CHAPTER 11

Structured Data

TOPICS

11.1 Abstract Data Types

CONCEPT: **Abstract data types (ADTs) are data types created by the programmer. ADTs have their own range (or domain) of data and their own sets of operations that may be performed on them.**

The term *abstract data type,* or ADT, is very important in computer science and is especially significant in object-oriented programming. This chapter introduces you to the structure, which is one of C++'s mechanisms for creating abstract data types.

Abstraction

An *abstraction* is a general model of something. It is a definition that includes only the general characteristics of an object. For example, the term "dog" is an abstraction. It defines a general type of animal. The term captures the essence of what all dogs are without specifying the detailed characteristics of any particular type of dog. According to *Webster's New Collegiate Dictionary*, a dog is a highly variable carnivorous domesticated mammal (*Canis familiaris*) probably descended from the common wolf.

In real life, however, there is no such thing as a mere "dog." There are specific types of dogs, each with its own set of characteristics. There are poodles, cocker spaniels, Great Danes,

rottweilers, and many other breeds. There are small dogs and large dogs. There are gentle dogs and ferocious dogs. They come in all shapes, sizes, and dispositions. A real-life dog is not abstract. It is concrete.

Data Types

C++ has several *primitive data types*, or data types that are defined as a basic part of the language, as shown in Table 11-1.

Table 11-1

`bool`	`int`	`unsigned long int`
`char`	`long int`	`float`
`unsigned char`	`unsigned short int`	`double`
`short int`	`unsigned int`	`long double`

A data type defines what values a variable may hold. Each data type listed in Table 11-1 has its own range of values, such as –32,768 to +32,767 for `shorts`, and so forth. Data types also define what values a variable may not hold. For example, integer variables may not be used to hold fractional numbers.

In addition to defining a range or domain of values that a variable may hold, data types also define the operations that may be performed on a value. All of the data types listed in Table 11-1 allow the following mathematical and relational operators to be used with them:

```
+   -   *   /   >   <   >=   <=   ==   !=
```

Only the integer data types, however, allow operations with the modulus operator (%). So, a data type defines what values an object may hold and the operations that may be performed on the object.

The primitive data types are abstract in the sense that a data type and an object of that data type are not the same thing. For example, consider the following variable definition:

```
int x = 1, y = 2, z = 3;
```

In the statement above the integer variables `x`, `y`, and `z` are defined. They are three separate instances of the data type `int`. Each variable has its own characteristics (`x` is set to 1, `y` is set to 2, and `z` is set to 3). In this example, the data type `int` is the abstraction, and the variables `x`, `y`, and `z` are concrete occurrences.

Abstract Data Types

An abstract data type (ADT) is a data type created by the programmer and is composed of one or more primitive data types. The programmer decides what values are acceptable for the data type, as well as what operations may be performed on the data type. In many cases, the programmer designs his or her own specialized operations.

For example, suppose a program is created to simulate a 12-hour clock. The program could contain three ADTs: `Hours`, `Minutes`, and `Seconds`. The range of values for the `Hours` data type would be the integers 1 through 12. The range of values for the `Minutes` and `Seconds` data types would be 0 through 59. If an `Hours` object is set to 12 and then incremented, it will then take on the value 1. Likewise if a `Minutes` object or a `Seconds` object is set to 59 and then incremented, it will take on the value 0.

Abstract data types often combine several values. In the clock program, the `Hours`, `Minutes`, and `Seconds` objects could be combined to form a single `Clock` object. In this chapter you will learn how to combine variables of primitive data types to form your own data structures, or ADTs.

11.2 Focus on Software Engineering: Combining Data into Structures

CONCEPT: **C++ allows you to group several variables together into a single item known as a structure.**

So far you've written programs that keep data in individual variables. If you need to group items together, C++ allows you to create arrays. The limitation of arrays, however, is that all the elements must be of the same data type. Sometimes a relationship exists between items of different types. For example, a payroll system might keep the variables shown in Table 11-2. These variables hold data for a single employee.

Table 11-2

Variable Definition	Data Held
`int empNumber;`	Employee number
`string name;`	Employee's name
`double hours;`	Hours worked
`double payRate;`	Hourly pay rate
`double grossPay;`	Gross pay

VideoNote
Creating a Structure

All of the variables listed in Table 11-2 are related because they can hold data about the same employee. Their definition statements, though, do not make it clear that they belong together. To create a relationship between variables, C++ gives you the ability to package them together into a *structure*.

Before a structure can be used, it must be declared. Here is the general format of a structure declaration:

```
struct tag
{
   variable declaration;
   // ... more declarations
   //     may follow...
};
```

The *tag* is the name of the structure. As you will see later, it's used like a data type name. The variable declarations that appear inside the braces declare *members* of the structure. Here is an example of a structure declaration that holds the payroll data listed in Table 11-2:

```
struct PayRoll
{
   int empNumber;     // Employee number
   string name;       // Employee's name
   double hours;      // Hours worked
   double payRate;    // Hourly pay rate
   double grossPay;   // Gross pay
};
```

This declaration declares a structure named `PayRoll`. The structure has five members: `empNumber`, `name`, `hours`, `payRate`, and `grossPay`.

WARNING! Notice that a semicolon is required after the closing brace of the structure declaration.

NOTE: In this text we begin the names of structure tags with an uppercase letter. Later you will see the same convention used with unions. This visually differentiates these names from the names of variables.

NOTE: The structure declaration shown contains three `double` members, each declared on a separate line. The three could also have been declared on the same line, as

```
struct PayRoll
{
   int empNumber;
   string name;
   double hours, payRate, grossPay;
};
```

Many programmers prefer to place each member declaration on a separate line, however, for increased readability.

It's important to note that the structure declaration in our example does not define a variable. It simply tells the compiler what a `PayRoll` structure is made of. In essence, it creates a new data type named `PayRoll`. You can define variables of this type with simple definition statements, just as you would with any other data type. For example, the following statement defines a variable named `deptHead`:

```
PayRoll deptHead;
```

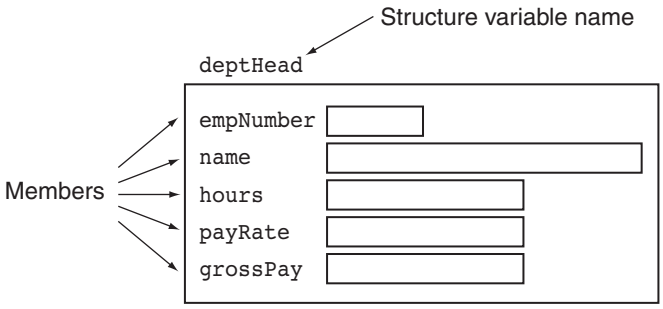
The data type of `deptHead` is the `PayRoll` structure. The structure tag, `PayRoll`, is listed before the variable name just as the word `int` or `double` would be listed to define variables of those types.

Remember that structure variables are actually made up of other variables known as members. Because `deptHead` is a `PayRoll` structure it contains the following members:

`empNumber`, an `int`
`name`, a `string` object
`hours`, a `double`
`payRate`, a `double`
`grossPay`, a `double`

Figure 11-1 illustrates this.

Figure 11-1

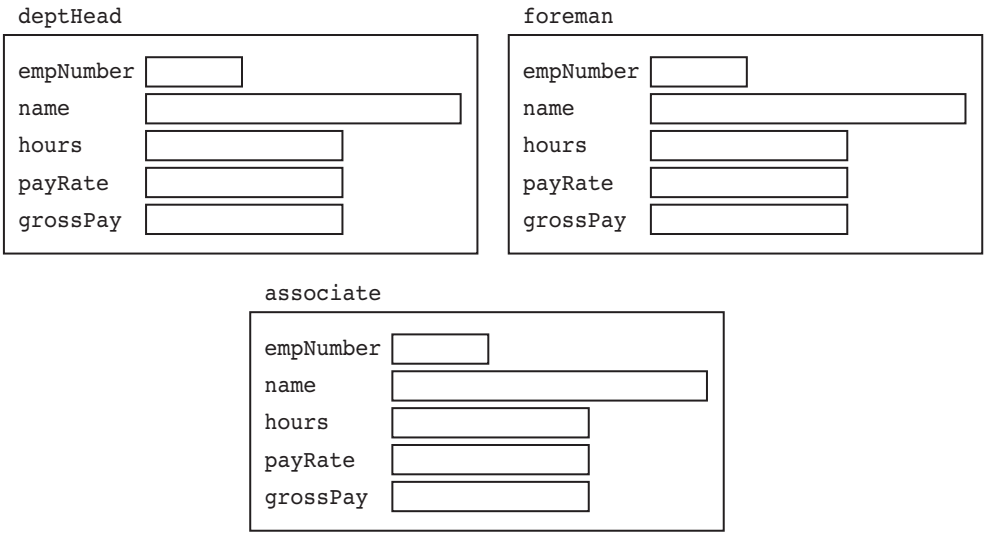


Just as it's possible to define multiple `int` or `double` variables, it's possible to define multiple structure variables in a program. The following statement defines three `PayRoll` variables: `deptHead`, `foreman`, and `associate`:

```
PayRoll deptHead, foreman, associate;
```

Figure 11-2 illustrates the existence of these three variables.

Figure 11-2



Each of the variables defined in this example is a separate *instance* of the `PayRoll` structure and contains its own members. An instance of a structure is a variable that exists in memory. It contains within it all the members described in the structure declaration.

Although the structure variables in the example are separate, each contains members with the same name. (In the next section you'll see how to access these members.) Here are some other examples of structure declarations and variable definitions:

```
struct Time
{
     int hour;
     int minutes;
     int seconds;
};

// Definition of the
// structure variable now.
Time now;
```

```
struct Date
{
    int day;
    int month;
    int year;
};

// Definition of the structure
// variable today.
Date today;
```

In review, there are typically two steps to implementing structures in a program:

- Create the structure declaration. This establishes the tag (or name) of the structure and a list of items that are members.
- Define variables (or instances) of the structure and use them in the program to hold data.

11.3 Accessing Structure Members

CONCEPT: **The *dot operator* (.) allows you to access structure members in a program.**

C++ provides the dot operator (a period) to access the individual members of a structure. Using our example of `deptHead` as a `PayRoll` structure variable, the following statement demonstrates how to access the `empNumber` member:

```
deptHead.empNumber = 475;
```

In this statement, the number 475 is assigned to the `empNumber` member of `deptHead`. The dot operator connects the name of the member variable with the name of the structure variable it belongs to. The following statements assign values to the `empNumber` members of the `deptHead`, `foreman`, and `associate` structure variables:

```
deptHead.empNumber = 475;
foreman.empNumber = 897;
associate.empNumber = 729;
```

With the dot operator you can use member variables just like regular variables. For example these statements display the contents of `deptHead`'s members:

```
cout << deptHead.empNumber << endl;
cout << deptHead.name << endl;
cout << deptHead.hours << endl;
cout << deptHead.payRate << endl;
cout << deptHead.grossPay << endl;
```

Program 11-1 is a complete program that uses the `PayRoll` structure.

Program 11-1

```
// This program demonstrates the use of structures.
#include <iostream>
#include <string>
#include <iomanip>
using namespace std;

struct PayRoll
{
   int empNumber;    // Employee number
   string name;      // Employee's name
   double hours;     // Hours worked
   double payRate;   // Hourly payRate
   double grossPay;  // Gross pay
};

int main()
{
    PayRoll employee; // employee is a PayRoll structure.

    // Get the employee's number.
    cout << "Enter the employee's number: ";
    cin >> employee.empNumber;

    // Get the employee's name.
    cout << "Enter the employee's name: ";
    cin.ignore();   // To skip the remaining '\n' character
    getline(cin, employee.name);

    // Get the hours worked by the employee.
    cout << "How many hours did the employee work? ";
    cin >> employee.hours;

    // Get the employee's hourly pay rate.
    cout << "What is the employee's hourly payRate? ";
    cin >> employee.payRate;

    // Calculate the employee's gross pay.
    employee.grossPay = employee.hours * employee.payRate;

    // Display the employee data.
    cout << "Here is the employee's payroll data:\n";
    cout << "Name: " << employee.name << endl;
    cout << "Number: " << employee.empNumber << endl;
    cout << "Hours worked: " << employee.hours << endl;
    cout << "Hourly payRate: " << employee.payRate << endl;
    cout << fixed << showpoint << setprecision(2);
    cout << "Gross Pay: $" << employee.grossPay << endl;
   return 0;
}
```

(program output continues)

Program 11-1 *(continued)*

Program Output with Example Input Shown in Bold

```
Enter the employee's number: 489 [Enter]
Enter the employee's name: Jill Smith [Enter]
How many hours did the employee work? 40 [Enter]
What is the employee's hourly pay rate? 20 [Enter]
Here is the employee's payroll data:
Name: Jill Smith
Number: 489
Hours worked: 40
Hourly pay rate: 20
Gross pay: $800.00
```

NOTE: Program 11-1 has the following call, in line 26, to `cin`'s ignore member function:

```
cin.ignore();
```

Recall that the `ignore` function causes `cin` to ignore the next character in the input buffer. This is necessary for the `getline` function to work properly in the program.

NOTE: The contents of a structure variable cannot be displayed by passing the entire variable to `cout`. For example, assuming `employee` is a `PayRoll` structure variable, the following statement will not work:

```
cout << employee << endl;  // Will not work!
```

Instead, each member must be separately passed to cout.

As you can see from Program 11-1, structure members that are of a primitive data type can be used with `cin`, `cout`, mathematical statements, and any operation that can be performed with regular variables. The only difference is that the structure variable name and the dot operator must precede the name of a member. Program 11-2 shows the member of a structure variable being passed to the `pow` function.

Program 11-2

```
1  // This program stores data about a circle in a structure.
2  #include <iostream>
3  #include <cmath> // For the pow function
4  #include <iomanip>
5  using namespace std;
6
7  // Constant for pi.
8  const double PI = 3.14159;
```

```
 9
10  // Structure declaration
11  struct Circle
12  {
13       double radius;     // A circle's radius
14       double diameter;   // A circle's diameter
15       double area;       // A circle's area
16  };
17
18  int main()
19  {
20       Circle c;     // Define a structure variable
21
22       // Get the circle's diameter.
23       cout << "Enter the diameter of a circle: ";
24       cin >> c.diameter;
25
26       // Calculate the circle's radius.
27       c.radius = c.diameter / 2;
28
29       // Calculate the circle's area.
30       c.area = PI * pow(c.radius, 2.0);
31
32       // Display the circle data.
33       cout << fixed << showpoint << setprecision(2);
34       cout << "The radius and area of the circle are:\n";
35       cout << "Radius: " << c.radius << endl;
36       cout << "Area: " << c.area << endl;
37       return 0;
38  }
```

Program Output with Example Input Shown in Bold

```
Enter the diameter of a circle: 10 [Enter]
The radius and area of the circle are:
Radius: 5
Area: 78.54
```

Comparing Structure Variables

You cannot perform comparison operations directly on structure variables. For example, assume that `circle1` and `circle2` are `Circle` structure variables. The following statement will cause an error.

```
if (circle1 == circle2)  // Error!
```

In order to compare two structures, you must compare the individual members, as shown in the following code.

```
if (circle1.radius == circle2.radius &&
    circle1.diameter == circle2.diameter &&
    circle1.area == circle2.area)
```

11.4 Initializing a Structure

CONCEPT: The members of a structure variable may be initialized with starting values when the structure variable is defined.

A structure variable may be initialized when it is defined, in a fashion similar to the initialization of an array. Assume the following structure declaration exists in a program:

```
struct CityInfo
{
   string cityName;
   string state;
   long population;
   int distance;
};
```

A variable may then be defined with an initialization list, as shown in the following:

```
CityInfo location = {"Asheville", "NC", 50000, 28};
```

This statement defines the variable `location`. The first value in the initialization list is assigned to the first declared member, the second value in the initialization list is assigned to the second member, and so on. The `location` variable is initialized in the following manner:

The string "Asheville" is assigned to `location.cityName`
The string "NC" is assigned to `location.state`
50000 is assigned to `location.population`
28 is assigned to `location.distance`

You do not have to provide initializers for all the members of a structure variable. For example, the following statement only initializes the `cityName` member of `location`:

```
CityInfo location = {"Tampa"};
```

The `state`, `population`, and `distance` members are left uninitialized. The following statement only initializes the `cityName` and `state` members, while leaving `population` and `distance` uninitialized:

```
CityInfo location = {"Atlanta", "GA"};
```

If you leave a structure member uninitialized, you must leave all the members that follow it uninitialized as well. C++ does not provide a way to skip members in a structure. For example, the following statement, which attempts to skip the initialization of the `population` member, is *not* legal:

```
CityInfo location = {"Knoxville", "TN", , 90};  // Illegal!
```

Program 11-3 demonstrates the use of partially initialized structure variables.

Program 11-3

```
1  // This program demonstrates partially initialized
2  // structure variables.
3  #include <iostream>
```

```
#include <string>
#include <iomanip>
using namespace std;

struct EmployeePay
{
     string name;         // Employee name
     int empNum;          // Employee number
     double payRate;      // Hourly pay rate
     double hours;        // Hours worked
     double grossPay;     // Gross pay
};

int main()
{
     EmployeePay employee1 = {"Betty Ross", 141, 18.75};
     EmployeePay employee2 = {"Jill Sandburg", 142, 17.50};

     cout << fixed << showpoint << setprecision(2);

     // Calculate pay for employee1
     cout << "Name: " << employee1.name << endl;
     cout << "Employee Number: " << employee1.empNum << endl;
     cout << "Enter the hours worked by this employee: ";
     cin >> employee1.hours;
     employee1.grossPay = employee1.hours * employee1.payRate;
     cout << "Gross Pay: " << employee1.grossPay << endl << endl;

     // Calculate pay for employee2
     cout << "Name: " << employee2.name << endl;
     cout << "Employee Number: " << employee2.empNum << endl;
     cout << "Enter the hours worked by this employee: ";
     cin >> employee2.hours;
     employee2.grossPay = employee2.hours * employee2.payRate;
     cout << "Gross Pay: " << employee2.grossPay << endl;
     return 0;
}
```

Program Output with Example Input Shown in Bold

```
Name: Betty Ross
Employee Number: 141
Enter the hours worked by this employee: 40 [Enter]
Gross Pay: 750.00

Name: Jill Sandburg
Employee Number: 142
Enter the hours worked by this employee: 20 [Enter]
Gross Pay: 350.00
```

It's important to note that you cannot initialize a structure member in the declaration of the structure. For instance, the following declaration is illegal:

```
// Illegal structure declaration
struct CityInfo
```

```
{
   string cityName = "Asheville";   // Error!
   string state = "NC";             // Error!
   long population = 50000;         // Error!
   int distance = 28;               // Error!
};
```

Remember that a structure declaration doesn't actually create the member variables. It only declares what the structure "looks like." The member variables are created in memory when a structure variable is defined. Because no variables are created by the structure declaration, there's nothing that can be initialized there.

Checkpoint

11.1 Write a structure declaration to hold the following data about a savings account:

Account Number (`string` object)
Account Balance (`double`)
Interest Rate (`double`)
Average Monthly Balance (`double`)

11.2 Write a definition statement for a variable of the structure you declared in Question 11.1. Initialize the members with the following data:

Account Number: ACZ42137-B12-7
Account Balance: $4512.59
Interest Rate: 4%
Average Monthly Balance: $4217.07

11.3 The following program skeleton, when complete, asks the user to enter these data about his or her favorite movie:

Name of movie
Name of the movie's director
Name of the movie's producer
The year the movie was released

Complete the program by declaring the structure that holds this data, defining a structure variable, and writing the individual statements necessary.

```
#include <iostream>
using namespace std;
// Write the structure declaration here to hold the movie data.

int main()
{
   // define the structure variable here.

   cout << "Enter the following data about your\n";
   cout << "favorite movie.\n";
   cout << "name: ";
   // Write a statement here that lets the user enter the
   // name of a favorite movie. Store the name in the
   // structure variable.
   cout << "Director: ";
   // Write a statement here that lets the user enter the
   // name of the movie's director. Store the name in the
   // structure variable.
```

```
    cout << "Producer: ";
    // Write a statement here that lets the user enter the
    // name of the movie's producer. Store the name in the
    // structure variable.
    cout << "Year of release: ";
    // Write a statement here that lets the user enter the
    // year the movie was released. Store the year in the
    // structure variable.
    cout << "Here is data on your favorite movie:\n";
    // Write statements here that display the data.
    // just entered into the structure variable.
    return 0;
}
```

11.5 Arrays of Structures

CONCEPT: Arrays of structures can simplify some programming tasks.

In Chapter 7 you saw that data can be stored in two or more arrays, with a relationship established between the arrays through their subscripts. Because structures can hold several items of varying data types, a single array of structures can be used in place of several arrays of regular variables.

An array of structures is defined like any other array. Assume the following structure declaration exists in a program:

```
struct BookInfo
{
   string title;
   string author;
   string publisher;
   double price;
};
```

The following statement defines an array, `bookList`, that has 20 elements. Each element is a `BookInfo` structure.

```
BookInfo bookList[20];
```

Each element of the array may be accessed through a subscript. For example, `bookList[0]` is the first structure in the array, `bookList[1]` is the second, and so forth. To access a member of any element, simply place the dot operator and member name after the subscript. For example, the following expression refers to the `title` member of `bookList[5]`:

```
bookList[5].title
```

The following loop steps through the array, displaying the data stored in each element:

```
for (int index = 0; index < 20; index++)
{
   cout << bookList[index].title << endl;
   cout << bookList[index].author << endl;
   cout << bookList[index].publisher << endl;
   cout << bookList[index].price << endl << endl;
}
```

Program 11-4 calculates and displays payroll data for three employees. It uses a single array of structures.

Program 11-4

```
// This program uses an array of structures.
#include <iostream>
#include <iomanip>
using namespace std;

struct PayInfo
{
    int hours;       // Hours worked
    double payRate;  // Hourly pay rate
};

int main()
{
    const int NUM_WORKERS = 3;    // Number of workers
    PayInfo workers[NUM_WORKERS]; // Array of structures
    int index;                    // Loop counter

    // Get employee pay data.
    cout << "Enter the hours worked by " << NUM_WORKERS
         << " employees and their hourly rates.\n";

    for (index = 0; index < NUM_WORKERS; index++)
    {
        // Get the hours worked by an employee.
        cout << "Hours worked by employee #" << (index + 1);
        cout << ": ";
        cin >> workers[index].hours;

        // Get the employee's hourly pay rate.
        cout << "Hourly pay rate for employee #";
        cout << (index + 1) << ": ";
        cin >> workers[index].payRate;
        cout << endl;
    }

    // Display each employee's gross pay.
    cout << "Here is the gross pay for each employee:\n";
    cout << fixed << showpoint << setprecision(2);
    for (index = 0; index < NUM_WORKERS; index++)
    {
        double gross;
        gross = workers[index].hours * workers[index].payRate;
        cout << "Employee #" << (index + 1);
        cout << ": $" << gross << endl;
    }
    return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter the hours worked by 3 employees and their hourly rates.
Hours worked by employee #1: 10 [Enter]
Hourly pay rate for employee #1: 9.75 [Enter]

Hours worked by employee #2: 20 [Enter]
Hourly pay rate for employee #2: 10.00 [Enter]

Hours worked by employee #3: 40 [Enter]
Hourly pay rate for employee #3: 20.00 [Enter]

Here is the gross pay for each employee:
Employee #1: $97.50
Employee #2: $200.00
Employee #3: $800.00
```

Initializing a Structure Array

To initialize a structure array, simply provide an initialization list for one or more of the elements. For example, the array in Program 11-4 could have been initialized as follows:

```
PayInfo workers[NUM_WORKERS] = {
                                  {10, 9.75 },
                                  {15, 8.62 },
                                  {20, 10.50},
                                  {40, 18.75},
                                  {40, 15.65}
                               };
```

As in all single-dimensional arrays, you can initialize all or part of the elements in an array of structures, as long as you do not skip elements.

11.6 Focus on Software Engineering: Nested Structures

CONCEPT: **It's possible for a structure variable to be a member of another structure variable.**

Sometimes it's helpful to nest structures inside other structures. For example, consider the following structure declarations:

```
struct Costs
{
   double wholesale;
   double retail;
};

struct Item
{
   string partNum;
   string description;
   Costs pricing;
};
```

The `Costs` structure has two members: `wholesale` and `retail`, both `doubles`. Notice that the third member of the `Item` structure, `pricing`, is a `Costs` structure. Assume the variable `widget` is defined as follows:

```
Item widget;
```

The following statements show examples of accessing members of the `pricing` variable, which is inside `widget`:

```
widget.pricing.wholesale = 100.0;
widget.pricing.retail = 150.0;
```

Program 11-5 gives a more elaborate illustration of nested structures.

Program 11-5

```
1  // This program uses nested structures.
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  // The Date structure holds data about a date.
7  struct Date
8  {
9      int month;
10     int day;
11     int year;
12 };
13
14 // The Place structure holds a physical address.
15 struct Place
16 {
17     string address;
18     string city;
19     string state;
20     string zip;
21 };
22
23 // The EmployeeInfo structure holds an employee's data.
24 struct EmployeeInfo
25 {
26     string name;
27     int employeeNumber;
28     Date birthDate;             // Nested structure
29     Place residence;            // Nested structure
30 };
31
32 int main()
33 {
34     // Define a structure variable to hold info about the manager.
35     EmployeeInfo manager;
36
```

```
37      // Get the manager's name and employee number
38      cout << "Enter the manager's name: ";
39      getline(cin, manager.name);
40      cout << "Enter the manager's employee number: ";
41      cin >> manager.employeeNumber;
42
43      // Get the manager's birth date
44      cout << "Now enter the manager's date of birth.\n";
45      cout << "Month (up to 2 digits): ";
46      cin >> manager.birthDate.month;
47      cout << "Day (up to 2 digits): ";
48      cin >> manager.birthDate.day;
49      cout << "Year: ";
50      cin >> manager.birthDate.year;
51      cin.ignore();  // Skip the remaining newline character
52
53      // Get the manager's residence information
54      cout << "Enter the manager's street address: ";
55      getline(cin, manager.residence.address);
56      cout << "City: ";
57      getline(cin, manager.residence.city);
58      cout << "State: ";
59      getline(cin, manager.residence.state);
60      cout << "ZIP Code: ";
61      getline(cin, manager.residence.zip);
62
63      // Display the information just entered
64      cout << "\nHere is the manager's information:\n";
65      cout << manager.name << endl;
66      cout << "Employee number " << manager.employeeNumber << endl;
67      cout << "Date of birth: ";
68      cout << manager.birthDate.month << "-";
69      cout << manager.birthDate.day << "-";
70      cout << manager.birthDate.year << endl;
71      cout << "Place of residence:\n";
72      cout << manager.residence.address << endl;
73      cout << manager.residence.city << ", ";
74      cout << manager.residence.state << " ";
75      cout << manager.residence.zip << endl;
76      return 0;
77  }
```

Program Output with Example Input Shown in Bold

```
Enter the manager's name: John Smith [Enter]
Enter the manager's employee number: 789 [Enter]
Now enter the manager's date of birth.
Month (up to 2 digits): 10 [Enter]
Day (up to 2 digits): 14 [Enter]
Year: 1970 [Enter]
Enter the manager's street address: 190 Disk Drive [Enter]
City: Redmond [Enter]
```

(program output continues)

Program 11-5 *(continued)*

```
State: WA [Enter]
ZIP Code: 98052 [Enter]
Here is the manager's information:
John Smith
Employee number 789
Date of birth: 10-14-1970
Place of residence:
190 Disk Drive
Redmond, WA 98052
```

Checkpoint

For Questions 11.4–11.7 below, assume the `Product` structure is declared as follows:

```
struct Product
{
     string description;   // Product description
     int partNum;          // Part number
     double cost;          // Product cost
};
```

11.4 Write a definition for an array of 100 `Product` structures. Do not initialize the array.

11.5 Write a loop that will step through the entire array you defined in Question 11.4, setting all the product descriptions to an empty string, all part numbers to zero, and all costs to zero.

11.6 Write the statements that will store the following data in the first element of the array you defined in Question 11.4:

Description: Claw hammer
Part Number: 547
Part Cost: $8.29

11.7 Write a loop that will display the contents of the entire array you created in Question 11.4.

11.8 Write a structure declaration named `Measurement`, with the following members:

`miles`, an integer
`meters`, a `long` integer

11.9 Write a structure declaration named `Destination`, with the following members:

`city`, a `string` object
`distance`, a `Measurement` structure (declared in Question 11.8)

Also define a variable of this structure type.

11.10 Write statements that store the following data in the variable you defined in Question 11.9:

City: Tupelo
Miles: 375
Meters: 603,375

11.7 Structures as Function Arguments

CONCEPT: **Structure variables may be passed as arguments to functions.**

VideoNote
Passing a Structure to a Function

Like other variables, the individual members of a structure variable may be used as function arguments. For example, assume the following structure declaration exists in a program:

```
struct Rectangle
{
   double length;
   double width;
   double area;
};
```

Let's say the following function definition exists in the same program:

```
double multiply(double x, double y)
{
   return x * y;
}
```

Assuming that `box` is a variable of the `Rectangle` structure type, the following function call will pass `box.length` into `x` and `box.width` into `y`. The return value will be stored in `box.area`.

```
box.area = multiply(box.length, box.width);
```

Sometimes it's more convenient to pass an entire structure variable into a function instead of individual members. For example, the following function definition uses a `Rectangle` structure variable as its parameter:

```
void showRect(Rectangle r)
{
   cout << r.length << endl;
   cout << r.width << endl;
   cout << r.area << endl;
}
```

The following function call passes the `box` variable into `r`:

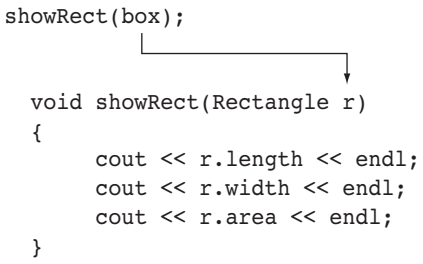
```
showRect(box);
```

Inside the function `showRect`, `r`'s members contain a copy of `box`'s members. This is illustrated in Figure 11-3.

Once the function is called, `r.length` contains a copy of `box.length`, `r.width` contains a copy of `box.width`, and `r.area` contains a copy of `box.area`.

Structures, like all variables, are normally passed by value into a function. If a function is to access the members of the original argument, a reference variable may be used as the parameter. Program 11-6 uses two functions that accept structures as arguments. Arguments are passed to the `getItem` function by reference and to the `showItem` function by value.

Figure 11-3



Program 11-6

```
1  // This program has functions that accept structure variables
2  // as arguments.
3  #include <iostream>
4  #include <string>
5  #include <iomanip>
6  using namespace std;
7
8  struct InventoryItem
9  {
10     int partNum;                 // Part number
11     string description;          // Item description
12     int onHand;                  // Units on hand
13     double price;                // Unit price
14 };
15
16 // Function Prototypes
17 void getItem(InventoryItem&);    // Argument passed by reference
18 void showItem(InventoryItem);    // Argument passed by value
19
20 int main()
21 {
22     InventoryItem part;
23
24     getItem(part);
25     showItem(part);
26     return 0;
27 }
28
29 //*********************************************************
30 // Definition of function getItem. This function uses     *
31 // a structure reference variable as its parameter. It asks *
32 // the user for information to store in the structure.    *
33 //*********************************************************
34
```

```
35  void getItem(InventoryItem &p)  // Uses a reference parameter
36  {
37       // Get the part number.
38       cout << "Enter the part number: ";
39       cin >> p.partNum;
40
41       // Get the part description.
42       cout << "Enter the part description: ";
43       cin.ignore();  // Ignore the remaining newline character
44       getline(cin, p.description);
45
46       // Get the quantity on hand.
47       cout << "Enter the quantity on hand: ";
48       cin >> p.onHand;
49
50       // Get the unit price.
51       cout << "Enter the unit price: ";
52       cin >> p.price;
53  }
54
55  //*********************************************************
56  // Definition of function showItem. This function accepts  *
57  // an argument of the InventoryItem structure type. The    *
58  // contents of the structure is displayed.                 *
59  //*********************************************************
60
61  void showItem(InventoryItem p)
62  {
63       cout << fixed << showpoint << setprecision(2);
64       cout << "Part Number: " << p.partNum << endl;
65       cout << "Description: " << p.description << endl;
66       cout << "Units On Hand: " << p.onHand << endl;
67       cout << "Price: $" << p.price << endl;
68  }
```

Program Output with Example Input Shown in Bold

```
Enter the part number: 800 [Enter]
Enter the part description: Screwdriver [Enter]
Enter the quantity on hand: 135 [Enter]
Enter the unit price: 1.25 [Enter]
Part Number: 800
Description: Screwdriver
Units on Hand: 135
Price: $1.25
```

Notice that the `InventoryItem` structure declaration in Program 11-6 appears before both the prototypes and the definitions of the `getItem` and `showItem` functions. This is because both functions use an `InventoryItem` structure variable as their parameter. The compiler must know what `InventoryItem` is before it encounters any definitions for variables of that type. Otherwise, an error will occur.

Constant Reference Parameters

Sometimes structures can be quite large. Passing large structures by value can decrease a program's performance because a copy of the structure has to be created. When a structure is passed by reference, however, it isn't copied. A reference that points to the original argument is passed instead. So, it's often preferable to pass large objects such as structures by reference.

Of course, the disadvantage of passing an object by reference is that the function has access to the original argument. It can potentially alter the argument's value. This can be prevented, however, by passing the argument as a constant reference. The `showItem` function from Program 11-6 is shown here, modified to use a constant reference parameter.

```
void showItem(const InventoryItem &p)
{
   cout << fixed << showpoint << setprecision(2);
   cout << "Part Number: " << p.partNum << endl;
   cout << "Description: " << p.description << endl;
   cout << "Units on Hand: " << p.onHand << endl;
   cout << "Price: $" << p.price << endl;
}
```

This version of the function is more efficient than the original version because the amount of time and memory consumed in the function call is reduced. Because the parameter is defined as a constant, the function cannot accidentally corrupt the value of the argument.

The prototype for this version of the function follows.

```
void showItem(const InventoryItem&);
```

11.8 Returning a Structure from a Function

CONCEPT: A function may return a structure.

Just as functions can be written to return an `int`, `long`, `double`, or other data type, they can also be designed to return a structure. Recall the following structure declaration from Program 11-2:

```
struct Circle
{
   double radius;
   double diameter;
   double area;
};
```

A function, such as the following, could be written to return a variable of the `Circle` data type:

```
Circle getCircleData()
{
   Circle temp;           // Temporary Circle structure

   temp.radius = 10.0;    // Store the radius
   temp.diameter = 20.0;  // Store the diameter
```

```
    temp.area = 314.159;   // Store the area
    return temp;           // Return the temporary structure
}
```

Notice that the `getCircleData` function has a return data type of `Circle`. That means the function returns an entire `Circle` structure when it terminates. The return value can be assigned to any variable that is a `Circle` structure. The following statement, for example, assigns the function's return value to the `Circle` structure variable named `myCircle`:

```
myCircle = getCircleData();
```

After this statement executes, `myCircle.radius` will be set to 10.0, `myCircle.diameter` will be set to 20.0, and `myCircle.area` will be set to 314.159.

When a function returns a structure, it is always necessary for the function to have a local structure variable to hold the member values that are to be returned. In the `getCircleData` function, the values for `diameter`, `radius`, and `area` are stored in the local variable `temp`. The `temp` variable is then returned from the function.

Program 11-7 is a modification of Program 11-2. The function `getInfo` gets the circle's diameter from the user and calculates the circle's radius. The diameter and radius are stored in a local structure variable, `round`, which is returned from the function.

Program 11-7

```
1  // This program uses a function to return a structure. This
2  // is a modification of Program 11-2.
3  #include <iostream>
4  #include <iomanip>
5  #include <cmath>  // For the pow function
6  using namespace std;
7
8  // Constant for pi.
9  const double PI = 3.14159;
10
11 // Structure declaration
12 struct Circle
13 {
14     double radius;       // A circle's radius
15     double diameter;     // A circle's diameter
16     double area;         // A circle's area
17 };
18
19 // Function prototype
20 Circle getInfo();
21
22 int main()
23 {
24     Circle c;         // Define a structure variable
25
26     // Get data about the circle.
27     c = getInfo();
```

(program continues)

Program 11-7 *(continued)*

```
28
29      // Calculate the circle's area.
30      c.area = PI * pow(c.radius, 2.0);
31
32      // Display the circle data.
33      cout << "The radius and area of the circle are:\n";
34      cout << fixed << setprecision(2);
35      cout << "Radius: " << c.radius << endl;
36      cout << "Area: " << c.area << endl;
37      return 0;
38  }
39
40  //*************************************************************
41  // Definition of function getInfo. This function uses a local   *
42  // variable, tempCircle, which is a circle structure. The user  *
43  // enters the diameter of the circle, which is stored in        *
44  // tempCircle.diameter. The function then calculates the radius *
45  // which is stored in tempCircle.radius. tempCircle is then     *
46  // returned from the function.                                  *
47  //*************************************************************
48
49  Circle getInfo()
50  {
51      Circle tempCircle;  // Temporary structure variable
52
53      // Store circle data in the temporary variable.
54      cout << "Enter the diameter of a circle: ";
55      cin >> tempCircle.diameter;
56      tempCircle.radius = tempCircle.diameter / 2.0;
57
58      // Return the temporary variable.
59      return tempCircle;
60  }
```

Program Output with Example Input Shown in Bold

```
Enter the diameter of a circle: 10 [Enter]
The radius and area of the circle are:
Radius: 5.00
Area: 78.54
```

NOTE: In Chapter 6 you learned that C++ only allows you to return a single value from a function. Structures, however, provide a way around this limitation. Even though a structure may have several members, it is technically a single value. By packaging multiple values inside a structure, you can return as many variables as you need from a function.

11.9 Pointers to Structures

CONCEPT: **You may take the address of a structure variable and create variables that are pointers to structures.**

Defining a variable that is a pointer to a structure is as simple as defining any other pointer variable: The data type is followed by an asterisk and the name of the pointer variable. Here is an example:

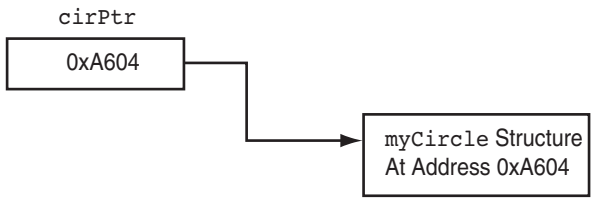
```
Circle *cirPtr = nullptr;
```

This statement defines `cirPtr` as a pointer to a `Circle` structure. Look at the following code:

```
Circle myCircle = { 10.0, 20.0, 314.159 };
Circle *cirPtr = nullptr;
cirPtr = &myCircle;
```

The first two lines define `myCircle`, a structure variable, and `cirPtr`, a pointer. The third line assigns the address of `myCircle` to `cirPtr`. After this line executes, `cirPtr` will point to the `myCircle` structure. This is illustrated in Figure 11-4.

Figure 11-4



Indirectly accessing the members of a structure through a pointer can be clumsy, however, if the indirection operator is used. One might think the following statement would access the `radius` member of the structure pointed to by `cirPtr`, but it doesn't:

```
*cirPtr.radius = 10;
```

The dot operator has higher precedence than the indirection operator, so the indirection operator tries to dereference `cirPtr.radius`, not `cirPtr`. To dereference the `cirPtr` pointer, a set of parentheses must be used.

```
(*cirPtr).radius = 10;
```

Because of the awkwardness of this notation, C++ has a special operator for dereferencing structure pointers. It's called the *structure pointer operator*, and it consists of a hyphen (-) followed by the greater-than symbol (>). The previous statement, rewritten with the structure pointer operator, looks like this:

```
cirPtr->radius = 10;
```

The structure pointer operator takes the place of the dot operator in statements using pointers to structures. The operator automatically dereferences the structure pointer on its left. There is no need to enclose the pointer name in parentheses.

NOTE: The structure pointer operator is supposed to look like an arrow, thus visually indicating that a "pointer" is being used.

Program 11-8 shows that a pointer to a structure may be used as a function parameter, allowing the function to access the members of the original structure argument.

Program 11-8

```
1  // This program demonstrates a function that uses a
2  // pointer to a structure variable as a parameter.
3  #include <iostream>
4  #include <string>
5  #include <iomanip>
6  using namespace std;
7
8  struct Student
9  {
10     string name;          // Student's name
11     int idNum;            // Student ID number
12     int creditHours;      // Credit hours enrolled
13     double gpa;           // Current GPA
14 };
15
16 void getData(Student *);  // Function prototype
17
18 int main()
19 {
20     Student freshman;
21
22     // Get the student data.
23     cout << "Enter the following student data:\n";
24     getData(&freshman);    // Pass the address of freshman.
25     cout << "\nHere is the student data you entered:\n";
26
27     // Now display the data stored in freshman
28     cout << setprecision(3);
29     cout << "Name: " << freshman.name << endl;
30     cout << "ID Number: " << freshman.idNum << endl;
31     cout << "Credit Hours: " << freshman.creditHours << endl;
32     cout << "GPA: " << freshman.gpa << endl;
33     return 0;
34 }
35
```

```
36  //*****************************************************
37  // Definition of function getData. Uses a pointer to a *
38  // Student structure variable. The user enters student *
39  // information, which is stored in the variable.       *
40  //*****************************************************
41
42  void getData(Student *s)
43  {
44       // Get the student name.
45       cout << "Student name: ";
46       getline(cin, s->name);
47
48       // Get the student ID number.
49       cout << "Student ID Number: ";
50       cin >> s->idNum;
51
52       // Get the credit hours enrolled.
53       cout << "Credit Hours Enrolled: ";
54       cin >> s->creditHours;
55
56       // Get the GPA.
57       cout << "Current GPA: ";
58       cin >> s->gpa;
59  }
```

Program Output with Example Input Shown in Bold

```
Enter the following student data:
Student Name: Frank Smith [Enter]
Student ID Number: 4876 [Enter]
Credit Hours Enrolled: 12 [Enter]
Current GPA: 3.45 [Enter]

Here is the student data you entered:
Name: Frank Smith
ID Number: 4876
Credit Hours: 12
GPA: 3.45
```

Dynamically Allocating a Structure

You can also use a structure pointer and the `new` operator to dynamically allocate a structure. For example, the following code defines a `Circle` pointer named `cirPtr` and dynamically allocates a `Circle` structure. Values are then stored in the dynamically allocated structure's members.

```
Circle *cirPtr = nullptr; // Define a Circle pointer
cirPtr = new Circle;      // Dynamically allocate a Circle structure
cirPtr->radius = 10;      // Store a value in the radius member
cirPtr->diameter = 20;    // Store a value in the diameter member
cirPtr->area = 314.159;   // Store a value in the area member
```

You can also dynamically allocate an array of structures. The following code shows an array of five `Circle` structures being allocated.

```
Circle *circles = nullptr;
circles = new Circle[5];
for (int count = 0; count < 5; count++)
{
   cout << "Enter the radius for circle "
        << (count + 1) << ": ";
   cin >> circles[count].radius;
}
```

11.10 Focus on Software Engineering: When to Use ., When to Use ->, and When to Use *

Sometimes structures contain pointers as members. For example, the following structure declaration has an `int` pointer member:

```
struct GradeInfo
{
   string name;                  // Student names
   int *testScores;              // Dynamically allocated array
   float average;                // Test average
};
```

It is important to remember that the structure pointer operator (`->`) is used to dereference a pointer to a structure, not a pointer that is a member of a structure. If a program dereferences the `testScores` pointer in this structure, the indirection operator must be used. For example, assume that the following variable has been defined:

```
GradeInfo student1;
```

The following statement will display the value pointed to by the `testScores` member:

```
cout << *student1.testScores;
```

It is still possible to define a pointer to a structure that contains a pointer member. For instance, the following statement defines `stPtr` as a pointer to a `GradeInfo` structure:

```
GradeInfo *stPtr = nullptr;
```

Assuming that `stPtr` points to a valid `GradeInfo` variable, the following statement will display the value pointed to by its `testScores` member:

```
cout << *stPtr->testScores;
```

In this statement, the `*` operator dereferences `stPtr->testScores`, while the `->` operator dereferences `stPtr`. It might help to remember that the following expression:

```
stPtr->testScores
```

is equivalent to

```
(*stPtr).testScores
```

So, the expression

```
*stPtr->testScores
```

is the same as

```
*(*stPtr).testScores
```

The awkwardness of this last expression shows the necessity of the -> operator. Table 11-3 lists some expressions using the *, ->, and . operators and describes what each references.

Table 11-3

Expression	Description
`s->m`	`s` is a structure pointer and `m` is a member. This expression accesses the `m` member of the structure pointed to by `s`.
`*a.p`	`a` is a structure variable and `p`, a pointer, is a member. This expression dereferences the value pointed to by `p`.
`(*s).m`	`s` is a structure pointer and `m` is a member. The `*` operator dereferences `s`, causing the expression to access the `m` member of the structure pointed to by `s`. This expression is the same as `s->m` .
`*s->p`	`s` is a structure pointer and `p`, a pointer, is a member of the structure pointed to by `s`. This expression accesses the value pointed to by `p`. (The `->` operator dereferences `s` and the `*` operator dereferences `p`.)
`*(*s).p`	`s` is a structure pointer and `p`, a pointer, is a member of the structure pointed to by `s`. This expression accesses the value pointed to by `p`. `(*s)` dereferences `s` and the outermost `*` operator dereferences `p`. The expression `*s->p` is equivalent.

Checkpoint

Assume the following structure declaration exists for Questions 11.11–11.15:

```
struct Rectangle
{
   int length;
   int width;
};
```

11.11 Write a function that accepts a `Rectangle` structure as its argument and displays the structure's contents on the screen.

11.12 Write a function that uses a `Rectangle` structure reference variable as its parameter and stores the user's input in the structure's members.

11.13 Write a function that returns a `Rectangle` structure. The function should store the user's input in the members of the structure before returning it.

11.14 Write the definition of a pointer to a `Rectangle` structure.

11.15 Assume `rptr` is a pointer to a `Rectangle` structure. Which of the expressions, A, B, or C, is equivalent to the following expression:

```
rptr->width
```

A) `*rptr.width`
B) `(*rptr).width`
C) `rptr.(*width)`

11.11 Unions

CONCEPT: A *union* is like a structure, except all the members occupy the same memory area.

A union, in almost all regards, is just like a structure. The difference is that all the members of a union use the same memory area, so only one member can be used at a time. A union might be used in an application where the program needs to work with two or more values (of different data types), but only needs to use one of the values at a time. Unions conserve memory by storing all their members in the same memory location.

Unions are declared just like structures, except the key word `union` is used instead of `struct`. Here is an example:

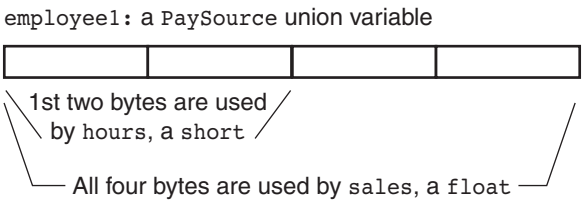
```
union PaySource
{
   short hours;
   float sales;
};
```

A union variable of the data type shown above can then be defined as

```
PaySource employee1;
```

The `PaySource` union variable defined here has two members: `hours` (a `short`), and `sales` (a `float`). The entire variable will only take up as much memory as the largest member (in this case, a `float`). The way this variable is stored on a typical PC is illustrated in Figure 11-5.

Figure 11-5



As shown in Figure 11-5, the union uses four bytes on a typical PC. It can store a `short` or a `float`, depending on which member is used. When a value is stored in the `sales` member, all four bytes are needed to hold the data. When a value is stored in the `hours` member, however, only the first two bytes are used. Obviously, both members can't hold values at the same time. This union is demonstrated in Program 11-9.

Program 11-9

```
1  // This program demonstrates a union.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  union PaySource
7  {
8       int hours;         // Hours worked
9       float sales;       // Amount of sales
10 };
11
12 int main()
13 {
14      PaySource employee1;     // Define a union variable
15      char payType;            // To hold the pay type
16      float payRate;           // Hourly pay rate
17      float grossPay;          // Gross pay
18
19      cout << fixed << showpoint << setprecision(2);
20      cout << "This program calculates either hourly wages or\n";
21      cout << "sales commission.\n";
22
23      // Get the pay type, hourly or commission.
24      cout << "Enter H for hourly wages or C for commission: ";
25      cin >> payType;
26
27      // Determine the gross pay, depending on the pay type.
28      if (payType == 'H' || payType == 'h')
29      {
30           // This is an hourly paid employee. Get the
31           // pay rate and hours worked.
32           cout << "What is the hourly pay rate? ";
33           cin >> payRate;
34           cout << "How many hours were worked? ";
35           cin >> employee1.hours;
36
37           // Calculate and display the gross pay.
38           grossPay = employee1.hours * payRate;
39           cout << "Gross pay: $" << grossPay << endl;
40      }
41      else if (payType == 'C' || payType == 'c')
42      {
43           // This is a commission-paid employee. Get the
44           // amount of sales.
45           cout << "What are the total sales for this employee? ";
46           cin >> employee1.sales;
47
48           // Calculate and display the gross pay.
49           grossPay = employee1.sales * 0.10;
50           cout << "Gross pay: $" << grossPay << endl;
51      }
```

(program continues)

Program 11-9 *(continued)*

```
52       else
53       {
54            // The user made an invalid selection.
55            cout << payType << " is not a valid selection.\n";
56       }
57       return 0;
58  }
```

Program Output with Example Input Shown in Bold

```
This program calculates either hourly wages or
sales commission.
Enter H for hourly wages or C for commission: C [Enter]
What are the total sales for this employee? 5000 [Enter]
Gross pay: $500.00
```

Program Output with Different Example Input Shown in Bold

```
This program calculates either hourly wages or
sales commission.
Enter H for hourly wages or C for commission: H [Enter]
What is the hourly pay rate? 20 [Enter]
How many hours were worked? 40 [Enter]
Gross pay: $800.00
```

Everything else you already know about structures applies to unions. For example, arrays of unions may be defined. A union may be passed as an argument to a function or returned from a function. Pointers to unions may be defined, and the members of the union referenced by the pointer can be accessed with the -> operator.

Anonymous Unions

The members of an anonymous union have names, but the union itself has no name. Here is the general format of an anonymous union declaration:

```
union
{
     member declaration;
     ...
};
```

An anonymous union declaration actually creates the member variables in memory, so there is no need to separately define a union variable. Anonymous unions are simple to use because the members may be accessed without the dot operator. Program 11-10, which is a modification of Program 11-9, demonstrates the use of an anonymous union.

Program 11-10

```
1  // This program demonstrates an anonymous union.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
```

```
 5
 6  int main()
 7  {
 8      union                       // Anonymous union
 9      {
10          int hours;
11          float sales;
12      };
13
14      char payType;               // To hold the pay type
15      float payRate;              // Hourly pay rate
16      float grossPay;             // Gross pay
17
18      cout << fixed << showpoint << setprecision(2);
19      cout << "This program calculates either hourly wages or\n";
20      cout << "sales commission.\n";
21
22      // Get the pay type, hourly or commission.
23      cout << "Enter H for hourly wages or C for commission: ";
24      cin >> payType;
25
26      // Determine the gross pay, depending on the pay type.
27      if (payType == 'H' || payType == 'h')
28      {
29          // This is an hourly paid employee. Get the
30          // pay rate and hours worked.
31          cout << "What is the hourly pay rate? ";
32          cin >> payRate;
33          cout << "How many hours were worked? ";
34          cin >> hours; // Anonymous union member
35
36          // Calculate and display the gross pay.
37          grossPay = hours * payRate;
38          cout << "Gross pay: $" << grossPay << endl;
39      }
40      else if (payType == 'C' || payType == 'c')
41      {
42          // This is a commission-paid employee. Get the
43          // amount of sales.
44          cout << "What are the total sales for this employee? ";
45          cin >> sales; // Anonymous union member
46
47          // Calculate and display the gross pay.
48          grossPay = sales * 0.10;
49          cout << "Gross pay: $" << grossPay << endl;
50      }
51      else
52      {
53          // The user made an invalid selection.
54          cout << payType << " is not a valid selection.\n";
55      }
56      return 0;
57  }
```

(program output continues)

Program 11-10 *(continued)*

Program Output with Example Input Shown in Bold

```
This program calculates either hourly wages or
sales commission.
Enter H for hourly wages or C for commission: C [Enter]
What are the total sales for this employee? 12000 [Enter]
Gross pay: $1200.00
```

NOTE: Notice the anonymous union in Program 11-10 is declared inside function `main`. If an anonymous union is declared globally (outside all functions), it must be declared static. This means the word `static` must appear before the word `union`.

Checkpoint

11.16 Declare a union named `ThreeTypes` with the following members:

`letter:`	A character
`whole:`	An integer
`real:`	A `double`

11.17 Write the definition for an array of 50 of the `ThreeTypes` structures you declared in Question 11.16.

11.18 Write a loop that stores the floating point value 2.37 in all the elements of the array you defined in Question 11.17.

11.19 Write a loop that stores the character 'A' in all the elements of the array you defined in Question 11.17.

11.20 Write a loop that stores the integer 10 in all the elements of the array you defined in Question 11.17.

11.12 Enumerated Data Types

CONCEPT: An enumerated data type is a programmer-defined data type. It consists of values known as enumerators, which represent integer constants.

Using the `enum` key word you can create your own data type and specify the values that belong to that type. Such a type is known as an *enumerated data type*. Here is an example of an enumerated data type declaration:

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
```

An enumerated type declaration begins with the key word `enum`, followed by the name of the type, followed by a list of identifiers inside braces, and is terminated with a semicolon. The example declaration creates an enumerated data type named `Day`. The identifiers `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, and `FRIDAY`, which are listed inside the braces,

are known as *enumerators*. They represent the values that belong to the `Day` data type. Here is the general format of an enumerated type declaration:

```
enum TypeName { One or more enumerators };
```

Note that the enumerators are not enclosed in quotation marks; therefore they are not strings. Enumerators must be legal C++ identifiers.

Once you have created an enumerated data type in your program, you can define variables of that type. For example, the following statement defines `workDay` as a variable of the `Day` type:

```
Day workDay;
```

Because `workDay` is a variable of the `Day` data type, we may assign any of the enumerators `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, or `FRIDAY` to it. For example, the following statement assigns the value `WEDNESDAY` to the `workDay` variable.

```
Day workDay = WEDNESDAY;
```

So just what are these enumerators `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, and `FRIDAY`? You can think of them as integer named constants. Internally, the compiler assigns integer values to the enumerators, beginning with 0. The enumerator `MONDAY` is stored in memory as the number 0, `TUESDAY` is stored in memory as the number 1, `WEDNESDAY` is stored in memory as the number 2, and so forth. To prove this, look at the following code.

```
cout << MONDAY << endl << TUESDAY << endl
     << WEDNESDAY << endl << THURSDAY << endl
     << FRIDAY << endl;
```

This statement will produce the following output:

```
0
1
2
3
4
```

NOTE: When making up names for enumerators, it is not required that they be written in all uppercase letters. For example, we could have written the enumerators of the `Days` type as `monday`, `tuesday`, etc. Because they represent constant values, however, many programmers prefer to write them in all uppercase letters. This is strictly a preference of style.

Assigning an Integer to an `enum` Variable

Even though the enumerators of an enumerated data type are stored in memory as integers, you cannot directly assign an integer value to an `enum` variable. For example, assuming that `workDay` is a variable of the `Day` data type previously described, the following assignment statement is illegal.

```
workDay = 3;  // Error!
```

Compiling this statement will produce an error message such as "Cannot convert `int` to `Day`." When assigning a value to an `enum` variable, you should use a valid enumerator. However, if circumstances require that you store an integer value in an `enum` variable, you can do so by casting the integer. Here is an example:

```
workDay = static_cast<Day>(3);
```

This statement will produce the same results as:

```
workDay = THURSDAY;
```

Assigning an Enumerator to an `int` Variable

Although you cannot directly assign an integer value to an `enum` variable, you can directly assign an enumerator to an integer variable. For example, the following code will work just fine.

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
int x;
x = THURSDAY;
cout << x << endl;
```

When this code runs it will display 3. You can also assign a variable of an enumerated type to an integer variable, as shown here:

```
Day workDay = FRIDAY;
int x = workDay;
cout << x << endl;
```

When this code runs it will display 4.

Comparing Enumerator Values

Enumerator values can be compared using the relational operators. For example, using the `Day` data type we have been discussing, the following expression is `true`.

```
FRIDAY > MONDAY
```

The expression is `true` because the enumerator `FRIDAY` is stored in memory as 4 and the enumerator `MONDAY` is stored as 0. The following code will display the message "Friday is greater than Monday."

```
if (FRIDAY > MONDAY)
   cout << "Friday is greater than Monday. \n";
```

You can also compare enumerator values with integer values. For example, the following code will display the message "Monday is equal to zero."

```
if (MONDAY == 0)
   cout << "Monday is equal to zero.\n";
```

Let's look at a complete program that uses much of what we have learned so far. Program 11-11 uses the `Day` data type that we have been discussing.

Program 11-11

```
1  // This program demonstrates an enumerated data type.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
7
8  int main()
9  {
10     const int NUM_DAYS = 5;     // The number of days
11     double sales[NUM_DAYS];     // To hold sales for each day
12     double total = 0.0;         // Accumulator
13     int index;                  // Loop counter
14
15     // Get the sales for each day.
16     for (index = MONDAY; index <= FRIDAY; index++)
17     {
18          cout << "Enter the sales for day "
19               << index << ": ";
20          cin >> sales[index];
21     }
22
23     // Calculate the total sales.
24     for (index = MONDAY; index <= FRIDAY; index++)
25        total += sales[index];
26
27     // Display the total.
28     cout << "The total sales are $" << setprecision(2)
29          << fixed << total << endl;
30
31     return 0;
32 }
```

Program Output with Example Input Shown in Bold

```
Enter the sales for day 0: 1525.00 [Enter]
Enter the sales for day 1: 1896.50 [Enter]
Enter the sales for day 2: 1975.63 [Enter]
Enter the sales for day 3: 1678.33 [Enter]
Enter the sales for day 4: 1498.52 [Enter]
The total sales are $8573.98
```

Anonymous Enumerated Types

Notice that Program 11-11 does not define a variable of the `Day` data type. Instead it uses the `Day` data type's enumerators in the `for` loops. The counter variable `index` is initialized to `MONDAY` (which is 0), and the loop iterates as long as `index` is less than or equal to `FRIDAY` (which is 4). When you do not need to define variables of an enumerated type, you can actually make the type anonymous. An *anonymous enumerated type* is simply one that does not have a name. For example, in Program 11-11 we could have declared the enumerated type as:

```
enum { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
```

This declaration still creates the enumerators. We just can't use the data type to define variables because the type does not have a name.

Using Math Operators to Change the Value of an `enum` Variable

Even though enumerators are really integers, and `enum` variables really hold integer values, you can run into problems when trying to perform math operations with them. For example, look at the following code.

```
Day day1, day2;    // Defines two Day variables.
day1 = TUESDAY;    // Assign TUESDAY to day1.
day2 = day1 + 1;   // ERROR! This will not work!
```

The third statement causes a problem because the expression `day1 + 1` results in the integer value 2. The assignment operator then attempts to assign the integer value 2 to the `enum` variable `day2`. Because C++ cannot implicitly convert an `int` to a `Day`, an error occurs. You can fix this by using a cast to explicitly convert the result to `Day`, as shown here:

```
day2 = static_cast<Day>(day1 + 1);  // This works.
```

Using an `enum` Variable to Step Through an Array's Elements

Because enumerators are stored in memory as integers, you can use them as array subscripts. For example, look at the following code.

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
const int NUM_DAYS = 5;
double sales[NUM_DAYS];
sales[MONDAY] = 1525.0;     // Stores 1525.0 in sales[0].
sales[TUESDAY] = 1896.5;    // Stores 1896.5 in sales[1].
sales[WEDNESDAY] = 1975.63; // Stores 1975.63 in sales[2].
sales[THURSDAY] = 1678.33;  // Stores 1678.33 in sales[3].
sales[FRIDAY] = 1498.52;    // Stores 1498.52 in sales[4].
```

This code stores values in all five elements of the `sales` array. Because enumerator values can be used as array subscripts, you can use an `enum` variable in a loop to step through the elements of an array. However, using an `enum` variable for this purpose is not as straightforward as using an `int` variable. This is because you cannot use the `++` or `--` operators directly on an `enum` variable. To understand what I mean, first look at the following code taken from Program 11-11:

```
for (index = MONDAY; index <= FRIDAY; index++)
{
   cout << "Enter the sales for day "
        << index << ": ";
   cin >> sales[index];
}
```

In this code, `index` is an `int` variable used to step through each element of the array. It is reasonable to expect that we could use a `Day` variable instead, as shown in the following code.

```
Day workDay;  // Define a Day variable

// ERROR!!! This code will NOT work.
for (workDay = MONDAY; workDay <= FRIDAY; workDay++)
{
   cout << "Enter the sales for day "
        << workDay << ": ";
   cin >> sales[workDay];
}
```

Notice that the `for` loop's update expression uses the `++` operator to increment `workDay`. Although this works fine with an `int` variable, the `++` operator cannot be used with an `enum` variable. Instead, you must convert `workDay++` to an equivalent expression that will work. The expression `workDay++` attempts to do the same thing as:

```
workDay = workDay + 1;  // Good idea, but still won't work.
```

However, this still will not work. We have to use a cast to explicitly convert the expression `workDay + 1` to the `Day` data type, like this:

```
workDay = static_cast<Day>(workDay + 1);
```

This is the expression that we must use in the `for` loop instead of `workDay++`. The corrected `for` loop looks like this:

```
for (workDay = MONDAY; workDay <= FRIDAY;
                workDay = static_cast<Day>(workDay + 1))
{
   cout << "Enter the sales for day "
        << workDay << ": ";
   cin >> sales[workDay];
}
```

Program 11-12 is a version of Program 11-11 that is modified to use a `Day` variable to step through the elements of the `sales` array.

Program 11-12

```
1  // This program demonstrates an enumerated data type.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
7
8  int main()
9  {
10     const int NUM_DAYS = 5;     // The number of days
11     double sales[NUM_DAYS];     // To hold sales for each day
12     double total = 0.0;         // Accumulator
13     Day workDay;                // Loop counter
14
```

(program continues)

Program 11-12 *(continued)*

```
15      // Get the sales for each day.
16      for (workDay = MONDAY; workDay <= FRIDAY;
17                             workDay = static_cast<Day>(workDay + 1))
18      {
19          cout << "Enter the sales for day "
20               << workDay << ": ";
21          cin >> sales[workDay];
22      }
23
24      // Calculate the total sales.
25      for (workDay = MONDAY; workDay <= FRIDAY;
26                             workDay = static_cast<Day>(workDay + 1))
27          total += sales[workDay];
28
29      // Display the total.
30      cout << "The total sales are $" << setprecision(2)
31           << fixed << total << endl;
32
33      return 0;
34  }
```

Program Output with Example Input Shown in Bold

```
Enter the sales for day 0: 1525.00 [Enter]
Enter the sales for day 1: 1896.50 [Enter]
Enter the sales for day 2: 1975.63 [Enter]
Enter the sales for day 3: 1678.33 [Enter]
Enter the sales for day 4: 1498.52 [Enter]
The total sales are $8573.98
```

Using Enumerators to Output Values

As you have already seen, sending an enumerator to `cout` causes the enumerator's integer value to be displayed. For example, assuming we are using the `Day` type previously described, the following statement displays 0.

```
cout << MONDAY << endl;
```

If you wish to use the enumerator to display a string such as "Monday," you'll have to write code that produces the desired string. For example, in the following code assume that `workDay` is a `Day` variable that has been initialized to some value. The `switch` statement displays the name of a day, based upon the value of the variable.

```
switch(workDay)
{
   case MONDAY    : cout << "Monday";
                    break;
   case TUESDAY   : cout << "Tuesday";
                    break;
   case WEDNESDAY : cout << "Wednesday";
                    break;
   case THURSDAY  : cout << "Thursday";
                    break;
   case FRIDAY    : cout << "Friday";
}
```

Program 11-13 shows this type of code used in a function. Instead of asking the user to enter the sales for day 0, day 1, and so forth, it displays the names of the days.

Program 11-13

```
1  // This program demonstrates an enumerated data type.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
7
8  // Function prototype
9  void displayDayName(Day);
10
11 int main()
12 {
13     const int NUM_DAYS = 5;     // The number of days
14     double sales[NUM_DAYS];     // To hold sales for each day
15     double total = 0.0;         // Accumulator
16     Day workDay;                // Loop counter
17
18     // Get the sales for each day.
19     for (workDay = MONDAY; workDay <= FRIDAY;
20                            workDay = static_cast<Day>(workDay + 1))
21     {
22         cout << "Enter the sales for day ";
23         displayDayName(workDay);
24         cout << ": ";
25         cin >> sales[workDay];
26     }
27
28     // Calculate the total sales.
29     for (workDay = MONDAY; workDay <= FRIDAY;
30                            workDay = static_cast<Day>(workDay + 1))
31         total += sales[workDay];
32
33     // Display the total.
34     cout << "The total sales are $" << setprecision(2)
35          << fixed << total << endl;
36
37     return 0;
38 }
39
40 //********************************************************
41 // Definition of the displayDayName function             *
42 // This function accepts an argument of the Day type and *
43 // displays the corresponding name of the day.           *
44 //********************************************************
45
```

(program continues)

Program 11-13 *(continued)*

```
46  void displayDayName(Day d)
47  {
48      switch(d)
49      {
50          case MONDAY    : cout << "Monday";
51                           break;
52          case TUESDAY   : cout << "Tuesday";
53                           break;
54          case WEDNESDAY : cout << "Wednesday";
55                           break;
56          case THURSDAY  : cout << "Thursday";
57                           break;
58          case FRIDAY    : cout << "Friday";
59      }
60  }
```

Program Output with Example Input Shown in Bold

```
Enter the sales for Monday: 1525.00 [Enter]
Enter the sales for Tuesday: 1896.50 [Enter]
Enter the sales for Wednesday: 1975.63 [Enter]
Enter the sales for Thursday: 1678.33 [Enter]
Enter the sales for Friday: 1498.52 [Enter]
The total sales are $8573.98
```

Specifying Integer Values for Enumerators

By default, the enumerators in an enumerated data type are assigned the integer values 0, 1, 2, and so forth. If this is not appropriate, you can specify the values to be assigned, as in the following example.

```
enum Water { FREEZING = 32, BOILING = 212 };
```

In this example, the `FREEZING` enumerator is assigned the integer value 32 and the `BOILING` enumerator is assigned the integer value 212. Program 11-14 demonstrates how this enumerated type might be used.

Program 11-14

```
1   // This program demonstrates an enumerated data type.
2   #include <iostream>
3   #include <iomanip>
4   using namespace std;
5
6   int main()
7   {
8       enum Water { FREEZING = 32, BOILING = 212 };
9       int waterTemp; // To hold the water temperature
10
```

```
11      cout << "Enter the current water temperature: ";
12      cin >> waterTemp;
13      if (waterTemp <= FREEZING)
14          cout << "The water is frozen.\n";
15      else if (waterTemp >= BOILING)
16          cout << "The water is boiling.\n";
17      else
18          cout << "The water is not frozen or boiling.\n";
19
20      return 0;
21  }
```

Program Output with Example Input Shown in Bold

```
Enter the current water temperature: 10 [Enter]
The water is frozen.
```

Program Output with Example Input Shown in Bold

```
Enter the current water temperature: 300 [Enter]
The water is boiling.
```

Program Output with Example Input Shown in Bold

```
Enter the current water temperature: 92 [Enter]
The water is not frozen or boiling.
```

If you leave out the value assignment for one or more of the enumerators, it will be assigned a default value. Here is an example:

```
enum Colors { RED, ORANGE, YELLOW = 9, GREEN, BLUE };
```

In this example the enumerator `RED` will be assigned the value 0, `ORANGE` will be assigned the value 1, `YELLOW` will be assigned the value 9, `GREEN` will be assigned the value 10, and `BLUE` will be assigned the value 11.

Enumerators Must Be Unique Within the Same Scope

Enumerators are identifiers just like variable names, named constants, and function names. As with all identifiers, they must be unique within the same scope. For example, an error will result if both of the following enumerated types are declared within the same scope. The reason is that `ROOSEVELT` is declared twice.

```
enum Presidents { MCKINLEY, ROOSEVELT, TAFT };
enum VicePresidents { ROOSEVELT, FAIRBANKS, SHERMAN };  // Error!
```

The following declarations will also cause an error if they appear within the same scope.

```
enum Status { OFF, ON };
const int OFF = 0;        // Error!
```

Declaring the Type and Defining the Variables in One Statement

The following code uses two lines to declare an enumerated data type and define a variable of the type.

```
enum Car { PORSCHE, FERRARI, JAGUAR };
Car sportsCar;
```

C++ allows you to declare an enumerated data type and define one or more variables of the type in the same statement. The previous code could be combined into the following statement:

```
enum Car { PORSCHE, FERRARI, JAGUAR } sportsCar;
```

The following statement declares the `Car` data type and defines two variables: `myCar` and `yourCar`.

```
enum Car { PORSCHE, FERRARI, JAGUAR } myCar, yourCar;
```

Using Strongly Typed `enum`s in C++ 11

Earlier we mentioned that you cannot have multiple enumerators with the same name, within the same scope. In C++ 11, you can use a new type of `enum`, known as a *strongly typed enum* (also known as an *enum class*), to get around this limitation. Here are two examples of a strongly typed `enum` declaration:

```
enum class Presidents { MCKINLEY, ROOSEVELT, TAFT };
enum class VicePresidents { ROOSEVELT, FAIRBANKS, SHERMAN };
```

These statements declare two strongly typed `enum`s: `Presidents` and `VicePresidents`. Notice that they look like regular `enum` declarations, except that the word `class` appears after `enum`. Although both `enum`s contain the same enumerator (`ROOSEVELT`), these declarations will compile without an error.

When you use a strongly typed enumerator, you must prefix the enumerator with the name of the `enum`, followed by the `::` operator. Here are two examples:

```
Presidents prez = Presidents::ROOSEVELT;
VicePresidents vp = VicePresidents::ROOSEVELT;
```

The first statement defines a `Presidents` variable named `prez`, and initializes it with the `Presidents::ROOSEVELT` enumerator. The second statement defines a `VicePresidents` variable named `vp`, and initializes it with the `VicePresidents::ROOSEVELT` enumerator. Here is an example of an `if` statement that compares the `prez` variable with an enumerator:

```
if (prez == Presidents::ROOSEVELT)
   cout << "Roosevelt is president!\n";
```

Strongly typed enumerators are stored as integers like regular enumerators. However, if you want to retrieve a strongly typed enumerator's underlying integer value, you must use a cast operator. Here is an example:

```
int x = static_cast<int>(Presidents::ROOSEVELT);
```

This statement assigns the underlying integer value of the `Presidents::ROOSEVELT` enumerator to the variable `x`. Here is another example:

```
cout << static_cast<int>(Presidents::TAFT) << endl
     << static_cast<int>(Presidents::MCKINLEY) << endl;
```

This statement displays the integer values for the `Presidents::TAFT` and the `Presidents::MCKINLEY` enumerators.

When you declare a strongly typed `enum`, you can optionally specify any integer data type as the underlying type. You simply write a colon (:) after the enum name, followed by the desired data type. For example, the following statement declares an `enum` that uses the `char` data type for its enumerators:

```
enum class Day : char { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
```

The following statement shows another example. This statement declares an `enum` named `Water` that uses `unsigned` as the data type of its enumerators. Additionally, values are assigned to the enumerators.

```
enum class Water : unsigned { FREEZING = 32, BOILING = 212 };
```

Checkpoint

11.21 Look at the following declaration.

```
enum Flower { ROSE, DAISY, PETUNIA };
```

In memory, what value will be stored for the enumerator `ROSE`? For `DAISY`? For `PETUNIA`?

11.22 What will the following code display?

```
enum { HOBBIT, ELF = 7, DRAGON };
cout << HOBBIT << " " << ELF << " " << DRAGON << endl;
```

11.23 Does the enumerated data type declared in Checkpoint Question 11.22 have a name, or is it anonymous?

11.24 What will the following code display?

```
enum Letters { Z, Y, X };
if (Z > X)
   cout << "Z is greater than X. \n";
else
   cout << "Z is not greater than X. \n";
```

11.25 Will the following code cause an error, or will it compile without any errors? If it causes an error, rewrite it so it compiles.

```
enum Color { RED, GREEN, BLUE };
Color c;
c = 0;
```

11.26 Will the following code cause an error, or will it compile without any errors? If it causes an error, rewrite it so it compiles.

```
enum Color { RED, GREEN, BLUE };
Color c = RED;
c++;
```

NOTE: For an additional example of this chapter's topics, see the High Adventure Travel Part 2 Case Study on this book's companion Web site at pearsonhighered.com/gaddis.

Review Questions and Exercises

Short Answer

1. What is a primitive data type?
2. Does a structure declaration cause a structure variable to be created?
3. Both arrays and structures are capable of storing multiple values. What is the difference between an array and a structure?
4. Look at the following structure declaration.

```
struct Point
{
   int x;
   int y;
};
```

   Write statements that

   A) define a `Point` structure variable named `center`
   B) assign 12 to the `x` member of `center`
   C) assign 7 to the `y` member of `center`
   D) display the contents of the `x` and `y` members of `center`

5. Look at the following structure declaration.

```
struct FullName
{
   string lastName;
   string middleName;
   string firstName;
};
```

   Write statements that

   A) Define a `FullName` structure variable named `info`
   B) Assign your last, middle, and first name to the members of the `info` variable
   C) Display the contents of the members of the `info` variable

6. Look at the following code.

```
struct PartData
{
   string partName;
   int idNumber;
};
PartData inventory[100];
```

   Write a statement that displays the contents of the `partName` member of element 49 of the `inventory` array.

7. Look at the following code.

```
struct Town
{
   string townName;
   string countyName;
   double population;
   double elevation;
};
```

```
Town t = { "Canton", "Haywood", 9478 };
```

A) What value is stored in `t.townName`?

B) What value is stored in `t.countyName`?

C) What value is stored in `t.population`?

D) What value is stored in `t.elevation`?

8. Look at the following code.

```
structure Rectangle
{
   int length;
   int width;
};
Rectangle *r = nullptr
```

Write statements that

A) Dynamically allocate a `Rectangle` structure variable and use `r` to point to it.

B) Assign 10 to the structure's `length` member and 14 to the structure's `width` member.

9. What is the difference between a union and a structure?

10. Look at the following code.

```
union Values
{
   int ivalue;
   double dvalue;
};

Values v;
```

Assuming that an `int` uses four bytes and a `double` uses eight bytes, how much memory does the variable `v` use?

11. What will the following code display?

```
enum { POODLE, BOXER, TERRIER };
cout << POODLE << " " << BOXER << " " << TERRIER << endl;
```

12. Look at the following declaration.

```
enum Person { BILL, JOHN, CLAIRE, BOB };
Person p;
```

Indicate whether each of the following statements or expressions is valid or invalid.

A) `p = BOB;`

B) `p++;`

C) `BILL > BOB`

D) `p = 0;`

E) `int x = BILL;`

F) `p = static_cast<Person>(3);`

G) `cout << CLAIRE << endl;`

Fill-in-the-Blank

13. Before a structure variable can be created, the structure must be _____.
14. The _____ is the name of the structure type.
15. The variables declared inside a structure declaration are called _____.
16. A(n) _____ is required after the closing brace of a structure declaration.
17. In the definition of a structure variable, the _____ is placed before the variable name, just like the data type of a regular variable is placed before its name.
18. The _____ operator allows you to access structure members.

Algorithm Workbench

19. The structure `Car` is declared as follows:

```
struct Car
{
   string carMake;
   string carModel;
   int yearModel;
   double cost;
};
```

Write a definition statement that defines a `Car` structure variable initialized with the following data:

Make: Ford
Model: Mustang
Year Model: 1968
Cost: $20,000

20. Define an array of 25 of the `Car` structure variables (the structure is declared in Question 19).
21. Define an array of 35 of the `Car` structure variables. Initialize the first three elements with the following data:

Make	Model	Year	Cost
Ford	Taurus	1997	$ 21,000
Honda	Accord	1992	$ 11,000
Lamborghini	Countach	1997	$200,000

22. Write a loop that will step through the array you defined in Question 21, displaying the contents of each element.
23. Declare a structure named `TempScale`, with the following members:

```
fahrenheit: a double
centigrade: a double
```

Next, declare a structure named `Reading`, with the following members:

```
windSpeed: an int
humidity: a double
temperature: a TempScale structure variable
```

Next define a `Reading` structure variable.

24. Write statements that will store the following data in the variable you defined in Question 23.

    Wind Speed: 37 mph
    Humidity: 32%
    Fahrenheit temperature: 32 degrees
    Centigrade temperature: 0 degrees

25. Write a function called `showReading`. It should accept a `Reading` structure variable (see Question 23) as its argument. The function should display the contents of the variable on the screen.

26. Write a function called `findReading`. It should use a `Reading` structure reference variable (see Question 23) as its parameter. The function should ask the user to enter values for each member of the structure.

27. Write a function called `getReading`, which returns a `Reading` structure (see Question 23). The function should ask the user to enter values for each member of a `Reading` structure, then return the structure.

28. Write a function called `recordReading`. It should use a `Reading` structure pointer variable (see Question 23) as its parameter. The function should ask the user to enter values for each member of the structure pointed to by the parameter.

29. Rewrite the following statement using the structure pointer operator:

```
(*rptr).windSpeed = 50;
```

30. Rewrite the following statement using the structure pointer operator:

```
*(*strPtr).num = 10;
```

31. Write the declaration of a union called `Items` with the following members:

`alpha`	a character
`num`	an integer
`bigNum`	a `long` integer
`real`	a `float`

    Next, write the definition of an `Items` union variable.

32. Write the declaration of an anonymous union with the same members as the union you declared in Question 31.

33. Write a statement that stores the number 452 in the `num` member of the anonymous union you declared in Question 32.

34. Look at the following statement.

```
enum Color { RED, ORANGE, GREEN, BLUE };
```

    A) What is the name of the data type declared by this statement?
    B) What are the enumerators for this type?
    C) Write a statement that defines a variable of this type and initializes it with a valid value.

35. A pet store sells dogs, cats, birds, and hamsters. Write a declaration for an anonymous enumerated data type that can represent the types of pets the store sells.

True or False

36. T F A semicolon is required after the closing brace of a structure or union declaration.
37. T F A structure declaration does not define a variable.
38. T F The contents of a structure variable can be displayed by passing the structure variable to the `cout` object.
39. T F Structure variables may not be initialized.
40. T F In a structure variable's initialization list, you do not have to provide initializers for all the members.
41. T F You may skip members in a structure's initialization list.
42. T F The following expression refers to element 5 in the array `carInfo`:
    `carInfo.model[5]`
43. T F An array of structures may be initialized.
44. T F A structure variable may not be a member of another structure.
45. T F A structure member variable may be passed to a function as an argument.
46. T F An entire structure may not be passed to a function as an argument.
47. T F A function may return a structure.
48. T F When a function returns a structure, it is always necessary for the function to have a local structure variable to hold the member values that are to be returned.
49. T F The indirection operator has higher precedence than the dot operator.
50. T F The structure pointer operator does not automatically dereference the structure pointer on its left.
51. T F In a `union`, all the members are stored in different memory locations.
52. T F All the members of a `union` may be used simultaneously.
53. T F You may define arrays of `unions`.
54. T F You may not define pointers to `unions`.
55. T F An anonymous `union` has no name.
56. T F If an anonymous `union` is defined globally (outside all functions), it must be declared `static`.

Find the Errors

Each of the following declarations, programs, and program segments has errors. Locate as many as you can.

57.
```
struct
{
   int x;
   float y;
};
```

58.
```
struct Values
{
   string name;
   int age;
}
```

59.
```
struct TwoVals
{
   int a, b;
};
int main ()
{
   TwoVals.a = 10;
   TwoVals.b = 20;
   return 0;
}
```

60.
```
#include <iostream>
using namespace std;

struct ThreeVals
{
   int a, b, c;
};
int main()
{
   ThreeVals vals = {1, 2, 3};
   cout << vals << endl;
   return 0;
}
```

61.
```
#include <iostream>
#include <string>
using namespace std;

struct names
{
   string first;
   string last;
};
int main ()
{
   names customer = "Smith", "Orley";
   cout << names.first << endl;
   cout << names.last << endl;
   return 0;
}
```

62.
```
struct FourVals
{
   int a, b, c, d;
};
int main ()
{
   FourVals nums = {1, 2, , 4};
   return 0;
}
```

63.
```
#include <iostream>
using namespace std;
```

```
struct TwoVals
{
   int a = 5;
   int b = 10;
};
int main()
{
   TwoVals v;

   cout << v.a << " " << v.b;
   return 0;
}
```

64.
```
struct TwoVals
{
   int a = 5;
   int b = 10;
};

int main()
{
   TwoVals varray[10];

   varray.a[0] = 1;
   return 0;
}
```

65.
```
struct TwoVals
{
   int a;
   int b;
};
TwoVals getVals()
{
   TwoVals.a = TwoVals.b = 0;
}
```

66.
```
struct ThreeVals
{
   int a, b, c;
};
int main ()
{
   TwoVals s, *sptr = nullptr;
   sptr = &s;
   *sptr.a = 1;
   return 0;
}
```

67.
```
#include <iostream>
using namespace std;

union Compound
{
   int x;
   float y;
};
```

```
int main()
{
   Compound u;
   u.x = 1000;
   cout << u.y << endl;
   return 0;
}
```

Programming Challenges

1. **Movie Data**

   Write a program that uses a structure named `MovieData` to store the following information about a movie:

   Title
   Director
   Year Released
   Running Time (in minutes)

   The program should create two `MovieData` variables, store values in their members, and pass each one, in turn, to a function that displays the information about the movie in a clearly formatted manner.

2. **Movie Profit**

   Modify the Movie Data program written for Programming Challenge 1 to include two additional members that hold the movie's production costs and first-year revenues. Modify the function that displays the movie data to display the title, director, release year, running time, and first year's profit or loss.

3. **Corporate Sales Data**

   Write a program that uses a structure to store the following data on a company division:

   Division Name (such as East, West, North, or South)
   First-Quarter Sales
   Second-Quarter Sales
   Third-Quarter Sales
   Fourth-Quarter Sales
   Total Annual Sales
   Average Quarterly Sales

   The program should use four variables of this structure. Each variable should represent one of the following corporate divisions: East, West, North, and South. The user should be asked for the four quarters' sales figures for each division. Each division's total and average sales should be calculated and stored in the appropriate member of each structure variable. These figures should then be displayed on the screen.

   Input Validation: Do not accept negative numbers for any sales figures.

4. Weather Statistics

VideoNote
Solving the Weather Statistics Problem

Write a program that uses a structure to store the following weather data for a particular month:

Total Rainfall
High Temperature
Low Temperature
Average Temperature

The program should have an array of 12 structures to hold weather data for an entire year. When the program runs, it should ask the user to enter data for each month. (The average temperature should be calculated.) Once the data are entered for all the months, the program should calculate and display the average monthly rainfall, the total rainfall for the year, the highest and lowest temperatures for the year (and the months they occurred in), and the average of all the monthly average temperatures.

Input Validation: Only accept temperatures within the range between –100 and +140 degrees Fahrenheit.

5. Weather Statistics Modification

Modify the program that you wrote for Programming Challenge 4 so it defines an enumerated data type with enumerators for the months (`JANUARY`, `FEBRUARY`, etc.). The program should use the enumerated type to step through the elements of the array.

6. Soccer Scores

Write a program that stores the following data about a soccer player in a structure:

Player's Name
Player's Number
Points Scored by Player

The program should keep an array of 12 of these structures. Each element is for a different player on a team. When the program runs it should ask the user to enter the data for each player. It should then show a table that lists each player's number, name, and points scored. The program should also calculate and display the total points earned by the team. The number and name of the player who has earned the most points should also be displayed.

Input Validation: Do not accept negative values for players' numbers or points scored.

7. Customer Accounts

Write a program that uses a structure to store the following data about a customer account:

Name
Address
City, State, and ZIP
Telephone Number
Account Balance
Date of Last Payment

The program should use an array of at least 10 structures. It should let the user enter data into the array, change the contents of any element, and display all the data stored in the array. The program should have a menu-driven user interface.

Input Validation: When the data for a new account is entered, be sure the user enters data for all the fields. No negative account balances should be entered.

8. **Search Function for Customer Accounts Program**

   Add a function to Programming Challenge 7 that allows the user to search the structure array for a particular customer's account. It should accept part of the customer's name as an argument and then search for an account with a name that matches it. All accounts that match should be displayed. If no account matches, a message saying so should be displayed.

9. **Speakers' Bureau**

   Write a program that keeps track of a speakers' bureau. The program should use a structure to store the following data about a speaker:

   Name
   Telephone Number
   Speaking Topic
   Fee Required

   The program should use an array of at least 10 structures. It should let the user enter data into the array, change the contents of any element, and display all the data stored in the array. The program should have a menu-driven user interface.

   Input Validation: When the data for a new speaker is entered, be sure the user enters data for all the fields. No negative amounts should be entered for a speaker's fee.

10. **Search Function for the Speakers' Bureau Program**

    Add a function to Programming Challenge 9 that allows the user to search for a speaker on a particular topic. It should accept a key word as an argument and then search the array for a structure with that key word in the Speaking Topic field. All structures that match should be displayed. If no structure matches, a message saying so should be displayed.

11. **Monthly Budget**

    A student has established the following monthly budget:

Housing	500.00
Utilities	150.00
Household Expenses	65.00
Transportation	50.00
Food	250.00
Medical	30.00
Insurance	100.00
Entertainment	150.00
Clothing	75.00
Miscellaneous	50.00

Write a program that has a `MonthlyBudget` structure designed to hold each of these expense categories. The program should pass the structure to a function that asks the user to enter the amounts spent in each budget category during a month. The program should then pass the structure to a function that displays a report indicating the amount over or under in each category, as well as the amount over or under for the entire monthly budget.

12. Course Grade

Write a program that uses a structure to store the following data:

Member Name	Description
Name	Student name
Idnum	Student ID number
Tests	Pointer to an array of test scores
Average	Average test score
Grade	Course grade

The program should keep a list of test scores for a group of students. It should ask the user how many test scores there are to be and how many students there are. It should then dynamically allocate an array of structures. Each structure's `Tests` member should point to a dynamically allocated array that will hold the test scores.

After the arrays have been dynamically allocated, the program should ask for the ID number and all the test scores for each student. The average test score should be calculated and stored in the `average` member of each structure. The course grade should be computed on the basis of the following grading scale:

Average Test Grade	Course Grade
91–100	A
81–90	B
71–80	C
61–70	D
60 or below	F

The course grade should then be stored in the `Grade` member of each structure. Once all this data is calculated, a table should be displayed on the screen listing each student's name, ID number, average test score, and course grade.

Input Validation: Be sure all the data for each student is entered. Do not accept negative numbers for any test score.

13. Drink Machine Simulator

Write a program that simulates a soft drink machine. The program should use a structure that stores the following data:

Drink Name
Drink Cost
Number of Drinks in Machine

The program should create an array of five structures. The elements should be initialized with the following data:

Drink Name	Cost	Number in Machine
Cola	.75	20
Root Beer	.75	20
Lemon-Lime	.75	20
Grape Soda	.80	20
Cream Soda	.80	20

Each time the program runs, it should enter a loop that performs the following steps: A list of drinks is displayed on the screen. The user should be allowed to either quit the program or pick a drink. If the user selects a drink, he or she will next enter the amount of money that is to be inserted into the drink machine. The program should display the amount of change that would be returned and subtract one from the number of that drink left in the machine. If the user selects a drink that has sold out, a message should be displayed. The loop then repeats. When the user chooses to quit the program it should display the total amount of money the machine earned.

Input Validation: When the user enters an amount of money, do not accept negative values or values greater than $1.00.

14. Inventory Bins

Write a program that simulates inventory bins in a warehouse. Each bin holds a number of the same type of parts. The program should use a structure that keeps the following data:

Description of the part kept in the bin
Number of parts in the bin

The program should have an array of 10 bins, initialized with the following data:

Part Description	Number of Parts in the Bin
Valve	10
Bearing	5
Bushing	15
Coupling	21
Flange	7
Gear	5
Gear Housing	5
Vacuum Gripper	25
Cable	18
Rod	12

The program should have the following functions:

AddParts: a function that increases a specific bin's part count by a specified number.

RemoveParts: a function that decreases a specific bin's part count by a specified number.

When the program runs, it should repeat a loop that performs the following steps: The user should see a list of what each bin holds and how many parts are in each bin. The user can choose to either quit the program or select a bin. When a bin is selected, the user can either add parts to it or remove parts from it. The loop then repeats, showing the updated bin data on the screen.

Input Validation: No bin can hold more than 30 parts, so don't let the user add more than a bin can hold. Also, don't accept negative values for the number of parts being added or removed.

15. Multipurpose Payroll

Write a program that calculates pay for either an hourly paid worker or a salaried worker. Hourly paid workers are paid their hourly pay rate times the number of hours worked. Salaried workers are paid their regular salary plus any bonus they may have earned. The program should declare two structures for the following data:

Hourly Paid:
HoursWorked
HourlyRate

Salaried:
Salary
Bonus

The program should also declare a `union` with two members. Each member should be a structure variable: one for the hourly paid worker and another for the salaried worker.

The program should ask the user whether he or she is calculating the pay for an hourly paid worker or a salaried worker. Regardless of which the user selects, the appropriate members of the `union` will be used to store the data that will be used to calculate the pay.

Input Validation: Do not accept negative numbers. Do not accept values greater than 80 for HoursWorked.

CHAPTER 12

Advanced File Operations

TOPICS

12.1 File Operations

CONCEPT: A file is a collection of data that is usually stored on a computer's disk. Data can be saved to files and then later reused.

Almost all real-world programs use files to store and retrieve data. Here are a few examples of familiar software packages that use files extensively.

- **Word Processors:** Word processing programs are used to write letters, memos, reports, and other documents. The documents are then saved in files so they can be edited and reprinted.
- **Database Management Systems:** DBMSs are used to create and maintain databases. Databases are files that contain large collections of data, such as payroll records, inventories, sales statistics, and customer records.
- **Spreadsheets:** Spreadsheet programs are used to work with numerical data. Numbers and mathematical formulas can be inserted into the rows and columns of the spreadsheet. The spreadsheet can then be saved to a file for use later.
- **Compilers:** Compilers translate the source code of a program, which is saved in a file, into an executable file. Throughout the previous chapters of this book you have created many C++ source files and compiled them to executable files.

Chapter 5 provided enough information for you to write programs that perform simple file operations. This chapter covers more advanced file operations and focuses primarily

on the `fstream` data type. As a review, Table 12-1 compares the `ifstream`, `ofstream`, and `fstream` data types. All of these data types require the `fstream` header file.

Table 12-1 File Stream

Data Type	Description
`ifstream`	Input File Stream. This data type can be used only to read data from files into memory.
`ofstream`	Output File Stream. This data type can be used to create files and write data to them.
`fstream`	File Stream. This data type can be used to create files, write data to them, and read data from them.

Using the `fstream` Data Type

You define an `fstream` object just as you define objects of other data types. The following statement defines an `fstream` object named `dataFile`.

```
fstream dataFile;
```

As with `ifstream` and `ofstream` objects, you use an `fstream` object's `open` function to open a file. An `fstream` object's `open` function requires two arguments, however. The first argument is a string containing the name of the file. The second argument is a file access flag that indicates the mode in which you wish to open the file. Here is an example.

```
dataFile.open("info.txt", ios::out);
```

The first argument in this function call is the name of the file, `info.txt`. The second argument is the file access flag `ios::out`. This tells C++ to open the file in output mode. Output mode allows data to be written to a file. The following statement uses the `ios::in` access flag to open a file in input mode, which allows data to be read from the file.

```
dataFile.open("info.txt", ios::in);
```

There are many file access flags, as listed in Table 12-2.

Table 12-2

File Access Flag	Meaning
`ios::app`	Append mode. If the file already exists, its contents are preserved and all output is written to the end of the file. By default, this flag causes the file to be created if it does not exist.
`ios::ate`	If the file already exists, the program goes directly to the end of it. Output may be written anywhere in the file.
`ios::binary`	Binary mode. When a file is opened in binary mode, data are written to or read from it in pure binary format. (The default mode is text.)
`ios::in`	Input mode. Data will be read from the file. If the file does not exist, it will not be created and the open function will fail.
`ios::out`	Output mode. Data will be written to the file. By default, the file's contents will be deleted if it already exists.
`ios::trunc`	If the file already exists, its contents will be deleted (truncated). This is the default mode used by `ios::out`.

Several flags may be used together if they are connected with the | operator. For example, assume `dataFile` is an `fstream` object in the following statement:

```
dataFile.open("info.txt", ios::in | ios::out);
```

This statement opens the file `info.txt` in both input and output modes. This means data may be written to and read from the file.

NOTE: When used by itself, the `ios::out` flag causes the file's contents to be deleted if the file already exists. When used with the `ios::in` flag, however, the file's existing contents are preserved. If the file does not already exist, it will be created.

The following statement opens the file in such a way that data will only be written to its end:

```
dataFile.open("info.txt", ios::out | ios::app);
```

By using different combinations of access flags, you can open files in many possible modes.

Program 12-1 uses an `fstream` object to open a file for output, and then writes data to the file.

Program 12-1

```
1  // This program uses an fstream object to write data to a file.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8       fstream dataFile;
9
10      cout << "Opening file...\n";
11      dataFile.open("demofile.txt", ios::out);    // Open for output
12      cout << "Now writing data to the file.\n";
13      dataFile << "Jones\n";                      // Write line 1
14      dataFile << "Smith\n";                      // Write line 2
15      dataFile << "Willis\n";                     // Write line 3
16      dataFile << "Davis\n";                      // Write line 4
17      dataFile.close();                           // Close the file
18      cout << "Done.\n";
19      return 0;
20 }
```

Program Output

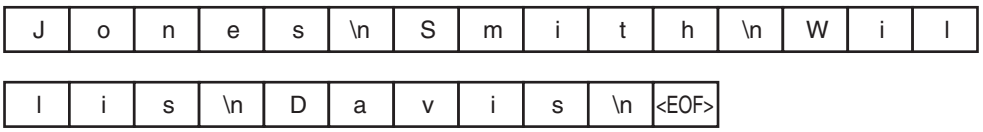
```
Opening file...
Now writing data to the file.
Done.
```

Output to File `demofile.txt`

```
Jones
Smith
Willis
Davis
```

The file output is shown for Program 12-1 the way it would appear if the file contents were displayed on the screen. The `\n` characters cause each name to appear on a separate line. The actual file contents, however, appear as a stream of characters as shown in Figure 12-1.

Figure 12-1



As you can see from the figure, `\n` characters are written to the file along with all the other characters. The characters are added to the file sequentially, in the order they are written by the program. The very last character is an *end-of-file marker*. It is a character that marks the end of the file and is automatically written when the file is closed. (The actual character used to mark the end of a file depends upon the operating system being used. It is always a nonprinting character. For example, some systems use control-Z.)

Program 12-2 is a modification of Program 12-1 that further illustrates the sequential nature of files. The file is opened, two names are written to it, and it is closed. The file is then reopened by the program in append mode (with the `ios::app` access flag). When a file is opened in append mode, its contents are preserved, and all subsequent output is appended to the file's end. Two more names are added to the file before it is closed and the program terminates.

Program 12-2

```
1  // This program writes data to a file, closes the file,
2  // then reopens the file and appends more data.
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
7  int main()
8  {
9      ofstream dataFile;
10
11     cout << "Opening file...\n";
12     // Open the file in output mode.
13     dataFile.open("demofile.txt", ios::out);
14     cout << "Now writing data to the file.\n";
15     dataFile << "Jones\n";                        // Write line 1
16     dataFile << "Smith\n";                        // Write line 2
17     cout << "Now closing the file.\n";
18     dataFile.close();                             // Close the file
19
20     cout << "Opening the file again...\n";
21     // Open the file in append mode.
22     dataFile.open("demofile.txt", ios::out | ios::app);
23     cout << "Writing more data to the file.\n";
24     dataFile << "Willis\n";                       // Write line 3
25     dataFile << "Davis\n";                        // Write line 4
26     cout << "Now closing the file.\n";
27     dataFile.close();                             // Close the file
```

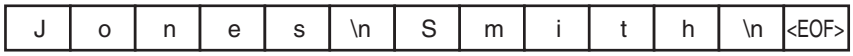
```
28
29       cout << "Done.\n";
30       return 0;
31   }
```

Output to File `demofile.txt`

```
Jones
Smith
Willis
Davis
```

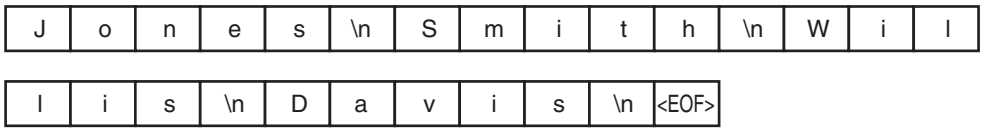
The first time the file is opened, the names are written as shown in Figure 12-2.

Figure 12-2



The file is closed, and an end-of-file character is automatically written. When the file is reopened, the new output is appended to the end of the file, as shown in Figure 12-3.

Figure 12-3



NOTE: If the `ios::out` flag had been alone, without `ios::app` the second time the file was opened, the file's contents would have been deleted. If this had been the case, the names Jones and Smith would have been erased, and the file would only have contained the names Willis and Davis.

File Open Modes with `ifstream` and `ofstream` Objects

The `ifstream` and `ofstream` data types each have a default mode in which they open files. This mode determines the operations that may be performed on the file and what happens if the file that is being opened already exists. Table 12-3 describes each data type's default open mode.

Table 12-3

File Type	Default Open Mode
`ofstream`	The file is opened for output only. Data may be written to the file, but not read from the file. If the file does not exist, it is created. If the file already exists, its contents are deleted (the file is truncated).
`ifstream`	The file is opened for input only. Data may be read from the file, but not written to it. The file's contents will be read from its beginning. If the file does not exist, the open function fails.

You cannot change the fact that ifstream files may only be read from and ofstream files may only be written to. You can, however, vary the way operations are carried out on these files by providing a file access flag as an optional second argument to the open function. The following code shows an example using an ofstream object.

```
ofstream outputFile;
outputFile.open("values.txt", ios::out|ios::app);
```

The ios::app flag specifies that data written to the values.txt file should be appended to its existing contents.

Checking for a File's Existence Before Opening It

Sometimes you want to determine whether a file already exists before opening it for output. You can do this by first attempting to open the file for input. If the file does not exist, the open operation will fail. In that case, you can create the file by opening it for output. The following code gives an example.

```
fstream dataFile;
dataFile.open("values.txt", ios::in);
if (dataFile.fail())
{
        // The file does not exist, so create it.
        dataFile.open("values.txt", ios::out);
        //
        // Continue to process the file...
        //
}
else    // The file already exists.
{
        dataFile.close();
        cout << "The file values.txt already exists.\n";
}
```

Opening a File with the File Stream Object Definition Statement

An alternative to using the open member function is to use the file stream object definition statement to open the file. Here is an example:

```
fstream dataFile("names.txt", ios::in | ios::out);
```

This statement defines an fstream object named dataFile and uses it to open the file names.txt. The file is opened in both input and output modes. This technique eliminates the need to call the open function when your program knows the name and access mode of the file at the time the object is defined. You may also use this technique with ifstream and ofstream objects, as shown in the following examples.

```
ifstream inputFile("info.txt");
ofstream outputFile("addresses.txt");
ofstream dataFile("customers.txt", ios::out|ios::app);
```

You may also test for errors after you have opened a file with this technique. The following code shows an example.

```
ifstream inputFile("SalesData.txt");
if (!inputFile)
     cout << "Error opening SalesData.txt.\n";
```

Checkpoint

12.1 Which file access flag would you use if you want all output to be written to the end of an existing file?

12.2 How do you use more than one file access flag?

12.3 Assuming that `diskInfo` is an `fstream` object, write a statement that opens the file `names.dat` for output.

12.4 Assuming that `diskInfo` is an `fstream` object, write a statement that opens the file `customers.txt` for output, where all output will be written to the end of the file.

12.5 Assuming that `diskInfo` is an `fstream` object, write a statement that opens the file `payable.txt` for both input and output.

12.6 Write a statement that defines an `fstream` object named `dataFile` and opens a file named `salesfigures.txt` for input. (*Note:* The file should be opened with the definition statement, not an `open` function call.)

12.2 File Output Formatting

CONCEPT: File output may be formatted in the same way that screen output is formatted.

The same output formatting techniques that are used with `cout`, which are covered in Chapter 3, may also be used with file stream objects. For example, the `setprecision` and `fixed` manipulators may be called to establish the number of digits of precision that floating point values are rounded to. Program 12-3 demonstrates this.

Program 12-3

```
1  // This program uses the setprecision and fixed
2  // manipulators to format file output.
3  #include <iostream>
4  #include <iomanip>
5  #include <fstream>
6  using namespace std;
7
8  int main()
9  {
10     fstream dataFile;
11     double num = 17.816392;
12
13     dataFile.open("numfile.txt", ios::out);     // Open in output mode
14
```

(program continues)

Program 12-3 *(continued)*

```
15      dataFile << fixed;             // Format for fixed-point notation
16      dataFile << num << endl;       // Write the number
17
18      dataFile << setprecision(4);   // Format for 4 decimal places
19      dataFile << num << endl;       // Write the number
20
21      dataFile << setprecision(3);   // Format for 3 decimal places
22      dataFile << num << endl;       // Write the number
23
24      dataFile << setprecision(2);   // Format for 2 decimal places
25      dataFile << num << endl;       // Write the number
26
27      dataFile << setprecision(1);   // Format for 1 decimal place
28      dataFile << num << endl;       // Write the number
29
30      cout << "Done.\n";
31      dataFile.close();              // Close the file
32      return 0;
33  }
```

Contents of File `numfile.txt`

```
17.816392
17.8164
17.816
17.82
17.8
```

Notice the file output is formatted just as `cout` would format screen output. Program 12-4 shows the `setw` stream manipulator being used to format file output into columns.

Program 12-4

```
1   // This program writes three rows of numbers to a file.
2   #include <iostream>
3   #include <fstream>
4   #include <iomanip>
5   using namespace std;
6
7   int main()
8   {
9       const int ROWS = 3;    // Rows to write
10      const int COLS = 3;    // Columns to write
11      int nums[ROWS][COLS] = { 2897, 5, 837,
12                               34, 7, 1623,
13                               390, 3456, 12 };
14      fstream outFile("table.txt", ios::out);
15
```

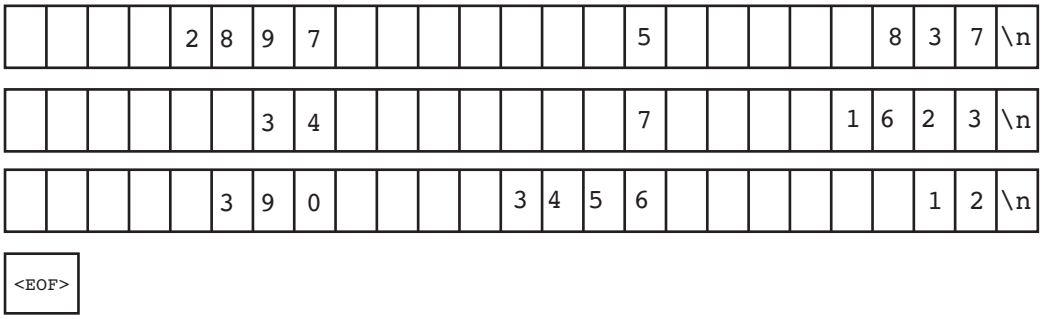
```
16      // Write the three rows of numbers with each
17      // number in a field of 8 character spaces.
18      for (int row = 0; row < ROWS; row++)
19      {
20          for (int col = 0; col < COLS; col++)
21          {
22              outFile << setw(8) << nums[row][col];
23          }
24          outFile << endl;
25      }
26      outFile.close();
27      cout << "Done.\n";
28      return 0;
29  }
```

Contents of File `table.txt`

```
    2897       5     837
      34       7    1623
     390    3456      12
```

Figure 12-4 shows the way the characters appear in the file.

Figure 12-4



12.3 Passing File Stream Objects to Functions

CONCEPT: File stream objects may be passed by reference to functions.

When writing actual programs, you'll want to create modularized code for handling file operations. File stream objects may be passed to functions, but they should always be passed by reference. The `openFile` function shown below uses an `fstream` reference object parameter:

VideoNote
Passing File Stream Objects to Functions

```
bool openFileIn(fstream &file, string name)
{
   bool status;
```

```
      file.open(name, ios::in);
      if (file.fail())
         status = false;
      else
         status = true;
      return status;
   }
```

The internal state of file stream objects changes with most every operation. They should always be passed to functions by reference to ensure internal consistency. Program 12-5 shows an example of how file stream objects may be passed as arguments to functions.

Program 12-5

```
1  // This program demonstrates how file stream objects may
2  // be passed by reference to functions.
3  #include <iostream>
4  #include <fstream>
5  #include <string>
6  using namespace std;
7
8  // Function prototypes
9  bool openFileIn(fstream &, string);
10 void showContents(fstream &);
11
12 int main()
13 {
14     fstream dataFile;
15
16     if (openFileIn(dataFile, "demofile.txt"))
17     {
18         cout << "File opened successfully.\n";
19         cout << "Now reading data from the file.\n\n";
20         showContents(dataFile);
21         dataFile.close();
22         cout << "\nDone.\n";
23     }
24     else
25         cout << "File open error!" << endl;
26
27     return 0;
28 }
29
30 //*********************************************************
31 // Definition of function openFileIn. Accepts a reference  *
32 // to an fstream object as an argument. The file is opened *
33 // for input. The function returns true upon success, false *
34 // upon failure.                                           *
35 //*********************************************************
36
```

```
37  bool openFileIn(fstream &file, string name)
38  {
39       file.open(name, ios::in);
40       if (file.fail())
41            return false;
42       else
43            return true;
44  }
45
46  //***********************************************************
47  // Definition of function showContents. Accepts an fstream  *
48  // reference as its argument. Uses a loop to read each name *
49  // from the file and displays it on the screen.             *
50  //***********************************************************
51
52  void showContents(fstream &file)
53  {
54       string line;
55
56       while (file >> line)
57       {
58            cout << line << endl;
59       }
60  }
```

Program Output

```
File opened successfully.
Now reading data from the file.

Jones
Smith
Willis
Davis

Done.
```

12.4 More Detailed Error Testing

CONCEPT: **All stream objects have error state bits that indicate the condition of the stream.**

All stream objects contain a set of bits that act as flags. These flags indicate the current state of the stream. Table 12-4 lists these bits.

These bits can be tested by the member functions listed in Table 12-5. (You've already learned about the `fail()` function.) One of the functions listed in the table, `clear()`, can be used to set a status bit.

Table 12-4

Bit	Description
`ios::eofbit`	Set when the end of an input stream is encountered.
`ios::failbit`	Set when an attempted operation has failed.
`ios::hardfail`	Set when an unrecoverable error has occurred.
`ios::badbit`	Set when an invalid operation has been attempted.
`ios::goodbit`	Set when all the flags above are not set. Indicates the stream is in good condition.

Table 12-5

Function	Description
`eof()`	Returns true (nonzero) if the `eofbit` flag is set, otherwise returns false.
`fail()`	Returns true (nonzero) if the `failbit` or `hardfail` flags are set, otherwise returns false.
`bad()`	Returns true (nonzero) if the `badbit` flag is set, otherwise returns false.
`good()`	Returns true (nonzero) if the `goodbit` flag is set, otherwise returns false.
`clear()`	When called with no arguments, clears all the flags listed above. Can also be called with a specific flag as an argument.

The function `showState`, shown here, accepts a file stream reference as its argument. It shows the state of the file by displaying the return values of the `eof()`, `fail()`, `bad()`, and `good()` member functions:

```
void showState(fstream &file)
{
   cout << "File Status:\n";
   cout << " eof bit: " << file.eof() << endl;
   cout << " fail bit: " << file.fail() << endl;
   cout << " bad bit: " << file.bad() << endl;
   cout << " good bit: " << file.good() << endl;
   file.clear();      // Clear any bad bits
}
```

Program 12-6 uses the `showState` function to display `testFile`'s status after various operations. First, the file is created and the integer value 10 is stored in it. The file is then closed and reopened for input. The integer is read from the file, and then a second read operation is performed. Because there is only one item in the file, the second read operation will result in an error.

Program 12-6

```
1  // This program demonstrates the return value of the stream
2  // object error testing member functions.
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
```

```
// Function prototype
void showState(fstream &);

int main()
{
     int num = 10;

     // Open the file for output.
     fstream testFile("stuff.dat", ios::out);
     if (testFile.fail())
     {
          cout << "ERROR: Cannot open the file.\n";
          return 0;
     }

     // Write a value to the file.
     cout << "Writing the value " << num << " to the file.\n";
     testFile << num;

     // Show the bit states.
     showState(testFile);

     // Close the file.
     testFile.close();

     // Reopen the file for input.
     testFile.open("stuff.dat", ios::in);
     if (testFile.fail())
     {
          cout << "ERROR: Cannot open the file.\n";
          return 0;
     }

     // Read the only value from the file.
     cout << "Reading from the file.\n";
     testFile >> num;
     cout << "The value " << num << " was read.\n";

     // Show the bit states.
     showState(testFile);

     // No more data in the file, but force an invalid read operation.
     cout << "Forcing a bad read operation.\n";
     testFile >> num;

     // Show the bit states.
     showState(testFile);

     // Close the file.
     testFile.close();
     return 0;
}
```

(program continues)

Program 12-6 *(continued)*

```
60  //*************************************************************
61  // Definition of function showState. This function uses       *
62  // an fstream reference as its parameter. The return values of *
63  // the eof(), fail(), bad(), and good() member functions are   *
64  // displayed. The clear() function is called before the function *
65  // returns.                                                    *
66  //*************************************************************
67
68  void showState(fstream &file)
69  {
70       cout << "File Status:\n";
71       cout << "  eof bit: " << file.eof() << endl;
72       cout << "  fail bit: " << file.fail() << endl;
73       cout << "  bad bit: " << file.bad() << endl;
74       cout << "  good bit: " << file.good() << endl;
75       file.clear(); // Clear any bad bits
76  }
```

Program Output

```
Writing the value 10 to the file.
File Status:
     eof bit: 0
     fail bit: 0
     bad bit: 0
     good bit: 1
Reading from the file.
The value 10 was read.
File Status:
     eof bit: 1
     fail bit: 0
     bad bit: 0
     good bit: 1
Forcing a bad read operation.
File Status:
     eof bit: 1
     fail bit: 1
     bad bit: 0
     good bit: 0
```

12.5 Member Functions for Reading and Writing Files

CONCEPT: File stream objects have member functions for more specialized file reading and writing.

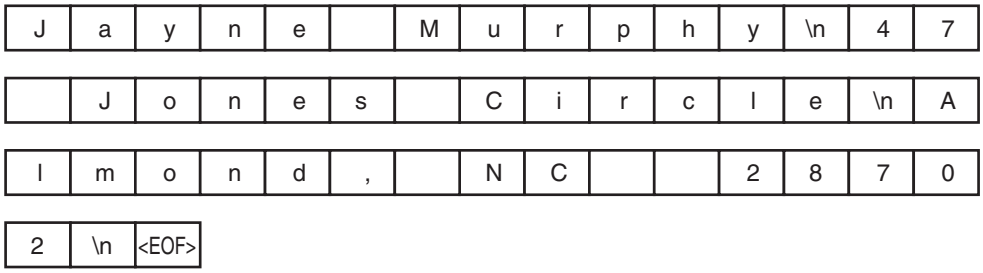
If whitespace characters are part of the data in a file, a problem arises when the file is read by the >> operator. Because the operator considers whitespace characters as delimiters, it

does not read them. For example, consider the file `murphy.txt`, which contains the following data:

Jayne Murphy
47 Jones Circle
Almond, NC 28702

Figure 12-5 shows the way the data is recorded in the file.

Figure 12-5



The problem that arises from the use of the >> operator is evident in the output of Program 12-7.

Program 12-7

```
1  // This program demonstrates how the >> operator should not
2  // be used to read data that contain whitespace characters
3  // from a file.
4  #include <iostream>
5  #include <fstream>
6  #include <string>
7  using namespace std;
8
9  int main()
10 {
11     string input;     // To hold file input
12     fstream nameFile; // File stream object
13
14     // Open the file in input mode.
15     nameFile.open("murphy.txt", ios::in);
16
17     // If the file was successfully opened, continue.
18     if (nameFile)
19     {
20         // Read the file contents.
21         while (nameFile >> input)
22         {
23             cout << input;
24         }
25
```

(program continues)

Program 12-7 *(continued)*

```
26          // Close the file.
27          nameFile.close();
28      }
29      else
30      {
31          cout << "ERROR: Cannot open file.\n";
32      }
33      return 0;
34  }
```

Program Output

```
JayneMurphy47JonesCircleAlmond,NC28702
```

The `getline` Function

The problem with Program 12-7 can be solved by using the `getline` function. The function reads a "line" of data, including whitespace characters. Here is an example of the function call:

```
getline(dataFile, str,'\n');
```

The three arguments in this statement are explained as follows:

`dataFile`	This is the name of the file stream object. It specifies the stream object from which the data is to be read.
`str`	This is the name of a `string` object. The data read from the file will be stored here.
`'\n'`	This is a delimiter character of your choice. If this delimiter is encountered, it will cause the function to stop reading. (This argument is optional. If it's left out, `'\n'` is the default.)

The statement is an instruction to read a line of characters from the file. The function will read until it encounters a `\n`. The line of characters will be stored in the `str` object.

Program 12-8 is a modification of Program 12-7. It uses the `getline` function to read whole lines of data from the file.

Program 12-8

```
1   // This program uses the getline function to read a line of
2   // data from the file.
3   #include <iostream>
4   #include <fstream>
5   #include <string>
6   using namespace std;
7
8   int main()
```

```
{
     string input;     // To hold file input
     fstream nameFile; // File stream object

     // Open the file in input mode.
     nameFile.open("murphy.txt", ios::in);

     // If the file was successfully opened, continue.
     if (nameFile)
     {
          // Read an item from the file.
          getline(nameFile, input);

          // While the last read operation
          // was successful, continue.
          while (nameFile)
          {
               // Display the last item read.
               cout << input << endl;

               // Read the next item.
               getline(nameFile, input);
          }

          // Close the file.
          nameFile.close();
     }
     else
     {
          cout << "ERROR: Cannot open file.\n";
     }
     return 0;
}
```

Program Output

```
Jayne Murphy
47 Jones Circle
Almond, NC 28702
```

Because the third argument of the `getline` function was left out in Program 12-8, its default value is `\n`. Sometimes you might want to specify another delimiter. For example, consider a file that contains multiple names and addresses and that is internally formatted in the following manner:

Contents of `names2.txt`

```
Jayne Murphy$47 Jones Circle$Almond, NC 28702\n$Bobbie Smith$
217 Halifax Drive$Canton, NC 28716\n$Bill Hammet$PO Box 121$
Springfield, NC 28357\n$
```

Think of this file as consisting of three records. A record is a complete set of data about a single item. Also, the records in the file above are made of three fields. The first field is the person's name. The second field is the person's street address or PO box number. The third field contains the person's city, state, and ZIP code. Notice that each field ends with a `$` character, and each record ends with a `\n` character. Program 12-9 demonstrates how a `getline` function can be used to detect the `$` characters.

Program 12-9

```
1  // This file demonstrates the getline function with
2  // a specified delimiter.
3  #include <iostream>
4  #include <fstream>
5  #include <string>
6  using namespace std;
7
8  int main()
9  {
10     string input; // To hold file input
11
12     // Open the file for input.
13     fstream dataFile("names2.txt", ios::in);
14
15     // If the file was successfully opened, continue.
16     if (dataFile)
17     {
18          // Read an item using $ as a delimiter.
19          getline(dataFile, input, '$');
20
21          // While the last read operation
22          // was successful, continue.
23          while (dataFile)
24          {
25               // Display the last item read.
26               cout << input << endl;
27
28               // Read an item using $ as a delimiter.
29               getline(dataFile, input, '$');
30          }
31
32          // Close the file.
33          dataFile.close();
34     }
35     else
36     {
37          cout << "ERROR: Cannot open file.\n";
38     }
39     return 0;
40 }
```

Program Output

```
Jayne Murphy
47 Jones Circle
Almond, NC 28702

Bobbie Smith
217 Halifax Drive
Canton, NC 28716

Bill Hammet
PO Box 121
Springfield, NC 28357
```

Notice that the \n characters, which mark the end of each record, are also part of the output. They cause an extra blank line to be printed on the screen, separating the records.

NOTE: When using a printable character, such as $, to delimit data in a file, be sure to select a character that will not actually appear in the data itself. Since it's doubtful that anyone's name or address contains a $ character, it's an acceptable delimiter. If the file contained dollar amounts, however, another delimiter would have been chosen.

The get Member Function

The file stream object's get member function is also useful. It reads a single character from the file. Here is an example of its usage:

```
inFile.get(ch);
```

In this example, ch is a char variable. A character will be read from the file and stored in ch. Program 12-10 shows the function used in a complete program. The user is asked for the name of a file. The file is opened and the get function is used in a loop to read the file's contents, one character at a time.

Program 12-10

```
1  // This program asks the user for a file name. The file is
2  // opened and its contents are displayed on the screen.
3  #include <iostream>
4  #include <fstream>
5  #include <string>
6  using namespace std;
7
8  int main()
9  {
10     string fileName;  // To hold the file name
11     char ch;          // To hold a character
12     fstream file;     // File stream object
13
14     // Get the file name
15     cout << "Enter a file name: ";
16     cin >> fileName;
17
18     // Open the file.
19     file.open(filename, ios::in);
20
21     // If the file was successfully opened, continue.
22     if (file)
23     {
24          // Get a character from the file.
25          file.get(ch);
26
27          // While the last read operation was
28          // successful, continue.
29          while (file)
30          {
```

(program continues)

Program 12-10 *(continued)*

```
31              // Display the last character read.
32              cout << ch;
33
34              // Read the next character
35              file.get(ch);
36          }
37
38          // Close the file.
39          file.close();
40      }
41      else
42          cout << fileName << " could not be opened.\n";
43      return 0;
44  }
```

Program 12-10 will display the contents of any file. The `get` function even reads whitespaces, so all the characters will be shown exactly as they appear in the file.

The `put` Member Function

The `put` member function writes a single character to the file. Here is an example of its usage:

```
outFile.put(ch);
```

In this statement, the variable `ch` is assumed to be a `char` variable. Its contents will be written to the file associated with the file stream object `outFile`. Program 12-11 demonstrates the `put` function.

Program 12-11

```
1   // This program demonstrates the put member function.
2   #include <iostream>
3   #include <fstream>
4   using namespace std;
5
6   int main()
7   {
8       char ch;  // To hold a character
9
10      // Open the file for output.
11      fstream dataFile("sentence.txt", ios::out);
12
13      cout << "Type a sentence and be sure to end it with a ";
14      cout << "period.\n";
15
16      // Get a sentence from the user one character at a time
17      // and write each character to the file.
18      cin.get(ch);
19      while (ch != '.')
20      {
21          dataFile.put(ch);
22          cin.get(ch);
23      }
```

```
24       dataFile.put(ch); // Write the period.
25
26       // Close the file.
27       dataFile.close();
28       return 0;
29   }
```

Program Output with Example Input Shown in Bold

```
Type a sentence and be sure to end it with a period.
I am on my way to becoming a great programmer. [Enter]
```

Resulting Contents of the File `sentence.txt`:

```
I am on my way to becoming a great programmer.
```

Checkpoint

12.7 Assume the file `input.txt` contains the following characters:

R	u	n		S	p	o	t		r	u	n	\n	S	e

e		S	p	o	t		r	u	n	\n	<EOF>

What will the following program display on the screen?

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
     fstream inFile("input.txt", ios::in);
     string item;
     inFile >> item;
     while (inFile)
     {
          cout << item << endl;
          inFile >> item;
     }
     inFile.close();
     return 0;
}
```

12.8 Describe the difference between reading a file with the >> operator and the `getline` function.

12.9 What will be stored in the file `out.txt` after the following program runs?

```
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
```

```
int main()
{
     const int SIZE = 5;
     ofstream outFile("out.txt");
     double nums[SIZE] = {100.279, 1.719, 8.602, 7.777, 5.099};

     outFile << fixed << setprecision(2);
     for (int count = 0; count < 5; count++)
     {
          outFile << setw(8) << nums[count];
     }
     outFile.close();
     return 0;
}
```

12.6 Focus on Software Engineering: Working with Multiple Files

CONCEPT: **It's possible to have more than one file open at once in a program.**

VideoNote
Working with Multiple Files

Quite often you will need to have multiple files open at once. In many real-world applications, data about a single item are categorized and written to several different files. For example, a payroll system might keep the following files:

`emp.dat`	A file that contains the following data about each employee: name, job title, address, telephone number, employee number, and the date hired.
`pay.dat`	A file that contains the following data about each employee: employee number, hourly pay rate, overtime rate, and number of hours worked in the current pay cycle.
`withhold.dat`	A file that contains the following data about each employee: employee number, dependents, and extra withholdings.

When the system is writing paychecks, you can see that it will need to open each of the files listed above and read data from them. (Notice that each file contains the employee number. This is how the program can locate a specific employee's data.)

In C++, you open multiple files by defining multiple file stream objects. For example, if you need to read from three files, you can define three file stream objects, such as:

```
ifstream file1, file2, file3;
```

Sometimes you will need to open one file for input and another file for output. For example, Program 12-12 asks the user for a file name. The file is opened and read. Each character is converted to uppercase and written to a second file called `out.txt`. This type of program can be considered a *filter*. Filters read the input of one file, changing the data in some fashion, and write it out to a second file. The second file is a modified version of the first file.

Program 12-12

```
1  // This program demonstrates reading from one file and writing
2  // to a second file.
3  #include <iostream>
4  #include <fstream>
5  #include <string>
6  #include <cctype> // Needed for the toupper function.
7  using namespace std;
8
9  int main()
10 {
11     string fileName;     // To hold the file name
12     char ch;             // To hold a character
13     ifstream inFile;     // Input file
14
15     // Open a file for output.
16     ofstream outFile("out.txt");
17
18     // Get the input file name.
19     cout << "Enter a file name: ";
20     cin >> fileName;
21
22     // Open the file for input.
23     inFile.open(filename);
24
25     // If the input file opened successfully, continue.
26     if (inFile)
27     {
28         // Read a char from file 1.
29         inFile.get(ch);
30
31         // While the last read operation was
32         // successful, continue.
33         while (inFile)
34         {
35             // Write uppercase char to file 2.
36             outFile.put(toupper(ch));
37
38             // Read another char from file 1.
39             inFile.get(ch);
40         }
41
42         // Close the two files.
43         inFile.close();
44         outFile.close();
45         cout << "File conversion done.\n";
46     }
47     else
48         cout << "Cannot open " << fileName << endl;
49     return 0;
50 }
```

(program output continues)

Program 12-12 *(continued)*

Program Output with Example Input Shown in Bold

```
Enter a file name: hownow.txt [Enter]
File conversion done.
```

Contents of `hownow.txt`

```
how now brown cow.
How Now?
```

Resulting Contents of `out.txt`

```
HOW NOW BROWN COW.
HOW NOW?
```

12.7 Binary Files

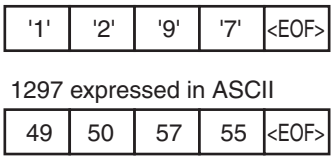
CONCEPT: **Binary files contain data that is not necessarily stored as ASCII text.**

All the files you've been working with so far have been text files. That means the data stored in the files has been formatted as ASCII text. Even a number, when stored in a file with the << operator, is converted to text. For example, consider the following program segment:

```
ofstream file("num.dat");
short x = 1297;
file << x;
```

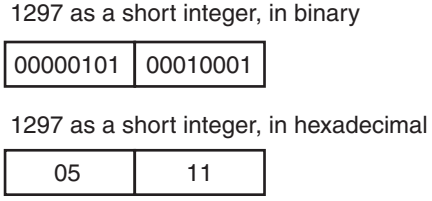
The last statement writes the contents of `x` to the file. When the number is written, however, it is stored as the characters `'1'`, `'2'`, `'9'`, and `'7'`. This is illustrated in Figure 12-6.

Figure 12-6



The number 1297 isn't stored in memory (in the variable `x`) in the fashion depicted in the figure above, however. It is formatted as a binary number, occupying two bytes on a typical PC. Figure 12-7 shows how the number is represented in memory, using binary or hexadecimal.

Figure 12-7



The representation of the number shown in Figure 12-7 is the way the "raw" data is stored in memory. Data can be stored in a file in its pure, binary format. The first step is to open the file in binary mode. This is accomplished by using the `ios::binary` flag. Here is an example:

```
file.open("stuff.dat", ios::out | ios::binary);
```

Notice the `ios::out` and `ios::binary` flags are joined in the statement with the | operator. This causes the file to be opened in both output and binary modes.

NOTE: By default, files are opened in text mode.

The `write` and `read` Member Functions

The file stream object's `write` member function is used to write binary data to a file. The general format of the `write` member function is

```
fileObject.write(address, size);
```

Let's look at the parts of this function call format.

- *`fileObject`* is the name of a file stream object.
- *`address`* is the starting address of the section of memory that is to be written to the file. This argument is expected to be the address of a `char` (or a pointer to a `char`).
- *`size`* is the number of bytes of memory to write. This argument must be an integer value.

For example, the following code uses a file stream object named `file` to write a character to a binary file.

```
char letter = 'A';
file.write(&letter, sizeof(letter));
```

The first argument passed to the `write` function is the address of the `letter` variable. This tells the `write` function where the data that is to be written to the file is located. The second argument is the size of the `letter` variable, which is returned from the `sizeof` operator. This tells the `write` function the number of bytes of data to write to the file. Because the sizes of data types can vary among systems, it is best to use the `sizeof` operator to determine the number of bytes to write. After this function call executes, the contents of the `letter` variable will be written to the binary file associated with the `file` object.

The following code shows another example. This code writes an entire `char` array to a binary file.

```
char data[] = {'A', 'B', 'C', 'D'};
file.write(data, sizeof(data));
```

In this code, the first argument is the name of the `data` array. By passing the name of the array we are passing a pointer to the beginning of the array. Because `data` is an array of `char` values, the name of the array is a pointer to a `char`. The second argument passes the name of the array to the `sizeof` operator. When the name of an array is passed to the `sizeof` operator, the operator returns the number of bytes allocated to the array. After this function call executes, the contents of the `data` array will be written to the binary file associated with the `file` object.

The `read` member function is used to read binary data from a file into memory. The general format of the `read` member function is

```
fileObject.read(address, size);
```

Here are the parts of this function call format:

- *`fileObject`* is the name of a file stream object.
- *`address`* is the starting address of the section of memory where the data being read from the file is to be stored. This is expected to be the address of a `char` (or a pointer to a `char`).
- *`size`* is the number of bytes of memory to read from the file. This argument must be an integer value.

For example, suppose we want to read a single character from a binary file and store that character in the `letter` variable. The following code uses a file stream object named `file` to do just that.

```
char letter;
file.read(&letter, sizeof(letter));
```

The first argument passed to the `read` function is the address of the `letter` variable. This tells the `read` function where to store the value that is read from the file. The second argument is the size of the `letter` variable. This tells the `read` function the number of bytes to read from the file. After this function executes, the `letter` variable will contain a character that was read from the file.

The following code shows another example. This code reads enough data from a binary file to fill an entire `char` array.

```
char data[4];
file.read(data, sizeof(data));
```

In this code, the first argument is the address of the `data` array. The second argument is the number of bytes allocated to the array. On a system that uses 1-byte characters, this function will read four bytes from the file and store them in the `data` array.

Program 12-13 demonstrates writing a `char` array to a file and then reading the data from the file back into memory.

Program 12-13

```
1  // This program uses the write and read functions.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8      const int SIZE = 4;
9      char data[SIZE] = {'A', 'B', 'C', 'D'};
10     fstream file;
11
```

```
12      // Open the file for output in binary mode.
13      file.open("test.dat", ios::out | ios::binary);
14
15      // Write the contents of the array to the file.
16      cout << "Writing the characters to the file.\n";
17      file.write(data, sizeof(data));
18
19      // Close the file.
20      file.close();
21
22      // Open the file for input in binary mode.
23      file.open("test.dat", ios::in | ios::binary);
24
25      // Read the contents of the file into the array.
26      cout << "Now reading the data back into memory.\n";
27      file.read(data, sizeof(data));
28
29      // Display the contents of the array.
30      for (int count = 0; count < SIZE; count++)
31          cout << data[count] << " ";
32      cout << endl;
33
34      // Close the file.
35      file.close();
36      return 0;
37  }
```

Program Output

```
Writing the characters to the file.
Now reading the data back into memory.
A B C D
```

Writing Data Other Than `char` to Binary Files

Because the `write` and `read` member functions expect their first argument to be a pointer to a `char`, you must use a type cast when writing and reading items that are of other data types. To convert a pointer from one type to another you should use the `reinterpret_cast` type cast. The general format of the type cast is

```
reinterpret_cast<dataType>(value)
```

where *`dataType`* is the data type that you are converting to, and *`value`* is the value that you are converting. For example, the following code uses the type cast to store the address of an `int` in a `char` pointer variable.

```
int x = 65;
char *ptr = nullptr;
ptr = reinterpret_cast<char *>(&x);
```

The following code shows how to use the type cast to pass the address of an integer as the first argument to the `write` member function.

```
int x = 27;
file.write(reinterpret_cast<char *>(&x), sizeof(x));
```

After the function executes, the contents of the variable `x` will be written to the binary file associated with the `file` object. The following code shows an `int` array being written to a binary file.

```
const int SIZE = 10;
int numbers[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
file.write(reinterpret_cast<char *>(numbers), sizeof(numbers));
```

After this function call executes, the contents of the `numbers` array will be written to the binary file. The following code shows values being read from the file and stored into the `numbers` array.

```
const int SIZE = 10;
int numbers[SIZE];
file.read(reinterpret_cast<char *>(numbers), sizeof(numbers));
```

Program 12-14 demonstrates writing an `int` array to a file and then reading the data from the file back into memory.

Program 12-14

```
1  // This program uses the write and read functions.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8      const int SIZE = 10;
9      fstream file;
10     int numbers[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
11
12     // Open the file for output in binary mode.
13     file.open("numbers.dat", ios::out | ios::binary);
14
15     // Write the contents of the array to the file.
16     cout << "Writing the data to the file.\n";
17     file.write(reinterpret_cast<char *>(numbers), sizeof(numbers));
18
19     // Close the file.
20     file.close();
21
22     // Open the file for input in binary mode.
23     file.open("numbers.dat", ios::in | ios::binary);
24
25     // Read the contents of the file into the array.
26     cout << "Now reading the data back into memory.\n";
27     file.read(reinterpret_cast<char *>(numbers), sizeof(numbers));
28
29     // Display the contents of the array.
30     for (int count = 0; count < SIZE; count++)
31         cout << numbers[count] << " ";
32     cout << endl;
33
34     // Close the file.
35     file.close();
36     return 0;
37 }
```

Program Output

```
Writing the data to the file.
Now reading the data back into memory.
1 2 3 4 5 6 7 8 9 10
```

12.8 Creating Records with Structures

CONCEPT: Structures may be used to store fixed-length records to a file.

Earlier in this chapter the concept of fields and records was introduced. A field is an individual piece of data pertaining to a single item. A record is made up of fields and is a complete set of data about a single item. For example, a set of fields might be a person's name, age, address, and phone number. Together, all those fields that pertain to one person make up a record.

In C++, structures provide a convenient way to organize data into fields and records. For example, the following code could be used to create a record containing data about a person.

```
const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;

struct Info
{
     char name[NAME_SIZE];
     int age;
     char address1[ADDR_SIZE];
     char address2[ADDR_SIZE];
     char phone[PHONE_SIZE];
};
```

Besides providing an organizational structure for data, structures also package data into a single unit. For example, assume the structure variable `person` is defined as

```
Info person;
```

Once the members (or fields) of `person` are filled with data, the entire variable may be written to a file using the `write` function:

```
file.write(reinterpret_cast<char *>(&person), sizeof(person));
```

The first argument is the address of the `person` variable. The `reinterpret_cast` operator is used to convert the address to a `char` pointer. The second argument is the `sizeof` operator with `person` as its `argument`. This returns the number of bytes used by the `person` structure. Program 12-15 demonstrates this technique.

NOTE: Because structures can contain a mixture of data types, you should always use the `ios::binary` mode when opening a file to store them.

Program 12-15

```
// This program uses a structure variable to store a record to a file.
#include <iostream>
#include <fstream>
using namespace std;

// Array sizes
const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;

// Declare a structure for the record.
struct Info
{
    char name[NAME_SIZE];
    int age;
    char address1[ADDR_SIZE];
    char address2[ADDR_SIZE];
    char phone[PHONE_SIZE];
};

int main()
{
    Info person;    // To hold info about a person
    char again;     // To hold Y or N

    // Open a file for binary output.
    fstream people("people.dat", ios::out | ios::binary);

    do
    {
        // Get data about a person.
        cout << "Enter the following data about a "
             << "person:\n";
        cout << "Name: ";
        cin.getline(person.name, NAME_SIZE);
        cout << "Age: ";
        cin >> person.age;
        cin.ignore(); // Skip over the remaining newline.
        cout << "Address line 1: ";
        cin.getline(person.address1, ADDR_SIZE);
        cout << "Address line 2: ";
        cin.getline(person.address2, ADDR_SIZE);
        cout << "Phone: ";
        cin.getline(person.phone, PHONE_SIZE);

        // Write the contents of the person structure to the file.
        people.write(reinterpret_cast<char *>(&person),
                     sizeof(person));

        // Determine whether the user wants to write another record.
        cout << "Do you want to enter another record? ";
        cin >> again;
        cin.ignore(); // Skip over the remaining newline.
    } while (again == 'Y' || again == 'y');
```

```
53
54      // Close the file.
55      people.close();
56      return 0;
57  }
```

Program Output with Example Input Shown in Bold

```
Enter the following data about a person:
Name: Charlie Baxter [Enter]
Age: 42 [Enter]
Address line 1: 67 Kennedy Blvd. [Enter]
Address line 2: Perth, SC 38754 [Enter]
Phone: (803)555-1234 [Enter]
Do you want to enter another record? Y [Enter]
Enter the following data about a person:
Name: Merideth Murney [Enter]
Age: 22 [Enter]
Address line 1: 487 Lindsay Lane [Enter]
Address line 2: Hazelwood, NC 28737 [Enter]
Phone: (828)555-9999 [Enter]
Do you want to enter another record? N [Enter]
```

Program 12-15 allows you to build a file by filling the members of the `person` variable, and then writing the variable to the file. Program 12-16 opens the file and reads each record into the `person` variable, then displays the data on the screen.

Program 12-16

```
1   // This program uses a structure variable to read a record from a file.
2   #include <iostream>
3   #include <fstream>
4   using namespace std;
5
6   const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;
7
8   // Declare a structure for the record.
9   struct Info
10  {
11      char name[NAME_SIZE];
12      int age;
13      char address1[ADDR_SIZE];
14      char address2[ADDR_SIZE];
15      char phone[PHONE_SIZE];
16  };
17
18  int main()
19  {
20      Info person;      // To hold info about a person
21      char again;       // To hold Y or N
22      fstream people;   // File stream object
23
24      // Open the file for input in binary mode.
25      people.open("people.dat", ios::in | ios::binary);
26
```

(program continues)

Program 12-16 *(continued)*

```
27      // Test for errors.
28      if (!people)
29      {
30          cout << "Error opening file. Program aborting.\n";
31          return 0;
32      }
33
34
35      cout << "Here are the people in the file:\n\n";
36      // Read the first record from the file.
37      people.read(reinterpret_cast<char *>(&person),
38                  sizeof(person));
39
40      // While not at the end of the file, display
41      // the records.
42      while (!people.eof())
43      {
44          // Display the record.
45          cout << "Name: ";
46          cout << person.name << endl;
47          cout << "Age: ";
48          cout << person.age << endl;
49          cout << "Address line 1: ";
50          cout << person.address1 << endl;
51          cout << "Address line 2: ";
52          cout << person.address2 << endl;
53          cout << "Phone: ";
54          cout << person.phone << endl;
55
56          // Wait for the user to press the Enter key.
57          cout << "\nPress the Enter key to see the next record.\n";
58          cin.get(again);
59
60          // Read the next record from the file.
61          people.read(reinterpret_cast<char *>(&person),
62                      sizeof(person));
63      }
64
65      cout << "That's all the data in the file!\n";
66      people.close();
67      return 0;
68  }
```

Program Output (Using the same file created by Program 12-15 as input)

```
Here are the people in the file:

Name: Charlie Baxter
Age: 42
Address line 1: 67 Kennedy Blvd.
Address line 2: Perth, SC 38754
Phone: (803)555-1234
```

```
Press the Enter key to see the next record.

Name: Merideth Murney
Age: 22
Address line 1: 487 Lindsay Lane
Address line 2: Hazelwood, NC 28737
Phone: (828)555-9999

Press the Enter key to see the next record.

That's all the data in the file!
```

NOTE: Structures containing pointers cannot be correctly stored to disk using the techniques of this section. This is because if the structure is read into memory on a subsequent run of the program, it cannot be guaranteed that all program variables will be at the same memory locations. Because `string` class objects contain implicit pointers, they cannot be a part of a structure that has to be stored.

12.9 Random-Access Files

CONCEPT: **Random access means nonsequentially accessing data in a file.**

All of the programs created so far in this chapter have performed *sequential file access*. When a file is opened, the position where reading and/or writing will occur is at the file's beginning (unless the `ios::app` mode is used, which causes data to be written to the end of the file). If the file is opened for output, bytes are written to it one after the other. If the file is opened for input, data is read beginning at the first byte. As the reading or writing continues, the file stream object's read/write position advances sequentially through the file's contents.

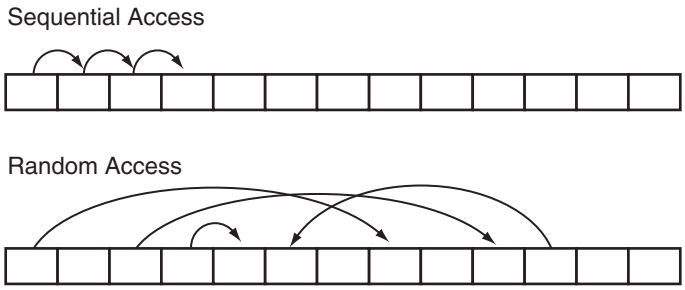
The problem with sequential file access is that in order to read a specific byte from the file, all the bytes that precede it must be read first. For instance, if a program needs data stored at the hundredth byte of a file, it will have to read the first 99 bytes to reach it. If you've ever listened to a cassette tape player, you understand sequential access. To listen to a song at the end of the tape, you have to listen to all the songs that come before it, or fast-forward over them. There is no way to immediately jump to that particular song.

Although sequential file access is useful in many circumstances, it can slow a program down tremendously. If the file is very large, locating data buried deep inside it can take a long time. Alternatively, C++ allows a program to perform *random file access*. In random file access, a program may immediately jump to any byte in the file without first reading the preceding bytes. The difference between sequential and random file access is like the difference between a cassette tape and a compact disc. When listening to a CD, there is no need to listen to or fast forward over unwanted songs. You simply jump to the track that you want to listen to. This is illustrated in Figure 12-8.

The `seekp` and `seekg` Member Functions

File stream objects have two member functions that are used to move the read/write position to any byte in the file. They are `seekp` and `seekg`. The `seekp` function is used with

Figure 12-8



files opened for output, and `seekg` is used with files opened for input. (It makes sense if you remember that "p" stands for "put" and "g" stands for "get." `seekp` is used with files that you put data into, and `seekg` is used with files you get data out of.)

Here is an example of `seekp`'s usage:

```
file.seekp(20L, ios::beg);
```

The first argument is a `long` integer representing an offset into the file. This is the number of the byte you wish to move to. In this example, `20L` is used. (Remember, the L suffix forces the compiler to treat the number as a long integer.) This statement moves the file's write position to byte number 20. (All numbering starts at 0, so byte number 20 is actually the twenty-first byte.)

The second argument is called the mode, and it designates where to calculate the offset *from*. The flag `ios::beg` means the offset is calculated from the beginning of the file. Alternatively, the offset can be calculated from the end of the file or the current position in the file. Table 12-6 lists the flags for all three of the random-access modes.

Table 12-6

Mode Flag	Description
`ios::beg`	The offset is calculated from the beginning of the file.
`ios::end`	The offset is calculated from the end of the file.
`ios::cur`	The offset is calculated from the current position.

Table 12-7 shows examples of `seekp` and `seekg` using the various mode flags.

Notice that some of the examples in Table 12-7 use a negative offset. Negative offsets result in the read or write position being moved backward in the file, while positive offsets result in a forward movement.

Assume the file `letters.txt` contains the following data:

```
abcdefghijklmnopqrstuvwxyz
```

Program 12-17 uses the `seekg` function to jump around to different locations in the file, retrieving a character after each stop.

Table 12-7

Statement	How It Affects the Read/Write Position
`file.seekp(32L, ios::beg);`	Sets the write position to the 33rd byte (byte 32) from the beginning of the file.
`file.seekp(-10L, ios::end);`	Sets the write position to the 10th byte from the end of the file.
`file.seekp(120L, ios::cur);`	Sets the write position to the 121st byte (byte 120) from the current position.
`file.seekg(2L, ios::beg);`	Sets the read position to the 3rd byte (byte 2) from the beginning of the file.
`file.seekg(-100L, ios::end);`	Sets the read position to the 100th byte from the end of the file.
`file.seekg(40L, ios::cur);`	Sets the read position to the 41st byte (byte 40) from the current position.
`file.seekg(0L, ios::end);`	Sets the read position to the end of the file.

Program 12-17

```
1  // This program demonstrates the seekg function.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8       char ch; // To hold a character
9
10      // Open the file for input.
11      fstream file("letters.txt", ios::in);
12
13      // Move to byte 5 from the beginning of the file
14      // (the 6th byte) and read the character there.
15      file.seekg(5L, ios::beg);
16      file.get(ch);
17      cout << "Byte 5 from beginning: " << ch << endl;
18
19      // Move to the 10th byte from the end of the file
20      // and read the character there.
21      file.seekg(-10L, ios::end);
22      file.get(ch);
23      cout << "10th byte from end: " << ch << endl;
24
25      // Move to byte 3 from the current position
26      // (the 4th byte) and read the character there.
27      file.seekg(3L, ios::cur);
28      file.get(ch);
29      cout << "Byte 3 from current: " << ch << endl;
30
31      file.close();
32      return 0;
33 }
```

(program output continues)

Program 12-17 *(continued)*

Program Screen Output

```
Byte 5 from beginning: f
10th byte from end: q
Byte 3 from current: u
```

Program 12-18 shows a more robust example of the `seekg` function. It opens the `people.dat` file created by Program 12-15. The file contains two records. Program 12-18 displays record 1 (the second record) first, then displays record 0.

The program has two important functions other than `main`. The first, `byteNum`, takes a record number as its argument and returns that record's starting byte. It calculates the record's starting byte by multiplying the record number by the size of the `Info` structure. This returns the offset of that record from the beginning of the file. The second function, `showRec`, accepts an `Info` structure as its argument and displays its contents on the screen.

Program 12-18

```
1  // This program randomly reads a record of data from a file.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;
7
8  // Declare a structure for the record.
9  struct Info
10 {
11     char name[NAME_SIZE];
12     int age;
13     char address1[ADDR_SIZE];
14     char address2[ADDR_SIZE];
15     char phone[PHONE_SIZE];
16 };
17
18 // Function Prototypes
19 long byteNum(int);
20 void showRec(Info);
21
22 int main()
23 {
24     Info person;     // To hold info about a person
25     fstream people;  // File stream object
26
27     // Open the file for input in binary mode.
28     people.open("people.dat", ios::in | ios::binary);
29
```

```
30      // Test for errors.
31      if (!people)
32      {
33          cout << "Error opening file. Program aborting.\n";
34          return 0;
35      }
36
37      // Read and display record 1 (the second record).
38      cout << "Here is record 1:\n";
39      people.seekg(byteNum(1), ios::beg);
40      people.read(reinterpret_cast<char *>(&person), sizeof(person));
41      showRec(person);
42
43      // Read and display record 0 (the first record).
44      cout << "\nHere is record 0:\n";
45      people.seekg(byteNum(0), ios::beg);
46      people.read(reinterpret_cast<char *>(&person), sizeof(person));
47      showRec(person);
48
49      // Close the file.
50      people.close();
51      return 0;
52  }
53
54  //*********************************************************
55  // Definition of function byteNum. Accepts an integer as    *
56  // its argument. Returns the byte number in the file of the *
57  // record whose number is passed as the argument.           *
58  //*********************************************************
59
60  long byteNum(int recNum)
61  {
62      return sizeof(Info) * recNum;
63  }
64
65  //*********************************************************
66  // Definition of function showRec. Accepts an Info structure *
67  // as its argument, and displays the structure's contents.  *
68  //*********************************************************
69
70  void showRec(Info record)
71  {
72      cout << "Name: ";
73      cout << record.name << endl;
74      cout << "Age: ";
75      cout << record.age << endl;
76      cout << "Address line 1: ";
77      cout << record.address1 << endl;
78      cout << "Address line 2: ";
79      cout << record.address2 << endl;
```

(program continues)

Program 12-18 *(continued)*

```
80       cout << "Phone: ";
81       cout << record.phone << endl;
82   }
```

Program Output (Using the same file created by Program 12–15 as input)

```
Here is record 1:
Name: Merideth Murney
Age: 22
Address line 1: 487 Lindsay Lane
Address line 2: Hazelwood, NC 28737
Phone: (828)555-9999

Here is record 0:
Name: Charlie Baxter
Age: 42
Address line 1: 67 Kennedy Blvd.
Address line 2: Perth, SC 38754
Phone: (803)555-1234
```

WARNING! If a program has read to the end of a file, you must call the file stream object's `clear` member function before calling `seekg` or `seekp`. This clears the file stream object's `eof` flag. Otherwise, the `seekg` or `seekp` function will not work.

The `tellp` and `tellg` Member Functions

File stream objects have two more member functions that may be used for random file access: `tellp` and `tellg`. Their purpose is to return, as a long integer, the current byte number of a file's read and write position. As you can guess, `tellp` returns the write position and `tellg` returns the read position. Assuming `pos` is a long integer, here is an example of the functions' usage:

```
pos = outFile.tellp();
pos = inFile.tellg();
```

One application of these functions is to determine the number of bytes that a file contains. The following example demonstrates how to do this using the `tellg` function.

```
file.seekg(0L, ios::end);
numBytes = file.tellg();
cout << "The file has " << numBytes << " bytes.\n";
```

First the `seekg` member function is used to move the read position to the last byte in the file. Then the `tellg` function is used to get the current byte number of the read position.

Program 12-19 demonstrates the `tellg` function. It opens the `letters.txt` file, which was also used in Program 12-17. The file contains the following characters:

```
abcdefghijklmnopqrstuvwxyz
```

Program 12-19

```
// This program demonstrates the tellg function.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    long offset;    // To hold an offset amount
    long numBytes;  // To hold the file size
    char ch;        // To hold a character
    char again;     // To hold Y or N

    // Open the file for input.
    fstream file("letters.txt", ios::in);

    // Determine the number of bytes in the file.
    file.seekg(0L, ios::end);
    numBytes = file.tellg();
    cout << "The file has " << numBytes << " bytes.\n";

    // Go back to the beginning of the file.
    file.seekg(0L, ios::beg);

    // Let the user move around within the file.
    do
    {
        // Display the current read position.
        cout << "Currently at position " << file.tellg() << endl;

        // Get a byte number from the user.
        cout << "Enter an offset from the beginning of the file: ";
        cin >> offset;

        // Move the read position to that byte, read the
        // character there, and display it.
        if (offset >= numBytes)   // Past the end of the file?
            cout << "Cannot read past the end of the file.\n";
        else
        {
            file.seekg(offset, ios::beg);
            file.get(ch);
            cout << "Character read: " << ch << endl;
        }

        // Does the user want to try this again?
        cout << "Do it again? ";
        cin >> again;
    } while (again == 'Y' || again == 'y');

    // Close the file.
    file.close();
    return 0;
}
```

(program output continues)

Program 12-19 *(continued)*

Program Output with Example Input Shown in Bold

```
The file has 26 bytes.
Currently at position 0
Enter an offset from the beginning of the file: 5 [Enter]
Character read: f
Do it again? y [Enter]
Currently at position 6
Enter an offset from the beginning of the file: 0 [Enter]
Character read: a
Do it again? y [Enter]
Currently at position 1
Enter an offset from the beginning of the file: 26 [Enter]
Cannot read past the end of the file.
Do it again? n [Enter]
```

Rewinding a Sequential-Access File with `seekg`

Sometimes when processing a sequential file, it is necessary for a program to read the contents of the file more than one time. For example, suppose a program searches a file for an item specified by the user. The program must open the file, read its contents, and determine if the specified item is in the file. If the user needs to search the file again for another item, the program must read the file's contents again.

One simple approach for reading a file's contents more than once is to close and reopen the file, as shown in the following code example.

```
dataFile.open("file.txt", ios::in);      // Open the file.

//
// Read and process the file's contents.
//

dataFile.close();                        // Close the file.
dataFile.open("file.txt", ios::in);      // Open the file again.

//
// Read and process the file's contents again.
//

dataFile.close();                        // Close the file.
```

Each time the file is reopened, its read position is located at the beginning of the file. The read position is the byte in the file that will be read with the next read operation.

Another approach is to "rewind" the file. This means moving the read position to the beginning of the file without closing and reopening it. This is accomplished with the file stream object's `seekg` member function to move the read position back to the beginning of the file. The following example code demonstrates.

```
dataFile.open("file.txt", ios::in);      // Open the file.

//
// Read and process the file's contents.
//
```

```
dataFile.clear();                          // Clear the eof flag.
dataFile.seekg(0L, ios::beg);              // Rewind the read position.

//
// Read and process the file's contents again.
//

dataFile.close();                          // Close the file.
```

Notice that prior to calling the `seekg` member function, the `clear` member function is called. As previously mentioned this clears the file object's `eof` flag and is necessary only if the program has read to the end of the file. This approach eliminates the need to close and reopen the file each time the file's contents are processed.

12.10 Opening a File for Both Input and Output

CONCEPT: **You may perform input and output on an `fstream` file without closing it and reopening it.**

Sometimes you'll need to perform both input and output on a file without closing and reopening it. For example, consider a program that allows you to search for a record in a file and then make changes to it. A read operation is necessary to copy the data from the file to memory. After the desired changes have been made to the data in memory, a write operation is necessary to replace the old data in the file with the new data in memory.

Such operations are possible with `fstream` objects. The `ios::in` and `ios::out` file access flags may be joined with the `|` operator, as shown in the following statement.

```
fstream file("data.dat", ios::in | ios::out)
```

The same operation may be accomplished with the `open` member function:

```
file.open("data.dat", ios::in | ios::out);
```

You may also specify the `ios::binary` flag if binary data is to be written to the file. Here is an example:

```
file.open("data.dat", ios::in | ios::out | ios::binary);
```

When an `fstream` file is opened with both the `ios::in` and `ios::out` flags, the file's current contents are preserved, and the read/write position is initially placed at the beginning of the file. If the file does not exist, it is created.

Programs 12-20, 12-21, and 12-22 demonstrate many of the techniques we have discussed. Program 12-20 sets up a file with five blank inventory records. Each record is a structure with members for holding a part description, quantity on hand, and price. Program 12-21 displays the contents of the file on the screen. Program 12-22 opens the file in both input and output modes and allows the user to change the contents of a specific record.

Program 12-20

```
1  // This program sets up a file of blank inventory records.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  // Constants
7  const int DESC_SIZE = 31;    // Description size
8  const int NUM_RECORDS = 5;   // Number of records
9
10 // Declaration of InventoryItem structure
11 struct InventoryItem
12 {
13     char desc[DESC_SIZE];
14     int qty;
15     double price;
16 };
17
18 int main()
19 {
20     // Create an empty InventoryItem structure.
21     InventoryItem record = { "", 0, 0.0 };
22
23     // Open the file for binary output.
24     fstream inventory("Inventory.dat", ios::out | ios::binary);
25
26     // Write the blank records
27     for (int count = 0; count < NUM_RECORDS; count++)
28     {
29         cout << "Now writing record " << count << endl;
30         inventory.write(reinterpret_cast<char *>(&record),
31                         sizeof(record));
32     }
33
34     // Close the file.
35     inventory.close();
36     return 0;
37 }
```

Program Output

```
Now writing record 0
Now writing record 1
Now writing record 2
Now writing record 3
Now writing record 4
```

Program 12-21 simply displays the contents of the inventory file on the screen. It can be used to verify that Program 12-20 successfully created the blank records, and that Program 12-22 correctly modified the designated record.

Program 12-21

```
// This program displays the contents of the inventory file.
#include <iostream>
#include <fstream>
using namespace std;

const int DESC_SIZE = 31; // Description size

// Declaration of InventoryItem structure
struct InventoryItem
{
    char desc[DESC_SIZE];
    int qty;
    double price;
};

int main()
{
    InventoryItem record; // To hold an inventory record

    // Open the file for binary input.
    fstream inventory("Inventory.dat", ios::in | ios::binary);

    // Now read and display the records
    inventory.read(reinterpret_cast<char *>(&record),
                   sizeof(record));
    while (!inventory.eof())
    {
        cout << "Description: ";
        cout << record.desc << endl;
        cout << "Quantity: ";
        cout << record.qty << endl;
        cout << "Price: ";
        cout << record.price << endl << endl;
        inventory.read(reinterpret_cast<char *>(&record),
                       sizeof(record));
    }

    // Close the file.
    inventory.close();
    return 0;
}
```

Here is the screen output of Program 12-21 if it is run immediately after Program 12-20 sets up the file of blank records.

Program 12-21

Program Output

```
Description:
Quantity: 0
Price: 0.0
```

(program output continues)

Program 12-21 *(continued)*

```
Description:
Quantity: 0
Price: 0.0

Description:
Quantity: 0
Price: 0.0

Description:
Quantity: 0
Price: 0.0

Description:
Quantity: 0
Price: 0.0
```

Program 12-22 allows the user to change the contents of an individual record in the inventory file.

Program 12-22

```
1  // This program allows the user to edit a specific record.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  const int DESC_SIZE = 31; // Description size
7
8  // Declaration of InventoryItem structure
9  struct InventoryItem
10 {
11     char desc[DESC_SIZE];
12     int qty;
13     double price;
14 };
15
16 int main()
17 {
18     InventoryItem record;  // To hold an inventory record
19     long recNum;           // To hold a record number
20
21     // Open the file in binary mode for input and output
22     fstream inventory("Inventory.dat",
23                       ios::in | ios::out | ios::binary);
24
25     // Get the record number of the desired record.
26     cout << "Which record do you want to edit? ";
27     cin >> recNum;
28
29     // Move to the record and read it.
30     inventory.seekg(recNum * sizeof(record), ios::beg);
31     inventory.read(reinterpret_cast<char *>(&record),
32                    sizeof(record));
```

```
33
34      // Display the record contents.
35      cout << "Description: ";
36      cout << record.desc << endl;
37      cout << "Quantity: ";
38      cout << record.qty << endl;
39      cout << "Price: ";
40      cout << record.price << endl;
41
42      // Get the new record data.
43      cout << "Enter the new data:\n";
44      cout << "Description: ";
45      cin.ignore();
46      cin.getline(record.desc, DESC_SIZE);
47      cout << "Quantity: ";
48      cin >> record.qty;
49      cout << "Price: ";
50      cin >> record.price;
51
52      // Move back to the beginning of this record's position.
53      inventory.seekp(recNum * sizeof(record), ios::beg);
54
55      // Write the new record over the current record.
56      inventory.write(reinterpret_cast<char *>(&record),
57                      sizeof(record));
58
59      // Close the file.
60      inventory.close();
61      return 0;
62  }
```

Program Output with Example Input Shown in Bold

```
Which record do you want to edit? 2 [Enter]
Description:
Quantity: 0
Price: 0.0
Enter the new data:
Description: Wrench [Enter]
Quantity: 10 [Enter]
Price: 4.67 [Enter]
```

Checkpoint

12.10 Describe the difference between the `seekg` and the `seekp` functions.

12.11 Describe the difference between the `tellg` and the `tellp` functions.

12.12 Describe the meaning of the following file access flags:

```
ios::beg
ios::end
ios::cur
```

12.13 What is the number of the first byte in a file?

12.14 Briefly describe what each of the following statements does:

```
file.seekp(100L, ios::beg);
file.seekp(-10L, ios::end);
file.seekg(-25L, ios::cur);
file.seekg(30L, ios::cur);
```

12.15 Describe the mode that each of the following statements causes a file to be opened in:

```
file.open("info.dat", ios::in | ios::out);
file.open("info.dat", ios::in | ios::app);
file.open("info.dat", ios::in | ios::out | ios::ate);
file.open("info.dat", ios::in | ios::out | ios::binary);
```

For another example of this chapter's topics, see the High Adventure Travel Part 3 Case Study, available on the book's companion Web site at www.pearsonhighered.com/gaddis.

Review Questions and Exercises

Short Answer

1. What capability does the `fstream` data type provide that the `ifstream` and `ofstream` data types do not?
2. Which file access flag do you use to open a file when you want all output written to the end of the file's existing contents?
3. Assume that the file `data.txt` already exists, and the following statement executes. What happens to the file?

   ```
   fstream file("data.txt", ios::out);
   ```

4. How do you combine multiple file access flags when opening a file?
5. Should file stream objects be passed to functions by value or by reference? Why?
6. Under what circumstances is a file stream object's `ios::hardfail` bit set? What member function reports the state of this bit?
7. Under what circumstances is a file stream object's `ios::eofbit` bit set? What member function reports the state of this bit?
8. Under what circumstances is a file stream object's `ios::badbit` bit set? What member function reports the state of this bit?
9. How do you read the contents of a text file that contains whitespace characters as part of its data?
10. What arguments do you pass to a file stream object's `write` member function?
11. What arguments do you pass to a file stream object's `read` member function?
12. What type cast do you use to convert a pointer from one type to another?
13. What is the difference between the `seekg` and `seekp` member functions?
14. How do you get the byte number of a file's current read position? How do you get the byte number of a file's current write position?
15. If a program has read to the end of a file, what must you do before using either the `seekg` or `seekp` member functions?

16. How do you determine the number of bytes that a file contains?
17. How do you rewind a sequential-access file?

Fill-in-the-Blank

18. The _____ file stream data type is for output files, input files, or files that perform both input and output.
19. If a file fails to open, the file stream object will be set to _____.
20. The same formatting techniques used with _____ may also be used when writing data to a file.
21. The _____ function reads a line of text from a file.
22. The _____ member function reads a single character from a file.
23. The _____ member function writes a single character to a file.
24. _____ files contain data that is unformatted and not necessarily stored as ASCII text.
25. _____ files contain data formatted as _____.
26. A(n) _____ is a complete set of data about a single item and is made up of _____.
27. In C++, _____ provide a convenient way to organize data into fields and records.
28. The _____ member function writes "raw" binary data to a file.
29. The _____ member function reads "raw" binary data from a file.
30. The _____ operator is necessary if you pass anything other than a pointer-to-char as the first argument of the two functions mentioned in questions 26 and 27.
31. In _____ file access, the contents of the file are read in the order they appear in the file, from the file's start to its end.
32. In _____ file access, the contents of a file may be read in any order.
33. The _____ member function moves a file's read position to a specified byte in the file.
34. The _____ member function moves a file's write position to a specified byte in the file.
35. The _____ member function returns a file's current read position.
36. The _____ member function returns a file's current write position.
37. The _____ mode flag causes an offset to be calculated from the beginning of a file.
38. The _____ mode flag causes an offset to be calculated from the end of a file.
39. The _____ mode flag causes an offset to be calculated from the current position in the file.
40. A negative offset causes the file's read or write position to be moved _____ in the file from the position specified by the mode.

Algorithm Workbench

41. Write a statement that defines a file stream object named `places`. The object will be used for both output and input.

42. Write two statements that use a file stream object named `people` to open a file named `people.dat`. (Show how to open the file with a member function and at the definition of the file stream object.) The file should be opened for output.
43. Write two statements that use a file stream object named `pets` to open a file named `pets.dat`. (Show how to open the file with a member function and at the definition of the file stream object.) The file should be opened for input.
44. Write two statements that use a file stream object named `places` to open a file named `places.dat`. (Show how to open the file with a member function and at the definition of the file stream object.) The file should be opened for both input and output.
45. Write a program segment that defines a file stream object named `employees`. The file should be opened for both input and output (in binary mode). If the file fails to open, the program segment should display an error message.
46. Write code that opens the file `data.txt` for both input and output, but first determines if the file exists. If the file does not exist, the code should create it, then open it for both input and output.
47. Write code that determines the number of bytes contained in the file associated with the file stream object `dataFile`.
48. The `infoFile` file stream object is used to sequentially access data. The program has already read to the end of the file. Write code that rewinds the file.

True or False

49. T F Different operating systems have different rules for naming files.
50. T F `fstream` objects are only capable of performing file output operations.
51. T F `ofstream` objects, by default, delete the contents of a file if it already exists when opened.
52. T F `ifstream` objects, by default, create a file if it doesn't exist when opened.
53. T F Several file access flags may be joined by using the | operator.
54. T F A file may be opened in the definition of the file stream object.
55. T F If a file is opened in the definition of the file stream object, no mode flags may be specified.
56. T F A file stream object's `fail` member function may be used to determine if the file was successfully opened.
57. T F The same output formatting techniques used with `cout` may also be used with file stream objects.
58. T F The >> operator expects data to be delimited by whitespace characters.
59. T F The `getline` member function can be used to read text that contains whitespaces.
60. T F It is not possible to have more than one file open at once in a program.
61. T F Binary files contain unformatted data, not necessarily stored as text.
62. T F Binary is the default mode in which files are opened.
63. T F The `tellp` member function tells a file stream object which byte to move its write position to.
64. T F It is possible to open a file for both input and output.

Find the Error

Each of the following programs or program segments has errors. Find as many as you can.

65.
```
fstream file(ios::in | ios::out);
file.open("info.dat");
if (!file)
{
   cout << "Could not open file.\n";
}
```

66.
```
ofstream file;
file.open("info.dat", ios::in);
if (file)
{
   cout << "Could not open file.\n";
}
```

67.
```
fstream file("info.dat");
if (!file)
{
   cout << "Could not open file.\n";
}
```

68.
```
fstream dataFile("info.dat", ios:in | ios:binary);
int x = 5;
dataFile << x;
```

69.
```
fstream dataFile("info.dat", ios:in);
char stuff[81];
dataFile.get(stuff);
```

70.
```
fstream dataFile("info.dat", ios:in);
char stuff[81] = "abcdefghijklmnopqrstuvwxyz";
dataFile.put(stuff);
```

71.
```
fstream dataFile("info.dat", ios:out);
struct Date
{
   int month;
   int day;
   int year;
};
Date dt = { 4, 2, 98 };
dataFile.write(&dt, sizeof(int));
```

72.
```
fstream inFile("info.dat", ios:in);
int x;
inFile.seekp(5);
inFile >> x;
```

Programming Challenges

1. **File Head Program**

   Write a program that asks the user for the name of a file. The program should display the first 10 lines of the file on the screen (the "head" of the file). If the file has fewer

than 10 lines, the entire file should be displayed, with a message indicating the entire file has been displayed.

NOTE: Using an editor, you should create a simple text file that can be used to test this program.

2. File Display Program

Write a program that asks the user for the name of a file. The program should display the contents of the file on the screen. If the file's contents won't fit on a single screen, the program should display 24 lines of output at a time, and then pause. Each time the program pauses, it should wait for the user to strike a key before the next 24 lines are displayed.

NOTE: Using an editor, you should create a simple text file that can be used to test this program.

3. Punch Line

Write a program that reads and prints a joke and its punch line from two different files. The first file contains a joke, but not its punch line. The second file has the punch line as its last line, preceded by "garbage." The `main` function of your program should open the two files and then call two functions, passing each one the file it needs. The first function should read and display each line in the file it is passed (the joke file). The second function should display only the last line of the file it is passed (the punch line file). It should find this line by seeking to the end of the file and then backing up to the beginning of the last line. Data to test your program can be found in the `joke.txt` and `punchline.txt` files.

4. Tail Program

Write a program that asks the user for the name of a file. The program should display the last 10 lines of the file on the screen (the "tail" of the file). If the file has fewer than 10 lines, the entire file should be displayed, with a message indicating the entire file has been displayed.

NOTE: Using an editor, you should create a simple text file that can be used to test this program.

5. Line Numbers

(This assignment could be done as a modification of the program in Programming Challenge 2.) Write a program that asks the user for the name of a file. The program should display the contents of the file on the screen. Each line of screen output should be preceded with a line number, followed by a colon. The line numbering should start at 1. Here is an example:

```
1:George Rolland
2:127 Academy Street
3:Brasstown, NC  28706
```

If the file's contents won't fit on a single screen, the program should display 24 lines of output at a time, and then pause. Each time the program pauses, it should wait for the user to strike a key before the next 24 lines are displayed.

NOTE: Using an editor, you should create a simple text file that can be used to test this program.

6. **String Search**

   Write a program that asks the user for a file name and a string to search for. The program should search the file for every occurrence of a specified string. When the string is found, the line that contains it should be displayed. After all the occurrences have been located, the program should report the number of times the string appeared in the file.

NOTE: Using an editor, you should create a simple text file that can be used to test this program.

7. **Sentence Filter**

   Write a program that asks the user for two file names. The first file will be opened for input and the second file will be opened for output. (It will be assumed that the first file contains sentences that end with a period.) The program will read the contents of the first file and change all the letters to lowercase except the first letter of each sentence, which should be made uppercase. The revised contents should be stored in the second file.

NOTE: Using an editor, you should create a simple text file that can be used to test this program.

8. **Array/File Functions**

   Write a function named `arrayToFile`. The function should accept three arguments: the name of a file, a pointer to an `int` array, and the size of the array. The function should open the specified file in binary mode, write the contents of the array to the file, and then close the file.

   Write another function named `fileToArray`. This function should accept three arguments: the name of a file, a pointer to an `int` array, and the size of the array. The function should open the specified file in binary mode, read its contents into the array, and then close the file.

   Write a complete program that demonstrates these functions by using the `arrayToFile` function to write an array to a file, and then using the `fileToArray` function to read the data from the same file. After the data are read from the file into the array, display the array's contents on the screen.

9. File Encryption Filter


VideoNote
Solving the File Encryption Filter Problem

File encryption is the science of writing the contents of a file in a secret code. Your encryption program should work like a filter, reading the contents of one file, modifying the data into a code, and then writing the coded contents out to a second file. The second file will be a version of the first file, but written in a secret code.

Although there are complex encryption techniques, you should come up with a simple one of your own. For example, you could read the first file one character at a time, and add 10 to the ASCII code of each character before it is written to the second file.

10. File Decryption Filter

Write a program that decrypts the file produced by the program in Programming Challenge 9. The decryption program should read the contents of the coded file, restore the data to its original state, and write it to another file.

11. Corporate Sales Data Output

Write a program that uses a structure to store the following data on a company division:

Division Name (such as East, West, North, or South)
Quarter (1, 2, 3, or 4)
Quarterly Sales

The user should be asked for the four quarters' sales figures for the East, West, North, and South divisions. The data for each quarter for each division should be written to a file.

Input Validation: Do not accept negative numbers for any sales figures.

12. Corporate Sales Data Input

Write a program that reads the data in the file created by the program in Programming Challenge 11. The program should calculate and display the following figures:

- Total corporate sales for each quarter
- Total yearly sales for each division
- Total yearly corporate sales
- Average quarterly sales for the divisions
- The highest and lowest quarters for the corporation

13. Inventory Program

Write a program that uses a structure to store the following inventory data in a file:

Item Description
Quantity on Hand
Wholesale Cost
Retail Cost
Date Added to Inventory

The program should have a menu that allows the user to perform the following tasks:

- Add new records to the file.
- Display any record in the file.
- Change any record in the file.

Input Validation: The program should not accept quantities, or wholesale or retail costs, less than 0. The program should not accept dates that the programmer determines are unreasonable.

14. Inventory Screen Report

Write a program that reads the data in the file created by the program in Programming Challenge 13. The program should calculate and display the following data:

- The total wholesale value of the inventory
- The total retail value of the inventory
- The total quantity of all items in the inventory

15. Average Number of Words

If you have downloaded this book's source code from the companion Web site, you will find a file named `text.txt` in the Chapter 12 folder. (The companion Web site is at www.pearsonhighered.com/gaddis.) The text that is in the file is stored as one sentence per line. Write a program that reads the file's contents and calculates the average number of words per sentence.

Group Project

16. Customer Accounts

This program should be designed and written by a team of students. Here are some suggestions:

- One student should design function `main`, which will call other program functions. The remainder of the functions will be designed by other members of the team.
- The requirements of the program should be analyzed so each student is given about the same workload.

Write a program that uses a structure to store the following data about a customer account:

Name
Address
City, State, and ZIP
Telephone Number
Account Balance
Date of Last Payment

The structure should be used to store customer account records in a file. The program should have a menu that lets the user perform the following operations:

- Enter new records into the file.
- Search for a particular customer's record and display it.
- Search for a particular customer's record and delete it.
- Search for a particular customer's record and change it.
- Display the contents of the entire file.

Input Validation: When the data for a new account is entered, be sure the user enters data for all the fields. No negative account balances should be entered.

This page intentionally left blank