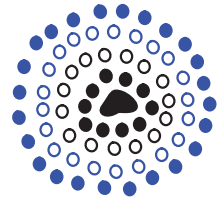
Chapter 1 A C++ Primer



Contents

1.1 Basic C++ Programming Elements

Building data structures and algorithms requires communicating instructions to a computer, and an excellent way to perform such communication is using a high-level computer language, such as C++. C++ evolved from the programming language C, and has, over time, undergone further evolution and development from its original definition. It has incorporated many features that were not part of C, such as symbolic constants, in-line function substitution, reference types, parametric polymorphism through templates, and exceptions (which are discussed later). As a result, C++ has grown to be a complex programming language. Fortunately, we do not need to know every detail of this sophisticated language in order to use it effectively.

In this chapter and the next, we present a quick tour of the C++ programming language and its features. It would be impossible to present a complete presentation of the language in this short space, however. Since we assume that the reader is already familiar with programming with some other language, such as C or Java, our descriptions are short. This chapter presents the language's basic features, and in the following chapter, we concentrate on those features that are important for object-oriented programming.

C++ is a powerful and flexible programming language, which was designed to build upon the constructs of the C programming language. Thus, with minor exceptions, C++ is a superset of the C programming language. C++ shares C's ability to deal efficiently with hardware at the level of bits, bytes, words, addresses, etc. In addition, C++ adds several enhancements over C (which motivates the name "C++"), with the principal enhancement being the object-oriented concept of a ***class***.

A class is a user-defined type that encapsulates many important mechanisms such as guaranteed initialization, implicit type conversion, control of memory management, operator overloading, and polymorphism (which are all important topics that are discussed later in this book). A class also has the ability to hide its underlying data. This allows a class to conceal its implementation details and allows users to conceptualize the class in terms of a well-defined interface. Classes enable programmers to break an application up into small, manageable pieces, or ***objects***. The resulting programs are easier to understand and easier to maintain.

1.1.1 A Simple C++ Program

Like many programming languages, creating and running a C++ program requires several steps. First, we create a C++ source file into which we enter the lines of our program. After we save this file, we then run a program, called a ***compiler***, which

creates a machine-code interpretation of this program. Another program, called a ***linker*** (which is typically invoked automatically by the compiler), includes any required library code functions needed and produces the final machine-executable file. In order to run our program, the user requests that the system execute this file.

Let us consider a very simple program to illustrate some of the language's basic elements. Don't worry if some elements in this example are not fully explained. We discuss them in greater depth later in this chapter. This program inputs two integers, which are stored in the variables x and y. It then computes their sum and stores the result in a variable sum, and finally it outputs this sum. (The line numbers are not part of the program; they are just for our reference.)

```
 1  #include <cstdlib>
 2  #include <iostream>
 3  /* This program inputs two numbers x and y and outputs their sum */
 4  int main( ) {
 5    int x, y;
 6    std::cout << "Please enter two numbers: ";
 7    std::cin >> x >> y;                    // input x and y
 8    int sum = x + y;                       // compute their sum
 9    std::cout << "Their sum is " << sum << std::endl;
10    return EXIT_SUCCESS;                   // terminate successfully
11  }
```

A few things about this C++ program should be fairly obvious. First, comments are indicated with two slashes (//). Each such comment extends to the end of the line. Longer block comments are enclosed between /* and */. Block comments may extend over multiple lines. The quantities manipulated by this program are stored in three integer variables, x, y, and sum. The operators ">>" and "<<" are used for input and output, respectively.

Program Elements

Let us consider the elements of the above program in greater detail. Lines 1 and 2 input the two ***header files***, "cstdlib" and "iostream." Header files are used to provide special declarations and definitions, which are of use to the program. The first provides some standard system definitions, and the second provides definitions needed for input and output.

The initial entry point for C++ programs is the function main. The statement "int main( )" on line 4 declares main to be a function that takes no arguments and returns an integer result. (In general, the main function may be called with the command-line arguments, but we don't discuss this.) The ***function body*** is given within curly braces ({...}), which start on line 4 and end on line 11. The program terminates when the return statement on line 10 is executed.

By convention, the function main returns the value zero to indicate success and returns a nonzero value to indicate failure. The include file cstdlib defines the constant EXIT_SUCCESS to be 0. Thus, the return statement on line 10 returns 0, indicating a successful termination.

The statement on line 6 prints a string using the output operator ("<<"). The statement on line 7 inputs the values of the variables x and y using the input operator (">>"). These variable values could be supplied, for example, by the person running our program. The name std::cout indicates that output is to be sent to the ***standard output stream***. There are two other important I/O streams in C++: ***standard input*** is where input is typically read, and ***standard error*** is where error output is written. These are denoted std::cin and std::cerr, respectively.

The prefix "std::" indicates that these objects are from the system's ***standard library***. We should include this prefix when referring to objects from the standard library. Nonetheless, it is possible to inform the compiler that we wish to use objects from the standard library—and so omit this prefix—by utilizing the "**using**" statement as shown below.

```
#include <iostream>
using namespace std;                          // makes std:: available
// ...
cout << "Please enter two numbers: ";         // (std:: is not needed)
cin >> x >> y;
```

We discuss the **using** statement later in Section 1.1.4. In order to keep our examples short, we often omit the **include** and **using** statements when displaying C++ code. We also use " //..." to indicate that some code has been omitted.

Returning to our simple example C++ program, we note that the statement on line 9 outputs the value of the variable sum, which in this case stores the computed sum of x and y. By default, the output statement does not produce an end of line. The special object std::endl generates a special end-of-line character. Another way to generate an end of line is to output the ***newline character***, '\n'.

If run interactively, that is, with the user inputing values when requested to do so, this program's output would appear as shown below. The user's input is indicated below in blue.

```
Please enter two numbers:  7 35
Their sum is 42
```

1.1.2 Fundamental Types

We continue our exploration of C++ by discussing the language's basic data types and how these types are represented as constants and variables. The fundamental

types are the basic building blocks from which more complex types are constructed. They include the following.

bool	Boolean value, either **true** or **false**
char	character
short	short integer
int	integer
long	long integer
float	single-precision floating-point number
double	double-precision floating-point number

There is also an enumeration, or **enum**, type to represent a set of discrete values. Together, enumerations and the types **bool**, **char**, and **int** are called ***integral types***. Finally, there is a special type **void**, which explicitly indicates the absence of any type information. We now discuss each of these types in greater detail.

Characters

A **char** variable holds a single character. A **char** in C++ is typically 8-bits, but the exact number of bits used for a **char** variable is dependent on the particular implementation. By allowing different implementations to define the meaning of basic types, such as **char**, C++ can tailor its generated code to each machine architecture and so achieve maximum efficiency. This flexibility can be a source of frustration for programmers who want to write machine-independent programs, however.

A ***literal*** is a constant value appearing in a program. Character literals are enclosed in single quotes, as in `'a'`, `'Q'`, and `'+'`. A backslash (\ ) is used to specify a number of special character literals as shown below.

`'\n'`	newline	`'\t'`	tab
`'\b'`	backspace	`'\0'`	null
`'\''`	single quote	`'\"'`	double quote
`'\\'`	backslash		

The null character, `'\0'`, is sometimes used to indicate the end of a string of characters. Every character is associated with an integer code. The function `int(ch)` returns the integer value associated with a character variable `ch`.

Integers

An **int** variable holds an integer. Integers come in three sizes: **short int**, (plain) **int**, and **long int**. The terms "**short**" and "**long**" are synonyms for "**short int**" and "**long int**," respectively. Decimal numbers such as 0, 25, 98765, and -3 are of type **int**. The suffix "l" or "L" can be added to indicate a long integer, as in 123456789L. Octal (base 8) constants are specified by prefixing the number with the zero digit, and hexadecimal (base 16) constants can be specified by prefixing the number with

"0x." For example, the literals 256, 0400, and 0x100 all represent the integer value 256 (in decimal).

When declaring a variable, we have the option of providing a ***definition***, or initial value. If no definition is given, the initial value is unpredictable, so it is important that each variable be assigned a value before being used. Variable names may consist of any combination of letters, digits, or the underscore (_) character, but the first character cannot be a digit. Here are some examples of declarations of integral variables.

```
short n;                                    // n's value is undefined
int   octalNumber = 0400;                   // 400 (base 8) = 256 (base 10)
char  newline_character = '\n';
long  BIGnumber = 314159265L;
short _aSTRANGE__1234_variABlE_NaMe;
```

Although it is legal to start a variable name with an underscore, it is best to avoid this practice, since some C++ compilers use this convention for defining their own internal identifiers.

C++ does not specify the exact number of bits in each type, but a **short** is at least 16 bits, and a **long** is at least 32 bits. In fact, there is no requirement that **long** be strictly longer than **short** (but it cannot be shorter!). Given a type T, the expression **sizeof**(T) returns the size of type T, expressed as some number of multiples of the size of **char**. For example, on typical systems, a **char** is 8 bits long, and an **int** is 32 bits long, and hence **sizeof**(int) is 4.

Enumerations

An enumeration is a user-defined type that can hold any of a set of discrete values. Once defined, enumerations behave much like an integer type. A common use of enumerations is to provide meaningful names to a set of related values. Each element of an enumeration is associated with an integer value. By default, these values count up from 0, but it is also possible to define explicit constant values as shown below.

```
enum Day { SUN, MON, TUE, WED, THU, FRI, SAT };
enum Mood { HAPPY = 3, SAD = 1, ANXIOUS = 4, SLEEPY = 2 };

Day today = THU;            // today may be any of MON ... SAT
Mood myMood = SLEEPY;       // myMood may be HAPPY, ..., SLEEPY
```

Since we did not specify values, SUN would be associated with 0, MON with 1, and so on. As a hint to the reader, we write enumeration names and other constants with all capital letters.

Floating Point

A variable of type **float** holds a single-precision floating-point number, and a variable of type **double** holds a double-precision floating-point number. As it does with integers, C++ leaves undefined the exact number of bits in each of the floating point types. By default, floating point literals, such as 3.14159 and -1234.567 are of type **double**. Scientific or exponential notation may by specified using either "e" or "E" to separate the mantissa from the exponent, as in 3.14E5, which means 3.14×10^5. To force a literal to be a **float**, add the suffix "f" or "F," as in 2.0f or 1.234e-3F.

1.1.3 Pointers, Arrays, and Structures

We next discuss how to combine fundamental types to form more complex ones.

Pointers

Each program variable is stored in the computer's memory at some location, or ***address***. A ***pointer*** is a variable that holds the value of such an address. Given a type T, the type T* denotes a pointer to a variable of type T. For example, **int*** denotes a pointer to an integer.

Two essential operators are used to manipulate pointers. The first returns the address of an object in memory, and the second returns the contents of a given address. In C++ the first task is performed by the ***address-of*** operator, &. For example if x is an integer variable in your program &x is the address of x in memory. Accessing an object's value from its address is called ***dereferencing***. This is done using the * operator. For example, if we were to declare q to be a pointer to an integer (that is, int*) and then set q = &x, we could access x's value with *q. Assigning an integer value to *q effectively changes the value of x.

Consider, for example, the code fragment below. The variable p is declared to be a pointer to a **char**, and is initialized to point to the variable ch. Thus, *p is another way of referring to ch. Observe that when the value of ch is changed, the value of *p changes as well.

```
char ch = 'Q';
char* p = &ch;              // p holds the address of ch
cout << *p;                 // outputs the character 'Q'
ch = 'Z';                   // ch now holds 'Z'
cout << *p;                 // outputs the character 'Z'
*p = 'X';                   // ch now holds 'X'
cout << ch;                 // outputs the character 'X'
```

We shall see that pointers are very useful when building data structures where objects are linked to one another through the use of pointers. Pointers need not point

only to fundamental types, such as **char** and **int**—they may also point to complex types and even to functions. Indeed, the popularity of C++ stems in part from its ability to handle low-level entities like pointers.

It is useful to have a pointer value that points to nothing, that is, a ***null pointer***. By convention, such a pointer is assigned the value zero. An attempt to dereference a null pointer results in a run-time error. All C++ implementations define a special symbol NULL, which is equal to zero. This definition is activated by inserting the statement "#include <cstdlib>" in the beginning of a program file.

We mentioned earlier that the special type **void** is used to indicate no type information at all. Although we cannot declare a variable to be of type **void**, we can declare a pointer to be of type **void***. Such a pointer can point to a variable of ***any*** type. Since the compiler is unable to check the correctness of such references, the use of **void*** pointers is strongly discouraged, except in unusual cases where direct access to the computer's memory is needed.

Caution

Beware when declaring two or more pointers on the same line. The * operator binds with the variable name, not with the type name. Consider the following misleading declaration.

```
int* x, y, z;          // same as: int* x; int y; int z;
```

This declares one pointer variable x, but the other two variables are plain integers. The simplest way to avoid this confusion is to declare one variable per statement.

Arrays

An ***array*** is a collection of elements of the same type. Given any type T and a constant N, a variable of type T[N] holds an array of N elements, each of type T. Each element of the array is referenced by its ***index***, that is, a number from 0 to $N-1$. The following statements declare two arrays; one holds three doubles and the other holds 10 double pointers.

```
double f[5];           // array of 5 doubles: f[0], ..., f[4]
int m[10];             // array of 10 ints: m[0], ..., m[9]
f[4] = 2.5;
m[2] = 4;
cout << f[m[2]];       // outputs f[4], which is 2.5
```

Once declared, it is not possible to increase the number of elements in an array. Also, C++ provides no built-in run-time checking for array subscripting out of bounds. This decision is consistent with C++'s general philosophy of not introducing any feature that would slow the execution of a program. Indexing an array outside of its declared bounds is a common programming error. Such an error often occurs "silently," and only much later are its effects noticed. In Section 1.5.5,

we see that the vector type of the C++ Standard Template Library (STL) provides many of the capabilities of a more complete array type, including run-time index checking and the ability to dynamically change the array's size.

A two-dimensional array is implemented as an "array of arrays." For example "int A[15][30]" declares A to be an array of 30 objects, each of which is an array of 15 integers. An element in such an array is indexed as A[i][j], where i is in the range 0 to 14 and j is in the range 0 to 29.

When declaring an array, we can initialize its values by enclosing the elements in curly braces ({...}). When doing so, we do not have to specify the size of the array, since the compiler can figure this out.

```
int a[]  = {10, 11, 12, 13};        // declares and initializes a[4]
bool b[] = {false, true};            // declares and initializes b[2]
char c[] = {'c', 'a', 't'};          // declares and initializes c[3]
```

Just as it is possible to declare an array of integers, it is possible to declare an array of pointers to integers. For example, int* r[17] declares an array r consisting of 17 pointers to objects of type **int**. Once initialized, we can dereference an element of this array using the * operator, for example, *r[16] is the value of the integer pointed to by the last element of this array.

Pointers and Arrays

There is an interesting connection between arrays and pointers, which C++ inherited from the C programming language—the name of an array is equivalent to a pointer to the array's initial element and vice versa. In the example below, c is an array of characters, and p and q are pointers to the first element of c. They all behave essentially the same, however.

```
char c[] = {'c', 'a', 't'};
char* p = c;                         // p points to c[0]
char* q = &c[0];                     // q also points to c[0]
cout << c[2] << p[2] << q[2];        // outputs "ttt"
```

Caution

This equivalence between array names and pointers can be confusing, but it helps to explain many of C++'s apparent mysteries. For example, given two arrays c and d, the comparison (c == d) does not test whether the contents of the two arrays are equal. Rather it compares the addresses of their initial elements, which is probably not what the programmer had in mind. If there is a need to perform operations on entire arrays (such as copying one array to another) it is a good idea to use the vector class, which is part of C++'s Standard Template Library. We discuss these concepts in Section 1.5.5.

Strings

A string literal, such as `"Hello World"`, is represented as a fixed-length array of characters that ends with the null character. Character strings represented in this way are called ***C-style strings***, since they were inherited from C. Unfortunately, this representation alone does not provide many string operations, such as concatenation and comparison. It also possesses all the peculiarities of C++ arrays, as mentioned earlier.

For this reason, C++ provides a string type as part of its Standard Template Library (STL). When we need to distinguish, we call these ***STL strings***. In order to use STL strings it is necessary to include the header file <string>. Since STL strings are part of the standard namespace (see Section 1.1.4), their full name is std::string. By adding the statement "**using** std::string," we inform the compiler that we want to access this definition directly, so we can omit the "std::" prefix. STL strings may be concatenated using the + operator, they may be compared with each other using lexicographic (or dictionary) order, and they may be input and output using the >> and << operators, respectively. For example:

```
#include <string>
using std::string;
// ...
string s = "to be";
string t = "not " + s;              // t = "not to be"
string u = s + " or " + t;          // u = "to be or not to be"
if (s > t)                          // true: "to be" > "not to be"
  cout << u;                        // outputs "to be or not to be"
```

There are other STL string operations, as well. For example, we can append one string to another using the += operator. Also, strings may be indexed like arrays and the number of characters in a string s is given by s.size(). Since some library functions require the old C-style strings, there is a conversion function s.c_str(), which returns a pointer to a C-style string. Here are some examples:

```
string s = "John";                  // s = "John"
int i = s.size();                   // i = 4
char c = s[3];                      // c = 'n'
s += " Smith";                      // now s = "John Smith"
```

The C++ STL provides many other string operators including operators for extracting, searching for, and replacing substrings. We discuss some of these in Section 1.5.5.

C-Style Structures

A ***structure*** is useful for storing an aggregation of elements. Unlike an array, the elements of a structure may be of different types. Each ***member***, or ***field***, of a

structure is referred to by a given name. For example, consider the following structure for storing information about an airline passenger. The structure includes the passenger's name, meal preference, and information as to whether this passenger is in the frequent flyer program. We create an enumerated type to handle meal preferences.

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

struct Passenger {
    string      name;          // passenger name
    MealType    mealPref;      // meal preference
    bool        isFreqFlyer;   // in the frequent flyer program?
    string      freqFlyerNo;   // the passenger's freq. flyer number
};
```

This defines a new type called Passenger. Let us declare and initialize a variable named "pass" of this type.

```
Passenger pass = { "John Smith", VEGETARIAN, true, "293145" };
```

The individual members of the structure are accessed using the ***member selection operator***, which has the form struct_name.member. For example, we could change some of the above fields as follows.

```
pass.name = "Pocahontas";       // change name
pass.mealPref = REGULAR;        // change meal preference
```

Structures of the same type may be assigned to one another. For example, if p1 and p2 are of type Passenger, then p2 = p1 copies the elements of p1 to p2.

What we have discussed so far might be called a ***C-style structure***. C++ provides a much more powerful and flexible construct called a class, in which both data and functions can be combined. We discuss classes in Section 1.5.

Pointers, Dynamic Memory, and the "new" Operator

We often find it useful in data structures to create objects dynamically as the need arises. The C++ run-time system reserves a large block of memory called the ***free store***, for this reason. (This memory is also sometimes called ***heap memory***, but this should not be confused with the heap data structure, which is discussed in Chapter 8.) The operator **new** dynamically allocates the correct amount of storage for an object of a given type from the free store and returns a pointer to this object. That is, the value of this pointer is the address where this object resides in memory. Indeed, C++ allows for pointer variables to any data type, even to other pointers or to individual cells in an array.

For example, suppose that in our airline system we encounter a new passenger. We would like to dynamically create a new instance using the **new** operator. Let p be a pointer to a Passenger structure. This implies that *p refers to the actual structure; hence, we could access one of its members, say the mealPref field, using the expression (*p).mealPref. Because complex objects like structures are often allocated dynamically, C++ provides a shorter way to access members using the "->" operator.

pointer_name->member is equivalent to (*pointer_name).member

For example, we could allocate a new passenger object and initialize its members as follows.

```
Passenger *p;
// ...
p = new Passenger;                  // p points to the new Passenger
p->name = "Pocahontas";             // set the structure members
p->mealPref = REGULAR;
p->isFreqFlyer = false;
p->freqFlyerNo = "NONE";
```

It would be natural to wonder whether we can initialize the members using the curly brace ({...}) notation used above. The answer is no, but we will see another more convenient way of initializing members when we discuss classes and constructors in Section 1.5.2.

This new passenger object continues to exist in the free store until it is explicitly deleted—a process that is done using the **delete** operator, which destroys the object and returns its space to the free store.

```
delete p;                           // destroy the object p points to
```

The **delete** operator should only be applied to objects that have been allocated through **new**. Since the object at p's address was allocated using the **new** operator, the C++ run-time system knows how much memory to deallocate for this **delete** statement. Unlike some programming languages such as Java, C++ does not provide automatic ***garbage collection***. This means that C++ programmers have the responsibility of explicitly deleting all dynamically allocated objects.

Arrays can also be allocated with **new**. When this is done, the system allocator returns a pointer to the first element of the array. Thus, a dynamically allocated array with elements of type T would be declared being of type *T. Arrays allocated in this manner cannot be deallocated using the standard **delete** operator. Instead, the operator delete[ ] is used. Here is an example that allocates a character buffer of 500 elements, and then later deallocates it.

```
char* buffer = new char[500];       // allocate a buffer of 500 chars
buffer[3] = 'a';                    // elements are still accessed using []
delete [] buffer;                   // delete the buffer
```

Memory Leaks

Failure to delete dynamically allocated objects can cause problems. If we were to change the (address) value of p without first deleting the structure to which it points, there would be no way for us to access this object. It would continue to exist for the lifetime of the program, using up space that could otherwise be used for other allocated objects. Having such inaccessible objects in dynamic memory is called a ***memory leak***. We should strongly avoid memory leaks, especially in programs that do a great deal of memory allocation and deallocation. A program with memory leaks can run out of usable memory even when there is a sufficient amount of memory present. An important rule for a disciplined C++ programmer is the following:

Remember

> If an object is allocated with **new**, it should eventually be deallocated with **delete**.

References

Pointers provide one way to refer indirectly to an object. Another way is through references. A ***reference*** is simply an alternative name for an object. Given a type T, the notation T& indicates a reference to an object of type T. Unlike pointers, which can be NULL, a reference in C++ must refer to an actual variable. When a reference is declared, its value must be initialized. Afterwards, any access to the reference is treated exactly as if it is an access to the underlying object.

```
string author = "Samuel Clemens";
string& penName = author;          // penName is an alias for author
penName = "Mark Twain";            // now author = "Mark Twain"
cout << author;                    // outputs "Mark Twain"
```

References are most often used for passing function arguments and are also often used for returning results from functions. These uses are discussed later.

1.1.4 Named Constants, Scope, and Namespaces

We can easily name variables without concern for naming conflicts in small problems. It is much harder for us to avoid conflicts in large software systems, which may consist of hundreds of files written by many different programmers. C++ has a number of mechanisms that aid in providing names and limiting their scope.

Constants and Typedef

Good programmers commonly like to associate names with constant quantities. By adding the keyword **const** to a declaration, we indicate that the value of the associated object cannot be changed. Constants may be used virtually anywhere that literals can be used, for example, in an array declaration. As a hint to the reader, we will use all capital letters when naming constants.

```
const double PI          = 3.14159265;
const int    CUT_OFF[]   = {90, 80, 70, 60};
const int    N_DAYS      = 7;
const int    N_HOURS     = 24*N_DAYS; // using a constant expression
int counter[N_HOURS];                 // an array of 168 ints
```

Note that enumerations (see Section 1.1.2) provide another convenient way to define integer-valued constants, especially within structures and classes.

In addition to associating names with constants, it is often useful to associate a name with a type. This association can be done with a **typedef** declaration. Rather than declaring a variable, a **typedef** defines a new type name.

```
typedef char* BufferPtr;          // type BufferPtr is a pointer to char
typedef double Coordinate;        // type Coordinate is a double

BufferPtr p;                      // p is a pointer to char
Coordinate x, y;                  // x and y are of type double
```

By using **typedef** we can provide shorter or more meaningful synonyms for various types. The type name Coordinate provides more of a hint to the reader of the meaning of variables x and y than does **double**. Also, if later we decide to change our coordinate representation to **int**, we need only change the **typedef** statement. We will follow the convention of indicating user-defined types by capitalizing the first character of their names.

Local and Global Scopes

When a group of C++ statements are enclosed in curly braces ({...}), they define a ***block***. Variables and types that are declared within a block are only accessible from within the block. They are said to be ***local*** to the block. Blocks can be nested within other blocks. In C++, a variable may be declared outside of any block. Such a variable is ***global***, in the sense that it is accessible from everywhere in the program. The portions of a program from which a given name is accessible are called its ***scope***.

Two variables of the same name may be defined within nested blocks. When this happens, the variable of the inner block becomes active until leaving the block.

Thus a local variable "hides" any global variables of the same name as shown in the following example.

```
const int Cat = 1;                    // global Cat

int main() {
  const int Cat = 2;                  // this Cat is local to main
  cout << Cat;                        // outputs 2 (local Cat)
  return EXIT_SUCCESS;
}

int dog = Cat;                        // dog = 1 (from the global Cat)
```

Namespaces

Global variables present many problems in large software systems because they can be accessed and possibly modified anywhere in the program. They also can lead to programming errors, since an important global variable may be hidden by a local variable of the same name. As a result, it is best to avoid global variables. We may not be able to avoid globals entirely, however. For example, when we perform output, we actually use the system's global standard output stream object, cout. If we were to define a variable with the same name, then the system's cout stream would be inaccessible.

A ***namespace*** is a mechanism that allows a group of related names to be defined in one place. This helps organize global objects into natural groups and minimizes the problems of globals. For example, the following declares a namespace myglobals containing two variables, cat and dog.

```
namespace myglobals {
  int cat;
  string dog = "bow wow";
}
```

Namespaces may generally contain definitions of more complex objects, including types, classes, and functions. We can access an object x in namespace group, using the notation group::x, which is called its ***fully qualified name***. For example, myglobals::cat refers to the copy of variable cat in the myglobals namespace.

We have already seen an example of a namespace. Many standard system objects, such as the standard input and output streams cin and cout, are defined in a system namespace called std. Their fully qualified names are std::cin and std::cout, respectively.

The Using Statement

If we are repeatedly using variables from the same namespace, it is possible to avoid entering namespace specifiers by telling the system that we want to "use" a particular specifier. We communicate this desire by utilizing the **using** statement, which makes some or all of the names from the namespace accessible, without explicitly providing the specifier. This statement has two forms that allow us to list individual names or to make every name in the namespace accessible as shown below.

```
using std::string;                  // makes just std::string accessible
using std::cout;                    // makes just std::cout accessible

using namespace myglobals;          // makes all of myglobals accessible
```

1.2 Expressions

An ***expression*** combines variables and literals with operators to create new values. In the following discussion, we group operators according to the types of objects they may be applied to. Throughout, we use var to denote a variable or anything to which a value may be assigned. (In official C++ jargon, this is called an ***lvalue***.) We use exp to denote an expression and type to denote a type.

Member Selection and Indexing

Some operators access a member of a structure, class, or array. We let class_name denote the name of a structure or class; pointer denotes a pointer to a structure or class and array denotes an array or a pointer to the first element of an array.

class_name . member	class/structure member selection
pointer –> member	class/structure member selection
array [exp]	array subscripting

Arithmetic Operators

The following are the binary arithmetic operators:

exp + exp	addition
exp – exp	subtraction
exp * exp	multiplication
exp / exp	division
exp % exp	modulo (remainder)

There are also unary minus (–x) and unary plus (+x) operations. Division between two integer operands results in an integer result by truncation, even if the

result is being assigned to a floating point variable. The modulo operator n%m yields the remainder that would result from the integer division n/m.

Increment and Decrement Operators

The ***post-increment*** operator returns a variable's value and then increments it by 1. The post-decrement operator is analogous but decreases the value by 1. The ***pre-increment*** operator first increments the variables and then returns the value.

var ++	post increment
var −−	post decrement
++ var	pre increment
−− var	pre decrement

The following code fragment illustrates the increment and decrement operators.

```
int a[] = {0, 1, 2, 3};
int i = 2;
int j = i++;                    // j = 2 and now i = 3
int k = --i;                    // now i = 2 and k = 2
cout << a[k++];                 // a[2] (= 2) is output; now k = 3
```

Relational and Logical Operators

C++ provides the usual comparison operators.

exp < exp	less than
exp > exp	greater than
exp <= exp	less than or equal
exp >= exp	greater than or equal
exp == exp	equal to
exp != exp	not equal to

These return a Boolean result—either **true** or **false**. Comparisons can be made between numbers, characters, and STL strings (but not C-style strings). Pointers can be compared as well, but it is usually only meaningful to test whether pointers are equal or not equal (since their values are memory addresses).

The following logical operators are also provided.

! exp	logical not
exp && exp	logical and
exp || exp	logical or

The operators && and || evaluate sequentially from left to right. If the left operand of && is false, the entire result is false, and the right operand is not evaluated. The || operator is analogous, but evaluation stops if the left operand is true.

This "short circuiting" is quite useful in evaluating a chain of conditional expressions where the left condition guards against an error committed by the right

condition. For example, the following code first tests that a Passenger pointer p is non-null before accessing it. It would result in an error if the execution were not stopped if the first condition is not satisfied.

```
if ((p != NULL) && p->isFreqFlyer) ...
```

Bitwise Operators

The following operators act on the representations of numbers as binary bit strings. They can be applied to any integer type, and the result is an integer type.

~ exp	bitwise complement
exp & exp	bitwise and
exp ^ exp	bitwise exclusive-or
exp | exp	bitwise or
exp1 << exp2	shift exp1 left by exp2 bits
exp1 >> exp2	shift exp1 right by exp2 bits

The left shift operator always fills with zeros. How the right shift fills depends on a variable's type. In C++ integer variables are "signed" quantities by default, but they may be declared as being "unsigned," as in "**unsigned int** x." If the left operand of a right shift is unsigned, the shift fills with zeros and otherwise the right shift fills with the number's sign bit (0 for positive numbers and 1 for negative numbers). Note that the input (>>) and output (<<) operators are not in this group. They are discussed later.

Assignment Operators

In addition to the familiar assignment operator (=), C++ includes a special form for each of the arithmetic binary operators (+, −, *, /, %) and each of the bitwise binary operators (&, |, ^, <<, >>), that combines a binary operation with assignment. For example, the statement "n += 2" means "n = n + 2." Some examples are shown below.

```
int     i = 10;
int     j = 5;
string  s = "yes";
i   -=   4;                    // i = i - 4 = 6
j   *=  -2;                    // j = j * (-2) = -10
s   +=   " or no";             // s = s + " or no" = "yes or no"
```

These assignment operators not only provide notational convenience, but they can be more efficient to execute as well. For example, in the string concatenation example above, the new text can just be appended to s without the need to generate a temporary string to hold the intermediate result.

Take care when performing assignments between aggregate objects (arrays, strings, and structures). Typically the programmer intends such an assignment to copy the contents of one object to the other. This works for STL strings and C-style structures (provided they have the same type). However, as discussed earlier, C-style strings and arrays cannot be copied merely through a single assignment statement.

Other Operators

Here are some other useful operators.

class_name :: member	class scope resolution
namespace_name :: member	namespace resolution
bool_exp ? true_exp : false_exp	conditional expression

We have seen the namespace resolution operator in Section 1.1.4. The conditional expression is a variant of "if-then-else" for expressions. If bool_exp evaluates to true, the value of true_exp is returned, and otherwise the value of false_exp is returned.

The following example shows how to use this to return the minimum of two numbers, x and y.

```
smaller = (x < y ? x : y);          // smaller = min(x,y)
```

We also have the following operations on input/output streams.

stream >> var	stream input
stream << exp	stream output

Although they look like the bitwise shift operators, the input (>>) and output (<<) stream operators are quite different. They are examples of C++'s powerful capability, called ***operator overloading***, which are discussed in Section 1.4.2. These operators are not an intrinsic part of C++, but are provided by including the file <iostream>. We refer the reader to the references given in the chapter notes for more information on input and output in C++.

The above discussion provides a somewhat incomplete list of all the C++ operators, but it nevertheless covers the most common ones. Later we introduce others, including casting operators.

Operator Precedence

Operators in C++ are assigned a ***precedence*** that determines the order in which operations are performed in the absence of parentheses. In Table 1.1, we show the precedence of some of the more common C++ operators, with the highest listed first. Unless parentheses are used, operators are evaluated in order from highest

to lowest. For example, the expression `0 < 4 + x * 3` would be evaluated as if it were parenthesized as `0 < (4 + (x * 3))`. If `p` is an array of pointers, then `*p[2]` is equivalent to `*(p[2])`. Except for `&&` and `||`, which guarantee left-to-right evaluation, the order of evaluation of subexpressions is dependent on the implementation. Since these rules are complex, it is a good idea to add parentheses to complex expressions to make your intent clear to someone reading your program.

Operator Precedences

Type	*Operators*
scope resolution	`namespace_name :: member`

selection/subscripting
function call
postfix operators
	`class_name.member  pointer->member  array[exp]`
	`function(args)`
	`var++  var--`

prefix operators dereference/address	`++var  --var  +exp  -exp  ~exp  !exp` `*pointer  &var`
multiplication/division	`*  /  %`
addition/subtraction	`+  -`
shift	`<<  >>`
comparison	`<  <=  >  >=`
equality	`==  !=`
bitwise and	`&`
bitwise exclusive-or	`^`
bitwise or	`|`
logical and	`&&`
logical or	`||`
conditional	`bool_exp ? true_exp : false_exp`
assignment	`=  +=  -=  *=  /=  %=  >>=  <<=  &=  ^=  |=`

Table 1.1: The C++ precedence rules. The notation "**exp**" denotes any expression.

1.2.1 Changing Types through Casting

Casting is an operation that allows us to change the type of a variable. In essence, we can take a variable of one type and ***cast*** it into an equivalent variable of another type. Casting is useful in many situations. There are two fundamental types of casting that can be done in C++. We can either cast with respect to the fundamental types or we can cast with respect to class objects and pointers. We discuss casting with fundamental types here, and we consider casting with objects in Section 2.2.4. We begin by introducing the traditional way of casting in C++, and later we present C++'s newer casting operators.

Traditional C-Style Casting

Let exp be some expression, and let T be a type. To cast the value of the expression to type T we can use the notation "(T)exp." We call this a ***C-style cast***. If the desired type is a type name (as opposed to a type expression), there is an alternate ***functional-style cast***. This has the form "T(exp)." Some examples are shown below. In both cases, the integer value 14 is cast to a double value 14.0.

```
int     cat = 14;
double  dog = (double) cat;          // traditional C-style cast
double  pig = double(cat);           // C++ functional cast
```

Both forms of casting are legal, but some authors prefer the functional-style cast.

Casting to a type of higher precision or size is often needed in forming expressions. The results of certain binary operators depend on the variable types involved. For example, division between integers always produces an integer result by truncating the fractional part. If a floating-point result is desired, we must cast the operands ***before*** performing the operation as shown below.

```
int     i1  = 18;
int     i2  = 16;
double  dv1 = i1 / i2;                    // dv1 has value 1.0
double  dv2 = double(i1) / double(i2);    // dv2 has value 1.125
double  dv3 = double( i1 / i2 );          // dv3 has value 1.0
```

When i1 and i2 are cast to doubles, double-precision division is performed. When i1 and i2 are not cast, truncated integer division is performed. In the case of dv3, the cast is performed after the integer division, so precision is still lost.

Explicit Cast Operators

Casting operations can vary from harmless to dangerous, depending on how similar the two types are and whether information is lost. For example, casting a **short** to an **int** is harmless, since no information is lost. Casting from a **double** to an **int** is more dangerous because the fractional part of the number is lost. Casting from a **double*** to **char*** is dangerous because the meaning of converting such a pointer will likely vary from machine to machine. One important element of good software design is that programs be ***portable***, meaning that they behave the same on different machines.

For this reason, C++ provides a number of casting operators that make the safety of the cast much more explicit. These are called the **static_cast**, **dynamic_cast**, **const_cast**, and **reinterpret_cast**. We discuss only the **static_cast** here and consider the others as the need arises.

Static Casting

Static casting is used when a conversion is made between two related types, for example numbers to numbers or pointers to pointers. Its syntax is given below.

static_cast < *desired_type* > (*expression*)

The most common use is for conversions between numeric types. Some of these conversions may involve the loss of information, for example a conversion from a **double** to an **int**. This conversion is done by truncating the fractional part (not rounding). For example, consider the following:

```
double  d1 = 3.2;
double  d2 = 3.9999;
int     i1 = static_cast<int>(d1);        // i1 has value 3
int     i2 = static_cast<int>(d2);        // i2 has value 3
```

This type of casting is more verbose than the C-style and functional-style casts shown earlier. But this form is appropriate, because it serves as a visible warning to the programmer that a potentially unsafe operation is taking place. In our examples in this book, we use the functional style for safe casts (such as integer to double) and these newer cast operators for all other casts. Some older C++ compilers may not support the newer cast operators, but then the traditional C-style and functional-style casts can be used instead.

Implicit Casting

There are many instances where the programmer has not requested an ***explicit cast***, but a change of types is required. In many of these cases, C++ performs an ***implicit cast***. That is, the compiler automatically inserts a cast into the machine-generated code. For example, when numbers of different types are involved in an operation, the compiler automatically casts to the stronger type. C++ allows an assignment that implicitly loses information, but the compiler usually issues a warning message.

```
int     i  = 3;
double  d  = 4.8;
double  d3 = i / d;          // d3 = 0.625 = double(i)/d
int     i3 = d3;             // i3 = 0 = int(d3)
                             // Warning! Assignment may lose information
```

A general rule with casting is to "play it safe." If a compiler's behavior regarding the implicit casting of a value is uncertain, then we are safest in using an explicit cast. Doing so makes our intentions clear.

1.3 Control Flow

Control flow in C++ is similar to that of other high-level languages. We review the basic structure and syntax of control flow in C++ in this section, including method returns, if statements, switch statements, loops, and restricted forms of "jumps" (the **break** and **continue** statements).

If Statement

Every programming language includes a way of making choices, and C++ is no exception. The most common method of making choices in a C++ program is through the use of an ***if statement***. The syntax of an ***if statement*** in C++ is shown below, together with a small example.

```
if ( condition )
    true_statement
else if ( condition )
    else_if_statement
else
    else_statement
```

Each of the conditions should return a Boolean result. Each statement can either be a single statement or a block of statements enclosed in braces ({...}). The "**else if**" and "**else**" parts are optional, and any number of else-if parts may be given. The conditions are tested one by one, and the statement associated with the first true condition is executed. All the other statements are skipped. Here is a simple example.

```
if ( snowLevel < 2 ) {
    goToClass();              // do this if snow level is less than 2
    comeHome();
}
else if ( snowLevel < 5 )
    haveSnowballFight();      // if level is at least 2 but less than 5
else if ( snowLevel < 10 )
    goSkiing();               // if level is at least 5 but less than 10
else
    stayAtHome();             // if snow level is 10 or more
```

Switch Statement

A ***switch statement*** provides an efficient way to distinguish between many different options according to the value of an integral type. In the following example, a

single character is input, and based on the character's value, an appropriate editing function is called. The comments explain the equivalent if-then-else structure, but the compiler is free to select the most efficient way to execute the statement.

```
char command;
cin >> command;                 // input command character
switch (command) {              // switch based on command value
  case 'I' :                    // if (command == 'I')
    editInsert();
    break;
  case 'D' :                    // else if (command == 'D')
    editDelete();
    break;
  case 'R' :                    // else if (command == 'R')
    editReplace();
    break;
  default :                     // else
    cout << "Unrecognized command\n";
    break;
}
```

The argument of the **switch** can be any integral type or enumeration. The "**default**" case is executed if none of the cases equals the switch argument.

Each case in a switch statement should be terminated with a **break** statement, which, when executed, exits the switch statement. Otherwise, the flow of control "falls through" to the next case.

While and Do-While Loops

C++ has two kinds of conditional loops for iterating over a set of statements as long as some specified condition holds. These two loops are the standard ***while loop*** and the ***do-while loop***. One loop tests a Boolean condition before performing an iteration of the loop body and the other tests a condition after. Let us consider the while loop first.

while (*condition*)
    loop_body_statement

At the beginning of each iteration, the loop tests the Boolean expression and then executes the loop body only if this expression evaluates to true. The loop body statement can also be a block of statements.

Consider the following example. It computes the sum of the elements of an array, until encountering the first negative value. Note the use of the += operator to increment the value of sum and the ++ operator which increments i after accessing

the current array element.

```
int a[100];
// ...
int i = 0;
int sum = 0;
while (i < 100 && a[i] >= 0) {
  sum += a[i++];
}
```

The do-while loop is similar to the while loop in that the condition is tested at the end of the loop execution rather than before. It has the following syntax.

do
    loop_body_statement
while (*condition*)

For Loop

Many loops involve three common elements: an initialization, a condition under which to continue execution, and an increment to be performed after each execution of the loop's body. A ***for loop*** conveniently encapsulates these three elements.

for (*initialization* ; *condition* ; *increment*)
    loop_body_statement

The *initialization* indicates what is to be done before starting the loop. Typically, this involves declaring and initializing a loop-control variable or counter. Next, the *condition* gives a Boolean expression to be tested in order for the loop to continue execution. It is evaluated before executing the loop body. When the condition evaluates to **false**, execution jumps to the next statement after the for loop. Finally, the *increment* specifies what changes are to be made at the end of each execution of the loop body. Typically, this involves incrementing or decrementing the value of the loop-control variable.

Here is a simple example, which prints the positive elements of an array, one per line. Recall that `'\n'` generates a newline character.

```
const int NUM_ELEMENTS = 100;
double b[NUM_ELEMENTS];
// ...
for (int i = 0; i < NUM_ELEMENTS; i++) {
  if (b[i] > 0)
    cout << b[i] << '\n';
}
```

In this example, the loop variable i was declared as int i = 0. Before each iteration, the loop tests the condition "i < NUM_ELEMENTS" and executes the loop body

only if this is true. Finally, at the end of each iteration the loop uses the statement i++ to increment the loop variable i before testing the condition again. Although the loop variable is declared outside the curly braces of the for loop, the compiler treats it as if it were a local variable within the loop. This implies that its value is not accessible outside the loop.

Break and Continue Statements

C++ provides statements to change control flow, including the **break**, **continue**, and **return** statements. We discuss the first two here, and leave the **return** statement for later. A break statement is used to "break" out of a loop or switch statement. When it is executed, it causes the flow of control to immediately exit the innermost switch statement or loop (for loop, while loop, or do-while loop). The break statement is useful when the condition for terminating the loop is determined inside the loop. For example, in an input loop, termination often depends on a specific value that has been input. The following example provides a different implementation of an earlier example, which sums the elements of an array until finding the first negative value.

```
int a[100];
// ...
int sum = 0;
for (int i = 0; i < 100; i++) {
  if (a[i] < 0) break;
  sum += a[i];
}
```

The other statement that is often useful for altering loop behavior is the **continue** statement. The continue statement can only be used inside loops (**for**, **while**, and **do-while**). The continue statement causes the execution to skip to the end of the loop, ready to start a new iteration.

1.4 Functions

A ***function*** is a chunk of code that can be called to perform some well-defined task, such as calculating the area of a rectangle, computing the weekly withholding tax for a company employee, or sorting a list of names in ascending order. In order to define a function, we need to provide the following information to the compiler:

Return type. This specifies the type of value or object that is returned by the function. For example, a function that computes the area of a rectangle might return a value of type **double**. A function is not required to return a value. For example, it may simply produce some output or modify some data structure.

If so, the return type is **void**. A function that returns no value is sometimes called a ***procedure***.

Function name. This indicates the name that is given to the function. Ideally, the function's name should provide a hint to the reader as to what the function does.

Argument list. This serves as a list of placeholders for the values that will be passed into the function. The actual values will be provided when the function is invoked. For example, a function that computes the area of a polygon might take four **double** arguments; the x- and y-coordinates of the rectangle's lower left corner and the x- and y-coordinates of the rectangle's upper right corner. The argument list is given as a comma-separated list enclosed in parentheses, where each entry consists of the name of the argument and its type. A function may have any number of arguments, and the argument list may even be empty.

Function body. This is a collection of C++ statements that define the actual computations to be performed by the function. This is enclosed within curly braces. If the function returns a value, the body will typically end with a **return** statement, which specifies the final function value.

Function specifications in C++ typically involve two steps, declaration and definition. A function is ***declared***, by specifying three things: the function's return type, its name, and its argument list. The declaration makes the compiler aware of the function's existence, and allows the compiler to verify that the function is being used correctly. This three-part combination of return type, function name, and argument types is called the function's ***signature*** or ***prototype***.

For example, suppose that we wanted to create a function, called evenSum, that is given two arguments, an integer array a and its length n. It determines whether the sum of array values is even, and if so it returns the value **true**. Otherwise, it returns **false**. Thus, it has a return value of type **bool**. The function could be declared as follows:

```
bool evenSum(int a[], int n);                    // function declaration
```

Second, the function is ***defined***. The definition consists both of the function's signature and the function body. The reason for distinguishing between the declaration and definition involves the manner in which large C++ programs are written. They are typically spread over many different files. The function declaration must appear in every file that invokes the function, but the definition must appear only

once. Here is how our evenSum function might be defined.

```
bool evenSum(int a[], int n) {                    // function definition
  int sum = 0;
  for (int i = 0; i < n; i++)                     // sum the array elements
    sum += a[i];
  return (sum % 2) == 0;                          // returns true if sum is even
}
```

The expression in the **return** statement may take a minute to understand. We use the mod operator (%) to compute the remainder when sum is divided by 2. If the sum is even, the remainder is 0, and hence the expression "(sum % 2) == 0" evaluates to **true**. Otherwise, it evaluates to **false**, which is exactly what we want.

To complete the example, let us provide a simple main program, which first declares the function, and then invokes it on an actual array.

```
bool evenSum(int a[], int n);                     // function declaration

int main() {
  int list[] = {4, 2, 7, 8, 5, 1};
  bool result = evenSum(list, 6);                 // invoke the function
  if (result)     cout << "the sum is even\n";
  else            cout << "the sum is odd\n";
  return EXIT_SUCCESS;
}
```

Let us consider this example in greater detail. The names "a" and "n" in the function definition are called ***formal arguments*** since they serve merely as placeholders. The variable "list" and literal "6" in the function call in the main program are the ***actual arguments***. Thus, each reference to "a" in the function body is translated into a reference to the actual array "list." Similarly, each reference to "n" can be thought of as taking on the actual value 6 in the function body. The types of the actual arguments must agree with the corresponding formal arguments. Exact type agreement is not always necessary, however, for the compiler may perform implicit type conversions in some cases, such as casting a **short** actual argument to match an **int** formal argument.

When we refer to function names throughout this book, we often include a pair of parentheses following the name. This makes it easier to distinguish function names from variable names. For example, we would refer to the above function as evenSum.

1.4.1 Argument Passing

By default, arguments in C++ programs are passed ***by value***. When arguments are passed by value, the system makes a copy of the variable to be passed to the

function. In the above example, the formal argument "n" is initialized to the actual value 6 when the function is called. This implies that modifications made to a formal argument in the function do not alter the actual argument.

Sometimes it is useful for the function to modify one of its arguments. To do so, we can explicitly define a formal argument to be a ***reference type*** (as introduced in Section 1.1.3). When we do this, any modifications made to an argument in the function modifies the corresponding actual argument. This is called passing the argument ***by reference***. An example is shown below, where one argument is passed by value and the other is passed by reference.

```
void f(int value, int& ref) {          // one value and one reference
  value++;                             // no effect on the actual argument
  ref++;                               // modifies the actual argument
  cout << value << endl;               // outputs 2
  cout << ref << endl;                 // outputs 6
}

int main() {
  int cat = 1;
  int dog = 5;
  f(cat, dog);                         // pass cat by value, dog by ref
  cout << cat << endl;                 // outputs 1
  cout << dog << endl;                 // outputs 6
  return EXIT_SUCCESS;
}
```

Observe that altering the value argument had no effect on the actual argument, whereas modifying the reference argument did.

Modifying function arguments is felt to be a rather sneaky way of passing information back from a function, especially if the function returns a nonvoid value. Another way to modify an argument is to pass the address of the argument, rather than the argument itself. Even though a pointer is passed by value (and, hence, the address of where it is pointing cannot be changed), we can access the pointer and modify the variables to which it points. Reference arguments achieve essentially the same result with less notational burden.

Constant References as Arguments

There is a good reason for choosing to pass structure and class arguments by reference. In particular, passing a large structure or class by value results in a copy being made of the entire structure. All this copying may be quite inefficient for large structures and classes. Passing such an argument by reference is much more efficient, since only the address of the structure need be passed.

Since most function arguments are not modified, an even better practice is to pass an argument as a "constant reference." Such a declaration informs the compiler

that, even though the argument is being passed by reference, the function cannot alter its value. Furthermore, the function is not allowed to pass the argument to another function that might modify its value. Here is an example using the Passenger structure, which we defined earlier in Section 1.1.3. The attempt to modify the argument would result in a compiler error message.

```
void someFunction(const Passenger& pass) {
  pass.name = "new name";          // ILLEGAL! pass is declared const
}
```

When writing small programs, we can easily avoid modifying the arguments that are passed by reference for the sake of efficiency. But in large programs, which may be distributed over many files, enforcing this rule is much harder. Fortunately, passing class and structure arguments as a constant reference allows the compiler to do the checking for us. Henceforth, when we pass a class or structure as an argument, we typically pass it as a reference, usually a constant reference.

Array Arguments

We have discussed passing large structures and classes by reference, but what about large arrays? Would passing an array by value result in making a copy of the entire array? The answer is no. When an array is passed to a function, it is converted to a pointer to its initial element. That is, an object of type T[ ] is converted to type T*. Thus, an assignment to an element of an array within a function does modify the actual array contents. In short, arrays are not passed by value.

By the same token, it is not meaningful to pass an array back as the result of a function call. Essentially, an attempt to do so will only pass a pointer to the array's initial element. If returning an array is our goal, then we should either explicitly return a pointer or consider returning an object of type vector from the C++ Standard Template Library.

1.4.2 Overloading and Inlining

Overloading means defining two or more functions or operators that have the same name, but whose effect depends on the types of their actual arguments.

Function Overloading

Function overloading occurs when two or more functions are defined with the same name but with different argument lists. Such definitions are useful in situations where we desire two functions that achieve essentially the same purpose, but do it with different types of arguments.

One convenient application of function overloading is in writing procedures that print their arguments. In particular, a function that prints an integer would be different from a function that prints a Passenger structure from Section 1.1.3, but both could use the same name, print, as shown in the following example.

```
void print(int x)                          // print an integer
  { cout << x; }

void print(const Passenger& pass) {   // print a Passenger
  cout << pass.name << " " << pass.mealPref;
  if (pass.isFreqFlyer)
    cout << " " << pass.freqFlyerNo;
}
```

When the print function is used, the compiler considers the types of the actual argument and invokes the appropriate function, that is, the one with signature closest to the actual arguments.

Operator Overloading

C++ also allows overloading of operators, such as +, *, +=, and <<. Not surprisingly, such a definition is called ***operator overloading***. Suppose we would like to write an equality test for two Passenger objects. We can denote this in a natural way by overloading the == operator as shown below.

```
bool operator==(const Passenger& x, const Passenger& y) {
  return x.name          == y.name
      && x.mealPref      == y.mealPref
      && x.isFreqFlyer   == y.isFreqFlyer
      && x.freqFlyerNo   == y.freqFlyerNo;
}
```

This definition is similar to a function definition, but in place of a function name we use "operator==." In general, the == is replaced by whatever operator is being defined. For binary operators we have two arguments, and for unary operators we have just one.

There are several useful applications of function and operator overloading. For example, overloading the == operator allows us to naturally test for the equality of two objects, p1 and p2, with the expression "p1==p2." Another useful application of operator overloading is for defining input and output operators for classes and structures. Here is how to define an output operator for our Passenger structure. The type ostream is the system's output stream type. The standard output, cout is

of this type.

```
ostream& operator<<(ostream& out, const Passenger& pass) {
  out << pass.name << " " << pass.mealPref;
  if (pass.isFreqFlyer) {
    out << " " << pass.freqFlyerNo;
  }
  return out;
}
```

The output in this case is not very pretty, but we could easily modify our output operator to produce nicer formatting.

There is much more that could be said about function and operator overloading, and indeed C++ functions in general. We refer the reader to a more complete reference on C++ for this information.

Operator overloading is a powerful mechanism, but it is easily abused. It can be very confusing for someone reading your program to find that familiar operations such as "+" and "/" have been assigned new and possibly confusing meanings. Good programmers usually restrict operator overloading to certain general purpose operators such as " <<" (output), "=" (assignment), "==" (equality), "[ ]" (indexing, for sequences).

In-line Functions

Very short functions may be defined to be "**inline**." This is a hint to the compiler it should simply expand the function code in place, rather than using the system's call-return mechanism. As a rule of thumb, in-line functions should be very short (at most a few lines) and should not involve any loops or conditionals. Here is an example, which returns the minimum of two integers.

```
inline int min(int x, int y) { return (x < y ? x : y); }
```

1.5 Classes

The concept of a ***class*** is fundamental to C++, since it provides a way to define new user-defined types, complete with associated functions and operators. By restricting access to certain class members, it is possible to separate out the properties that are essential to a class's correct use from the details needed for its implementation. Classes are fundamental to programming that uses an object-oriented approach, which is a programming paradigm we discuss in the next chapter.

1.5.1 Class Structure

A class consists of ***members***. Members that are variables or constants are ***data members*** (also called ***member variables***) and members that are functions are called ***member functions*** (also called ***methods***). Data members may be of any type, and may even be classes themselves, or pointers or references to classes. Member functions typically act on the member variables, and so define the behavior of the class.

We begin with a simple example, called Counter. It implements a simple counter stored in the member variable count. It provides three member functions. The first member function, called Counter, initializes the counter. The second, called getCount, returns the counter's current value. The third, called increaseBy, increases the counter's value.

```
class Counter {                          // a simple counter
public:
  Counter();                             // initialization
  int getCount();                        // get the current count
  void increaseBy(int x);                // add x to the count
private:
  int count;                             // the counter's value
};
```

Let's explore this class definition in a bit more detail. Observe that the class definition is separated into two parts by the keywords **public** and **private**. The public section defines the class's ***public interface***. These are the entities that users of the class are allowed to access. In this case, the public interface has the three member functions (Counter, getCount, and increaseBy). In contrast, the private section declares entities that cannot be accessed by users of the class. We say more about these two parts below.

So far, we have only declared the member functions of class Counter. Next, we present the definitions of these member functions. In order to make clear to the compiler that we are defining member functions of Counter (as opposed to member functions of some other class), we precede each function name with the scoping specifier "Counter::".

```
Counter::Counter()                       // constructor
  { count = 0; }
int Counter::getCount()                  // get current count
  { return count; }
void Counter::increaseBy(int x)          // add x to the count
  { count += x; }
```

The first of these functions has the same name as the class itself. This is a special member function called a ***constructor***. A constructor's job is to initialize the values

of the class's member variables. The function getCount is commonly referred to as a "getter" function. Such functions provide access to the private members of the class.

Here is an example how we might use our simple class. We declare a new object of type Counter, called ctr. This implicitly invokes the class's constructor, and thus initializes the counter's value to 0. To invoke one of the member functions, we use the notation ctr.function_name().

```
Counter ctr;                          // an instance of Counter
cout << ctr.getCount() << endl;       // prints the initial value (0)
ctr.increaseBy(3);                    // increase by 3
cout << ctr.getCount() << endl;       // prints 3
ctr.increaseBy(5);                    // increase by 5
cout << ctr.getCount() << endl;       // prints 8
```

Access Control

One important feature of classes is the notion of ***access control***. Members may be declared to be ***public***, which means that they are accessible from outside the class, or ***private***, which means that they are accessible only from within the class. (We discuss two exceptions to this later: protected access and friend functions.) In the previous example, we could not directly access the private member count from outside the class definition.

```
Counter ctr;                          // ctr is an instance of Counter
// ...
cout << ctr.count << endl;            // ILLEGAL - count is private
```

Why bother declaring members to be private? We discuss the reasons in detail in Chapter 2 when we discuss object-oriented programming. For now, suffice it to say that it stems from the desire to present users with a clean (public) interface from which to use the class, without bothering them with the internal (private) details of its implementation. All external access to class objects takes place through the public members, or the ***public interface*** as it is called. The syntax for a class is as follows.

class ⟨ *class_name* ⟩ {
public:
    public_members
private:
    private_members
};

Note that if no ***access specifier*** is given, the default is **private** for classes and **public** for structures. (There is a third specifier, called **protected**, which is discussed later in the book.) There is no required order between the private and public

sections, and in fact, it is possible to switch back and forth between them. Most C++ style manuals recommend that public members be presented first, since these are the elements of the class that are relevant to a programmer who wishes to use the class. We sometimes violate this convention in this book, particularly when we want to emphasize the private members.

Member Functions

Let us return to the Passenger structure, which was introduced earlier in Section 1.1.3, but this time we define it using a class structure. We provide the same member variables as earlier, but they are now private members. To this we add a few member functions. For this short example, we provide just a few of many possible member functions. The first member function is a constructor. Its job is to guarantee that each instance of the class is properly initialized. Notice that the constructor does not have a return type. The member function isFrequentFlyer tests whether the passenger is a frequent flyer, and the member function makeFrequentFlyer makes a passenger a frequent flyer and assigns a frequent flyer number. This is only a partial definition, and a number of member functions have been omitted. As usual we use " //. . ." to indicate omitted code.

```
class Passenger {                          // Passenger (as a class)
public:
  Passenger();                             // constructor
  bool isFrequentFlyer() const;            // is this a frequent flyer?
                                           // make this a frequent flyer
  void makeFrequentFlyer(const string& newFreqFlyerNo);
  // . . . other member functions
private:
  string      name;                        // passenger name
  MealType    mealPref;                    // meal preference
  bool        isFreqFlyer;                 // is a frequent flyer?
  string      freqFlyerNo;                 // frequent flyer number
};
```

Class member functions can be placed in two major categories: ***accessor functions***, which only read class data, and ***update functions***, which may alter class data. The keyword "**const**" indicates that the member function isFrequentFlyer is an accessor. This informs the user of the class that this function will not change the object contents. It also allows the compiler to catch a potential error should we inadvertently attempt to modify any class member variables.

We have declared two member functions, but we still need to define them. Member functions may either be defined inside or outside the class body. Most C++ style manuals recommend defining all member functions outside the class, in

order to present a clean public interface in the class's definition. As we saw above in the Counter example, when a member function is defined outside the class body, it is necessary to specify which class it belongs to, which is done by preceding the function name with the scoping specifier class_name::member_name.

```
bool Passenger::isFrequentFlyer() const {
  return isFreqFlyer;
}
void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
  isFreqFlyer = true;
  freqFlyerNo = newFreqFlyerNo;
}
```

Notice that when we are within the body of a member function, the member variables (such as isFreqFlyer and freqFlyerNo) are given without reference to a particular object. These functions will be invoked on a particular Passenger object. For example, let pass be a variable of type Passenger. We may invoke these public member functions on pass using the same member selection operator we introduced with structures as shown below. Only public members may be accessed in this way.

```
Passenger pass;                          // pass is a Passenger
// ...
if ( !pass.isFrequentFlyer() ) {         // not already a frequent flyer?
  pass.makeFrequentFlyer("392953");      // set pass's freq flyer number
}
pass.name = "Joe Blow";                  // ILLEGAL! name is private
```

In-Class Function Definitions

In the above examples, we have shown member functions being defined outside of the class body. We can also define members within the class body. When a member function is defined within a class it is compiled ***in line*** (recall Section 1.4.2). As with in-line functions, in-class function definitions should be reserved for short functions that do not involve loops or conditionals. Here is an example of how the isFrequentFlyer member function would be defined from within the class.

```
class Passenger {
public:
  // ...
  bool isFrequentFlyer() const { return isFreqFlyer; }
  // ...
};
```

1.5.2 Constructors and Destructors

The above declaration of the class variable pass suffers from the shortcoming that we have not initialized any of its classes members. An important aspect of classes is the capability to initialize a class's member data. A ***constructor*** is a special member function whose task is to perform such an initialization. It is invoked when a new class object comes into existence. There is an analogous ***destructor*** member function that is called when a class object goes out of existence.

Constructors

A constructor member function's name is the same as the class, and it has no return type. Because objects may be initialized in different ways, it is natural to define different constructors and rely on function overloading to determine which one is to be called.

Returning to our Passenger class, let us define three constructors. The first constructor has no arguments. Such a constructor is called a ***default constructor***, since it is used in the absence of any initialization information. The second constructor is given the values of the member variables to initialize. The third constructor is given a Passenger reference from which to copy information. This is called a ***copy constructor***.

```
class Passenger {
private:
  // ...
public:
  Passenger();                                  // default constructor
  Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
  Passenger(const Passenger& pass);  // copy constructor
  // ...
};
```

Look carefully at the second constructor. The notation ffn="NONE" indicates that the argument for ffn is a ***default argument***. That is, an actual argument need not be given, and if so, the value "NONE" is used instead. If a newly created passenger is not a frequent flyer, we simply omit this argument. The constructor tests for this special value and sets things up accordingly. Default arguments can be assigned any legal value and can be used for more than one argument. It is often useful to define default values for all the arguments of a constructor. Such a constructor is the default constructor because it is called if no arguments are given. Default arguments can be used with any function (not just constructors). The associated constructor definitions are shown below. Note that the default argument is given in

the declaration, but not in the definition.

```
Passenger::Passenger() {                    // default constructor
  name = "--NO NAME--";
  mealPref = NO_PREF;
  isFreqFlyer = false;
  freqFlyerNo = "NONE";
}
                                            // constructor given member values
Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
  name = nm;
  mealPref = mp;
  isFreqFlyer = (ffn != "NONE");            // true only if ffn given
  freqFlyerNo = ffn;
}
                                            // copy constructor
Passenger::Passenger(const Passenger& pass) {
  name = pass.name;
  mealPref = pass.mealPref;
  isFreqFlyer = pass.isFreqFlyer;
  freqFlyerNo = pass.freqFlyerNo;
}
```

Here are some examples of how the constructors above can be invoked to define Passenger objects. Note that in the cases of p3 and pp2 we have omitted the frequent flyer number.

```
Passenger p1;                                   // default constructor
Passenger p2("John Smith", VEGETARIAN, 293145); // 2nd constructor
Passenger p3("Pocahontas", REGULAR);            // not a frequent flyer
Passenger p4(p3);                               // copied from p3
Passenger p5 = p2;                              // copied from p2
Passenger* pp1 = new Passenger;                 // default constructor
Passenger* pp2 = new Passenger("Joe Blow", NO_PREF); // 2nd constr.
Passenger pa[20];                               // uses the default constructor
```

Although they look different, the declarations for p4 and p5 both call the copy constructor. These declarations take advantage of a bit of notational magic, which C++ provides to make copy constructors look more like the type definitions we have seen so far. The declarations for pp1 and pp2 create new passenger objects from the free store, and return a pointer to each. The declaration of pa declares an array of Passenger. The individual members of the array are always initialized from the default constructor.

Initializing Class Members with Initializer Lists

There is a subtlety that we glossed over in our presentations of the constructors. Recall that a string is a class in the standard template library. Our initialization using "name=nm" above relied on the fact that the string class has an assignment operator defined for it. If the type of name is a class without an assignment operator, this type of initialization might not be possible. In order to deal with the issue of initializing member variables that are themselves classes, C++ provides an alternate method of initialization called an ***initializer list***. This list is placed between the constructor's argument list and its body. It consists of a colon (:) followed by a comma-separated list of the form member_name(initial_value). To illustrate the feature, let us rewrite the second Passenger constructor so that its first three members are initialized by an initializer list. The initializer list is executed before the body of the constructor.

```
                                    // constructor using an initializer list
Passenger::Passenger(const string& nm, MealType mp, string ffn)
  : name(nm), mealPref(mp), isFreqFlyer(ffn != "NONE")
  { freqFlyerNo = ffn; }
```

Destructors

A constructor is called when a class object comes into existence. A ***destructor*** is a member function that is automatically called when a class object ceases to exist. If a class object comes into existence dynamically using the **new** operator, the destructor will be called when this object is destroyed using the **delete** operator. If a class object comes into existence because it is a local variable in a function that has been called, the destructor will be called when the function returns. The destructor for a class T is denoted ~T. It takes no arguments and has no return type. Destructors are needed when classes allocate resources, such as memory, from the system. When the object ceases to exist, it is the responsibility of the destructor to return these resources to the system.

Let us consider a class Vect, shown in the following code fragment, which stores a vector by dynamically allocating an array of integers. The dynamic array is referenced by the member variable data. (Recall from Section 1.1.3 that a dynamically allocated array is represented by a pointer to its initial element.) The member variable size stores the number of elements in the vector. The constructor for this class allocates an array of the desired size. In order to return this space to the system when a Vect object is removed, we need to provide a destructor to deallocate this space. (Recall that when an array is deleted we use "delete[ ]," rather than "**delete**.")

```
class Vect {                              // a vector class
public:
  Vect(int n);                            // constructor, given size
  ~Vect();                                // destructor
  // ... other public members omitted
private:
  int*        data;                       // an array holding the vector
  int         size;                       // number of array entries
};

Vect::Vect(int n) {                       // constructor
  size = n;
  data = new int[n];                      // allocate array
}

Vect::~Vect() {                           // destructor
  delete [] data;                         // free the allocated array
}
```

We are not strictly required by C++ to provide our own destructor. Nonetheless, if our class allocates memory, we should write a destructor to free this memory. If we did not provide the destructor in the example above, the deletion of an object of type Vect would cause a memory leak. (Recall from Section 1.1.3 that this is an inaccessible block of memory that cannot be removed). The job of explicitly deallocating objects that were allocated is one of the chores that C++ programmers must endure.

1.5.3 Classes and Memory Allocation

When a class performs memory allocation using **new**, care must be taken to avoid a number of common programming errors. We have shown above that failure to deallocate storage in a class's destructor can result in memory leaks. A somewhat more insidious problem occurs when classes that allocate memory fail to provide a copy constructor or an assignment operator. Consider the following example, using our Vect class.

```
Vect a(100);                              // a is a vector of size 100
Vect b = a;                               // initialize b from a (DANGER!)
Vect c;                                   // c is a vector (default size 10)
c = a;                                    // assign a to c (DANGER!)
```

It would seem that we have just created three separate vectors, all of size 100, but have we? In reality all three of these vectors share the same 100-element array. Let us see why this has occurred.

The declaration of object a invokes the vector constructor, which allocates an array of 100 integers and a.data points to this array. The declaration "Vect b=a" initializes b from a. Since we provided no copy constructor in Vect, the system uses its default, which simply copies each member of a to b. In particular it sets "b.data=a.data." Notice that this does not copy the contents of the array; rather it copies the pointer to the array's initial element. This default action is sometimes called a ***shallow copy***.

The declaration of c invokes the constructor with a default argument value of 10, and hence allocates an array of 10 elements in the free store. Because we have not provided an assignment operator, the statement "c=a," also does a shallow copy of a to c. Only pointers are copied, not array contents. Worse yet, we have lost the pointer to c's original 10-element array, thus creating a memory leak.

Now, a, b, and c all have members that point to the same array in the free store. If the contents of the arrays of one of the three were to change, the other two would mysteriously change as well. Worse yet, if one of the three were to be deleted before the others (for example, if this variable was declared in a nested block), the destructor would delete the shared array. When either of the other two attempts to access the now deleted array, the results would be disastrous. In short, there are many problems here.

Fortunately, there is a simple fix for all of these problems. The problems arose because we allocated memory and we used the system's default copy constructor and assignment operator. If a class allocates memory, you should provide a copy constructor and assignment operator to allocate new memory for making copies. A copy constructor for a class T is typically declared to take a single argument, which is a constant reference to an object of the same class, that is, T(const T& t). As shown in the code fragment below, it copies each of the data members from one class to the other while allocating memory for any dynamic members.

```
Vect::Vect(const Vect& a) {                  // copy constructor from a
  size = a.size;                             // copy sizes
  data = new int[size];                      // allocate new array
  for (int i = 0; i < size; i++) {           // copy the vector contents
    data[i] = a.data[i];
  }
}
```

The assignment operator is handled by overloading the = operator as shown in the next code fragment. The argument "a" plays the role of the object on the right side of the assignment operator. The assignment operator deletes the existing array storage, allocates a new array of the proper size, and copies elements into this new array. The if statement checks against the possibility of self assignment. (This can sometimes happen when different variables reference the same object.) We perform this check using the keyword **this**. For any instance of a class object,

"**this**" is defined to be the address of this instance. If **this** equals the address of a, then this is a case of self assignment, and we ignore the operation. Otherwise, we deallocate the existing array, allocate a new array, and copy the contents over.

```
Vect& Vect::operator=(const Vect& a) {   // assignment operator from a
  if (this != &a) {                      // avoid self-assignment
    delete [] data;                      // delete old array
    size = a.size;                       // set new size
    data = new int[size];                // allocate new array
    for (int i=0; i < size; i++) {       // copy the vector contents
      data[i] = a.data[i];
    }
  }
  return *this;
}
```

Notice that in the last line of the assignment operator we return a reference to the current object with the statement "return *this." Such an approach is useful for assignment operators, since it allows us to chain together assignments, as in "a=b=c." The assignment "b=c" invokes the assignment operator, copying variable c to b and then returns a reference to b. This result is then assigned to variable a.

The only other changes needed to complete the job would be to add the appropriate function declarations to the Vect class. By using the copy constructor and assignment operator, we avoid the above memory leak and the dangerous shared array. The lessons of the last two sections can be summarized in the following rule.

Remember

Every class that allocates its own objects using **new** should:

- Define a ***destructor*** to free any allocated objects.
- Define a ***copy constructor***, which allocates its own new member storage and copies the contents of member variables.
- Define an ***assignment operator***, which deallocates old storage, allocates new storage, and copies all member variables.

Some programmers recommend that these functions be included for every class, even if memory is not allocated, but we are not so fastidious. In rare instances, we may want to forbid users from using one or more of these operations. For example, we may not want a huge data structure to be copied inadvertently. In this case, we can define empty copy constructors and assignment functions and make them private members of the class.

1.5.4 Class Friends and Class Members

Complex data structures typically involve the interaction of many different classes. In such cases, there are often issues coordinating the actions of these classes to allow sharing of information. We discuss some of these issues in this section.

We said private members of a class may only be accessed from within the class, but there is an exception to this. Specifically, we can declare a function as a ***friend***, which means that this function may access the class's private data. There are a number of reasons for defining friend functions. One is that syntax requirements may forbid us from defining a member function. For example, consider a class SomeClass. Suppose that we want to define an overloaded output operator for this class, and this output operator needs access to private member data. To handle this, the class declares that the output operator is a friend of the class as shown below.

```
class SomeClass {
private:
  int secret;
public:
  // ...                                  // give << operator access to secret
  friend ostream& operator<<(ostream& out, const SomeClass& x);
};

ostream& operator<<(ostream& out, const SomeClass& x)
  { cout << x.secret; }
```

Another time when it is appropriate to use friends is when two different classes are closely related. For example, Code Fragment 1.1 shows two cooperating classes Vector and Matrix. The former stores a three-dimensional vector and the latter stores a 3×3 matrix. In this code fragment, we show just one example of the usefulness of class friendship. The class Vector stores is coordinates in a private array, called coord. The Matrix class defines a function that multiplies a matrix times a vector. Because coord is a private member of Vector, members of the class Matrix would not have access to coord. However, because Vector has declared Matrix to be a friend, class Matrix can access all the private members of class Vector.

The ability to declare friendship relationships between classes is useful, but the extensive use of friends often indicates a poor class structure design. For example, a better solution would be to have class Vector define a public subscripting operator. Then the multiply function could use this public member to access the vector class, rather than access private member data.

Note that "friendship" is not transitive. For example, if a new class Tensor was made a friend of Matrix, Tensor would not be a friend of Vector, unless class Vector were to explicitly declare it to be so.

```
class Vector {                                  // a 3-element vector
public: // ... public members omitted
private:
  double coord[3];                              // storage for coordinates
  friend class Matrix;                          // give Matrix access to coord
};

class Matrix {                                  // a 3x3 matrix
public:
  Vector multiply(const Vector& v);             // multiply by vector v
  // ... other public members omitted
private:
  double a[3][3];                               // matrix entries
};

Vector Matrix::multiply(const Vector& v) {      // multiply by vector v
  Vector w;
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      w.coord[i] += a[i][j] * v.coord[j];       // access to coord allowed
  return w;
}
```

Code Fragment 1.1: An example of class friendship.

Nesting Classes and Types within Classes

We know that classes may define member variables and member functions. Classes may also define their own types as well. In particular, we can nest a class definition within another class. Such a ***nested class*** is often convenient in the design of data structures. For example, suppose that we want to design a data structure, called Book, and we want to provide a mechanism for placing bookmarks to identify particular locations within our book. We could define a nested class, called Bookmark, which is defined within class Book.

```
class Book {
public:
  class Bookmark {
    // ... (Bookmark definition here)
  };
  // ... (Remainder of Book definition)
}
```

We might define a member function that returns a bookmark within the book, say, to the start of some chapter. Outside the class Book, we use the scope-resolution operator, Book::Bookmark, in order to refer to this nested class. We shall see many other examples of nested classes later in the book.

1.5.5 The Standard Template Library

The ***Standard Template Library*** (***STL***) is a collection of useful classes for common data structures. In addition to the string class, which we have seen many times, it also provides data structures for the following ***standard containers***. We discuss many of these data structures later in this book, so don't worry if their names seem unfamiliar.

stack	Container with last-in, first-out access
queue	Container with first-in, first-out access
deque	Double-ended queue
vector	Resizeable array
list	Doubly linked list
priority_queue	Queue ordered by value
set	Set
map	Associative array (dictionary)

Templates and the STL Vector Class

One of the important features of the STL is that each such object can store objects of any one type. Contrast this with the Vect class of Section 1.5.2, which can only hold integers. Such a class whose definition depends on a user-specified type is called a ***template***. We discuss templates in greater detail in Chapter 2, but we briefly mention how they are used with container objects here.

We specify the type of the object being stored in the container in angle brackets (<...>). For example, we could define vectors to hold 100 integers, 500 characters, and 20 passengers as follows:

```
#include <vector>
using namespace std;                    // make std accessible

vector<int> scores(100);                // 100 integer scores
vector<char> buffer(500);               // buffer of 500 characters
vector<Passenger> passenList(20);       // list of 20 Passengers
```

As usual, the include statement provides the necessary declarations for using the vector class. Each instance of an STL vector can only hold objects of one type.

STL vectors are superior to standard C++ arrays in many respects. First, as with arrays, individual elements can be indexed using the usual index operator ([ ]). They can also be accessed by the at member function. The advantage of the latter is that it performs range checking and generates an error exception if the index is out of bounds. (We discuss exceptions in Section 2.4.) Recall that standard arrays in C++ do not even know their size, and hence range checking is not even possible. In contrast, a vector object's size is given by its size member function. Unlike

standard arrays, one vector object can be assigned to another, which results in the contents of one vector object being copied to the other. A vector can be resized dynamically by calling the resize member function. We show several examples of uses of the STL vector class below.

```
int i = // ...
cout << scores[i];                    // index (range unchecked)
buffer.at(i) = buffer.at(2 * i);      // index (range checked)
vector<int> newScores = scores;       // copy scores to newScores
scores.resize(scores.size() + 10);    // add room for 10 more elements
```

We discuss the STL further in Chapter 3.

More on STL Strings

In Section 1.1.3, we introduced the STL string class. This class provides a number of useful utilities for manipulating character strings. Earlier, we discussed the use of the addition operator (" +") for concatenating strings, the operator " +=" for appending a string to the end of an existing string, the function size for determining the length of a string, and the indexing operator ("[]") for accessing individual characters of a string.

Let us present a few more string functions. In the table below, let s be an STL string, and let p be either an STL string or a standard C++ string. Let i and m be nonnegative integers. Throughout, we use i to denote the index of a position in a string and we use m to denote the number of characters involved in the operation. (A string's first character is at index $i = 0$.)

s.find(p)	Return the index of first occurrence of string p in s
s.find(p, i)	Return the index of first occurrence of string p in s on or after position i
s.substr(i,m)	Return the substring starting at position i of s and consisting of m characters
s.insert(i, p)	Insert string p just prior to index i in s
s.erase(i, m)	Remove the substring of length m starting at index i
s.replace(i, m, p)	Replace the substring of length m starting at index i with p
getline(is, s)	Read a single line from the input stream is and store the result in s

In order to indicate that a pattern string p is not found, the find function returns the special value string::npos. Strings can also be compared lexicographically, using the C++ comparison operators: $<$, $<=$, $>$, $>=$, $==$, and $!=$.

Here are some examples of the use of these functions.

```
string s = "a dog";                          // "a dog"
s += " is a dog";                            // "a dog is a dog"
cout << s.find("dog");                       // 2
cout << s.find("dog", 3);                    // 11
if (s.find("doug") == string::npos) { }      // true
cout << s.substr(7, 5);                      // "s a d"
s.replace(2, 3, "frog");                     // "a frog is a dog"
s.erase(6, 3);                               // "a frog a dog"
s.insert(0, "is ");                          // "is a frog a dog"
if (s == "is a frog a dog") { }              // true
if (s < "is a frog a toad") { }              // true
if (s < "is a frog a cat") { }               // false
```

1.6 C++ Program and File Organization

Let us now consider the broader issue of how to organize an entire C++ program. A typical large C++ program consists of many files, with related pieces of code residing within each file. For example, C++ programmers commonly place each major class in its own file.

Source Files

There are two common file types, source files and header files. ***Source files*** typically contain most of the executable statements and data definitions. This includes the bodies of functions and definitions of any global variables.

Different compilers use different file naming conventions. Source file names typically have distinctive suffixes, such as ".cc", ".cpp", and ".C". Source files may be compiled separately by the compiler, and then these files are combined into one program by a system program called a ***linker***.

Each nonconstant global variable and function may be defined only once. Other source files may share such a global variable or function provided they have a matching declaration. To indicate that a global variable is defined in another file, the type specifier "**extern**" is added. This keyword is not needed for functions. For example, consider the declarations extracted from two files below. The file Source1.cpp defines a global variable cat and function foo. The file Source2.cpp can access these objects by including the appropriate matching declarations and adding "**extern**" for variables.

File: Source1.cpp

```
int cat = 1;                          // definition of cat
int foo(int x) { return x+1; }        // definition of foo
```

File: Source2.cpp

```
extern int cat;                    // cat is defined elsewhere
int foo(int x);                    // foo is defined elsewhere
```

Header Files

Since source files using shared objects must provide identical declarations, we commonly store these shared declarations in a ***header file***, which is then read into each such source file using an #include statement. Statements beginning with # are handled by a special program, called the ***preprocessor***, which is invoked automatically by the compiler. A header file typically contains many declarations, including classes, structures, constants, enumerations, and typedefs. Header files generally do not contain the definition (body) of a function. In-line functions are an exception, however, as their bodies are given in a header file.

Except for some standard library headers, the convention is that header file names end with a ".h" suffix. Standard library header files are indicated with angle brackets, as in <iostream>, while other local header files are indicated using quotes, as in "myIncludes.h".

```
#include <iostream>                // system include file
#include "myIncludes.h"            // user-defined include file
```

As a general rule, we should avoid including namespace **using** directives in header files, because any source file that includes such a header file has its namespace expanded as a result. We make one exception to this in our examples, however. Some of our header files include a **using** directive for the STL string class because it is so useful.

1.6.1 An Example Program

To make this description more concrete, let us consider an example of a simple yet complete C++ program. Our example consists of one class, called CreditCard, which defines a credit card object and a procedure that uses this class.

The CreditCard Class

The credit card object defined by CreditCard is a simplified version of a traditional credit card. It has an identifying number, identifying information about the owner, and information about the credit limit and the current balance. It does not charge interest or late payments, but it does restrict charges that would cause a card's balance to go over its spending limit.

The main class structure is presented in the header file CreditCard.h and is shown in Code Fragment 1.2.

```
#ifndef CREDIT_CARD_H                          // avoid repeated expansion
#define CREDIT_CARD_H

#include <string>                              // provides string
#include <iostream>                            // provides ostream

class CreditCard {
public:
  CreditCard(const std::string& no,            // constructor
        const std::string& nm, int lim, double bal=0);
                                               // accessor functions
  std::string    getNumber() const     { return number; }
  std::string    getName() const       { return name; }
  double         getBalance() const    { return balance; }
  int            getLimit() const      { return limit; }

  bool chargeIt(double price);                 // make a charge
  void makePayment(double payment);            // make a payment
private:                                       // private member data
  std::string    number;                       // credit card number
  std::string    name;                         // card owner's name
  int            limit;                        // credit limit
  double         balance;                      // credit card balance
};
                                               // print card information
std::ostream& operator<<(std::ostream& out, const CreditCard& c);
#endif
```

Code Fragment 1.2: The header file CreditCard.h, which contains the definition of class CreditCard.

Before discussing the class, let us say a bit about the general file structure. The first two lines (containing #ifndef and #define) and the last line (containing #endif) are used to keep the same header file from being expanded twice. We discuss this later. The next lines include the header files for strings and standard input and output.

This class has four private data members. We provide a simple constructor to initialize these members. There are four ***accessor functions***, which provide access to read the current values of these member variables. Of course, we could have alternately defined the member variables as being public and saved the work of providing these accessor functions. However, this would allow users to modify any of these member variables directly. We usually prefer to restrict the modification of member variables to special ***update functions***. We include two such update functions, chargeIt and makePayment. We have also defined a stream output operator for the class.

The accessor functions and makePayment are short, so we define them within the class body. The other member functions and the output operator are defined outside the class in the file CreditCard.cpp, shown in Code Fragment 1.3. This approach of defining a header file with the class definition and an associated source file with the longer member function definitions is common in C++.

The Main Test Program

Our main program is in the file TestCard.cpp. It consists of a main function, but this function does little more than call the function testCard, which does all the work. We include CreditCard.h to provide the CreditCard declaration. We do not need to include iostream and string, since CreditCard.h does this for us, but it would not have hurt to do so.

The testCard function declares an array of pointers to CreditCard. We allocate three such objects and initialize them. We then perform a number of payments and print the associated information. We show the complete code for the Test class in Code Fragment 1.4.

The output of the Test class is sent to the standard output stream. We show this output in Code Fragment 1.5.

Avoiding Multiple Header Expansions

A typical C++ program includes many different header files, which often include other header files. As a result, the same header file may be expanded many times. Such repeated header expansion is wasteful and can result in compilation errors because of repeated definitions. To avoid this repeated expansion, most header files use a combination of preprocessor commands. Let us explain the process, illustrated in Code Fragment 1.2.

```
#include "CreditCard.h"                    // provides CreditCard

using namespace std;                       // make std:: accessible
                                           // standard constructor
CreditCard::CreditCard(const string& no, const string& nm, int lim, double bal) {
  number = no;
  name = nm;
  balance = bal;
  limit = lim;
}
                                           // make a charge
bool CreditCard::chargeIt(double price) {
  if (price + balance > double(limit))
    return false;                          // over limit
  balance += price;
  return true;                             // the charge goes through
}

void CreditCard::makePayment(double payment) { // make a payment
  balance -= payment;
}
                                           // print card information
ostream& operator<<(ostream& out, const CreditCard& c) {
  out << "Number = "    << c.getNumber()   << "\n"
      << "Name = "      << c.getName()     << "\n"
      << "Balance = "   << c.getBalance()  << "\n"
      << "Limit = "     << c.getLimit()    << "\n";
  return out;
}
```

Code Fragment 1.3: The file CreditCard.cpp, which contains the definition of the out-of-class member functions for class CreditCard.

```
#include <vector>                          // provides STL vector
#include "CreditCard.h"                    // provides CreditCard, cout, string

using namespace std;                       // make std accessible

void testCard() {                          // CreditCard test function
  vector<CreditCard*> wallet(10);          // vector of 10 CreditCard pointers
                                           // allocate 3 new cards
  wallet[0] = new CreditCard("5391 0375 9387 5309", "John Bowman", 2500);
  wallet[1] = new CreditCard("3485 0399 3395 1954", "John Bowman", 3500);
  wallet[2] = new CreditCard("6011 4902 3294 2994", "John Bowman", 5000);

  for (int j=1; j <= 16; j++) {            // make some charges
    wallet[0]->chargeIt(double(j));        // explicitly cast to double
    wallet[1]->chargeIt(2 * j);            // implicitly cast to double
    wallet[2]->chargeIt(double(3 * j));
  }

  cout << "Card payments:\n";
  for (int i=0; i < 3; i++) {              // make more charges
    cout << *wallet[i];
    while (wallet[i]->getBalance() > 100.0) {
      wallet[i]->makePayment(100.0);
      cout << "New balance = " << wallet[i]->getBalance() << "\n";
    }
    cout << "\n";
    delete wallet[i];                      // deallocate storage
  }
}

int main() {                               // main function
  testCard();
  return EXIT_SUCCESS;                     // successful execution
}
```

Code Fragment 1.4: The file TestCard.cpp.

Let us start with the second line. The #define statement defines a preprocessor variable CREDIT_CARD_H. This variable's name is typically based on the header file name, and by convention, it is written in all capitals. The name itself is not important as long as different header files use different names. The entire file is enclosed in a preprocessor "if" block starting with #ifndef on top and ending with #endif at the bottom. The "ifndef" is read "if not defined," meaning that the header file contents will be expanded only if the preprocessor variable CREDIT_CARD_H is ***not*** defined.

Here is how it works. The first time the header file is encountered, the variable

```
Card payments:
Number = 5391 0375 9387 5309
Name = John Bowman
Balance = 136
Limit = 2500
New balance = 36

Number = 3485 0399 3395 1954
Name = John Bowman
Balance = 272
Limit = 3500
New balance = 172
New balance = 72

Number = 6011 4902 3294 2994
Name = John Bowman
Balance = 408
Limit = 5000
New balance = 308
New balance = 208
New balance = 108
New balance = 8
```

Code Fragment 1.5: Sample program output.

CREDIT_CARD_H has not yet been seen, so the header file is expanded by the preprocessor. In the process of doing this expansion, the second line defines the variable CREDIT_CARD_H. Hence, any attempt to include the header file will find that CREDIT_CARD_H is defined, so the file will not be expanded.

Throughout this book we omit these preprocessor commands from our examples, but they should be included in each header file we write.

1.7 Writing a C++ Program

As with any programming language, writing a program in C++ involves three fundamental steps:

1. Design
2. Coding
3. Testing and Debugging.

We briefly discuss each of these steps in this section.

1.7.1 Design

The design step is perhaps the most important in the process of writing a program. In this step, we decide how to divide the workings of our program into classes, we decide how these classes will interact, what data each will store, and what actions each will perform. Indeed, one of the main challenges that beginning C++ programmers face is deciding what classes to define to do the work of their program. While general prescriptions are hard to come by, there are some general rules of thumb that we can apply when determining how to define our classes.

- ***Responsibilities***: Divide the work into different ***actors***, each with a different responsibility. Try to describe responsibilities using action verbs. These actors form the classes for the program.
- ***Independence***: Define the work for each class to be as independent from other classes as possible. Subdivide responsibilities between classes so that each class has autonomy over some aspect of the program. Give data (as member variables) to the class that has jurisdiction over the actions that require access to this data.
- ***Behaviors***: Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class are well understood by other classes with which it interacts. These behaviors define the member functions that this class performs. The set of behaviors for a class is sometimes referred to as a ***protocol***, since we expect the behaviors for a class to hold together as a cohesive unit.

Defining the classes, together with their member variables and member functions, determines the design of a C++ program. A good programmer will naturally develop greater skill in performing these tasks over time, as experience teaches him or her to notice patterns in the requirements of a program that match patterns that he or she has seen before.

1.7.2 Pseudo-Code

Programmers are often asked to describe algorithms in a way that is intended for human eyes only, prior to writing actual code. Such descriptions are called ***pseudo-code***. Pseudo-code is not a computer program, but is more structured than usual prose. Pseudo-code is a mixture of natural language and high-level programming constructs that describe the main ideas behind a generic implementation of a data structure or algorithm. There really is no precise definition of the ***pseudo-code*** language, however, because of its reliance on natural language. At the same time, to help achieve clarity, pseudo-code mixes natural language with standard programming language constructs. The programming language constructs we choose are those consistent with modern high-level languages such as C, C++, and Java.

These constructs include the following:

- ***Expressions:*** We use standard mathematical symbols to express numeric and Boolean expressions. We use the left arrow sign (←) as the assignment operator in assignment statements (equivalent to the = operator in C++) and we use the equal sign (=) as the equality relation in Boolean expressions (equivalent to the "==" relation in C++).
- ***Function declarations:*** **Algorithm** *name*(*arg1*,*arg2*, ...) declares a new function "name" and its arguments.
- ***Decision structures:*** **if** *condition* **then** *true-actions* [**else** *false-actions*]. We use indentation to indicate what actions should be included in the true-actions and false-actions.
- ***While-loops:*** **while** *condition* **do** *actions*. We use indentation to indicate what actions should be included in the loop actions.
- ***Repeat-loops:*** **repeat** *actions* **until** *condition*. We use indentation to indicate what actions should be included in the loop actions.
- ***For-loops:*** **for** *variable-increment-definition* **do** *actions*. We use indentation to indicate what actions should be included among the loop actions.
- ***Array indexing:*** $A[i]$ represents the ith cell in the array A. The cells of an n-celled array A are indexed from $A[0]$ to $A[n-1]$ (consistent with C++).
- ***Member function calls:*** *object.method*(*args*) (object is optional if it is understood).
- ***Function returns:*** **return** *value*. This operation returns the value specified to the method that called this one.
- ***Comments:*** { *Comment goes here.* }. We enclose comments in braces.

When we write pseudo-code, we must keep in mind that we are writing for a human reader, not a computer. Thus, we should strive to communicate high-level ideas, not low-level implementation details. At the same time, we should not gloss over important steps. Like many forms of human communication, finding the right balance is an important skill that is refined through practice.

1.7.3 Coding

As mentioned above, one of the key steps in coding up an object-oriented program is coding up the descriptions of classes and their respective data and member functions. In order to accelerate the development of this skill, we discuss various ***design patterns*** for designing object-oriented programs (see Section 2.1.3) at various points throughout this text. These patterns provide templates for defining classes and the interactions between these classes.

Many programmers do their initial coding not on a computer, but by using ***CRC cards***. Class-Responsibility-Collaborator (CRC) cards are simple index cards that subdivide the work required of a program. The main idea behind this tool is to

have each card represent a component, which will ultimately become a class in our program. We write the name of each component on the top of an index card. On the left-hand side of the card, we begin writing the responsibilities for this component. On the right-hand side, we list the collaborators for this component, that is, the other components that this component will have to interact with to perform its duties. The design process iterates through an action/actor cycle, where we first identify an action (that is, a responsibility), and we then determine an actor (that is, a component) that is best suited to perform that action. The design is complete when we have assigned all actions to actors.

By the way, in using index cards to begin our coding, we are assuming that each component will have a small set of responsibilities and collaborators. This assumption is no accident, since it helps keep our programs manageable.

An alternative to CRC cards is the use of UML (Unified Modeling Language) diagrams to express the organization of a program, and the use of pseudo-code to describe the algorithms. UML diagrams are a standard visual notation to express object-oriented software designs. Several computer-aided tools are available to build UML diagrams. Describing algorithms in pseudo-code, on the other hand, is a technique that we utilize throughout this book.

Once we have decided on the classes and their respective responsibilities for our programs, we are ready to begin coding. We create the actual code for the classes in our program by using either an independent text editor (such as emacs, notepad, or vi), or the editor embedded in an ***integrated development environment*** (IDE), such as Microsoft's Visual Studio and Eclipse.

Once we have completed coding for a program (or file), we then compile this file into working code by invoking a compiler. If our program contains syntax errors, they will be identified, and we will have to go back into our editor to fix the offending lines of code. Once we have eliminated all syntax errors and created the appropriate compiled code, we then run our program.

Readability and Style

Programs should be made easy to read and understand. Good programmers should therefore be mindful of their coding style and develop a style that communicates the important aspects of a program's design for both humans and computers. Much has been written about good coding style. Here are some of the main principles.

- Use meaningful names for identifiers. Try to choose names that can be read aloud and reflect the action, responsibility, or data each identifier is naming. The tradition in most C++ circles is to capitalize the first letter of each word in an identifier, except for the first word in an identifier for a variable or method. So, in this tradition, "Date," "Vector," and "DeviceManager"

would identify classes, and "isFull," "insertItem," "studentName," and "studentHeight" would respectively identify member functions and variables.

- Use named constants and enumerations instead of embedded values. Readability, robustness, and modifiability are enhanced if we include a series of definitions of named constant values in a class definition. These can then be used within this class and others to refer to special values for this class. Our convention is to fully capitalize such constants as shown below.

```
const int MIN_CREDITS = 12;        // min. credits in a term
const int MAX_CREDITS = 24;        // max. credits in a term
                                   // enumeration for year
enum Year { FRESHMAN, SOPHOMORE, JUNIOR, SENIOR };
```

- Indent statement blocks. Typically programmers indent each statement block by four spaces. (In this book, we typically use two spaces to avoid having our code overrun the book's margins.)
- Organize each class in a consistent order. In the examples in this book, we usually use the following order:
  1. Public types and nested classes
  2. Public member functions
  3. Protected member functions (internal utilities)
  4. Private member data

  Our class organizations do not always follow this convention. In particular, when we wish to emphasize the implementation details of a class, we present the private members first and the public functions afterwards.
- Use comments that add meaning to a program and explain ambiguous or confusing constructs. In-line comments are good for quick explanations and do not need to be sentences. Block comments are good for explaining the purpose of a method and complex code sections.

1.7.4 Testing and Debugging

Testing is the process of verifying the correctness of a program, while debugging is the process of tracking the execution of a program and discovering the errors in it. Testing and debugging are often the most time-consuming activity in the development of a program.

Testing

A careful testing plan is an essential part of writing a program. While verifying the correctness of a program over all possible inputs is usually not feasible, we should aim at executing the program on a representative subset of inputs. At the very minimum, we should make sure that every method in the program is tested

at least once (method coverage). Even better, each code statement in the program should be executed at least once (statement coverage).

Programs often tend to fail on ***special cases*** of the input. Such cases need to be carefully identified and tested. For example, when testing a method that sorts an array of integers (that is, arranges them in ascending order), we should consider the following inputs:

- The array has zero length (no elements)
- The array has one element
- All the elements of the array are the same
- The array is already sorted
- The array is reverse sorted

In addition to special inputs to the program, we should also consider special conditions for the structures used by the program. For example, if we use an array to store data, we should make sure that boundary cases, such as inserting/removing at the beginning or end of the subarray holding data, are properly handled. While it is essential to use hand-crafted test suites, it is also advantageous to run the program on a large collection of randomly generated inputs.

There is a hierarchy among the classes and functions of a program induced by the "caller-callee" relationship. Namely, a function A is above a function B in the hierarchy if A calls B. There are two main testing strategies, ***top-down*** and ***bottom-up***, which differ in the order in which functions are tested.

Bottom-up testing proceeds from lower-level functions to higher-level functions. Namely, bottom-level functions, which do not invoke other functions, are tested first, followed by functions that call only bottom-level functions, and so on. This strategy ensures that errors found in a method are not likely to be caused by lower-level functions nested within it.

Top-down testing proceeds from the top to the bottom of the method hierarchy. It is typically used in conjunction with ***stubbing***, a boot-strapping technique that replaces a lower-level method with a ***stub***, a replacement for the method that simulates the output of the original method. For example, if function A calls function B to get the first line of a file, we can replace B with a stub that returns a fixed string when testing A.

Debugging

The simplest debugging technique consists of using ***print statements*** (typically using the stream output operator, "<<") to track the values of variables during the execution of the program. The problem with this approach is that the print statements need to be removed or commented out before the program can be executed as part of a "production" software system.

A better approach is to run the program within a ***debugger***, which is a specialized environment for controlling and monitoring the execution of a program. The basic functionality provided by a debugger is the insertion of ***breakpoints*** within the code. When the program is executed within the debugger, it stops at each breakpoint. While the program is stopped, the current value of variables can be inspected. In addition to fixed breakpoints, advanced debuggers allow for specification of ***conditional breakpoints***, which are triggered only if a given expression is satisfied.

Many IDEs, such as Microsoft Visual Studio and Eclipse provide built-in debuggers.

1.8 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-1.1 Which of the following is ***not*** a valid C++ variable name? (There may be more than one.)

a. i_think_i_am_valid
b. i_may_have_2_many_digits_2_be_valid
c. _I_start_and_end_with_underscores_
d. I_Have_A_Dollar_$ign
e. I_AM_LONG_AND_HAVE_NO_LOWER_CASE_LETTERS

R-1.2 Write a pseudo-code description of a method for finding the smallest and largest numbers in an array of integers and compare that to a C++ function that would do the same thing.

R-1.3 Give a C++ definition of a **struct** called Pair that consists of two members. The first is an integer called first, and the second is a double called second.

R-1.4 What are the contents of string s after executing the following statements.

```
string s = "abc";
string t = "cde";
s += s + t[1] + s;
```

R-1.5 Consider the expression y + 2 * z ++ < 3 - w / 5. Add parentheses to show the precise order of evaluation given the C++ rules for operator precedence.

R-1.6 Consider the following attempt to allocate a 10-element array of pointers to doubles and initialize the associated double values to 0.0. Rewrite the following (***incorrect***) code to do this correctly. (Hint: Storage for the doubles needs to be allocated.)

```
double* dp[10]
for (int i = 0; i < 10; i++) dp[i] = 0.0;
```

R-1.7 Write a short C++ function that takes an integer n and returns the sum of all the integers smaller than n.

R-1.8 Write a short C++ function, isMultiple, that takes two positive **long** values, n and m, and returns true if and only if n is a multiple of m, that is, $n = mi$ for some integer i.

R-1.9 Write a C++ function printArray(A, m, n) that prints an $m \times n$ two-dimensional array A of integers, declared to be "int** A," to the standard output. Each of the m rows should appear on a separate line.

R-1.10 What (if anything) is different about the behavior of the following two functions f and g that increment a variable and print its value?

```
void f(int x)
  { std::cout << ++x; }
void g(int& x)
  { std::cout << ++x; }
```

R-1.11 Write a C++ class, Flower, that has three member variables of type **string**, **int**, and **float**, which respectively represent the name of the flower, its number of pedals, and price. Your class must include a constructor method that initializes each variable to an appropriate value, and your class should include functions for setting the value of each type, and getting the value of each type.

R-1.12 Modify the CreditCard class from Code Fragment 1.3 to check that the price argument passed to function chargeIt and the payment argument passed to function makePayment are positive.

R-1.13 Modify the CreditCard class from Code Fragment 1.2 to charge interest on each payment.

R-1.14 Modify the CreditCard class from Code Fragment 1.2 to charge a late fee for any payment that is past its due date.

R-1.15 Modify the CreditCard class from Code Fragment 1.2 to include ***modifier functions*** that allow a user to modify internal variables in a CreditCard class in a controlled manner.

R-1.16 Modify the declaration of the first **for** loop in the Test class in Code Fragment 1.4 so that its charges will eventually cause exactly one of the three credit cards to go over its credit limit. Which credit card is it?

R-1.17 Write a C++ class, AllKinds, that has three member variables of type **int**, **long**, and **float**, respectively. Each class must include a constructor function that initializes each variable to a nonzero value, and each class should include functions for setting the value of each type, getting the value of each type, and computing and returning the sum of each possible combination of types.

R-1.18 Write a short C++ function, isMultiple, that takes two **long** values, n and m, and returns **true** if and only if n is a multiple of m, that is, $n = m \cdot i$ for some integer i.

R-1.19 Write a short C++ function, isTwoPower, that takes an **int** i and returns **true** if and only if i is a power of 2. Do not use multiplication or division, however.

R-1.20 Write a short C++ function that takes an integer n and returns the sum of all the integers smaller than n.

R-1.21 Write a short C++ function that takes an integer n and returns the sum of all the odd integers smaller than n.

R-1.22 Write a short C++ function that takes a positive **double** value x and returns the number of times we can divide x by 2 before we get a number less than 2.

Creativity

C-1.1 Write a pseudo-code description of a method that reverses an array of n integers, so that the numbers are listed in the opposite order than they were before, and compare this method to an equivalent C++ method for doing the same thing.

C-1.2 Write a short C++ function that takes an array of **int** values and determines if there is a pair of numbers in the array whose product is even.

C-1.3 Write a C++ function that takes an STL vector of **int** values and determines if all the numbers are different from each other (that is, they are distinct).

C-1.4 Write a C++ function that takes an STL vector of **int** values and prints all the odd values in the vector.

C-1.5 Write a C++ function that takes an array containing the set of all integers in the range 1 to 52 and shuffles it into random order. Use the built-in function rand, which returns a pseudo-random integer each time it is called. Your function should output each possible order with equal probability.

C-1.6 Write a short C++ program that outputs all possible strings formed by using each of the characters `'a'`, `'b'`, `'c'`, `'d'`, `'e'`, and `'f'` exactly once.

C-1.7 Write a short C++ program that takes all the lines input to standard input and writes them to standard output in reverse order. That is, each line is output in the correct order, but the ordering of the lines is reversed.

C-1.8 Write a short C++ program that takes two arguments of type STL vector<double>, a and b, and returns the element-by-element product of a and b. That is, it returns a vector c of the same length such that $c[i] = a[i] \cdot b[i]$.

C-1.9 Write a C++ class Vector2, that stores the (x,y) coordinates of a two-dimensional vector, where x and y are of type **double**. Show how to override various C++ operators in order to implement the addition of two vectors (producing a vector result), the multiplication of a scalar times a vector (producing a vector result), and the dot product of two vectors (producing a double result).

C-1.10 Write an efficient C++ function that takes any integer value i and returns 2^i, as a **long** value. Your function should ***not*** multiply 2 by itself i times; there are much faster ways of computing 2^i.

C-1.11 The ***greatest common divisor***, or GCD, of two positive integers n and m is the largest number j, such that n and m are both multiples of j. Euclid proposed a simple algorithm for computing $\mathrm{GCD}(n,m)$, where $n > m$, which is based on a concept known as the Chinese Remainder Theorem. The main idea of the algorithm is to repeatedly perform modulo computations of consecutive pairs of the sequence that starts $(n,m,\ldots)$, until reaching zero. The last nonzero number in this sequence is the GCD of n and m. For example, for $n = 80,844$ and $m = 25,320$, the sequence is as follows:

$$\begin{aligned} 80,844 \bmod 25,320 &= 4,884 \\ 25,320 \bmod 4,884 &= 900 \\ 4,884 \bmod 900 &= 384 \\ 900 \bmod 384 &= 132 \\ 384 \bmod 132 &= 120 \\ 132 \bmod 120 &= 12 \\ 120 \bmod 12 &= 0 \end{aligned}$$

So, GCD of $80,844$ and $25,320$ is 12. Write a short C++ function to compute $\mathrm{GCD}(n,m)$ for two integers n and m.

Projects

P-1.1 A common punishment for school children is to write out the same sentence multiple times. Write a C++ stand-alone program that will write out the following sentence one hundred times: "I will always use object-oriented design." Your program should number each of the sentences and it should "accidentally" make eight different random-looking typos at various points in the listing, so that it looks like a human typed it all by hand.

P-1.2 Write a C++ program that, when given a starting day (Sunday through Saturday) as a string, and a four-digit year, prints a calendar for that year. Each month should contain the name of the month, centered over the dates for that month and a line containing the names of the days of the week, running from Sunday to Saturday. Each week should be printed on a separate line. Be careful to check for a leap year.

P-1.3 The ***birthday paradox*** says that the probability that two people in a room will have the same birthday is more than half as long as the number of

people in the room (n), is more than 23. This property is not really a paradox, but many people find it surprising. Design a C++ program that can test this paradox by a series of experiments on randomly generated birthdays, which test this paradox for $n = 5, 10, 15, 20, \ldots, 100$. You should run at least 10 experiments for each value of n and it should output, for each n, the number of experiments for that n, such that two people in that test have the same birthday.

Chapter Notes

For more detailed information about the C++ programming language and the Standard Template Library, we refer the reader to books by Stroustrup [91], Lippmann and Lajoie [67], Musser and Saini [81], and Horstmann [47]. Lippmann also wrote a short introduction to C++ [66]. For more advanced information of how to use C++'s features in the most effective manner, consult the books by Meyers [77, 76]. For an introduction to C++ assuming a background of C see the book by Pohl [84]. For an explanation of the differences between C++ and Java see the book by Budd [17].