CHAPTER 3

Writing Structured C++ Programs

After studying this chapter, you will be able to:

- Use structured flowcharts and pseudocode to write structured C++ programs
- Write simple modular programs in C++

In this chapter, you begin to learn how to write structured C++ programs. As you will see, creating a flowchart and writing pseudocode before you actually write the program ensures that you fully understand the program's intended design. You begin by looking at a structured flowchart and pseudocode from your text, *Programming Logic and Design, Eighth Edition*. You should do the exercises and labs in this chapter only after you have finished Chapters 2 and 3 of that book.

Using Flowcharts and Pseudocode to Write a C++ Program

In the first three chapters of *Programming Logic and Design*, you studied flowcharts and pseudocode for the Number-Doubling program. Figure 3-1 shows the functional, structured flowchart and pseudocode for this program.

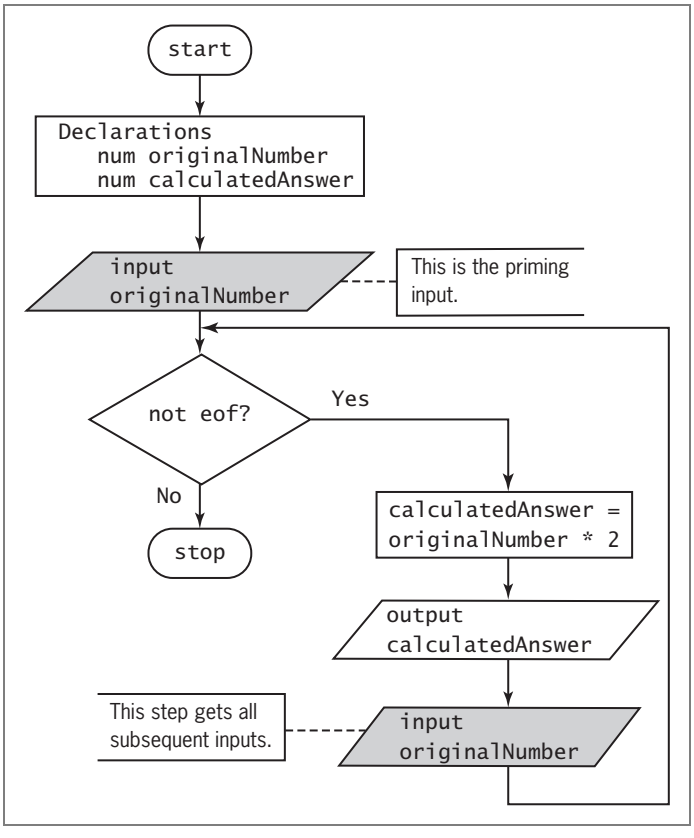

Figure 3-1 Functional, structured flowchart and pseudocode for the Number-Doubling program

By studying the flowchart and pseudocode, you can see that this program makes use of the sequence and loop structures you were introduced to in *Programming Logic and Design*. The remainder of this section walks you through the C++ code for this program. The explanations assume that you are simply reading along, but if you want, you can type the code as it is presented. The goal of this section is to help you get a feel for how flowcharts and pseudocode

can serve as a guide when you write C++ programs. You must learn more about C++ before you can expect to write this program by yourself.

In Figure 3-1, the first line of the pseudocode is the word `start`. How do you translate this pseudocode command into the C++ code that will start the Number-Doubling program? In Chapter 1 of this book, you learned that to start a C++ program, you first create a `main()` function. So to start the Number-Doubling program, you will first create a function named `main()` because it is always the first function that executes in a C++ program. Thus, the code that follows starts the Number-Doubling program by creating a `main()` function:

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

Notice that each opening curly brace is matched by a closing curly brace.

If you are typing the code as it is presented here, save the program in a file that has an appropriate name, such as `NumberDouble.cpp`. The complete program is also saved in a file named `NumberDouble.cpp` and is included in the student files for this chapter.

Next, you see that two variables, `originalNumber` and `calculatedAnswer`, are declared as data type `num`. The C++ code that follows adds the variable declarations, with the declarations shown in bold.

```
int main()
{
   int originalNumber;
   int calculatedAnswer;
}
```

The next line of the pseudocode instructs you to input the `originalNumber`. In other words, you need to write the input statement that primes the loop. You learned about priming read statements in Chapter 3 of *Programming Logic and Design*. In Chapter 2 of this book, you learned how to use interactive input statements in programs to allow the user to input data. You also learned to prompt the user by explaining what the program expects to receive as input. The following example includes the code that implements the priming read by displaying a prompt for the user and then retrieving the number the user wants doubled.

Note that the code in boldface has been added to the Number-Doubling program. If you were writing this code yourself, you would start by writing the code for the Number-Doubling program as shown above, and then edit it to add the boldface code shown here:

```
int main()
{
   int originalNumber;
   int calculatedAnswer;
   cout << "Enter a number to double: " << endl;
   cin >> originalNumber;
}
```


You have not learned enough about `while` loops to write this code yourself, but you can observe how it is done in this example. You will learn more about loops in Chapter 5 of this book.

Next, the pseudocode instructs you to begin a `while` loop with `eof` (end of file) used as the condition to exit the loop.

Since you are using interactive input in this program, it requires no `eof` marker. Instead you will use the number 0 (zero) to indicate the end of input. You use 0 because 0 doubled will always be 0. The use of 0 to indicate the end of input also requires you to change the prompt to tell the user how to end the program. Review the following code. Again, the newly added code is formatted in bold.

```
int main()
{
   int originalNumber;
   int calculatedAnswer;
   cout << "Enter a number to double or 0 to end: " << endl;
   cin >> originalNumber;
   while(originalNumber != 0)
   {
   }
}
```

Notice that a beginning curly brace (`{`) and an ending curly brace (`}`) are used in C++ to mark the beginning and end of code that executes as part of a loop.

According to the pseudocode, the body of the loop is made up of three sequential statements. The first statement calculates the `originalNumber` multiplied by 2, the second statement prints the `calculatedAnswer`, and the third statement retrieves the next `originalNumber` from the user. In C++, you actually need to add an additional statement between the curly braces that mark the body of the `while` loop. This additional statement prompts the user to enter the next number to be doubled.

In the following example, the code that makes up the body of the loop is in bold:

```
int main()
{
   int originalNumber;
   int calculatedAnswer;
   cout << "Enter a number to double or 0 to end: " << endl;
   cin >> originalNumber;
   while(originalNumber != 0)
   {
      calculatedAnswer = originalNumber * 2;
```

```
      cout << originalNumber << " doubled is "
           << calculatedAnswer << endl;
      cout << "Enter a number to double or 0 to end: " << endl;
      cin >> originalNumber;
   }
}
```

The last line of the pseudocode instructs you to end the program. In C++, the closing curly brace (}) for the `main()` function signifies the end of the `main()` function and therefore the end of the program. Note that the preceding code includes two closing curly braces. The last one is the one that ends the `main()` function and the second to last one ends the `while` loop.

You are almost finished translating the pseudocode into C++ code. The code shown in bold below is the `#include` preprocessor directive and the `using` statement that you learned about in Chapter 1 of this book. Additionally, you see the `return 0;` statement. This statement instructs the compiler to return the value 0 from the `main()` function.

```
#include <iostream>
using namespace std;
int main()
{
   int originalNumber;
   int calculatedAnswer;
   cout << "Enter a number to double or 0 to end: " << endl;
   cin >> originalNumber;
   while(originalNumber != 0)
   {
      calculatedAnswer = originalNumber * 2;
      cout << originalNumber << " doubled is "
           << calculatedAnswer << endl;
      cout << "Enter a number to double or 0 to end: " << endl;
      cin >> originalNumber;
   }
   return 0;
}
```

At this point, the program is ready to be compiled. Assuming there are no syntax errors, it should execute as planned. Figure 3-2 displays the input and output of the Number-Doubling program.

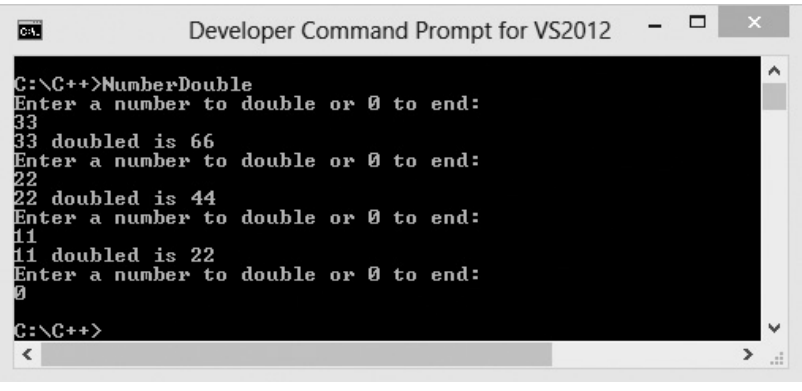

Figure 3-2 Number Double program input and output

Although you have not learned everything you need to know to write this program yourself, you can see from this example that writing the program in C++ is easier if you start with a well designed, functional, structured flowchart or pseudocode.

Lab 3-1: Using Flowcharts and Pseudocode to Write a C++ Program

In this lab, you use the flowchart and pseudocode in Figure 3-3 to add code to a partially created C++ program. When completed, college admissions officers should be able to use the C++ program to determine whether to accept or reject a student, based on his or her test score and class rank.

```
start
   input testScore, classRank
   if testScore >= 90 then
      if classRank >= 25 then
        output "Accept"
      else
        output "Reject"
      endif
   else
      if testScore >= 80 then
        if classRank >= 50 then
          output "Accept"
        else
          output "Reject"
        endif
      else
        if testScore >= 70 then
          if classRank >= 75 then
            output "Accept"
          else
            output "Reject"
          endif
        else
          output "Reject"
        endif
      endif
   endif
stop
```

Figure 3-3 Pseudocode for the College Admission program

1. Study the pseudocode in Figure 3-3.
2. Open the source code file named `CollegeAdmission.cpp` using Notepad or the text editor of your choice.
3. Declare two integer variables named `testScore` and `classRank`.

4. Write the interactive input statements to retrieve a student's test score and class rank from the user of the program. Do not forget to prompt the user for the test score and class rank.
5. The rest of the program is written for you. Save this source code file in a directory of your choice, and then make that directory your working directory.
6. Compile the source code file `CollegeAdmission.cpp`.
7. Execute the program by entering 87 for the test score and 60 for the class rank. Record the output of this program.

_____

8. Execute the program by entering 60 for the test score and 87 for the class rank. Record the output of this program.

_____

Writing a Modular Program in C++

In Chapter 2 of your book, *Programming Logic and Design,* you learned about local and global variables and named constants. To review briefly, you declare **local variables** and local constants within the module—or, in C++ terminology, the function—that uses them. Further, you can only use a local variable or a local constant within the function in which it is declared. **Global variables** and global constants are known to the entire program. They are declared at the program level and are visible to and usable in all the functions called by the program. It is not considered a good programming practice to use global variables, so the C++ program below uses local variables (as well as local constants). A good reason for using local variables and local constants is that source code is easier to understand when variables are declared where they are used. In addition, global variables can be accessed and altered by any part of the program, which can make the program difficult to read and maintain; it also makes the program prone to error.

Recall from Chapter 2 that most programs consist of a `main()` module or function that contains the mainline logic. The `main()` function then calls other methods to get specific work done in the program. The mainline logic of most procedural programs follows this general structure:

1. Declarations of variables and constants
2. **Housekeeping tasks**, such as displaying instructions to users, displaying report headings, opening files the program requires, and inputting the first data item
3. **Detail loop tasks** that do the main work of the program, such as processing many records and performing calculations
4. **End-of-job tasks**, such as displaying totals and closing any open files

In Chapter 2 of *Programming Logic and Design,* you studied a flowchart for a modular program that prints a payroll report with global variables and constants. This flowchart is shown in Figure 3-4.

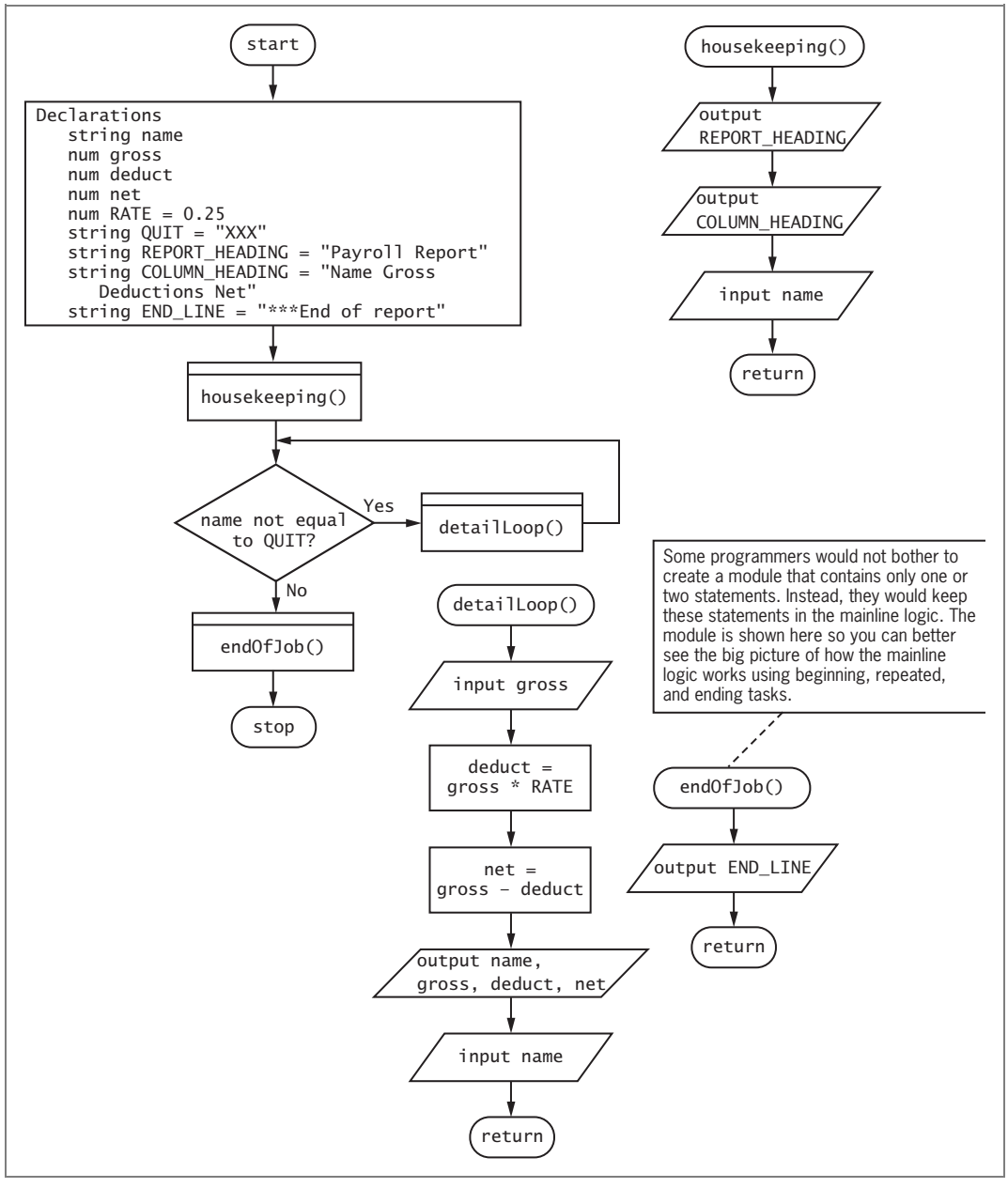

Figure 3-4 Flowchart for the Payroll Report program

In this section, you walk through the process of creating a C++ program that implements the logic illustrated in the flowchart in Figure 3-4. According to the flowchart, the program begins with the execution of the mainline method. The mainline method declares four global variables (`name`, `gross`, `deduct`, and `net`) and five global constants (`RATE`, `QUIT`, `REPORT_HEADING`, `COLUMN_HEADING`, and `END_LINE`).

The C++ code that follows shows the Payroll Report program with the `main()` function, as well as the variable and constant declarations. Notice that a `#include` preprocessor directive you have not seen before has been added at the beginning of the program. The preprocessor directive `#include <string>` allows you to use `string` objects in your C++ program. You will learn more about `string` objects in subsequent chapters.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   // Declare variables and constants local to main()
   string name;
   double gross, deduct, net;
   const double RATE = 0.25;
   const string QUIT = "XXX";
   const string REPORT_HEADING = "Payroll Report ";
   const string END_LINE = "**End of report ";
} // End of main() function
```

If you are typing the code as it is presented here, remember to save the program in a file that is named appropriately, for example, `PayrollReport.cpp`. The complete program is also saved in a file named `PayrollReport.cpp` and is included in the student files for this chapter.

By convention, in C++ the names of constants appear in all uppercase letters. Multiple words are separated with underscores. Some programmers believe using all uppercase makes it easier for you to spot named constants in a long block of code.

Notice that one of the declarations shown in the flowchart, `String COLUMN_HEADING = "Name Gross Deductions Net"` is not included in the C++ code. Since you have not yet learned about the C++ statements needed to line up values in report format, the C++ program shown above prints information on separate lines rather than in the column format used in the flowchart.

It is important for you to understand that the variables and constants declared in the flowchart are global variables that can be used in all modules that are part of the program. However, as mentioned earlier, it is not considered a good programming practice to use global variables. The variables and constants declared in the C++ version are local, which means they can only be used in the `main()` function.

After the declarations, the flowchart makes a call to the `housekeeping()` module that prints the `REPORT_HEADING` and `COLUMN_HEADING` constants and retrieves the first employee's name entered by the user of the program. The code that follows shows how these tasks are translated to C++ statements. The added code is shown in bold.

```
#include <iostream>
#include <string>
using namespace std;
int main()
```

```
{
   // Declare variables and constants local to main()
   string name;
   double gross, deduct, net;
   const double RATE = 0.25;
   const string QUIT = "XXX";
   const string REPORT_HEADING = "Payroll Report";
   const string END_LINE = "**End of report";
   // Work done in the housekeeping() function
   cout << REPORT_HEADING << endl;
   cout << "Enter employee's name: ";
   cin >> name;
} // End of main() function
```

Since it is not considered a good programming practice to use global variables, all of the variables and constants declared for this program are local variables that are available only in the `main()` function. If you were to create an additional function for the housekeeping tasks, that function would not have access to the `name` variable to store an employee's name. So, for now, the C++ programs that you write will have only one function (module), the `main()` function. Additional modules, such as the `housekeeping()` module, will be simulated through the use of comments. As shown in the preceding code, the statements that would execute as part of a `housekeeping()` function have been grouped together in the C++ program and preceded by a comment. You will learn how to create additional functions and pass data to functions in Chapter 9 of this book.

In the flowchart, the next statement to execute after the `housekeeping()` module finishes its work is a `while` loop in the main module that continues to execute until the user enters `XXX` when prompted for an employee's name. Within the loop, the `detailLoop()` module is called. The work done in the `detailLoop()` consists of retrieving an employee's gross pay; calculating deductions; calculating net pay; printing the employee's name, gross pay, deductions, and net pay on the user's screen; and retrieving the name of the next employee. The following code shows the C++ statements that have been added to the `PayrollReport` program to implement this logic. The added statements are shown in bold.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   // Declare variables and constants local to main()
   string name;
   double gross, deduct, net;
   const double RATE = 0.25;
   const string QUIT = "XXX";
   const string REPORT_HEADING = "Payroll Report";
   const string END_LINE = "**End of report ";
   // Work done in the housekeeping() function
   cout << REPORT_HEADING << endl;
   cout << "Enter employee's name: ";
   cin >> name;
   while(name != QUIT)
```

```
   {
         // work done in the detailLoop() function
         cout << "Enter employee's gross pay: ";
         cin >> gross;
         deduct = gross * RATE;
         net = gross - deduct;
         cout << "Name: " << name << endl;
         cout << "Gross Pay: " << gross << endl;
         cout << "Deductions: " << deduct << endl;
         cout << "Net Pay: " << net << endl;
         cout << "Enter employee's name: ";
         cin >> name;
   }
} // End of main() function
```

The `while` loop in the C++ program compares the name entered by the user with the value of the constant named `QUIT`. As long as the name is not equal to `XXX` (the value of `QUIT`), the loop executes. The statements that make up the simulated `detailLoop()` method include retrieving the employee's gross pay; calculating deductions and net pay; printing the employee's name, gross pay, deductions, and net pay; and retrieving the name of the next employee to process.

In the flowchart, when a user enters `XXX` for the employee's name, the program exits the `while` loop and then calls the `endOfJob()` module. The `endOfJob()` module is responsible for printing the value of the `END_LINE` constant. When the `endOfJob()` module finishes, control returns to the mainline module, and the program stops. The completed C++ program is shown next with the additional statements shown in bold.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   // Declare variables and constants local to main()
   string name;
   double gross, deduct, net;
   const double RATE = 0.25;
   const string QUIT = "XXX";
   const string REPORT_HEADING = "Payroll Report";
   const string END_LINE = "**End of report ";
   // Work done in the housekeeping() function
   cout << REPORT_HEADING << endl;
   cout << "Enter employee's name: ";
   cin >> name;
   while(name != QUIT)
   {
      // work done in the detailLoop() function
      cout << "Enter employee's gross pay: ";
      cin >> gross;
      deduct = gross * RATE;
      net = gross - deduct;
      cout << "Name: " << name << endl;
      cout << "Gross Pay: " << gross << endl;
```

```
      cout << "Deductions: " << deduct <<endl;
      cout << "Net Pay: " << net << endl;
      cout << "Enter employee's name: ";
      cin >> name;
   }
   // work done in the endOfJob() function
   cout << END_LINE;
   return 0;
} // End of main() function
```


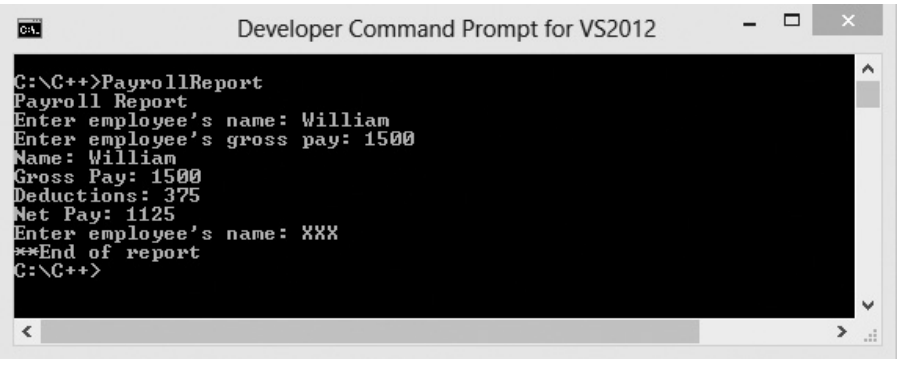

Figure 3-5 Output of the Payroll Report program when the input is `William` and `1500`

This program is now complete. Figure 3-5 shows the program's output in response to the input `William` (for the name), and `1500` (for the gross).

Lab 3-2: Writing a Modular Program in C++

In this lab, you add the input and output statements to a partially completed C++ program. When completed, the user should be able to enter a year, a month, and a day. The program then determines if the date is valid. Valid years are those that are greater than 0, valid months include the values 1 through 12, and valid days include the values 1 through 31.

1. Open the source code file named `BadDate.cpp` using Notepad or the text editor of your choice.
2. Notice that variables have been declared for you.
3. Write the simulated `housekeeping()` function that contains the prompts and input statements to retrieve a year, a month, and a day from the user.
4. Include the output statements in the simulated `endOfJob()` function. The format of the output is as follows:

   month/day/year `is a valid date.`
   or
   month/day/year `is an invalid date.`

5. Save this source code file in a directory of your choice, and then make that directory your working directory.
6. Compile the source code file `BadDate.cpp`.
7. Execute the program entering the following date: month = **5**, day = **32**, year = **2014**. Record the output of this program.
8. Execute the program entering the following date: month = **9**, day = **21**, year = **2002**. Record the output of this program.