable to optimization for numerical computation:

```
template<typename T>
class valarray {
        // ...
};
```

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

DRAFT

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

For more details, see §40.5. In particular, valarray offers stride access to help implement multidimensional computations.

5.6.5 Numeric Limits [tour4.limits]

In <limits>, the standard library provides classes that describe the properties of built-in types – such as the maximum exponent of a float or the number of bytes in an int; see §40.2. For example, we can assert that a char is signed:

```
static_assert(numeric_limits<char>::is_signed,"unsigned characters!");
static_assert(100000<numeric_limits<int>::max(),"small ints!");
```

Note that the second assert (only) works because numeric_limits<int>::max() is a constexpr function (§2.2.3, §10.4).

5.7 Advice [tour4.advice]

[1] Use resource handles to manage resources (RAII); §5.2.
[1] Use unique_ptr to refer to objects of polymorphic type;§5.2.1.
[2] Use shared_ptr to refer to shared objects;§5.2.1.
[3] Use type-safe mechanisms for concurrency; §5.3.
[4] Minimize the use of shared data; §5.3.4.
[5] Don't choose shared data for communication because of ‘‘efficiency’’ without thought and preferably not without measurement; §5.3.4.
[6] Think in terms of concurrent tasks, rather than threads; §5.3.5.
[7] A library doesn't have to be large or complicated to be useful; §5.4.
[8] Time your programs before making claims about efficiency; §5.4.1.
[9] You can write code to explicitly depend of properties of types; §5.4.2.
[10] Use regular expressions for simple pattern matching §5.5.
[11] Don't try to do serious numeric computation using only the bare language; use libraries; §5.6.
[12] Properties of numeric types are accessible through numeric_limits; §5.6.5.

D R A F T

blank page

DRAFT

D-R-A-F-T: *The C++ Programming Language, 4th edition* ©2013 Bjarne Stroustrup.