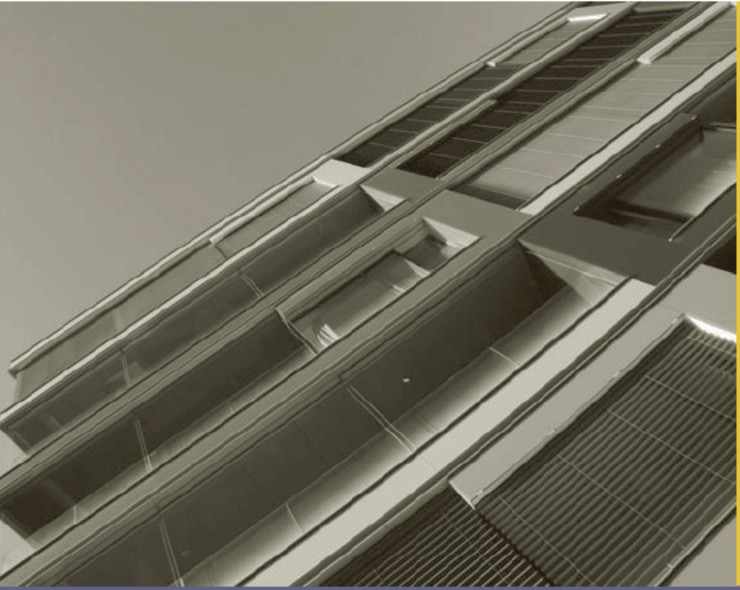
© HunThomas/Shutterstock.com

CHAPTER 9

Records (`structs`)

IN THIS CHAPTER, YOU WILL:

1. Learn about records (`structs`)
2. Examine various operations on a `struct`
3. Explore ways to manipulate data using a `struct`
4. Learn about the relationship between a `struct` and functions
5. Examine the difference between arrays and `structs`
6. Discover how arrays are used in a `struct`
7. Learn how to create an array of `struct` items
8. Learn how to create `structs` within a `struct`

In Chapter 8, you learned how to group values of the same type by using arrays. You also learned how to process data stored in an array and how to perform list operations, such as searching and sorting.

NOTE This chapter may be skipped without experiencing any discontinuation.

In this chapter, you will learn how to group related values that are of different types. C++ provides another structured data type, called a `struct` (some languages use the term "record"), to group related items of different types. An array is a homogeneous data structure; a `struct` is typically a heterogeneous data structure. The treatment of a `struct` in this chapter is similar to the treatment of a `struct` in C. A `struct` in this chapter, therefore, is a C-like `struct`. Chapter 10 introduces and discusses another structured data type, called a `class`.

Records (`structs`)

Suppose that you want to write a program to process student data. A student record consists of, among other things, the student's name, student ID, GPA, courses taken, and course grades. Thus, various components are associated with a student. However, these components are all of different types. For example, the student's name is a string, and the GPA is a floating-point number. Because these components are of different types, you cannot use an array to group all of the items associated with a student. C++ provides a structured data type called `struct` to group items of different types. Grouping components that are related but of different types offers several advantages. For example, a single variable can pass all the components as parameters to a function.

`struct`: A collection of a fixed number of components in which the components are accessed by name. The components may be of different types.

The components of a `struct` are called the members of the `struct`. The general syntax of a `struct` in C++ is:

```
struct structName
{
    dataType1 identifier1;
    dataType2 identifier2;
      .
      .
      .
    dataTypen identifiern;
};
```

In C++, struct is a reserved word. The members of a struct, even though enclosed in braces (that is, they form a block), are not considered to form a compound statement. Thus, a semicolon (after the right brace) is essential to end the struct statement. A semicolon at the end of the struct is, therefore, a part of the syntax.

The statement:

```
struct houseType
{
    string style;
    int numOfBedrooms;
    int numOfBathrooms;
    int numOfCarsGarage;
    int yearBuilt;
    int finishedSquareFootage;
    double price;
    double tax;
};
```

defines a struct **houseType** with eight members. The member **style** is of type **string**, the members **numOfBedrooms**, **numOfBathrooms**, **numOfCarsGarage**, **yearBuilt**, and **finishedSquareFootage** are of type int, and the members **price** and **tax** are of type double.

Like any type definition, a struct is a definition, not a declaration. That is, it defines only a data type; no memory is allocated.

Once a data type is defined, you can declare variables of that type.

For example, the following statement defines **newHouse** to be a struct variable of type **houseType**:

```
  //variable declaration
houseType newHouse;
```

The memory allocated is large enough to store **style**, **numOfBedrooms**, **numOfBathrooms**, **numOfCarsGarage**, **yearBuilt**, **finishedSquareFootage**, **price**, and **tax** (see Figure 9-1).

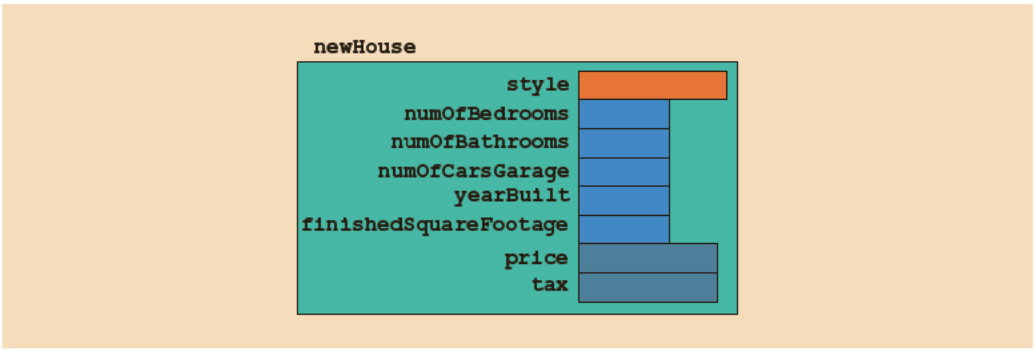

FIGURE 9-1 struct newHouse

NOTE

You can also declare struct variables when you define the struct. For example, consider the following statements:

```
struct houseType
{
    string style;
    int numOfBedrooms;
    int numOfBathrooms;
    int numOfCarsGarage;
    int yearBuilt;
    int finishedSquareFootage;
    double price;
    double tax;
} tempHouse;
```

These statements define the struct **houseType** and also declare **tempHouse** to be a variable of type **houseType**.

Typically, in a program, a struct is defined before the definitions of all the functions in the program, so that the struct can be used throughout the program. Therefore, if you define a struct and also simultaneously declare a struct variable (as in the preceding statements), then that struct variable becomes a global variable and thus can be accessed anywhere in the program. Keeping in mind the side effects of global variables, you should first only define a struct and then declare the struct variables.

Accessing struct Members

In arrays, you access a component by using the array name together with the relative position (index) of the component. The array name and index are separated using square brackets. To access a structure member (component), you use the struct variable name together with the member name; these names are separated by a dot (period). The syntax for accessing a struct member is:

```
structVariableName.memberName
```

The **structVariableName.memberName** is just like any other variable. For example, **newStudent.courseGrade** is a variable of type char, **newStudent.firstName** is a string variable, and so on. As a result, you can do just about anything with struct members that you normally do with variables. You can, for example, use them in assignment statements or input/output (where permitted) statements.

In C++, the dot (.) is an operator called the **member access operator**.

Consider the following statements:

```
struct studentType
{
    string firstName;
    string lastName;
    char courseGrade;
    int testScore;
    int programmingScore;
    double GPA;
};

    //variables
studentType newStudent;
studentType student;
```

Suppose you want to initialize the member **GPA** of **newStudent** to **0.0**. The following statement accomplishes this task:

```
newStudent.GPA = 0.0;
```

Similarly, the statements:

```
newStudent.firstName = "John";
newStudent.lastName = "Brown";
```

store **"John"** in the member **firstName** and **"Brown"** in the member **lastName** of **newStudent**.

After the preceding three assignment statements execute, **newStudent** is as shown in Figure 9-2.

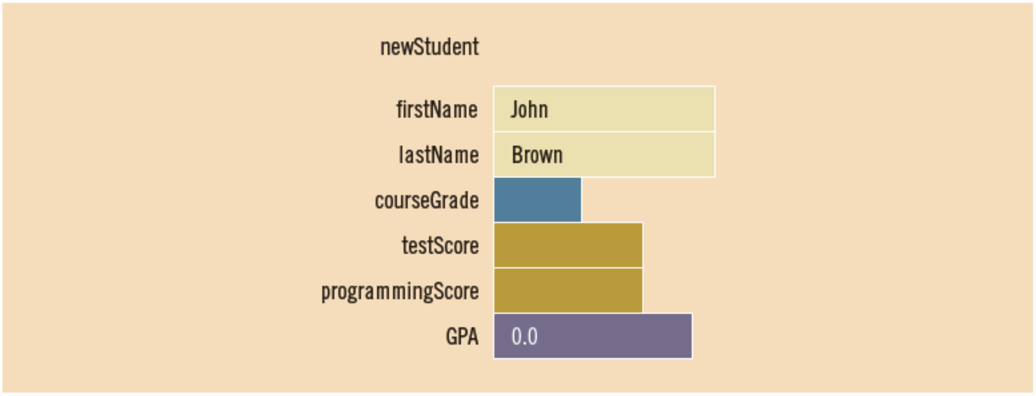

FIGURE 9-2 struct newStudent

The statement:

```
cin >> newStudent.firstName;
```

reads the next string from the standard input device and stores it in:

```
newStudent.firstName
```

The statement:

```
cin >> newStudent.testScore >> newStudent.programmingScore;
```

reads two integer values from the keyboard and stores them in `newStudent.testScore` and `newStudent.programmingScore`, respectively.

Suppose that `score` is a variable of type `int`. The statement:

```
score = (newStudent.testScore + newStudent.programmingScore) / 2;
```

assigns the average of `newStudent.testScore` and `newStudent.programmingScore` to `score`.

The following statement determines the course grade and stores it in `newStudent.courseGrade`:

```
if (score >= 90)
    newStudent.courseGrade = 'A';
else if (score >= 80)
    newStudent.courseGrade = 'B';
else if (score >= 70)
    newStudent.courseGrade = 'C';
else if (score >= 60)
    newStudent.courseGrade = 'D';
else
    newStudent.courseGrade = 'F';
```

EXAMPLE 9-1

Consider the definition of the `struct` `houseType` given in the previous section and the following statements:

```
houseType ryanHouse;
houseType anitaHouse;

ryanHouse.style = "Colonial";
ryanHouse.numOfBedrooms = 3;
ryanHouse.numOfBathrooms = 2;
ryanHouse.numOfCarsGarage = 2;
ryanHouse.yearBuilt = 2005;
ryanHouse.finishedSquareFootage = 2250;
ryanHouse.price = 290000;
ryanHouse.tax = 5000.50;
```

The first two statements declare `ryanHouse` and `anitaHouse` to be variables of `houseType`. The next eight statements store the string `"Colonial"` into `ryanHouse.style`, `3` into `ryanHouse.numOfBedrooms`, `2` into `ryanHouse.numOfBathrooms`, and so on.

Next, consider the following statements:

```
cin >> anitaHouse.style >> anitaHouse.numOfBedrooms
    >> anitaHouse.price;
```

If the input is:

```
Ranch 4 350000
```

then the string **"Ranch"** is stored into **anitaHouse.style**, **4** is stored into **anitaHouse.numOfBedrooms**, and **350000** is stored into **anitaHouse.price**.

Assignment

We can assign the value of one `struct` variable to another `struct` variable of the same type by using an assignment statement. Suppose that **newStudent** is as shown in Figure 9-3.

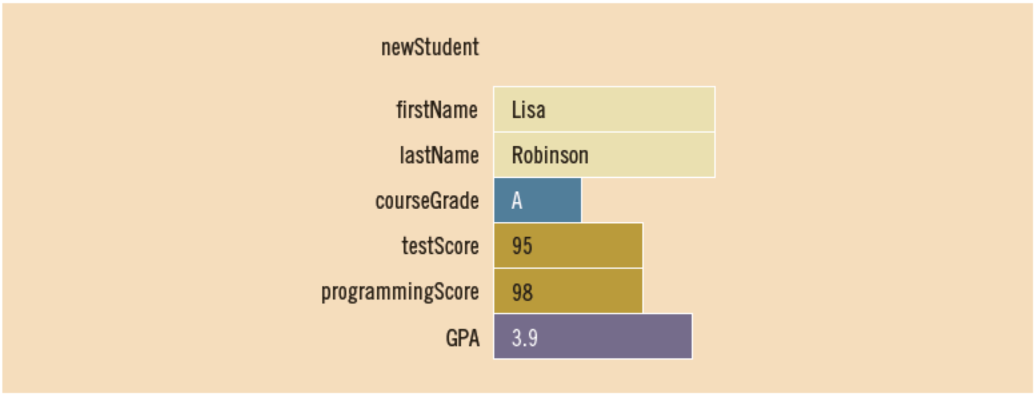

FIGURE 9-3 `struct newStudent`

The statement:

```
student = newStudent;
```

copies the contents of **newStudent** into **student**. After this assignment statement executes, the values of **student** are as shown in Figure 9-4.

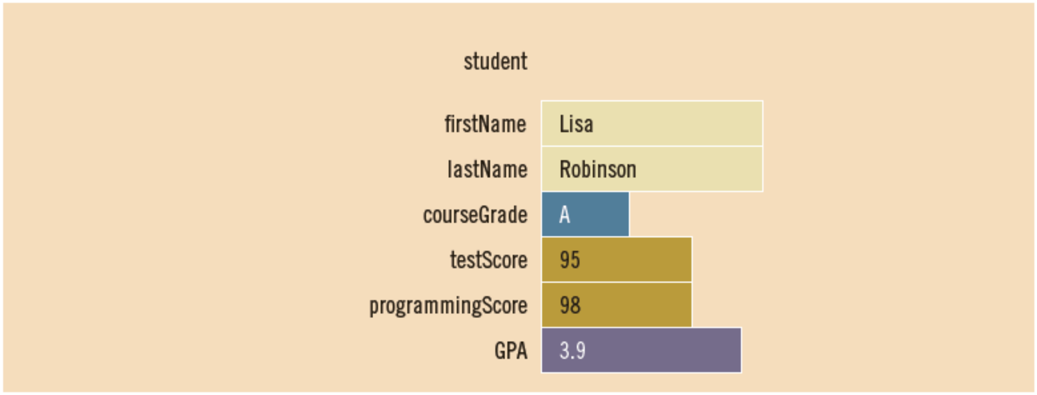

FIGURE 9-4 `student` after `student = newStudent`

In fact, the assignment statement:

```
student = newStudent;
```

is equivalent to the following statements:

```
student.firstName = newStudent.firstName;
student.lastName = newStudent.lastName;
student.courseGrade = newStudent.courseGrade;
student.testScore = newStudent.testScore;
student.programmingScore = newStudent.programmingScore;
student.GPA = newStudent.GPA;
```

Comparison (Relational Operators)

To compare struct variables, you compare them member-wise. As with an array, no aggregate relational operations are performed on a struct. For example, suppose that **newStudent** and **student** are declared as shown earlier. Furthermore, suppose that you want to see whether **student** and **newStudent** refer to the same student. Now **newStudent** and **student** refer to the same student if they have the same first name and the same last name. To compare the values of **student** and **newStudent**, you must compare them member-wise, as follows:

```
if (student.firstName == newStudent.firstName &&
    student.lastName == newStudent.lastName)
.
.
.
```

Although you can use an assignment statement to copy the contents of one struct into another struct of the same type, you cannot use relational operators on struct variables. Therefore, the following would be illegal:

```
if (student == newStudent)    //illegal
.
.
.
```

Input/Output

No aggregate input/output operations are allowed on a struct variable. Data in a struct variable must be read one member at a time. Similarly, the contents of a struct variable must be written one member at a time.

We have seen how to read data into a struct variable. Let us now see how to output a struct variable. The statement:

```
cout << newStudent.firstName << " " << newStudent.lastName
     << " " << newStudent.courseGrade
     << " " << newStudent.testScore
     << " " << newStudent.programmingScore
     << " " << newStudent.GPA << endl;
```

outputs the contents of the struct variable **newStudent**.

struct Variables and Functions

Recall that arrays are passed by reference only, and a function cannot return a value of type **array**. However:

- A struct variable can be passed as a parameter either by value or by reference, and
- A function can return a value of type struct.

The following function reads and stores a student's first name, last name, test score, programming score, and GPA. It also determines the student's course grade and stores it in the member **courseGrade**.

```
void readIn(studentType& student)
{
    int score;

    cin >> student.firstName >> student.lastName;
    cin >> student.testScore >> student.programmingScore;
    cin >> student.GPA;

    score = (student.testScore + student.programmingScore) / 2;

    if (score >= 90)
        student.courseGrade = 'A';
    else if (score >= 80)
        student.courseGrade = 'B';
    else if (score >= 70)
        student.courseGrade = 'C';
    else if (score >= 60)
        student.courseGrade = 'D';
    else
        student.courseGrade = 'F';
}
```

The statement:

```
readIn(newStudent);
```

calls the function **readIn**. The function **readIn** stores the appropriate information in the variable **newStudent**.

Similarly, we can write a function that will print the contents of a struct variable. For example, the following function outputs the contents of a struct variable of type **studentType** on the screen:

```
void printStudent(studentType student)
{
    cout << student.firstName << " " << student.lastName
         << " " << student.courseGrade
         << " " << student.testScore
         << " " << student.programmingScore
         << " " << student.GPA << endl;
}
```

Arrays versus structs

The previous discussion showed us that a `struct` and an array have similarities as well as differences. Table 9-1 summarizes this discussion.

TABLE 9-1 Arrays vs. `structs`

Aggregate Operation	Array	`struct`
Arithmetic	No	No
Assignment	No	Yes
Input/output	No (except strings)	No
Comparison	No	No
Parameter passing	By reference only	By value or by reference
Function returning a value	No	Yes

Arrays in structs

A list is a set of elements of the same type. Thus, a list has two things associated with it: the values (that is, elements) and the length. Because the values and the length are both related to a list, we can define a `struct` containing both items.

```
const int ARRAY_SIZE = 1000;

struct listType
{
    int listElem[ARRAY_SIZE];   //array containing the list
    int listLength;             //length of the list
};
```

The following statement declares **`intList`** to be a `struct` variable of type **`listType`** (see Figure 9-5):

```
listType intList;
```

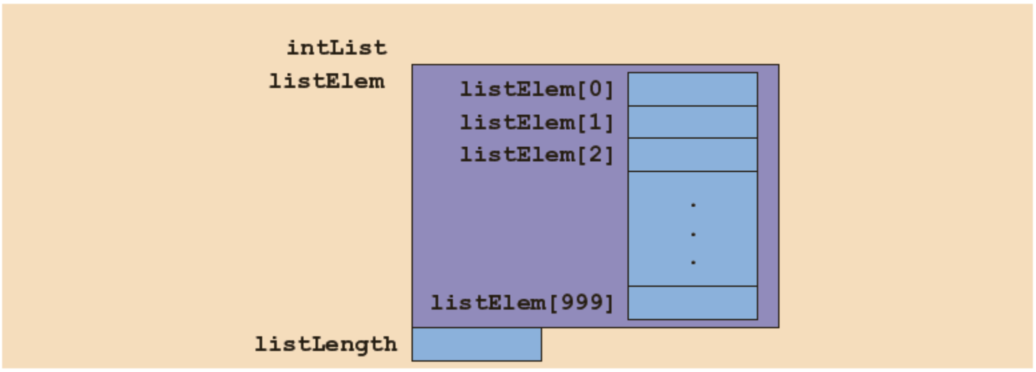

FIGURE 9-5 `struct` variable `intList`

The variable **`intList`** has two members: **`listElem`**, an array of 1,000 components of type `int`, and **`listLength`** of type `int`. Moreover, **`intList.listElem`** accesses the member **`listElem`**, and **`intList.listLength`** accesses the member **`listLength`**.

Consider the following statements:

```
intList.listLength = 0;             //Line 1
intList.listElem[0] = 12;           //Line 2
intList.listLength++;               //Line 3
intList.listElem[1] = 37;           //Line 4
intList.listLength++;               //Line 5
```

The statement in Line 1 sets the value of the member **`listLength`** to **`0`**. The statement in Line 2 stores **`12`** in the first component of the array **`listElem`**. The statement in Line 3 increments the value of **`listLength`** by **`1`**. The meaning of the other statements is similar. After these statements execute, **`intList`** is as shown in Figure 9-6.

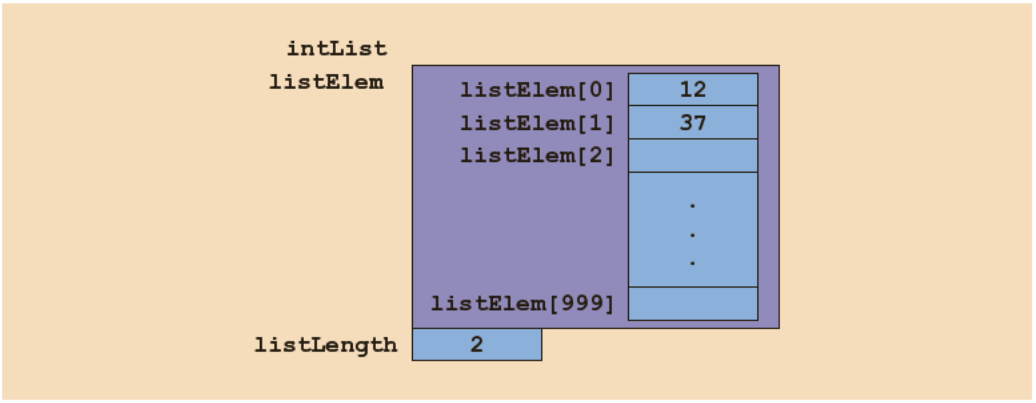

FIGURE 9-6 `intList` after the statements in Lines 1 through 5 execute

Next, we write the sequential search algorithm to determine whether a given item is in the list. If **searchItem** is found in the list, then the function returns its location in the list; otherwise, the function returns **-1**.

```
int seqSearch(const listType& list, int searchItem)
{
    int loc;

    bool found = false;

    for (loc = 0; loc < list.listLength; loc++)
        if (list.listElem[loc] == searchItem)
        {
            found = true;
            break;
        }

    if (found)
        return loc;
    else
        return -1;
}
```

In this function, because **listLength** is a member of **list**, we access this by **list.listLength**. Similarly, we can access an element of **list** via **list.listElem[loc]**.

Notice that the formal parameter **list** of the function **seqSearch** is declared as a constant reference parameter. This means that **list** receives the address of the corresponding actual parameter, but **list** cannot modify the actual parameter.

Recall that when a variable is passed by value, the formal parameter copies the value of the actual parameter. Therefore, if the formal parameter modifies the data, the modification has no effect on the data of the actual parameter.

Suppose that a struct has several data members requiring a large amount of memory to store the data, and you need to pass a variable of that struct type by value. The corresponding formal parameter then receives a copy of the data of the variable. The compiler must then allocate memory for the formal parameter in order to copy the value of the actual parameter. This operation might require, in addition to a large amount of storage space, a considerable amount of computer time to copy the value of the actual parameter into the formal parameter.

On the other hand, if a variable is passed by reference, the formal parameter receives only the address of the actual parameter. Therefore, an efficient way to pass a variable as a parameter is by reference. If a variable is passed by reference, then when the formal parameter changes, the actual parameter also changes. Sometimes, however, you do not want the function to be able to change the values of the actual parameter. In C++, you can pass a variable by reference and still prevent the function from changing its value. This is done by using the keyword const in the formal parameter declaration, as shown in the definition of the function **seqSearch**.

Likewise, we can also rewrite the sorting and other list-processing functions.

structs in Arrays

Suppose a company has 50 full-time employees. We need to print their monthly paychecks and keep track of how much money has been paid to each employee in the year-to-date. First, let's define an employee's record:

```
struct employeeType
{
    string firstName;
    string lastName;
    int    personID;
    string deptID;
    double yearlySalary;
    double monthlySalary
    double yearToDatePaid;
    double monthlyBonus;
};
```

Each employee has the following members (components): first name, last name, personal ID, department ID, yearly salary, monthly salary, year-to-date paid, and monthly bonus.

Because we have 50 employees and the data type of each employee is the same, we can use an array of 50 components to process the employees' data.

```
employeeType employees[50];
```

This statement declares the array **employees** of **50** components of type **employeeType** (see Figure 9-7). Every element of **employees** is a struct. For example, Figure 9-7 also shows **employees[2]**.

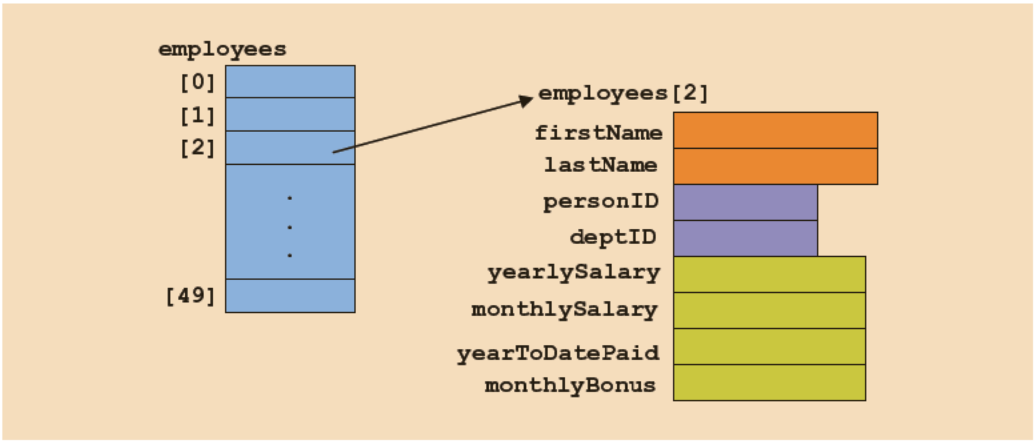

FIGURE 9-7 Array of employees

Suppose that every employee's initial data—first name, last name, personal ID, department ID, and yearly salary—are provided in a file. For our discussion, we assume that each employee's data is stored in a file, say, **employee.dat**. The following C++ code loads the data into the employees' array. We assume that, initially, **yearToDatePaid** is **0** and that the monthly bonus is determined each month based on performance.

```
ifstream infile; //input stream variable
                 //assume that the file employee.dat
                 //has been opened

for (int counter = 0; counter < 50; counter++)
{
    infile >> employees[counter].firstName
           >> employees[counter].lastName
           >> employees[counter].personID
           >> employees[counter].deptID
           >> employees[counter].yearlySalary;
    employees[counter].monthlySalary =
                 employees[counter].yearlySalary / 12;
    employees[counter].yearToDatePaid = 0.0;
    employees[counter].monthlyBonus = 0.0;
}
```

Suppose that for a given month, the monthly bonus is already stored in each employee's record, and we need to calculate the monthly paycheck and update the **yearToDatePaid** amount. The following loop computes and prints the employee's paycheck for the month:

```
double payCheck; //variable to calculate the paycheck

for (int counter = 0; counter < 50; counter++)
{
    cout << employees[counter].firstName << " "
         << employees[counter].lastName << " ";

    payCheck = employees[counter].monthlySalary +
               employees[counter].monthlyBonus;

    employees[counter].yearToDatePaid =
                         employees[counter].yearToDatePaid +
                         payCheck;

    cout << setprecision(2) << payCheck << endl;
}
```

structs within a struct

You have seen how the struct and array data structures can be combined to organize information. You also saw examples wherein a member of a struct is an array, and the array type is a struct. In this section, you will learn about situations for which it is beneficial to organize data in a struct by using another struct.

Let us consider the following employee record:

```
struct employeeType
{
    string firstname;
    string middlename;
    string lastname;
    string empID;
    string address1;
    string address2;
    string city;
    string state;
    string zip;
    int hiremonth;
    int hireday;
    int hireyear;
    int quitmonth;
    int quitday;
    int quityear;
    string phone;
    string cellphone;
    string fax;
    string pager;
    string email;
    string deptID;
    double salary;
};
```

As you can see, a lot of information is packed into one struct. This struct has 22 members. Some members of this struct will be accessed more frequently than others, and some members are more closely related than others. Moreover, some members will have the same underlying structure. For example, the hire date and the quit date are of type int. Let us reorganize this struct as follows:

```
struct nameType
{
    string first;
    string middle;
    string last;
};

struct addressType
{
    string address1;
    string address2;
    string city;
    string state;
    string zip;
};
```

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

struct contactType
{
    string phone;
    string cellphone;
    string fax;
    string pager;
    string email;
};
```

We have separated the employee's name, address, and contact type into subcategories. Furthermore, we have defined a `struct` **`dateType`**. Let us rebuild the employee's record as follows:

```
struct employeeType
{
    nameType name;
    string empID;
    addressType address;
    dateType hireDate;
    dateType quitDate;
    contactType contact;
    string deptID;
    double salary;
};
```

The information in this employee's `struct` is easier to manage than the previous one. Some of this `struct` can be reused to build another `struct`. For example, suppose that you want to define a customer's record. Every customer has a first name, last name, and middle name, as well as an address and a way to be contacted. You can, therefore, quickly put together a customer's record by using the `struct`s **`nameType, addressType, contactType`**, and the members specific to the customer.

Next, let us declare a variable of type **`employeeType`** and discuss how to access its members.

Consider the following statement:

```
employeeType newEmployee;
```

This statement declares **`newEmployee`** to be a `struct` variable of type **`employeeType`** (see Figure 9-8).

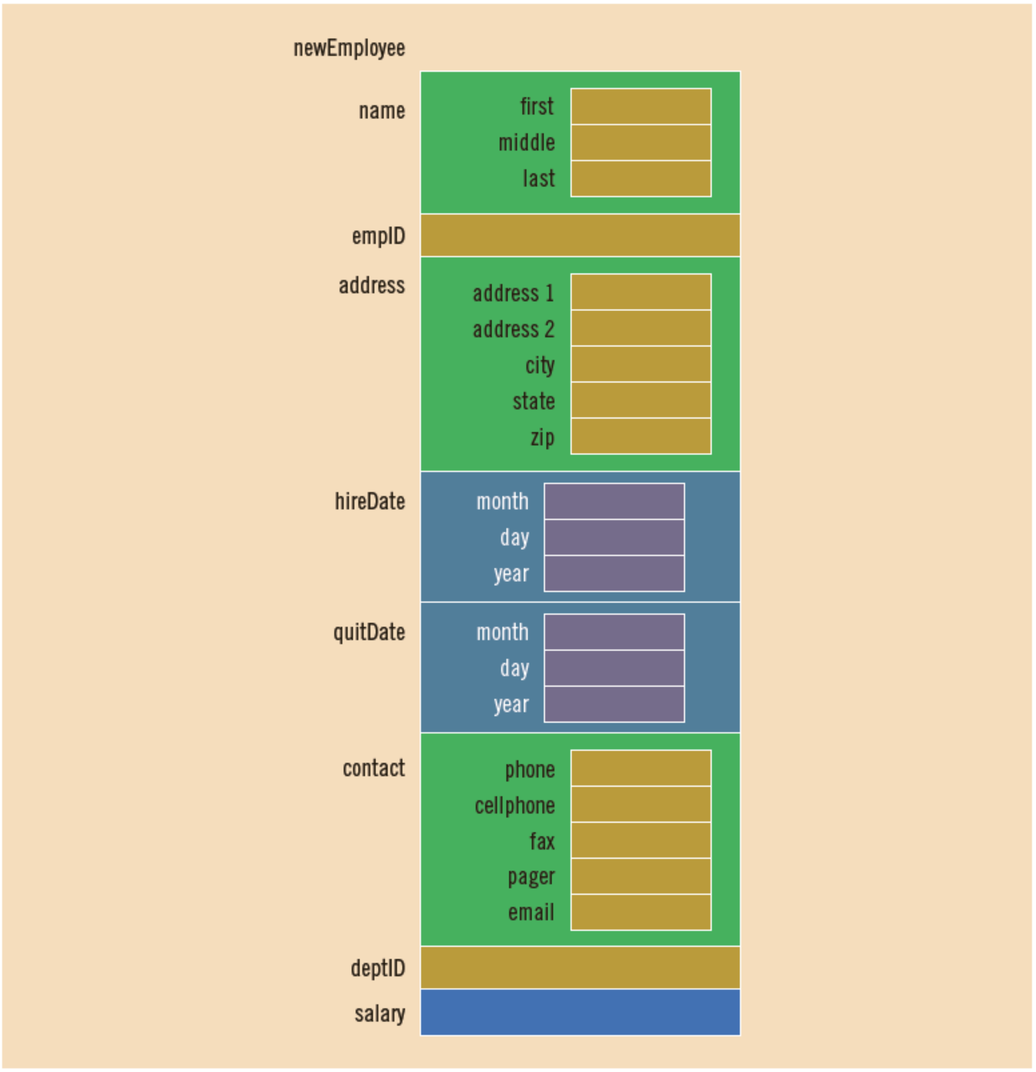

FIGURE 9-8 `struct` variable `newEmployee`

The statement:

```
newEmployee.salary = 45678.00;
```

sets the salary of **newEmployee** to **45678.00**. The statements:

```
newEmployee.name.first = "Mary";
newEmployee.name.middle = "Beth";
newEmployee.name.last = "Simmons";
```

set the **first**, **middle**, and **last** name of **newEmployee** to **"Mary"**, **"Beth"**, and **"Simmons"**, respectively. Note that **newEmployee** has a member called **name**. We access this member via **newEmployee.name**. Note also that **newEmployee.name** is a **struct** and has three members. We apply the member access criteria to access the member `first` of the `struct` `newEmployee.name`. So, `newEmployee.name.first` is the member where we store the first name.

The statement:

```
cin >> newEmployee.name.first;
```

reads and stores a string into `newEmployee.name.first`. The statement:

```
newEmployee.salary = newEmployee.salary * 1.05;
```

updates the salary of `newEmployee`.

The following statement declares `employees` to be an array of `100` components, wherein each component is of type `employeeType`:

```
employeeType employees[100];
```

The `for` loop:

```
for (int j = 0; j < 100; j++)
    cin >> employees[j].name.first >> employees[j].name.middle
        >> employees[j].name.last;
```

reads and stores the names of 100 employees in the array `employees`. Because `employees` is an array, to access a component, we use the index. For example, `employees[50]` is the 51st component of the array `employees` (recall that an array index starts with `0`). Because `employees[50]` is a `struct`, we apply the member access criteria to select a particular member.

PROGRAMMING EXAMPLE: Sales Data Analysis

A company has six salespeople. Every month, they go on road trips to sell the company's product. At the end of each month, the total sales for each salesperson, together with that salesperson's ID and the month, is recorded in a file. At the end of each year, the manager of the company wants to see this report in this following tabular format:

```
-----------Annual Sales Report ----------

   ID        QT1         QT2         QT3         QT4         Total
____________________________________________________________________
12345     1892.00        0.00      494.00      322.00      2708.00
32214      343.00      892.00     9023.00        0.00     10258.00
23422     1395.00     1901.00        0.00        0.00      3296.00
57373      893.00      892.00     8834.00        0.00     10619.00
35864     2882.00     1221.00        0.00     1223.00      5326.00
54654      893.00        0.00      392.00     3420.00      4705.00
Total     8298.00     4906.00    18743.00     4965.00

Max Sale by SalesPerson: ID = 57373, Amount = $10619.00
Max Sale by Quarter: Quarter = 3, Amount = $18743.00
```

In this report, **QT1** stands for quarter 1 (months 1 to 3), **QT2** for quarter 2 (months 4 to 6), **QT3** for quarter 3 (months 7 to 9), and **QT4** for quarter 4 (months 10 to 12).

The salespeople's IDs are stored in one file; the sales data is stored in another file. The sales data is in the following form:

```
salesPersonID month saleAmount
.
.
.
```

Furthermore, the sales data is in no particular order; it is not ordered by ID.

A sample sales data is:

```
12345 1 893
32214 1 343
23422 3 903
57373 2 893
.
.
.
```

Let us write a program that produces the output in the specified format.

Input One file containing each salesperson's ID and a second file containing the sales data.

Output A file containing the annual sales report in the above format.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

Based on the problem's requirements, it is clear that the main components for each salesperson are the salesperson's ID, quarterly sales amount, and total annual sales amount. Because the components are of different types, we can group them with the help of a `struct`, defined as follows:

```
struct salesPersonRec
{
    string ID;       //salesperson's ID
    double saleByQuarter[4];  //array to store the total
                              //sales for each quarter
    double totalSale;  //salesperson's yearly sales amount
};
```

Because there are six salespeople, we use an array of six components, wherein each component is of type `salesPersonRec`, defined as follows:

```
salesPersonRec salesPersonList[NO_OF_SALES_PERSON];
```

wherein the value of `NO_OF_SALES_PERSON` is `6`.

Because the program requires us to find the company's total sales for each quarter, we need an array of four components to store the data. Note that this data will be used to determine the quarter in which the maximum sales were made. Therefore, the program also needs the following array:

```
double totalSaleByQuarter[4];
```

Recall that in C++, the array index starts with `0`. Therefore, `totalSaleByQuarter[0]` stores data for quarter 1, `totalSaleByQuarter[1]` stores data for quarter 2, and so on.

We will refer to these variables throughout the discussion.

The array `salesPersonList` is as shown in Figure 9-9.

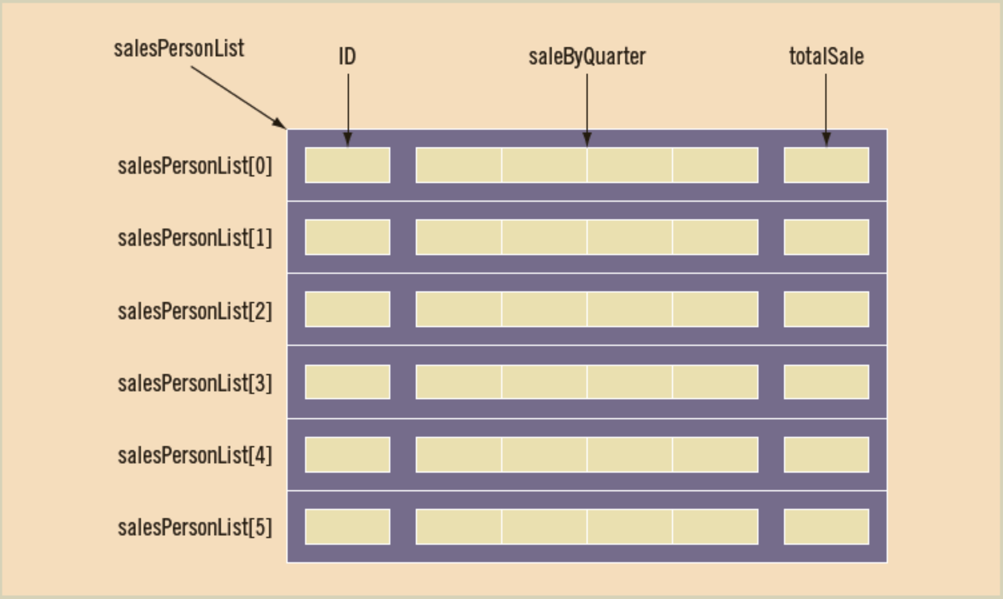

FIGURE 9-9 Array `salesPersonList`

The first step of the program is to read the salespeople's IDs into the array `salesPersonList` and initialize the quarterly sales and total sales for each salesperson to `0`. After this step, the array `salesPersonList` is as shown in Figure 9-10.

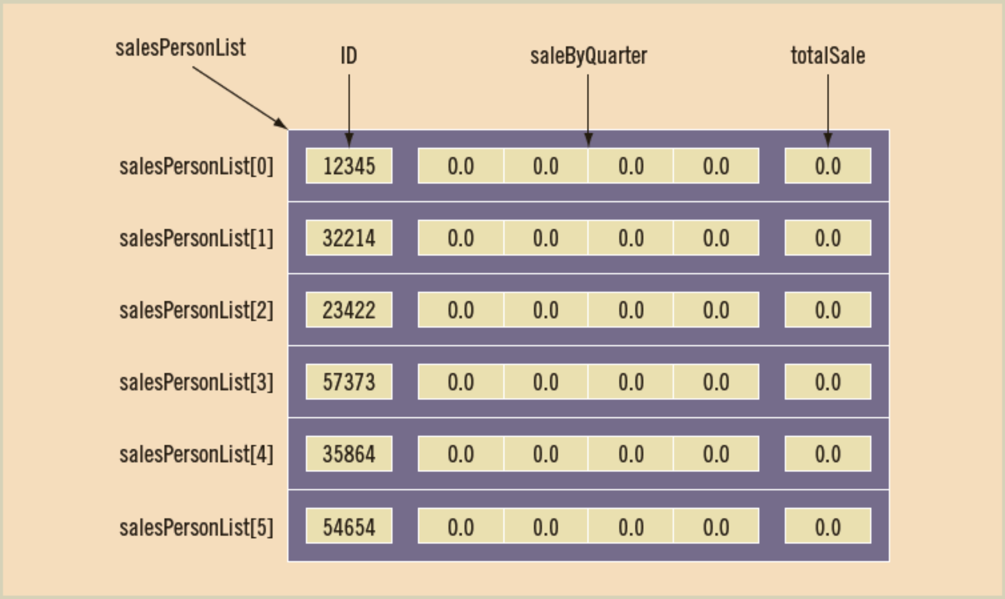

FIGURE 9-10 Array `salesPersonList` after initialization

The next step is to process the sales data. Processing the sales data is quite straightforward. For each entry in the file containing the sales data:

1. Read the salesperson's ID, month, and sale amount for the month.
2. Search the array `salesPersonList` to locate the component corresponding to this salesperson.
3. Determine the quarter corresponding to the month.
4. Update the sales for the quarter by adding the sale amount for the month.

Once the sales data file is processed:

1. Calculate the total sales by salesperson.
2. Calculate the total sales by quarter.
3. Print the report.

This discussion translates into the following algorithm:

1. Initialize the array `salesPersonList`.
2. Process the sales data.
3. Calculate the total sales by quarter.
4. Calculate the total sales by salesperson.

5. Print the report.
6. Calculate and print the maximum sales by salesperson.
7. Calculate and print the maximum sales by quarter.

To reduce the complexity of the main program, let us write a separate function for each of these seven steps.

Function initialize

This function reads the salesperson's ID from the input file and stores the salesperson's ID in the array `salesPersonList`. It also initializes the quarterly sales amount and the total sales amount for each salesperson to `0`. The definition of this function is:

```
void initialize(ifstream& indata, salesPersonRec list[],
                int listSize)
{
    for (int index = 0; index < listSize; index++)
    {
        indata >> list[index].ID; //get salesperson's ID

        for (int quarter = 0; quarter < 4; quarter++)
            list[index].saleByQuarter[quarter] = 0.0;

        list[index].totalSale = 0.0;
    }
} //end initialize
```

Function getData

This function reads the sales data from the input file and stores the appropriate information in the array `salesPersonList`. The algorithm for this function is as follows:

1. Read the salesperson's ID, month, and sales amount for the month.
2. Search the array `salesPersonList` to locate the component corresponding to the salesperson. (Because the salespeople's IDs are not sorted, we will use a sequential search to search the array.)
3. Determine the quarter corresponding to the month.
4. Update the sales for the quarter by adding the sales amount for the month.

Suppose that the entry read is:

```
57373 2 350
```

Here, the salesperson's ID is `57373`, the month is `2`, and the sales amount is `350`. Suppose that the array `salesPersonList` is as shown in Figure 9-11.

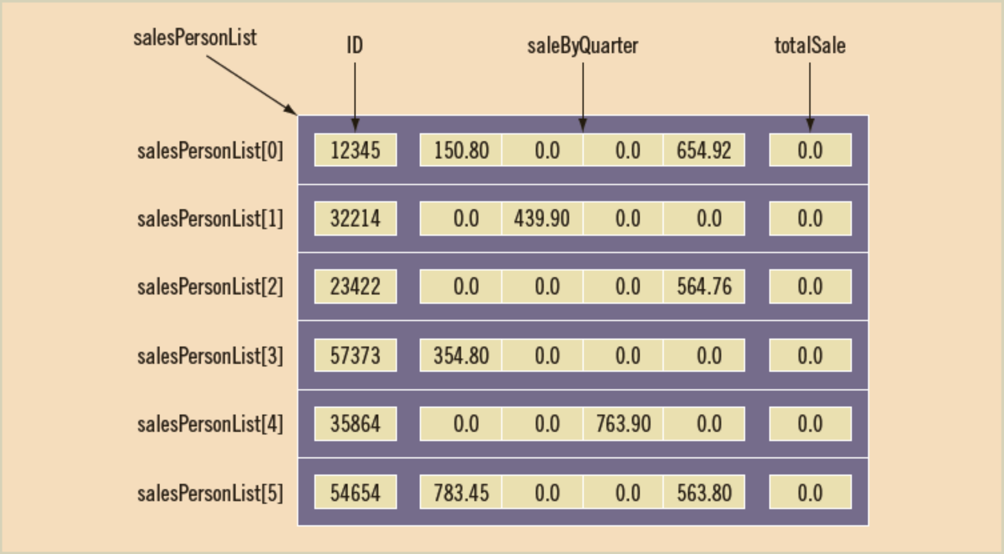

FIGURE 9-11 Array **`salesPersonList`**

Now, ID **`57373`** corresponds to the array component **`salesPersonList[3]`**, and month **`2`** corresponds to quarter **`1`**. Therefore, you add **`350`** to **`354.80`** to get the new amount, **`704.80`**. After processing this entry, the array **`salesPersonList`** is as shown in Figure 9-12.

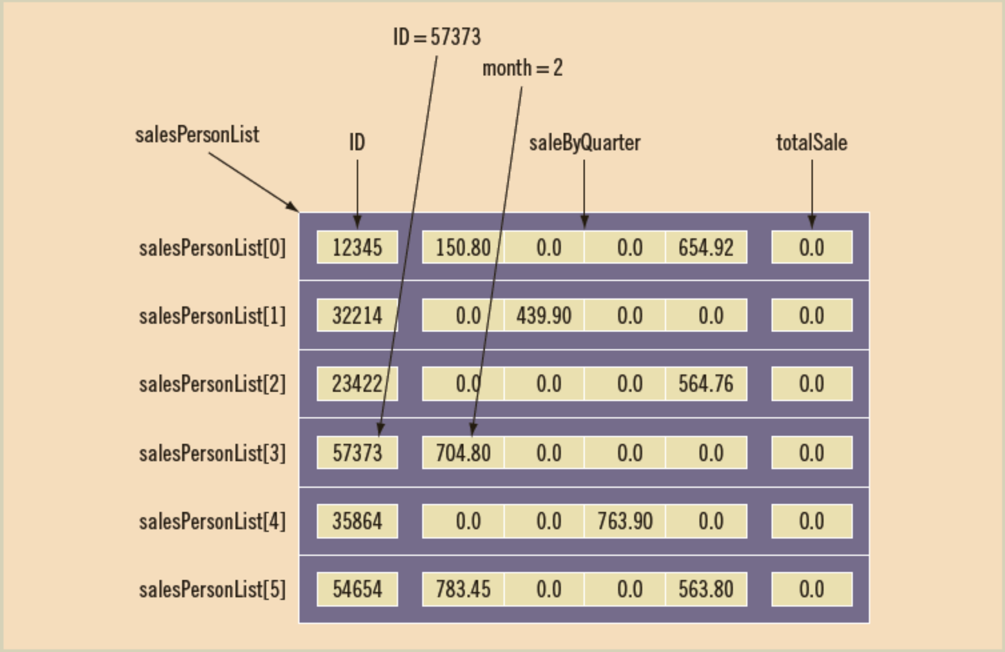

FIGURE 9-12 Array **`salesPersonList`** after processing entry **`57373 2 350`**

The definition of the function getData is:

```
void getData(ifstream& infile, salesPersonRec list[],
             int listSize)
{
    int index;
    int quarter;
    string sID;
    int month;
    double amount;

    infile >> sID;  //get salesperson's ID

    while (infile)
    {
        infile >> month >> amount; //get the sale month and
                                   //the sales amount

        for (index = 0; index < listSize; index++)
            if (sID == list[index].ID)
                break;

        if (1 <= month && month <= 3)
            quarter = 0;
        else if (4 <= month && month <= 6)
            quarter = 1;
        else if (7 <= month && month <= 9)
            quarter = 2;
        else
            quarter = 3;

        if (index < listSize)
            list[index].saleByQuarter[quarter] += amount;
        else
            cout << "Invalid salesperson's ID." << endl;

        infile >> sID;
    } //end while
} //end getData
```

Function saleByQuarter

This function finds the company's total sales for each quarter. To find the total sales for each quarter, we add the sales amount of each salesperson for that quarter. Clearly, this function must have access to the array salesPersonList and the array totalSaleByQuarter. This function also needs to know the number of rows in each array. Thus, this function has three parameters. The definition of this function is:

```
void saleByQuarter(salesPersonRec list[], int listSize,
                   double totalByQuarter[])
{
    for (int quarter = 0; quarter < 4; quarter++)
        totalByQuarter[quarter] = 0.0;

    for (int quarter = 0; quarter < 4; quarter++)
        for (int index = 0; index < listSize; index++)
            totalByQuarter[quarter] +=
                    list[index].saleByQuarter[quarter];
} //end saleByQuarter
```

Function `totalSaleByPerson`

This function finds each salesperson's yearly sales amount. To find an employee's yearly sales amount, we add that employee's sales amount for the four quarters. Clearly, this function must have access to the array **`salesPersonList`**. This function also needs to know the size of the array. Thus, this function has two parameters.

The definition of this function is:

```
void totalSaleByPerson(salesPersonRec list[], int listSize)
{
    for (int index = 0; index < listSize; index++)
        for (int quarter = 0; quarter < 4; quarter++)
            list[index].totalSale +=
                     list[index].saleByQuarter[quarter];
} //end totalSaleByPerson
```

Function `printReport`

This function prints the annual report in the specified format. The algorithm in pseudocode is as follows:

```
1. Print the heading—that is, the first three lines of output.
2. Print the data for each salesperson.
3. Print the last line of the table.
```

Note that the next two functions will produce the final two lines of output.

Clearly, the **`printReport`** function must have access to the array **`salesPersonList`** and the array **`totalSaleByQuarter`**. Also, because the output will be stored in a file, this function must have access to the **`ofstream`** variable associated with the output file. Thus, this function has four parameters: a parameter corresponding to the array **`salesPersonList`**, a parameter corresponding to the array **`totalSaleByQuarter`**, a parameter specifying the size of the array, and a parameter corresponding to the **`ofstream`** variable. The definition of this function is:

```
void printReport(ofstream& outfile, salesPersonRec list[],
                 int listSize, double saleByQuarter[])
{
    outfile << "-----------  Annual Sales Report ---------"
            << "----" << endl;
    outfile << endl;
    outfile << "  ID         QT1        QT2       QT3       "
            << "QT4       Total" << endl;
    outfile << "____________________________________________"
            << "_________________" << endl;

    for (int index = 0; index < listSize; index++)
    {
        outfile << list[index].ID << "   ";

        for (int quarter = 0; quarter < 4; quarter++)
            outfile << setw(10)
                    << list[index].saleByQuarter[quarter];

        outfile << setw(10) << list[index].totalSale << endl;
    }

    outfile << "Total   ";

    for (int quarter = 0; quarter < 4; quarter++)
        outfile << setw(10) << saleByQuarter[quarter];

    outfile << endl << endl;
} //end printReport
```

Function `maxSaleByPerson`

This function prints the name of the salesperson who produces the maximum sales amount. To identify this salesperson, we look at the sales total for each salesperson and find the largest sales amount. Because each employee's sales total is maintained in the array **salesPersonList**, this function must have access to the array **salesPersonList**. Also, because the output will be stored in a file, this function must have access to the **ofstream** variable associated with the output file. Therefore, this function has three parameters: a parameter corresponding to the array **salesPersonList**, a parameter specifying the size of this array, and a parameter corresponding to the output file.

The algorithm to find the largest sales amount is similar to the algorithm to find the largest element in an array (discussed in Chapter 8). The definition of this function is:

```
void maxSaleByPerson(ofstream& outData, salesPersonRec list[],
                     int listSize)
{
    int maxIndex = 0;

    for (int index = 1; index < listSize; index++)
        if (list[maxIndex].totalSale < list[index].totalSale)
            maxIndex = index;

    outData << "Max Sale by SalesPerson: ID = "
            << list[maxIndex].ID
            << ", Amount = $" << list[maxIndex].totalSale
            << endl;
} //end maxSaleByPerson
```

Function **maxSaleByQuarter**

This function prints the quarter in which the maximum sales were made. To identify this quarter, we look at the total sales for each quarter and find the largest sales amount. Because the sales total for each quarter is in the array `totalSaleByQuarter`, this function must have access to the array `totalSaleByQuarter`. Also, because the output will be stored in a file, this function must have access to the `ofstream` variable associated with the output file. Therefore, this function has two parameters: a parameter corresponding to the array `totalSaleByQuarter` and a parameter corresponding to the output file.

The algorithm to find the largest sales amount is the same as the algorithm to find the largest element in an array (discussed in Chapter 8). The definition of this function is:

```
void maxSaleByQuarter(ofstream& outData,
                      double saleByQuarter[])
{
    int maxIndex = 0;

    for (int quarter = 0; quarter < 4; quarter++)
        if (saleByQuarter[maxIndex] < saleByQuarter[quarter])
            maxIndex = quarter;

    outData << "Max Sale by Quarter: Quarter = "
            << maxIndex + 1
            << ", Amount = $" << saleByQuarter[maxIndex]
            << endl;
} //end maxSaleByQuarter
```

To make the program more flexible, we will prompt the user to specify the input and output files during its execution.

We are now ready to write the algorithm for the function `main`.

Main Algorithm

1. Declare the variables.
2. Prompt the user to enter the name of the file containing the salesperson's ID data.
3. Read the name of the input file.
4. Open the input file.
5. If the input file does not exist, exit the program.
6. Initialize the array `salesPersonList`. Call the function `initialize`.
7. Close the input file containing the salesperson's ID data and clear the input stream.
8. Prompt the user to enter the name of the file containing the sales data.
9. Read the name of the input file.
10. Open the input file.
11. If the input file does not exist, exit the program.
12. Prompt the user to enter the name of the output file.
13. Read the name of the output file.
14. Open the output file.
15. To output floating-point numbers in a fixed decimal format with the decimal point and trailing zeroes, set the manipulators `fixed` and `showpoint`. Also, to output floating-point numbers to two decimal places, set the precision to two decimal places.
16. Process the sales data. Call the function `getData`.
17. Calculate the total sales by quarter. Call the function `saleByQuarter`.
18. Calculate the total sales for each salesperson. Call the function `totalSaleByPerson`.
19. Print the report in a tabular format. Call the function `printReport`.
20. Find and print the salesperson who produces the maximum sales for the year. Call the function `maxSaleByPerson`.
21. Find and print the quarter that produces the maximum sales for the year. Call the function `maxSaleByQuarter`.
22. Close the files.

PROGRAM LISTING

```
//*************************************************************
// Author: D.S. Malik
//
// Program: Sales Data Analysis
// This program processes sales data for a company. For each
// salesperson, it outputs the ID, the total sales by each
// quarter, and the total sales for the year. It also outputs
// the salesperson's ID generating the maximum sale for the
// year and the sales amount. The quarter generating the
// maximum sale and the sales amount is also output.
//*************************************************************

#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>

using namespace std;

const int NO_OF_SALES_PERSON = 6;

struct salesPersonRec
{
    string ID;      //salesperson's ID
    double saleByQuarter[4]; //array to store the total
                             //sales for each quarter
    double totalSale;  //salesperson's yearly sales amount
};

void initialize(ifstream& indata, salesPersonRec list[],
                int listSize);
void getData(ifstream& infile, salesPersonRec list[],
             int listSize);
void saleByQuarter(salesPersonRec list[], int listSize,
                   double totalByQuarter[]);
void totalSaleByPerson(salesPersonRec list[], int listSize);
void printReport(ofstream& outfile, salesPersonRec list[],
                 int listSize, double saleByQuarter[]);
void maxSaleByPerson(ofstream& outData, salesPersonRec list[],
                     int listSize);
void maxSaleByQuarter(ofstream& outData, double saleByQuarter[]);

int main()
{
        //Step 1
    ifstream infile;    //input file stream variable
    ofstream outfile;   //output file stream variable
```

9

```
string inputFile;   //variable to hold the input file name
string outputFile;  //variable to hold the output file name

double totalSaleByQuarter[4];   //array to hold the
                                //sale by quarter

salesPersonRec salesPersonList[NO_OF_SALES_PERSON]; //array
                           //to hold the salesperson's data

cout << "Enter the salesPerson ID file name: "; //Step 2
cin >> inputFile;                               //Step 3
cout << endl;

infile.open(inputFile.c_str());                 //Step 4

if (!infile)                                    //Step 5
{
    cout << "Cannot open the input file."
         << endl;
    return 1;
}

initialize(infile, salesPersonList,
           NO_OF_SALES_PERSON);                 //Step 6

infile.close();                                 //Step 7
infile.clear();                                 //Step 7

cout << "Enter the sales data file name: ";     //Step 8
cin >> inputFile;                               //Step 9
cout << endl;

infile.open(inputFile.c_str());                 //Step 10

if (!infile)                                    //Step 11
{
    cout << "Cannot open the input file."
         << endl;
    return 1;
}

cout << "Enter the output file name: ";         //Step 12
cin >> outputFile;                              //Step 13
cout << endl;

outfile.open(outputFile.c_str());               //Step 14

outfile << fixed << showpoint
        << setprecision(2);                     //Step 15
    getData(infile, salesPersonList,
            NO_OF_SALES_PERSON);                        //Step 16
    saleByQuarter(salesPersonList,
                  NO_OF_SALES_PERSON,
                  totalSaleByQuarter);                  //Step 17
    totalSaleByPerson(salesPersonList,
                      NO_OF_SALES_PERSON);              //Step 18

    printReport(outfile, salesPersonList,
                NO_OF_SALES_PERSON,
                totalSaleByQuarter);                    //Step 19
    maxSaleByPerson(outfile, salesPersonList,
                    NO_OF_SALES_PERSON);                //Step 20
    maxSaleByQuarter(outfile, totalSaleByQuarter);      //Step 21

    infile.close();                                     //Step 22
    outfile.close();                                    //Step 22

    return 0;
}

//Place the definitions of the functions initialize,
//getData, saleByQuarter, totalSaleByPerson,
//printReport, maxSaleByPerson, and maxSaleByQuarter here.
```

Sample Run: In this sample run, the user input is shaded.

```
Enter the salesPerson ID file name: Ch9_SalesManID.txt

Enter the sales data file name: Ch9_SalesData.txt

Enter the output file name: Ch9_SalesDataAnalysis.txt
```

Input File: Salespeople's IDs

```
12345
32214
23422
57373
35864
54654
```

Input File: Salespeople's Data

```
12345 1 893
32214 1 343
23422 3 903
57373 2 893
35864 5 329
54654 9 392
12345 2 999
32214 4 892
```

```
23422 4 895
23422 2 492
57373 6 892
35864 10 1223
54654 11 3420
12345 12 322
35864  5 892
54654  3 893
12345 8 494
32214 8 9023
23422 6 223
23422 4 783
57373 8 8834
35864 3 2882
```

Output File: `Ch9_SalesDataAnalysis.txt`

```
------------ Annual Sales Report ------------

  ID         QT1         QT2         QT3          QT4         Total
___________________________________________________________________
 12345     1892.00        0.00      494.00      322.00      2708.00
 32214      343.00      892.00     9023.00        0.00     10258.00
 23422     1395.00     1901.00        0.00        0.00      3296.00
 57373      893.00      892.00     8834.00        0.00     10619.00
 35864     2882.00     1221.00        0.00     1223.00      5326.00
 54654      893.00        0.00      392.00     3420.00      4705.00
 Total     8298.00     4906.00    18743.00     4965.00

Max Sale by SalesPerson: ID = 57373, Amount = $10619.00
Max Sale by Quarter: Quarter = 3, Amount = $18743.00
```

QUICK REVIEW

1. A `struct` is a collection of a fixed number of components.
2. Components of a `struct` can be of different types.
3. The syntax to define a `struct` is:

```
struct structName
{
    dataType1 identifier1;
    dataType2 identifier2;
      .
      .
      .
   dataTypen identifiern;
};
```

4. In C++, `struct` is a reserved word.
5. In C++, `struct` is a definition; no memory is allocated. Memory is allocated for the `struct` variables only when you declare them.
6. Components of a `struct` are called members of the `struct`.
7. Components of a `struct` are accessed by name.
8. In C++, the dot (`.`) operator is called the member access operator.
9. Members of a `struct` are accessed by using the dot (.) operator. For example, if **`employeeType`** is a `struct`, **`employee`** is a variable of type **`employeeType`**, and **`name`** is a member of **`employee`**, then the expression **`employee.name`** accesses the member **`name`**. That is, **`employee.name`** is a variable and can be manipulated like other variables.
10. The only built-in operations on a `struct` are the assignment and member access operations.
11. Neither arithmetic nor relational operations are allowed on `struct` (s).
12. As a parameter to a function, a `struct` can be passed either by value or by reference.
13. A function can return a value of type `struct`.
14. A `struct` can be a member of another `struct`.

EXERCISES

The number in parentheses at the end of an exercise refers to the learning objective listed at the beginning of the chapter.

1. Mark the following statements as true or false.
    a. All members of a `struct` must be of different types. (1)
    b. A `struct` is a definition, not a declaration. (1)
    c. A `struct` variable must be declared after the `struct` definition. (1)
    d. A `struct` member is accessed by using the operator **`:`**. (2)
    e. The only allowable operations on a `struct` are assignment and member selection. (2)
    f. Because a `struct` has a finite number of components, relational operations are allowed on a `struct`. (2)
    g. Some aggregate input/output operations are allowed on a `struct` variable. (2)
    h. A `struct` variable can be passed as a parameter either by value or by reference. (4)
    i. A function cannot return a value of the type `struct`. (4)

j. An array can be a member of a struct. (6, 7)

k. A member of a struct can be another struct. (8)

2. Define a struct **computerType** to store the following data about a computer: Manufacturer (**string**), model type (**string**), processor type (**string**), ram (int) in GB, hard drive size (int) in GB, year when the computer was built (int), and the price (double). (1)

3. Assume the definition of Exercise 2. Declare a **computerType** variable and write C++ statements to store the following information: Manufacturer—Computer Corporation, model—Desk Top, processor type—Core I 7, RAM—12 GB, hard drive size—500 GB, year when the computer was built—2016, and the price—875.00. (2)

4. Consider the declaration of the struct **houseType** given in this chapter. Write C++ statements to do the following: (2, 3)

   a. Declare variables **oldHouse** and **newHouse** of type **houseType**.

   b. Store the following information into **oldHouse**: Style—Two-story, number of bedrooms—5, number of bathrooms—3, number of cars garage—4, year built—1975, finished square footage—3500, price—675000, and tax = 12500.

   c. Copy the values of the components of **oldHouse** into the corresponding components of **newHouse**.

5. Consider the declaration of the struct **houseType** given in this chapter. Suppose **firstHouse** and **secondHouse** are variables of **houseType**. Write C++ statement(s) to compare the style and price of **firstHouse** and **secondHouse**. Output true if the corresponding values are the same; false otherwise. (2, 3)

6. Define a struct **fruitType** to store the following data about a fruit: Fruit name (**string**), color (**string**), fat (int), sugar (int), and carbohydrate (int). (1)

7. Assume the definition of Exercise 6. Declare a variable of type **fruitType** to store the following data: Fruit name—**banana**, color—**yellow**, fat—**1**, sugar—**15**, carbohydrate—**22**. (2)

8. Assume the definition of Exercise 6.

   a. Write a C++ function, **getFruitInput** to read and store data into a variable of **fruitType**.

   b. Write a C++ function, **printFruitInfo** to output data stored into a variable of **fruitType**. Use appropriate labels to identify each component. (4)

9. Which aggregate operations allowed on struct variables are not allowed on an array variable? (5)

10. Consider the following statements:

```
struct nameType          struct courseType        struct studentType
{                        {                        {
    string first;            string name;             nameType name;
    string last;             int callNum;             double gpa;
};                           int credits;             courseType course;
                             char grade;          };
                         };
```

```
studentType student;
studentType classList[100];
courseType course;
nameType name;
```

Mark the following statements as valid or invalid. If a statement is invalid, explain why. (2, 3, 6, 7, 8)

a. `student.gpa = 3.76;`

b. `student.name.last= "Anderson";`

c. `classList[1].name = student;`

d. `classList[0].callNum = 0;`

e. `student.name = classList[10].name;`

f. `course = classList[0];`

g. `cin << classList[0];`

h.
```
for (int j = 0; j < 100; j++)
    classList[j].course = course;
```

i. `classList.name.last = " ";`

j. `course.credits = studentType.course.credits;`

11. a. Assume the declarations of Exercise 10. Write C++ statements to store the following information in `classList[0]`. (2, 3, 6, 7, 8)

```
name: Jessica Miller
gpa:  3.8
course name: Data Structure
course call number: 8340
course credits: 3
course grade: B
```

b. Write a C++ statement to copy the value of `classList[0]` into the variable `student`.

12. Assume the declarations of Exercise 10. Write C++ statements that do the following. (2, 3, 6, 7, 8)

a. Store the following information in `course`:

```
name: Programming I
callNum:  13452
credits: 3
grade: ""
```

b. In the array `classList`, initialize each `gpa` to `0.0`.

c. Copy the information of the 31st component of the array `classList` into `student`.

d. Update the `gpa` of the 10th student in the array `classList` by adding `0.75` to its previous value.

13. Consider the following statements (`nameType` is as defined in Exercise 10):

```
struct employeeType
{
    nameType name;
    int performanceRating;
    int pID;
    string dept;
    double salary;
};
employeeType employees[100];
employeeType newEmployee;
```

Mark the following statements as valid or invalid. If a statement is invalid, explain why. (2, 3, 6, 7, 8)

a. `newEmployee.name = "John Smith";`

b. `cout << newEmployee.name;`

c. `employees[35] = newEmployee;`

d.
```
if (employees[45].pID == 555334444)
    employees[45].performanceRating = 1;
```

e. `employees.salary = 0;`

14. Assume the declarations of Exercises 10 and 13. Write C++ statements that do the following. (2, 3, 6, 7, 8)

a. Store the following information in `newEmployee`:

```
name: Mickey Doe
pID:  111111111
performanceRating: 2
dept: ACCT
salary: 34567.78
```

b. In the array `employees`, initialize each `performanceRating` to `0`.

c. Copy the information of the 20th component of the array `employees` into `newEmployee`.

d. Update the salary of the 50th employee in the array `employees` by adding `5735.87` to its previous value.

15. Assume that you have the following definition of a `struct`.

```
struct sportsType
{
    string sportName;
    string teamName;
    int numberOfPlayers;
    double teamPayroll;
    double coachSalary;
};
```

Declare an array, `soccer`, of 20 components of type `sportsType`. (7)

16. Assume the definition of Exercise 15. (7)
    a. Write a C++ code to initialize each component of `soccer` as follows: `sportName` to null string, `teamName` to null string, `numberOfPlayers` to `0`, `teamPayroll` to `0.0`, and `coachSalary` to `0.0`.
    b. Write a C++ code that uses a loop to input data into the array `soccer`. Assume that the variable `length` indicates the number of elements in `soccer`.
    c. Write a C++ code that outputs the names of soccer teams whose pay roll is greater than or equal to 10,000,000.

17. Assume the definition and declaration of Exercise 15. (4, 7)
    a. Write the definition of a `void` function that can be used to input data in a variable of type `sportsType`. Also write a C++ code that uses your function to input data in `sportsType`.
    b. Write the definition of a `void` function that can be used to output data in a variable of type `sportsType`. Also write a C++ code that uses your function to output data in `soccer`.

18. Suppose that you have the following definitions: (8)

```
struct timeType                  struct tourType
{                                {
    int hr;                          string cityName;
    double min;                      int distance;
    int sec;                         timeType travelTime;
};                               };
```

    a. Declare the variable `destination` of type `tourType`.
    b. Write C++ statements to store the following data in `destination`: `cityName`—Chicago, `distance`—550 miles, `travelTime`—9 hours and 30 minutes.
    c. Write the definition of a function to output that data stored in a variable of type `tourType`.

9

d. Write the definition of a value-returning function that inputs data into a variable of type **tourType**.

e. Write the definition of a void function with a reference parameter of type **tourType** to input data in a variable of type **tourType**.

PROGRAMMING EXERCISES

1. Assume the definition of Exercise 2, which defines the struct **computerType**. Write a program that declares a variable of type **computerType**, prompts the user to input data about a computer, and outputs computer's data.

2. Write a program that reads students' names followed by their test scores. The program should output each student's name followed by the test scores and the relevant grade. It should also find and print the highest test score and the name of the students having the highest test score.

   Student data should be stored in a struct variable of type **studentType**, which has four components: **studentFName** and **studentLName** of type **string**, **testScore** of type int (**testScore** is between **0** and **100**), and **grade** of type char. Suppose that the class has 20 students. Use an array of 20 components of type **studentType**.

   Your program must contain at least the following functions:

   a. A function to read the students' data into the array.

   b. A function to assign the relevant grade to each student.

   c. A function to find the highest test score.

   d. A function to print the names of the students having the highest test score.

   Your program must output each student's name in this form: last name followed by a comma, followed by a space, followed by the first name; the name must be left justified. Moreover, other than declaring the variables and opening the input and output files, the function **main** should only be a collection of function calls.

3. Define a struct **menuItemType** with two components: **menuItem** of type **string** and **menuPrice** of type double.

4. Write a program to help a local restaurant automate its breakfast billing system. The program should do the following:

   a. Show the customer the different breakfast items offered by the restaurant.

   b. Allow the customer to select more than one item from the menu.

   c. Calculate and print the bill.

Assume that the restaurant offers the following breakfast items (the price of each item is shown to the right of the item):

```
Plain Egg                    $1.45
Bacon and Egg                $2.45
Muffin                       $0.99
French Toast                 $1.99
Fruit Basket                 $2.49
Cereal                       $0.69
Coffee                       $0.50
Tea                          $0.75
```

Use an array `menuList` of type `menuItemType`, as defined in Programming Exercise 3. Your program must contain at least the following functions:

- Function `getData`: This function loads the data into the array `menuList`.
- Function `showMenu`: This function shows the different items offered by the restaurant and tells the user how to select the items.
- Function `printCheck`: This function calculates and prints the check. (Note that the billing amount should include a 5% tax.)
  A sample output is:

```
Welcome to Johnny's Restaurant
Bacon and Egg                $2.45
Muffin                       $0.99
Coffee                       $0.50
Tax                          $0.20
Amount Due                   $4.14
```

Format your output with two decimal places. The name of each item in the output must be left justified. You may assume that the user selects only one item of a particular type.

5. Redo Exercise 4 so that the customer can select multiple items of a particular type. A sample output in this case is:

```
Welcome to Johnny's Restaurant
1   Bacon and Egg            $2.45
2   Muffin                   $1.98
1   Coffee                   $0.50
    Tax                      $0.25
    Amount Due               $5.18
```

6. Write a program whose main function is merely a collection of variable declarations and function calls. This program reads a text and outputs the letters, together with their counts, as explained below in the function `printResult`. (There can be no global variables! All information must be passed in and out of the functions. Use a structure to store the information.) Your program must consist of at least the following functions:
   - Function `openFile`: Opens the input and output files. You must pass the file streams as parameters (by reference, of course). If the file does not exist, the program should print an appropriate message and exit. The program must ask the user for the names of the input and output files.
   - Function `count`: Counts every occurrence of capital letters `A-Z` and small letters `a-z` in the text file opened in the function `openFile`. This information must go into an array of structures. The array must be passed as a parameter, and the file identifier must also be passed as a parameter.
   - Function `printResult`: Prints the number of capital letters and small letters, as well as the percentage of capital letters for every letter `A-Z` and the percentage of small letters for every letter `a-z`. The percentages should look like this: `"25%"`. This information must come from an array of structures, and this array must be passed as a parameter.

7. Write a program that declares a `struct` to store the data of a football player (player's name, player's position, number of touchdowns, number of catches, number of passing yards, number of receiving yards, and the number of rushing yards). Declare an array of 10 components to store the data of 10 football players. Your program must contain a function to input data and a function to output data. Add functions to search the array to find the index of a specific player, and update the data of a player. (You may assume that the input data is stored in a file.) Before the program terminates, give the user the option to save data in a file. Your program should be menu driven, giving the user various choices.