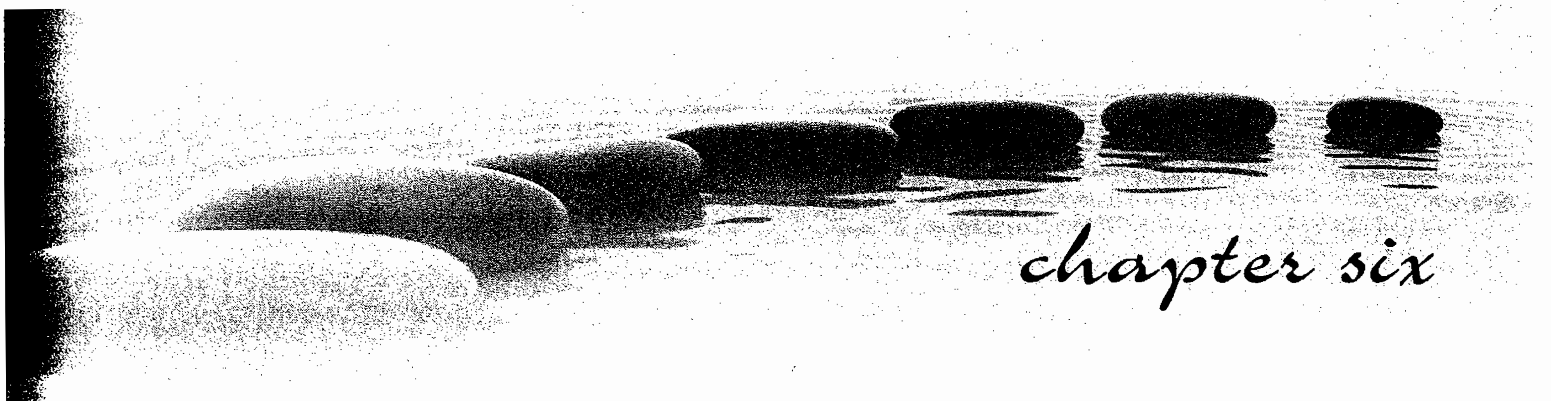


chapter six

Modular Programming

Learning Objectives

- To learn how to return function results through a function's arguments
- To understand the differences between call-by-value and call-by-reference
- To understand the distinction between input, inout, and output parameters and when to use each kind
- To learn how to modularize a program system and pass information between system modules (functions)
- To understand how to document the flow of information using structure charts
- To learn testing and debugging techniques appropriate for a program system with several modules

A CAREFULLY DESIGNED PROGRAM that has been constructed using functions has many of the properties of a stereo system. Each stereo component is an independent device that performs a specific operation. Electronic audio signals move back and forth over wires linking the stereo components. Plugs on the back of the stereo receiver are marked as inputs or outputs. Wires attached to the input plugs carry electronic signals into the receiver, where receiver sends new electronic signals through the output plugs to the speakers or back to the cassette deck for recording. You can connect the components and listen to your favorite music without knowing what electronic parts each component contains or how it works.

In Chapter 3 you learned how to write the separate components—the functions—of a program. The functions correspond to the individual steps in a problem solution. You also learned how to provide inputs to a function and how to return a single output. In this chapter you complete your study of functions, learning how to connect the functions to create a program system—an arrangement of separate modules that pass information from one to the other.

6.1 Value and Reference Parameters

At this point we know how to write functions that return up to one result. In this section, we'll learn how to write functions that use output parameters to return more than one result.

EXAMPLE 6.1

Function `computeSumAve` at the bottom of Listing 6.1 has four parameters: two for input (`num1` and `num2`) and two for output (`sum` and `average`). The symbol `&` (ampersand) in the function header indicates that the parameters `sum` and `average` are output parameters. The function computes the sum and average of its inputs but doesn't display them. Instead, these values are assigned to formal parameters `sum` and `average` and returned as function results to the `main` function that displays them.

The function call

```
computeSumAve(x, y, sum, mean);
```

sets up the argument correspondence below.

Actual Argument	Corresponds to formal parameter
`x`	`num1 (input)`
`y`	`num2 (input)`
`sum`	`sum (output)`
`mean`	`average (output)`

The values of `x` and `y` are passed into the function when it's first called. These values are associated with formal input parameters `num1` and `num2`. The statement

```
sum = num1 + num2;
```

stores the sum of the function inputs in the calling function variable `sum` (the third actual argument). The statement

```
average = sum / 2.0;
```

divides the value stored in the calling function variable `sum` by `2.0` and stores the quotient in the calling function variable `mean` (the fourth actual argument).

Listing 6.1 Function to compute sum and average

```
// File: computeSumAve.cpp
// Tests function computeSumAve.

#include <iostream>
using namespace std;

// Function prototype
void computeSumAve(float, float, float&, float&);

int main()
{
   float x,           // input - first number
         y,           // input - second number
         sum,         // output - their sum
         mean;        // output - their average

   cout << "Enter 2 numbers: ";
   cin >> x >> y;

   // Compute sum and average of x and y
   computeSumAve(x, y, sum, mean);

   // Display results
   cout << "Sum is " << sum << endl;
   cout << "Average is " << mean << endl;

   return 0;
}

// Computes the sum and average of num1 and num2.
// Pre: num1 and num2 are assigned values.
// Post: The sum and average of num1 and num2 are
//       computed and returned as function outputs.
void computeSumAve
   (float num1,          // IN -   values used in
    float num2,          //        computation
    float& sum,          // OUT - sum of num1 and num2
    float& average)      // OUT - average of num1 and num2
{
    sum = num1 + num2;
    average = sum / 2.0;
} // end computeSumAve
```

```
Enter 2 numbers: 8 10
Sum is 18
Average is 9
```

Figure 6.1 shows the main function data area and function `computeSumAve`'s data area after the function call but before the execution of `computeSumAve` begins; Figure 6.2 shows these data areas just after `computeSumAve` finishes execution. The execution of `computeSumAve` sets the values of calling function variables `sum` and `mean` to 18.0 and 9.0, respectively. We explain how this happens next.

Call-by-Value and Call-by-Reference Parameters

In C++, you insert the symbol `&` immediately following the type of a formal parameter to declare an output parameter. Therefore, in function `computeSumAve` in Listing 6.1, the formal parameters `sum` and `average` are output parameters, and `num1` and `num2` are input parameters.

call by-value
An argument-passing mechanism in which the value of an actual argument is stored in the called function's data area.

The compiler uses the information in the parameter declaration list to set up the correct *argument-passing mechanism* for each function parameter. For parameters used only as input, C++ uses **call-by-value** because the *value* of the argument is copied to the called function's data area and there's no further connection between the formal parameter and its corresponding actual argument. In Figures 6.1 and 6.2, the dashed arrows pointing to `num1` and `num2` indicate this situation. The top dashed arrow shows that the value of the

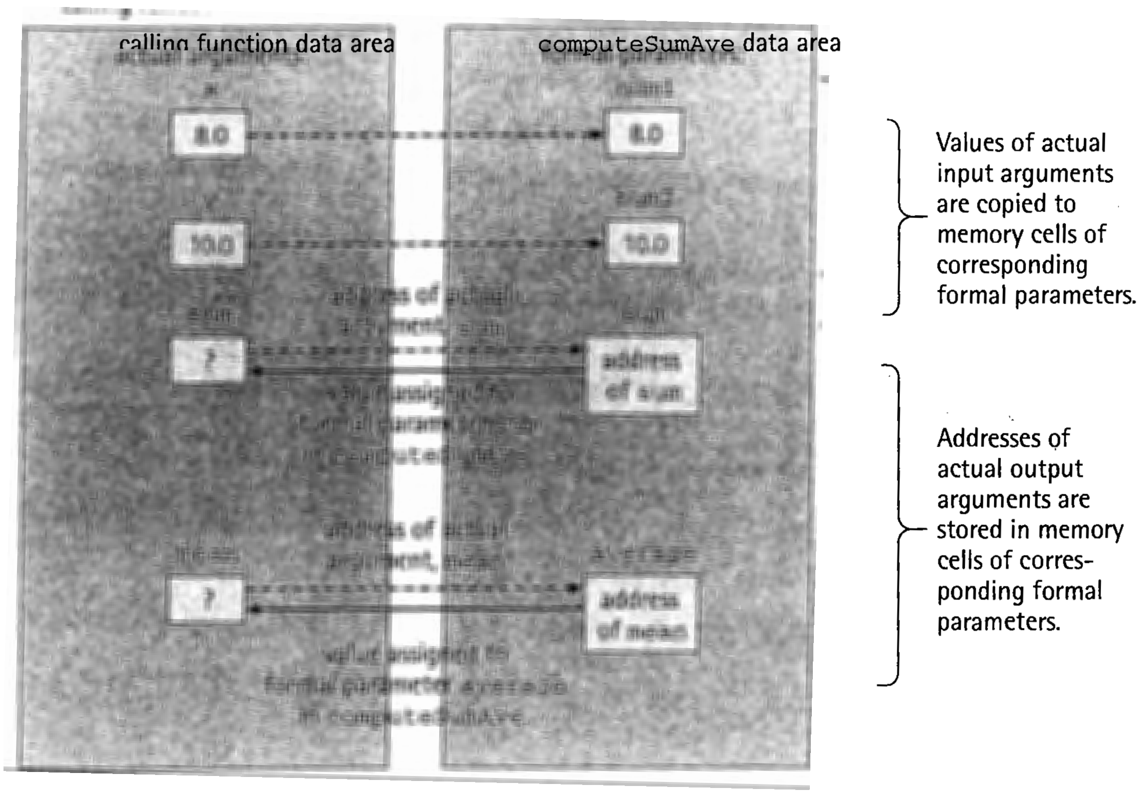
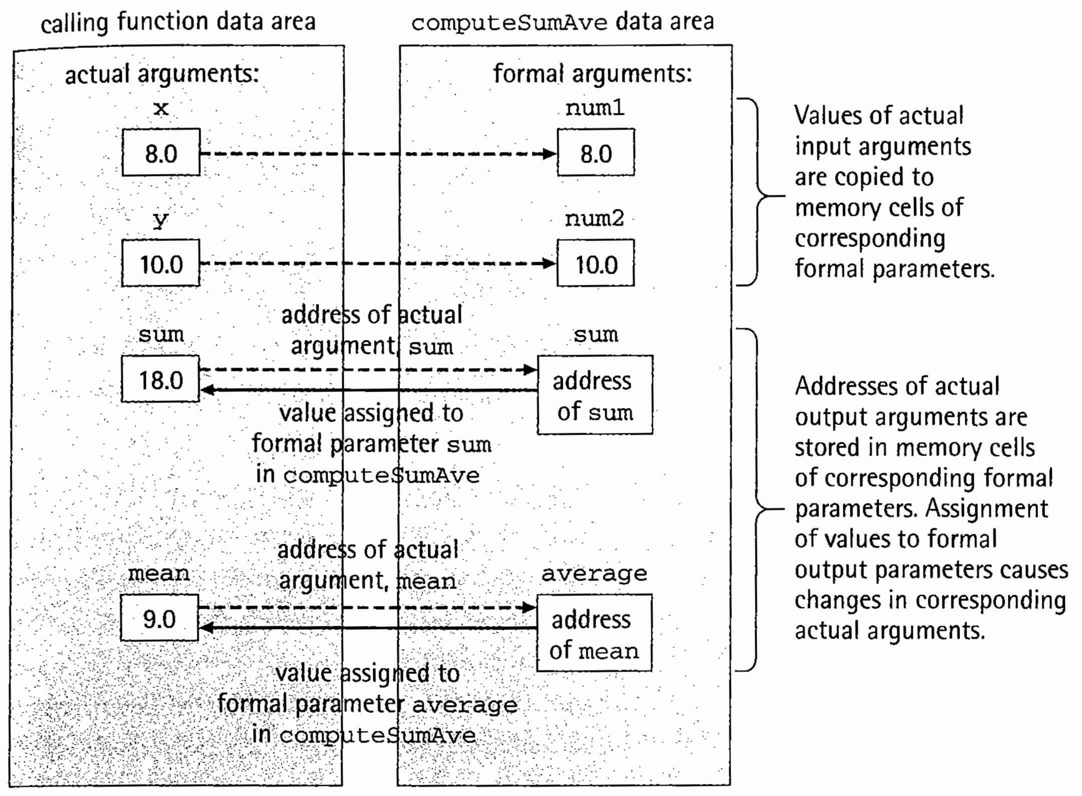

calling function data area
computeSumAve data area
8.0
8.0
10.0
10.0
?
address of sum
?
address of mean
Values of actual input arguments are copied to memory cells of corresponding formal parameters.
Addresses of actual output arguments are stored in memory cells of corresponding formal parameters.

Figure 6.1 Data areas after call to `computeSumAve` (before execution)



calling function data area
computeSumAve data area
actual arguments:
formal arguments:
x
8.0
num1
8.0
y
10.0
num2
10.0
address of actual argument sum
sum
18.0
sum
address of sum
value assigned to formal parameter sum in computeSumAve
address of actual argument, mean
mean
9.0
average
address of mean
value assigned to formal parameter average in computeSumAve
Values of actual input arguments are copied to memory cells of corresponding formal parameters.
Addresses of actual output arguments are stored in memory cells of corresponding formal parameters. Assignment of values to formal output parameters causes changes in corresponding actual arguments.

Figure 6.2 Data areas after execution of `computeSumAve`

actual argument `x` is passed to the formal parameter `num1` in function `computeSumAve`.

For output parameters, C++ uses **call-by-reference**. For *reference parameters,* the compiler stores in the called function's data area the memory *address* of the actual variable that corresponds to each reference parameter. Through this address, the called function can access the actual argument in the calling function, which enables the called function to modify the actual argument value or to use it in a computation. In Figure 6.2, a dashed arrow and a solid arrow connect a reference parameter in `computeSumAve` with its corresponding actual argument in the calling function. The dashed arrow in color shows that the address of variable `mean` in the calling function is passed to formal parameter `average` in `computeSumAve`; the solid arrow in color shows that the value assigned to formal parameter `average` in `computeSumAve` is actually stored in variable `mean` in the calling function.

call-by-reference
An argument-passing mechanism in which the address of an actual argument is stored in the called function's data area.

Remember to place the `&` (for reference parameters) in the formal parameter list only, not in the actual argument list. Also, you must insert `&`s in the function prototype to indicate that the third and fourth formal parameters are reference parameters:

```
void computeSumAve(float, float, float&, float&);
```

`void` Functions Can Return Results

Beginning programmers are sometimes confused by the fact that `computeSumAve` is declared as a `void` function but still returns results to the calling function. By declaring `computeSumAve` as `void`, we're only telling C++ that it doesn't return a value through execution of a `return` statement. But `computeSumAve` is still permitted to return results through its output parameters.

When to Use a Reference or a Value Parameter

How do you decide whether to use a reference parameter or a value parameter? Here are some rules of thumb:

- If information is to be passed into a function and doesn't have to be returned or passed out of the function, then the formal parameter representing that information should be a value parameter (for example, `num1` and `num2` in Figure 6.1). A parameter used in this way is called an **input parameter.**
- If information is to be returned to the calling function through a parameter, then the formal parameter representing that information must be a reference parameter (`sum` and `average` in Figure 6.1). A parameter used in this way is called an **output parameter**.
- If information is to be passed into a function, perhaps modified, and a new value returned, then the formal parameter representing that information must be a reference parameter. A parameter used in this way is called an **input/output parameter** (or **inout parameter**).

input parameter
A parameter used to receive a data value from the calling function.

output parameter
A parameter used to return a result to the calling function.

input/output (inout) parameter
A parameter that receives a data value from the calling function and returns a value to it.

Although we make a distinction between output parameters and input/output parameters, C++ does not. Both must be specified as reference parameters (using the ampersand), so that the address of the corresponding actual argument is stored in the called function data area when the function is called. For an input/output parameter (as well as for an input parameter), we assume there are some meaningful data in the actual argument before the function executes; for an output parameter, we make no such assumption.

PROGRAM STYLE Writing Formal Parameter Lists

In Listing 6.1, the formal parameter list

```
(float num1,          // IN - values used
 float num2,          //      in computation
 float& sum,          // OUT - sum of num1 and num2
 float& average)      // OUT - average of num1 and num2
```

is written on four lines to improve program readability. The value parameters are on the first two lines with comments that document their use as input parameters. The reference parameters are on the next two lines followed by comments that document their use as function outputs

Generally, we follow this practice in writing formal parameter lists. We list input parameters first, input/output parameters next, and output parameters last.

Comparison of Value and Reference Parameters

Table 6.1 summarizes what we've learned so far about value and reference parameters.

Protection Afforded by Value Parameters

Reference parameters are more versatile than value parameters because their values can be used in a computation as well as changed by the function's execution. Why not make all parameters, even input parameters, reference parameters? The reason is that value parameters offer some protection for data integrity. Because copies of value parameters are stored locally in the called function data area, C++ protects the actual argument's value and prevents it from being erroneously changed by the function's execution. For example, if we insert the statement

```
num1 = -5.0;
```

at the end of function `computeSumAve`, the value of formal parameter `num1` will be changed to `-5.0`, but the value stored in `x` (the corresponding actual argument) will still be `8.0`.

Table 6.1 Comparison of Value and Reference Parameters

Value Parameters	Reference Parameters
• Value of corresponding actual argument is stored in the called function.	• Address of corresponding actual argument is stored in the called function.
• The function execution cannot change the actual argument value.	• The function execution can change the actual argument value.
• Actual argument can be an expression, variable, or constant.	• Actual argument must be a variable.
• Formal parameter type must be specified in the formal parameter list.	• Formal parameter type must be followed by `&` in the formal parameter list.
• Parameters are used to store the data passed to a function (input parameters).	• Parameters are used to return outputs from a function (output parameters) or to change the value of a function argument (input / output parameters).

If you forget to declare a formal output parameter as a reference parameter (using the ampersand), then its value (not its address) will be passed to the function when it's called. The argument value is stored locally, and any change to its value will not be returned to the calling function. This is a very common error in parameter usage.

Argument/Parameter List Correspondence Revisited

We first discussed argument/parameter list correspondence in Section 3.5. We used the acronym **not** to indicate that an argument list used in a function call must agree with the called function's parameter list in **n**umber, **o**rder, and **t**ype. We repeat these three rules below and add one more.

Argument/Parameter List Correspondence

- Number: The number of actual arguments used in a call to a function must be the same as the number of formal parameters listed in the function prototype.
- Order: The order of arguments in the lists determines correspondence. The first actual argument corresponds to the first formal parameter, the second actual argument corresponds to the second formal parameter, and so on.
- Type: Each actual argument must be of a data type that can be assigned to the corresponding formal parameter with no unexpected loss of information.
- For reference parameters, an actual argument must be a variable. For value parameters, an actual argument may be a variable, a constant, or an expression.

The last rule states that you can use expressions (or variables or constants) as actual arguments corresponding to value parameters but not to reference parameters. For example, the function call

```
computeSumAve(x + y, 10, mySum, myAve);
```

calls `computeSumAve` to compute the `sum` (returned in `mySum`) and the `average` (returned in `myAve`) of the expression `x + y` and the integer `10`.

Only variables can correspond to reference parameters, so `mySum` and `myAve` must be declared as type `float` variables in the calling function. This restriction is imposed because an actual argument corresponding to a formal reference parameter may be modified when the called function executes; it makes no sense to allow a function to change the value of either a constant or an expression.

EXAMPLE 6.2

Listing 6.2 shows the outline of a `main` function and a lower-level function called `test`.

The prototype for function `test` shows that `test` has two type `int` value parameters (`a` and `b`), two type `float` reference parameters (`c` and `d`), and one type `char` reference parameter (`e`). You could use any of the following function calls in the `main` function:

```
test(m + 3, 10, x, y, next);
test(m, -63, y, x, next);
test(35, m * 10, y, x, next);
```

In each call above, the first two arguments are type `int` variables, constants, or expressions; the next two arguments are type `float` variables; the last parameter is a type `char` variable (`next`). Table 6.2 shows the correspondence specified by the first argument list.

Listing 6.2 Functions `main` and `test`

```
// Functions used . . .
void test(int, int, float&, float&, char&);

int main()
{ . . .
   float x, y;
   int m;
   char next;
   . . .
}

void test(int a, int b,                      // IN
          float& c, float& d, char& e)  // OUT
{
    . . .
}  // end test
```

Table 6.2 Argument/Parameter Correspondence for `test(m + 3, 10, x, y, next)`

Actual Argument	Formal Parameter	Description
`m + 3`	`a`	`int`, value
`10`	`b`	`int`, value
`x`	`c`	`float`, reference
`y`	`d`	`float`, reference
`next`	`e`	`char`, reference

Table 6.3 Invalid Function Calls

Function Call	Error
`test(30, 10, m, 19, next);`	The constant `19` cannot correspond to reference parameter `d`. Note that the integer `m` would be converted to floating point by the compiler.
`test(m, 19, x, y);`	Not enough actual arguments.
`test(m, 10, 35, y, 'E');`	Constants `35` and `'E'` cannot correspond to reference parameters.
`test(m, 3.3, x, y, next);`	This is legal; however, the type of `3.3` is not an integer so the fractional part will be lost.
`test(30, 10, x, x + y, next);`	Expression `x + y` cannot correspond to a reference parameter.
`test(30, 10, c, d, e);`	`c`, `d`, and `e` are not declared in the `main` function.

All the function calls in Table 6.3 contain errors. The last function call points out an error often made in using functions. The last three actual argument names (`c`, `d`, `e`) are the same as their corresponding formal parameters. However, they are not declared as variables in the `main` function, so they cannot be used as actual arguments.

When writing relatively long argument lists such as those in this example, be careful not to transpose two actual arguments. If you transpose arguments, you may get a syntax error. If no syntax rule is violated, the function execution will probably generate incorrect results.

EXERCISES FOR SECTION 6.1

Self-Check

1. The function definitions below are from Example 6.2.

```
int main()
{
   float x, y;
   int m;
   char next;
   . . .
}
void test(int a, int b,                          // IN
          float& c, float& d, char& e)           // OUT
{
   . . .
} // end test
```

For each argument list below, provide a table similar to Table 6.2 if the argument list is correct. Otherwise, indicate the reason(s) the argument list would not be correct.

```
test(m, -63, y, x, next);
test(35, m * 10, y, x, next);
test(m, m, x, m, 'a');
```

2. Correct the syntax errors in the prototype parameter lists below.

```
(int&, int&; float)
(value int, char x, y)
(float x + y, int account&)
```

3. Assume that you've been given the following declarations:

```
// Functions used ...
void massage (float&, float&, int);

   // Local data ...
   const int MAX = 32767;

   float x, y, z;
   int m, n;
```

Determine which of the following function calls are invalid and indicate why. If any standard conversions are required, indicate which one(s) and specify the result of the conversion.

a. `massage(x, y, z);`

b. `massage(x, y, 8);`

c. `massage(y, x, n);`

d. `massage(m, y, n);`

e. `massage(25.0, 15, x);`

f. `massage(x, y, m+n);`

g. `massage(a, b, x);`

h. `massage(y, z, m);`

i. `massage(y+z, y-z, m);`

j. `massage(z, y, x);`

k. `massage(x, y, m, 10);`

l. `massage(z, y, MAX);`

4. Trace the execution of the program in Listing 6.1 for the data items 5, 3.5.

Programming

1. Write a function that accepts a real number as input and returns its whole and fractional parts as outputs. For example, if the input is –5.32, the function outputs should be the integer –5 and the real value –0.32.
2. Write a function that accepts an input argument consisting of two words with a space between them and returns each word through its two output parameters. All three parameters should be type `string`. If the input argument has only one word, the second output parameter should store the string `""`.
3. Write a function that accepts a real number as input and returns the largest integer that is smaller than the real number and the smallest integer that is larger than the real number. (Hint: Use `cmath` functions `floor` and `ceil`.)
4. Write a function that has two input parameters, `x` and `y`, and two output parameters, `small` and `big`. The function should return the smaller of its inputs through `small` and the larger through `big`.

6.2 Functions with Output and Input Parameters

In previous examples, we passed information to a function through its input parameters and returned a result from a function through its output parameters. In this section, we study three functions that have only output or inout (input/output) parameters.

EXAMPLE 6.3

Function `getFrac` in Listing 6.3 reads a common fraction typed in by a user and returns the numerator and denominator through two type `int` output parameters. For example, if the user types in `4 / 7`, the function returns `4` through `numerator` and `7` through `denominator`. The `main` function calls `getFrac` to read a fraction and then displays the fraction read.

This example illustrates a curious phenomenon. Notice that function `getFrac's` outputs (`numerator` and `denominator`) are actually data items that are entered at the keyboard. Although this seems strange at first, it's a fairly common occurrence in programming. It illustrates that data elements in a program system can have different purposes in different modules. For example, `numerator` and `denominator` are output parameters of function `getFrac` and return their values to `main` program variables `num` and `denom`. In the `main` function, the values of `num` and `denom` are considered problem inputs because they are entered by the program user.

Listing 6.3 Testing function getFrac

```
// File: testGetFrac
// Test function getFrac

#include <iostream>
using namespace std;

void getFrac(int&, int&);

int main()
{
   int num,            // input - fraction numerator
       denom;          // input - fraction denominator

   cout << "Enter a common fraction "
        << "as 2 integers separated by a slash: ";

   getFrac(num, denom);

   cout << "Fraction is " << num
        << " / " << denom << endl;

   return 0;
}

// Reads a fraction.
// Pre: none
// Post: Returns fraction numerator through numerator
//       Returns fraction denominator through denominator
void getFrac(int& numerator,            // OUT
             int& denominator)          // OUT
{
   char slash;                          // temporary storage for slash
   cin >> numerator >> slash >> denominator;
}
```

```
Enter a fraction as 2 integers separated by a slash: 3 / 4
Fraction is 3 / 4
```

Similarly, if we wrote a function to display the fraction, we'd have to pass the fraction's numerator and denominator values to the display function through its input parameters. The display function would then display its inputs on the screen. In this case, the numerator and denominator are problem outputs, but they are inputs to the display function.

EXAMPLE 6.4

In this example, we illustrate multiple calls to a function. Function `readFracProblem` in Listing 6.4 reads a problem involving two common fractions. A sample problem would be 2/4 + 5/6 . The function outputs would be the numerator and denominators of both fractions and the operation as a character ('+' for the sample). Function `readFracProblem` calls `getFrac` twice. Programming Project 10 at the end of this chapter describes a program that uses `readFracProblem`.

Notice that the data for a fraction numerator (or denominator) has different names in each function. We use the name `numerator` in function `getFrac`, and we use the names `num1` and `num2` in function `getFracProblem`. The parameter `numerator` in `getFrac` corresponds to parameter `num1` in the first call and parameter `num2` in the second. If we use the statement

```
readFracProblem(n1, d1, n2, d2, op);
```

to call `readFracProblem`, argument `n1` will receive the value returned through `num1` and argument `n2` will receive the value returned through `num2`. Table 6.4 summarizes the argument/parameter list correspondence.

Listing 6.4 Function `readFracProblem`

```
// Reads a fraction problem
// Pre: none
// Post: Returns first fraction through num1, denom1
//       Returns second fraction through num2, denom2
//       Returns operation through op
void readFracProblem(
   int& num1, int& denom1,          // OUT - 1st fraction
   int& num2, int& denom2,          // OUT - 2nd fraction
   char& op)                        // OUT - operator
{
    getFrac(num1, denom1);
    cin >> op;
    getFrac(num2, denom2);
}
```

Table 6.4 Argument/Parameter Correspondence for `readFracProblem(n1, d1, n2, d2, op);`

Actual Argument	`readFracProblem` Parameter	`getFrac` Parameter
`n1`	`num1`	`numerator` (in first call)
`d1`	`denom1`	`denominator` (in first call)
`n2`	`num2`	`numerator` (in second call)
`d2`	`denom2`	`denominator` (in second call)

EXAMPLE 6.5

In this example, we illustrate multiple calls to a function with inout parameters. The `main` function in Listing 6.5 reads three data values into `num1`, `num2`, and `num3` and rearranges the data so that they are in increasing sequence, with the smallest value in `num1`. The three calls to function `order` perform an operation known as **sorting**.

sorting Rearranging data values so they are in an ordered sequence.

Each time that function `order` executes, the smaller of its two argument values is stored in its first actual argument and the larger is stored in its second actual argument. Therefore, the first function call

```
order(num1, num2);        // order the data in num1 & num2
```

stores the smaller of `num1` and `num2` in `num1` and the larger in `num2`. In the sample run shown, `num1` is `7.5` and `num2` is `9.6`, so these values are not changed by the function execution. However, the function call

```
order(num1, num3);        // order the data in num1 & num3
```

switches the values of `num1` (initial value is `7.5`) and `num3` (initial value is `5.5`). Table 6.5 traces the `main` function execution.

The body of function `order` is based on the `if` statement from Example 4.13. The function heading

```
void order(float& x, float& y)  // INOUT - numbers to sort
```

identifies `x` and `y` as inout (input/output) parameters because the function uses the current actual argument values as inputs and may return new values.

During the execution of the second function call

```
order(num1, num3);        // order the data in num1 & num3
```

the formal parameter `x` contains the address of the actual argument `num1`, and the formal parameter `y` contains the address of actual argument `num3`. In testing the condition

```
(x > y)
```

the variable corresponding to `x` (`num1`) has value `7.5` and the variable corresponding to `y` (`num3`) has value `5.5` so the condition is true. Executing the first assignment statement in the true task

```
temp = x;
```

causes `7.5` to be copied into the local variable `temp`. Figure 6.3 shows a snapshot of the values in memory immediately after execution of this assignment statement.

Execution of the next assignment statement

```
x = y;
```

replaces the `7.5` in the variable corresponding to `x (num1)` with the value `(5.5)` of the variable corresponding to `y (num3)`. The final assignment statement

```
y = temp;
```

copies the contents of the temporary variable `(7.5)` into the variable corresponding to `y (num3)`. This completes the swap of values.

Listing 6.5 Function to order three numbers

```
// File: sort3Numbers.cpp
// reads three numbers and sorts them in ascending order

#include <iostream>
using namespace std;

// Functions used . . .
// Sorts a pair of numbers
void order(float&, float&);  // INOUT - numbers to sort

int main()
{

   float num1, num2, num3;   // user input - numbers to sort

   // Read 3 numbers.
   cout << "Enter 3 numbers to sort: "
   cin >> num1 >> num2 >> num3;

   // Sort them.
   order(num1, num2);           // order data in num1 & num2
   order(num1, num3);           // order data in num1 & num3
   order(num2, num3);           // order data in num2 & num3

   // Display results.
   cout << "The three numbers in order are:" << endl;
   cout << num1 << " " << num2 << " " << num3 << endl;

   return 0;
}

// Sorts a pair of numbers represented by x and y
// Pre: x and y are assigned values.
// Post: x is the smaller of the pair and y is the larger.
void order(float& x, float& y)        // INOUT - numbers to sort
{
```

(continued)

Listing 6.5 Function to order three numbers (continued)

```
      // Local data . . .
      float temp;             // storage for number in x

      // Compare x and y, exchange values if not in order.
      if (x > y)
      {                       // exchange the values in x and y
         temp = x;            // store old x in temp
         x = y;               // store old y in x
         y = temp;            // store old x in y
      }
   }  // end order

   Enter 3 numbers to sort: 7.5 9.6 5.5
   The three numbers in order are:
   5.5 7.5 9.6
```

Table 6.5 Trace of Program to Sort Three Numbers

Statement	num1	num2	num3	Effect
`cin >> num1 >> num2 >> num3;`	7.5	9.6	5.5	Enters data
`order(num1, num2);`				No change
`order(num1, num3);`	5.5		7.5	Switches num1 and num3
`order(num2, num3);`		7.5	9.6	Switches num2 and num3
`cout << num1 << num2 << num3 << endl;`				Displays `5.5, 7.5, 9.6`

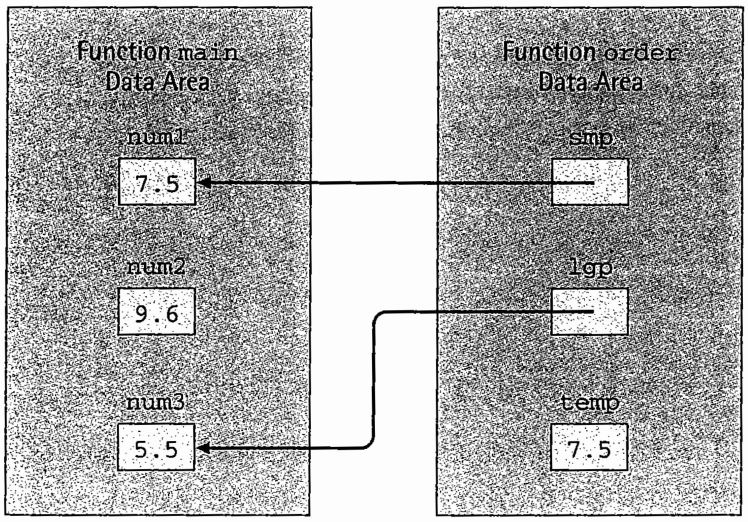


Figure 6.3 Data areas after `temp = x;` for call `order(num1, num3);`

EXERCISES FOR SECTION 6.2

Self-Check

1. Examine the functions below and answer the questions after function `sumDiff`.

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

void sumDiff(int, int);

int main()
{
   int w, x, y, z;

   x = 5;  y = 3;  z = 7;  w = 9;
   cout << "   x    y    z    w " << endl;
   cout << "   " << x << "   " << y << "   " << z
        << " " << w << endl;

   sumDiff(x, y, z, w);
   cout << " " << x << " " << y << " " << z
        << " " << w << endl;

   sumDiff(y, x, w, z);
   cout << " " << x << " " << y << " " << z
        << " " << w << endl;

   sumDiff(y, x, y, x);
   cout << " " << x << " " << y << " " << z
        << " " << w << endl;

   sumDiff(z, w, y, x);
   cout << " " << x << " " << y << " " << z
        << " " << w << endl;

   return 0;
}

void sumDiff(
  int num1, int num2,            // IN:
  int& num3, int& num4)          // INOUT:
{
   num3 = num1 + num2;
   num4 = num1 - num2;
}  // end sumDiff
```

a. Show the output displayed by function show in the form of a table of values for w, x, y, and z.

b. Briefly describe what function `sumDiff` computes. Include a description of how the input and input/output parameters to `sumDiff` are used.

c. Write the preconditions and postconditions for function `sumDiff`.

2. a. Refer to Listing 6.5 and trace the execution of the three function calls

```
order(num3, num2);
order(num3, num1);
order(num2, num1);
```

for the data sets: `15 20 4` and `4 15 20`.

b. What is the effect of this sequence of calls?

3. Write an algorithm for a function `addFrac` that computes and returns through its two outputs the sum of two fractions passed as its inputs.

Programming

1. Write the function `sumDiff` in Self-Check Exercise 1 as two separate functions `sum` and `diff` that take two input arguments. Function `sum` performs the addition, and function `diff` performs the subtraction. Rewrite the `main` function to use `sum` and `diff` to compute the same results as before.

2. Write function `doFrac` that returns a string representing a common fraction given its numerator and denominator as input arguments and also returns the decimal value of the function. If the input arguments are `1` and `4`, the string should be `"1/4"` and the decimal value should be `0.25`. Make sure you don't use integer division to calculate the decimal value (1 / 4 is 0 using integer division).

3. Write and test a function `multiplyFrac` that has four inputs representing the numerator and denominator of two fractions. The function should return the numerator and denominator of their product fraction.

4. Write and test the function for Self-Check Exercise 3.

6.3 Stepwise Design with Function

Using argument lists to pass information to and from functions improves problem-solving skills. If the solution to a subproblem cannot be written easily using just a few C++ statements, code it as a function. The case study demonstrates stepwise design of programs using functions.

case study General Sum and Average Problem

PROBLEM

You have been asked to accumulate a sum and to average a list of data values using functions. Because these tasks surface in many problems, design a general set of functions you can reuse in other programs.

ANALYSIS

Let's look again at the loop in Figure 5.2, which computed a company's total payroll. We can use a similar loop here to sum a collection of data values. To compute an average, divide a sum by the total number of items, being careful not to perform this division if the number of items is zero.

DATA REQUIREMENTS

Problem Input

`int numItems`	*// number of data items*
the data items	

Problem Output

`float sum`	*// accumulated sum of data items*
`float average`	*// average of all data items*

FORMULA

Average = Sum of data / number of data items

DESIGN

INITIAL ALGORITHM

1. Read the number of items.
2. Read the data items and compute the sum of the data (`computeSum`).
3. Compute the average of the data (`computeAve`).
4. Print the sum and the average (`printSumAve`).

The structure chart in Figure 6.4 documents the data flow between the main problem and its subproblems. We'll implement each step as a separate function; the label under a step denotes the name of the function implementing that step.

Figure 6.4 clarifies the data flow between the main function and each subordinate function. All variables whose values are set by a function are *function outputs* (indicated by an arrow pointing out of the function). All variables whose values are used in a computation but are not changed by a function are *function inputs* (indicated by an arrow pointing into the function). The role of each variable depends on its usage in a function and changes from step to step in the structure chart.

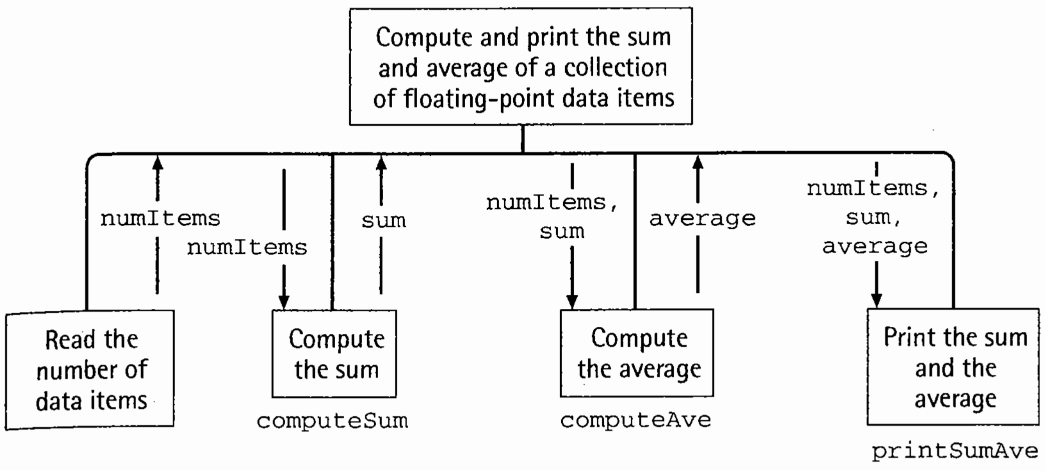

Compute and print the sum and average of a collection of floating-point data items
numItems
numItems
sum
numItems, sum
average
numItems, sum, average
Read the number of data items
Compute the sum
computeSum
Compute the average
computeAve
Print the sum and the average
printSumAve

Figure 6.4 Structure chart for general `sum` and `average` problem

Because the step "Read the number of data items" defines the value of `numItems`, this variable is an output of this step. Function `computeSum` needs the value of `numItems` to know how many data items to read and sum; consequently, `numItems` is an input to function `computeSum`. The variable `sum` is an output of function `computeSum` but is an input to functions `computeAve` and `printSumAve`. The variable `average` is an output of function `computeAve` but is an input to function `printSumAve`.

IMPLEMENTATION

Using the data flow information in the structure chart, you can write the main function even before refining the algorithm. Follow the approach described in Section 3.1 to write function `main`. Begin by converting the data requirements shown earlier into local declarations for function `main`. Declare all the variables that appear in the structure chart in the main function, because they store data passed to a function or results returned from a function. Omit the declaration for a variable that stores the individual data items, since it doesn't appear in the structure chart. However, remember to declare this variable later in the function that uses it (`computeSum`). Next, move the initial algorithm into the main function body, writing each algorithm step as a comment.

To complete the main function (see Listing 6.6), code each algorithm step *in-line* (as part of the main program code) or as a function call. Code the data entry step in-line because it consists of a simple prompt and data entry operation.

The data flow information in Figure 6.4 tells you the actual arguments to use in each function call. It also tells you the name of the main program variable that will hold the function result. For example, use the assignment statement

```
sum = computeSum(numItems);
```

Listing 6.6 Main function for general sum and average problem

```
// File: computeSumAve.cpp
// Computes and prints the sum and average of a collection of data.

#include <iostream>
using namespace std;

// Functions used . . .
// Computes sum of data
float computeSum
   (int);                    // IN - number of data items

// Computes average of data
float computeAve
   (int,                     // IN - number of data items
    float);                  // IN - sum of data items

// Prints number of items, sum, and average
void printSumAve
   (int,                     // IN - number of data items
    float,                   // IN - sum of the data
    float);                  // IN - average of the data

int main()
{
   int numItems;            // input - number of items to be added
   float sum;               // output - accumulated sum of data
   float average;           // output - average of data being processed

   // Read the number of items to process.
   cout << "Enter the number of items to process: ";
   cin >> numItems;

   // Compute the sum of the data.
   sum = computeSum(numItems);

   // Compute the average of the data.
   average = computeAve(numItems, sum);

   // Print the sum and the average.
   printSumAve(numItems, sum, average);

   return 0;
}

// Insert definitions for functions computeSum, computeAve,
// and printSumAve here.
```

to call `computeSum` and set the value of `sum`. In this call, `numItems` is passed as an input argument to function `computeSum` and the function result (determined by the `return` statement) is assigned to `sum`. Similarly, use the statement

```
average = computeAve(numItems, sum);
```

to call `computeAve`. Finally, use the statement

```
printSumAve(numItems, sum, average);
```

to call `printSumAve`, which has three input parameters and returns no result.

ANALYSIS FOR **COMPUTESUM**

In specifying the data requirements for `computeSum`, begin with the function interface information. This function is given the number of items to be processed as an input parameter (`numItems`). It's responsible for reading and computing the sum of this number of values. This sum is then returned using the `return` statement.

FUNCTION INTERFACE FOR **COMPUTESUM**

Input Parameters

`int numItems` *// number of items to process*

Output Parameters

(none)

Function Return Value

the sum (`float`) of the data items processed

Besides a variable to store the sum, `computeSum` needs two more local variables: one for storing each data item (`item`) and one for loop control (`count`).

Local Data

`float item`	*// Contains each data item as it is read in*
`float sum`	*// Used to accumulate the sum of data read in*
`int count`	*// The number of data items processed so far*

DESIGN OF **COMPUTESUM**

The loop control steps must ensure that the correct number of data items are read and summed. Since you know the number of items to sum beforehand (`numItems`), use a counting loop. Use these steps to write the algorithm for `computeSum`; Listing 6.7 shows the code for `computeSum`.

INITIAL ALGORITHM FOR **COMPUTESUM**

```
1. Initialize sum to zero.
2. For each value of count from 0, as long as count < numItems
```

Listing 6.7 Function computeSum

```
// Computes sum of data.
// Pre: numItems is assigned a value.
// Post: numItems data items read; their sum is stored in sum.
// Returns: Sum of all data items read if numItems >= 1;
//          otherwise, 0.

float computeSum
   (int numItems) // IN: number of data items
{
   // Local data . . .
   float item;        // input: contains current data item
   float sum;         // output: used to accumulate sum of data
                      //         read in

   // Read each data item and accumulate it in sum.
   sum = 0.0;
   for (int count = 0; count < numItems; count++)
   {
      cout << "Enter a number to be added: ";
      cin >> item;
      sum += item;
   }  // end for

   return sum;
}  // end computeSum
```

```
    2.1. Read in a data item.
    2.2. Add the data item to sum.
3. Return sum.
```

ANALYSIS FOR `COMPUTEAVE` AND `PRINTSUMAVE`

Both `computeAve` and `printSumAve` are relatively straightforward. We list their interface information and algorithms next. Neither function requires any local data, but both algorithms include a test of `numItems`. If `numItems` isn't positive, it makes no sense to compute or display the average of the data items.

FUNCTION INTERFACE FOR `COMPUTEAVE`

Input Parameters

`int numItems`	*// the number of data items to be processed*
`float sum`	*// the sum of all data processed*

Output Parameters

(none)

Function Return Value

the average of all the data (`float`)

DESIGN OF `computeAve`

INITIAL ALGORITHM

```
1. If the number of items is less than 1
    1.1. display "invalid number of items" message.
    1.2. return a value of 0.
2. Return the value of the sum divided by numItems.
```

FUNCTION INTERFACE FOR **PRINTSUMAVE**

Input Parameters

```
int numItems       // the number of data items to be processed
float sum          // the sum of all data processed
float average      // the average of all the data
```

Output Parameters

(none)

DESIGN OF `printSumAve`

INITIAL ALGORITHM

```
1. If the number of items is positive
    1.1. Display the number of items and the sum and average of the
         data.
   Else
    1.2. Display "invalid number of items" message.
```

IMPLEMENTATION OF `computeAve` AND `printSumAve`

The implementation of the `computeAve` and `printSumAve` functions is shown in Listings 6.8 and 6.9.

TESTING

The program that solves the general `sum` and `average` problem consists of four separate functions. In testing the complete program, you should make sure that `sum` and `average` are displayed correctly when `numItems` is positive and that a meaningful diagnostic is displayed when `numItems` is zero or negative. Listing 6.10 shows a sample run of the complete program system.

Listing 6.8 Function computeAve

```
// Computes average of data
// Pre: numItems and sum are defined; numItems must be
//      greater than 0.
// Post: If numItems is positive, the average is computed
//      as sum / numItems;
// Returns: The average if numItems is positive;
// otherwise, 0.

float computeAve
   (int numItems,          // IN: number of data items
    float sum)             // IN: sum of data
{
   // Compute the average of the data.
   if (numItems < 1)       // test for invalid input
   {
      cout << "Invalid value for numItems = " << numItems
          << endl;
      cout << "Average not computed." << endl;
      return 0.0;          // return for invalid input
   } // end if

   return sum / numItems;
   }  // end computeAve
```

Listing 6.9 Function printSumAve

```
// Prints number of items, sum, and average of data
// Pre: numItems, sum, and average are defined.
// Post: Displays numItems, sum and average if numItems > 0.

void printSumAve
   (int numItems,            // IN: number of data items
    float sum,               // IN: sum of the data
    float average)           // IN: average of the data
{
   // Display results if numItems is valid.
   if (numItems > 0)
   {
      cout << "The number of items is " << numItems << endl;
      cout << "The sum of the data is " << sum << endl;
      cout << "The average of the data is " << average << endl;
   }
```

(continued)

Listing 6.9 Function `printSumAve` (continued)

```
    else
    {
      cout << "Invalid number of items = " << numItems << endl;
      cout << "Sum and average are not defined." << endl;
      cout << "No printing done. Execution terminated." << endl;
    }  // end if
} // end printSumAve
```

Listing 6.10 Sample run of general `sum` and `average` problem

```
Enter the number of items to process: 3
Enter a number to be added: 5
Enter a number to be added: 6
Enter a number to be added: 17
The number of items is 3
The sum of the data is 28.00
The average of the data is 9.3333
```

Multiple Declarations of Identifiers in a Program

The identifiers `sum` and `numItems` are declared as variables in the `main` function and as formal parameters in the three subordinate functions. From the discussion of scope of names (see Section 3.6), you know that each of these declarations has its own scope, and the scope for each formal parameter is the function that declares it. The argument lists associate the main function variable `sum` with each of the other identifiers named `sum`. The value of variable `sum` (in function `main`) is initially defined when function `computeSum` finishes execution because variable `sum` is assigned the function result. This value is passed into function `computeAve` because variable `sum` corresponds to `computeAve's` input parameter `sum`, and so on.

To avoid the possible confusion of seeing the identifier `sum` in multiple functions, we could have introduced different names in each function (for example, `total`, `mySum`). But the program is easier to read if the name `sum` is used throughout to refer to the sum of the data values. Make sure that you remember to link these separate uses of identifier `sum` through argument list correspondence.

PROGRAM STYLE Use of Functions for Relatively Simple Algorithm Steps

All but the first step of the general sum and average problem are performed by separate functions. Even though it was relatively easy to implement the step for computing the average, we used a function (computeAve) rather than writing the code inline in the main function. We want to encourage the use of separate functions even for relatively easy-to-implement algorithm steps. This helps you keep the details of these steps separate and hidden and makes the program system easier to debug, test, and even modify at some future date. Also, you may be able to reuse this function later. From this point on, your main functions should consist primarily of a sequence of function calls.

PROGRAM STYLE Cohesive Functions

Function computeSum only computes the sum. It doesn't read in the number of data items (Step 1 of the main function), nor does it display the sum of the data (role of function printSumAve).

Functions that perform a single operation are called *functionally cohesive.* It's good programming style to write such single-purpose, highly cohesive functions, as this helps to keep each function relatively compact and easy to read, write, and debug. You can determine whether a function is highly cohesive from the comment describing what the function does. If the comment consists of a short sentence or phrase with no connectives such as "and" or "or," then the function should be highly cohesive. If more than one or two connectives or separate sentences are needed to describe the purpose of a function, this may be a hint that the function is doing too many things and should be further decomposed into subfunctions.

EXERCISES FOR SECTION 6.3

Self-Check

1. Function computeAve returns a single value using a return statement. Rewrite this function to return this result through an output parameter. Why is this not as good a solution as the one used in the case study?
2. Draw the before and after data areas for the main function and revised computeAve (see Self-Check Exercise 1) assuming computeAve is called with sum equal to 150.0 and numItems equal to 8.
3. Draw the main function and printSumAve data areas given the data value assumptions in Self-Check Exercise 2.

4. Consider the three functions `computeSum`, `computeAve`, and `printSumAve` as though the code to validate the value of the parameter `numItems` had been omitted. For each of these functions, describe what would happen now that the function would be allowed to proceed with its work even if `numItems` were zero or negative.
5. Design an algorithm for `readNumItems` that uses a loop to ensure that the user enters a positive value. The loop should continue reading numbers until the user enters a positive number.

Programming

1. Implement the solution to Self-Check Exercise 5.

6.4 Using Objects with Functions

C++ provides two ways that you can use functions to modify objects.

1. You can use dot notation to apply a member function to an object (also called "sending the object a message"). The member function may modify one or more data attributes of the object to which it is applied.
2. You can pass an object as an argument to a function.

We illustrate both approaches in the next example.

EXAMPLE 6.6

Listing 6.11 shows function `moneyToNumberString`, which has a single inout argument of type `string`. This function removes a dollar sign and any commas that appear in its argument string. For example, if the argument string passed to the function is `"-$5,405,123.65"`, the argument string will be changed to `"-5405123.65"` by the function execution.

The `if` statement checks for the presence of a `$`. If the argument string begins with a `$`, the true task removes it. The `else` clause removes only the `$` from a string that begins with the substring `"-$"`. Next, the `while` loop removes all commas from the argument string, starting with the leftmost one. If a comma is found, the loop repetition condition is true, so the comma is removed and the next one is searched for. The loop repetition condition fails when there are no commas left in the argument string. We will say more about objects as arguments in Chapter 10.

Listing 6.11 Testing function `moneyToNumberString`

```
// File: moneyToNumberTest.cpp
// Tests function moneyToNumberString
#include <string>
#include <iostream>
using namespace std;

// Function prototype
void moneyToNumberString(string&);

int main()
{
   string mString;              // input - a"money"string

   cout << "Enter a dollar amount with $ and commas: ";
   cin >> mString;

   moneyToNumberString(mString);

   cout << "The dollar amount as a number is " << mString
        << endl;

   return 0;
}

// Removes the $ and commas from a money string.
// Pre: moneyString is defined and may contain commas and
//      begin with $ or -$.
// Post: $ and all commas are removed from moneyString.
void moneyToNumberString
  (string& moneyString) // INOUT - string with possible $ and commas
{
   // Local data . . .
   int posComma; // position of next comma

   // Remove $ from moneyString
   if (moneyString.at(0) == '$')          // Starts with $ ?
      moneyString.erase(0, 1);            // Remove $
   else if (moneyString.find("-$") == 0)  // Starts with -$ ?
      moneyString.erase(1, 1);            // Remove $

   // Remove all commas
   posComma = moneyString.find(",");      // Find first ,
```

(continued)

Listing 6.11 Testing function `moneyToNumberString` (continued)

```
      while (posComma >= 0 &&
             posComma < moneyString.length())    // Is posComma valid ?
      {
         moneyString.erase(posComma, 1);         // Remove ,
         posComma = moneyString.find(",");       // Find next ,
      }
   }  // end moneyToNumberString

   Enter a dollar amount with $ and commas: -$5,405,123.65
   The dollar amount as a number is -5405123.65
```

EXERCISES FOR SECTION 6.4

Self-Check

1. Trace the execution of function `moneyToNumberString` for the data value shown in Listing 6.11. Show all values assigned to `posComma`.

Programming

1. Write a function `compress` that removes all blanks from its string argument. If the argument is `"  this  Is  One"` it should be changed to `"thisIsOne"`.
2. Write a function `doRemove` that removes the first occurrence of a substring (an input argument) from a second argument string (input/output).

6.5 Debugging and Testing a Program System

As the number of statements in a program system grows, the possibility of error also increases. If we keep each function to a manageable size, the likelihood of error increases much more slowly. It's also easier to read and test each function.

Just as you can simplify the overall programming process by writing a large program as a set of independent functions, you can simplify testing and debugging if you test in stages as the program evolves. Two kinds of testing are used: top-down testing and bottom-up testing. You should use a combination of these methods to test a program and its functions.

Top-Down Testing and Stubs

Though a single programmer or a programming team may be developing a program system, not all functions will be ready at the same time. It's still possible to test the overall flow of control between the main program and its level-1 functions and to test and debug the level-1 functions that are complete. Testing the flow of control between a `main` function and its subordinate functions is called **top-down testing**.

top-down testing
The process of testing flow of control between a `main` function and its subordinate functions.

stub
A function with a heading and a minimal body used in testing flow of control.

Because the `main` function calls all level-1 functions, we need for all functions that are not yet coded a substitute called a **stub**—a function heading followed by a minimal body, which should display a message identifying the function being executed and should assign simple values to any outputs. Listing 6.12 shows a stub for function `computeSum` that could be used in a test of the `main` function in Listing 6.6. The stub arbitrarily returns a value of `100.0`, which is reasonable data for the remaining functions to process. Examining the program output tells us whether the `main` function calls its level-1 functions in the required sequence and whether data flows correctly between the `main` function and its level-1 functions.

Bottom-Up Testing and Drivers

unit test
A test of an individual function.

When a function is completed, it can be substituted for its stub in the program. However, we often perform a preliminary **unit test** of a new function before substitution because it's easier to locate and correct errors when dealing with a single function rather than with a complete program system. We can perform such a unit test by writing a short driver function to call it.

It isn't a good idea to spend a lot of time creating an elegant driver function, because it will be discarded as soon as the new function is tested. A driver function should contain only the declarations and executable statements necessary to test a single function. A driver should begin by reading or assigning values to all input arguments and to input/output arguments.

Listing 6.12 Stub for function computeSum

```
// Computes sum of data - stub
// Pre: numItems is assigned a value.
// Post: numItems data items read; their sum is stored in sum.
// Returns: Sum of all data items read if numItems >= 1;
//    otherwise 0.0.
float computeSum
  (int numItems)      // IN - number of data items

{
    cout << "Function computeSum entered" << endl;
    return 100.0;
}  // end computeSum stub
```

Next comes the call to the function being tested. After calling the function, the driver should display the function results. Listing 6.13 shows a driver for the completed function `computeSum`. Since we have no need to save the function result, we inserted the function call directly in the output statement.

Once you're confident that a function works properly, you can substitute it for its stub in the program system. The process of separately testing individual functions before inserting them in a program system is called **bottom-up testing**. Tests of the entire system are called **system integration tests**.

bottom-up testing
The process of separately testing individual functions of a program system.

system integration tests
Testing a system after replacing all its stubs with functions that have been pretested.

By following a combination of top-down and bottom-up testing, a programming team can be fairly confident that the complete function system will be relatively free of errors when it's finally put together. Consequently, the final debugging sessions should proceed quickly and smoothly.

Debugging Tips for Program Systems

The following suggestions will prove helpful when debugging a program system.

1. Carefully document each function parameter and local variable using comments as you write the code. Also describe the function's purpose using comments.
2. Create a trace of execution by displaying the function name as you enter it.
3. Trace or display the values of all input and input/output parameters upon entry to a function. Check that these values make sense.

Listing 6.13 A driver to test `computeSum`

```
int main()
{

    int n;
    // Keep calling computeSum and displaying the result.
    do
    {
       cout << "Enter number of items or 0 to quit: ";
       cin >> n;
       cout << "The sum is " << computeSum(n) << endl;
    } while (n != 0);

    return 0;
}   // end driver
```

4. Trace or display the values of all function outputs after returning from a function. Verify that these values are correct by hand computation. Make sure you declare all input/output and output parameters as reference parameters (using the & symbol).
5. Make sure that the function stub assigns a value to each output parameter.

You should plan for debugging as you write each function rather than adding debugging statements later. Include any output statements that you might need to help determine that the function is working. When you're satisfied that the function works as desired, you can remove these debugging statements. In Section 5.9, it was suggested that you turn them into comments.

Another way to "turn debugging off and on" is to introduce a type bool flag called DEBUG. First, make each debugging statement a dependent statement that executes only if DEBUG is true.

```
if (DEBUG)
   cout << "***** score is " << score << " and sum is "
          << sum << endl;
```

To "turn on debugging," insert the statement

```
const bool DEBUG = true;
```

before your main function, making it visible in all your functions. Setting the debugging flag to true causes the debugging statements to execute. To "turn off debugging," set the constant DEBUG to false instead of true.

Identifier Scope and Watch Window Variables

You can use a debugger to trace values passed into a function's input arguments and to trace the values returned by a function. The values displayed in the Watch window are determined by the normal scope rules for identifiers. Consequently, a function's local variables and formal parameters will be considered undefined until that function begins execution. Upon exit from the function, its local variables and formal parameters will again become undefined.

Black-Box Versus White-Box Testing

black-box (or specification-based) testing A testing process that assumes the tester has no knowledge of the code; the tester must compare a function or the system's performance with its specification.

There are two basic ways to test a completed function or system: (1) black-box testing and (2) white-box testing. In **black-box** (or **specification-based**) **testing**, we assume that the program tester has no information about the code inside the function or system. The tester's job is to verify that the function or system meets its specifications. For For each function, the tester must ensure that its postconditions are satisfied whenever its preconditions are met.

Because the tester cannot look inside the function or system, the tester must prepare sufficient sets of test data to verify that the system output is correct for all valid input values. The tester should also find out whether the function or system will crash for invalid data. The tester should especially check the *boundaries* of the system, or particular values where the system performance changes. For example, a boundary for a payroll program would be the value of hours worked that triggers overtime pay. Black-box testing is most often done by a special testing team or by program users.

In **white-box** (or **glass-box**) **testing**, the tester has full knowledge of the code for the function or system and must ensure that each section of code has been thoroughly tested. For a selection statement (`if` or `switch`), this means checking all possible paths through the selection statement. The tester must determine that the correct path is chosen for all possible values of the selection variable, taking special care at the boundary values where the path changes.

white box (or glass box) testing
A testing process that assumes the tester knows how the system is coded and requires checking all possible execution paths.

For a repetition statement, the tester must make sure that the loop always performs the correct number of iterations and that the number of iterations isn't off by one. Also, the tester should verify that the computations inside the loop are correct at the boundaries—that is, for the initial and final values of the loop control variable. Finally, the tester should make sure that the function or system still meets its specification when a loop executes zero times and that under no circumstances can the loop execute forever.

EXERCISES FOR SECTION 6.5

Self-Check

1. Show the output you'd expect to see for the `sum` and `average` program (Listing 6.6) using the stub in Listing 6.12 when `numItems` is `10`.
2. Write a driver program to test function `computeAve`.
3. Explain the difference between black-box and white-box testing, between bottom-up and top-down testing, between unit tests and system integration tests.
4. Explain the difference between stubs and drivers. Which would be more useful in bottom-up testing? Which would be more useful in top-down testing?
5. Compare function prototypes and stubs. How are they similar, and how are they different?

Programming

1. Write a driver function to test function `computeAve`.
2. Rewrite the driver in Listing 6.13 using a `while` statement.

6.6 Recursive Functions (Optional)

C++ allows a function to call itself. A function that calls itself is a *recursive function*. Sometimes it's simpler to implement a repeated operation using recursion instead of iteration. A recursive function calls itself repeatedly, but with different argument values for each call.

Just as we did for a loop, we need to identify a situation (called a **stopping case**) that stops the recursion; otherwise, the function will call itself forever. Usually a recursive function has the following form:

stopping case
An alternative task in a recursive function that leads to no further recursive calls.

Template for a Recursive Function

```
1. If the stopping case is reached
       1.1. Return a value for the stopping case
   Else
       1.2. Return a value computed by calling the function again with
            different arguments.
```

The `if` statement tests whether the stopping case has been reached. When it is reached, the recursion stops and the function returns a value to the caller. If the stopping case isn't reached, the function calls itself again with different arguments. The arguments in successive calls should bring us closer and closer to reaching the stopping case.

In this section we describe a recursive function that returns an integer value representing the factorial of its argument. The *factorial of n* is the product of all positive integers less than or equal to n and is written in mathematics as $n!$. For example, 4! is the product $4 \times 3 \times 2 \times 1$ or 24. We provide a recursive definition for $n!$ next.

$$n! = 1 \qquad \text{for } n = 0 \text{ or } 1$$
$$n! = n \times (n-1)! \qquad \text{for } n > 1$$

We can translate this definition into pseudocode using the template shown earlier:

```
1. If $n$ is 0 or 1
       1.1. Return 1
   Else
       1.2. Return $n \times (n-1)!$
```

Listing 6.14 shows function `factorial` rewritten as a recursive function. The stopping case is reached when n is less than or equal to 1. When n is greater than 1, the statement

```
return n * factorial(n-1);
```

executes, which is the C++ form of the second formula. The expression part of this statement contains a valid function call, `factorial(n-1)`, which calls function `factorial` with an argument that is 1 less than the current argument. This function call is a *recursive call.* If the argument in the initial call to `factorial` is `3`, the following chain of recursive calls occurs:

```
factorial(3)→3 * factorial(2)→3 * (2 * factorial(1))
```

In the last call above, `n` is equal to `1`, so the statement

```
return 1;
```

executes, stopping the chain of recursive calls.

When it finishes the last function call, C++ must return a value from each recursive call, starting with the last one. This process is called **unwinding the recursion**. The last call was `factorial(1)` and it returns a value of 1. To find the value returned by each call for `n` greater than 1, multiply `n` by the value returned from `factorial(n-1)`. Therefore, the value returned from `factorial(2)` is `2 *` the value returned from `factorial(1)` or `2`; the value returned from `factorial(3)` is `3 *` the value returned from `factorial(2)` or `6` (see Figure 6.5).

unwinding the recursion
The process of returning a value from each recursive call.

For comparison purposes, Listing 6.15 shows an iterative factorial function that uses a loop to accumulate partial products in local variable `productSoFar`. The `for` statement repeats the multiplication step when `n` is greater than `1`. If `n` is `0` or `1`, the `for` loop body doesn't execute, so `productSoFar` retains its initial value of 1. After loop exit, the last value of `productSoFar` is returned as the function result.

Listing 6.14 Recursive function factorial

```
// Pre: n is >= 0
// Returns: The product 1 * 2 * 3 * . . . * n for n > 1;
//          otherwise 1.

int factorial(int n)
{
   if (n <= 1)
      return 1;
   else
      return n * factorial(n-1);
}
```

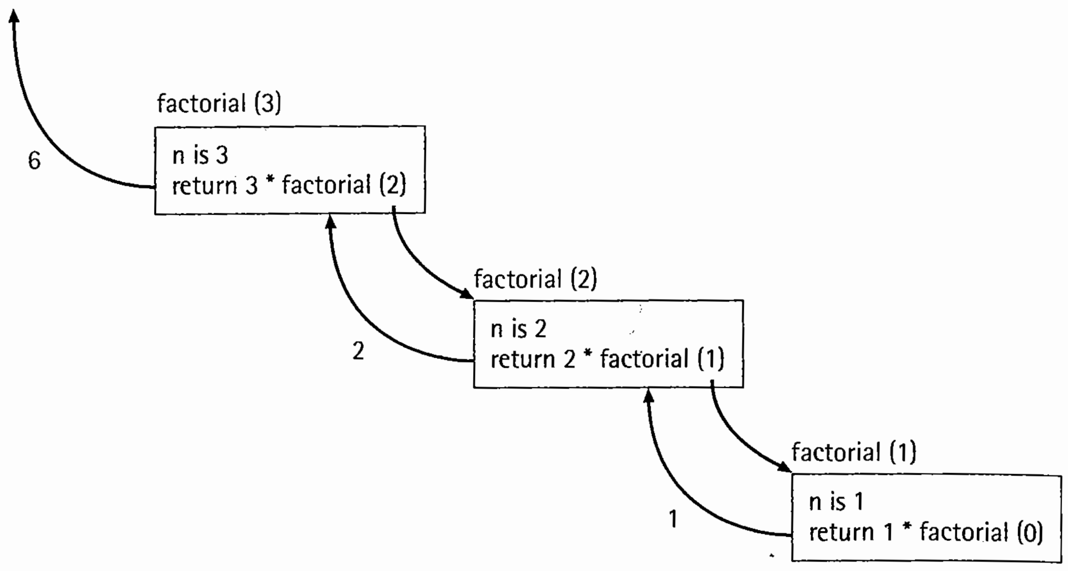


Figure 6.5 Unwinding the recursion

Listing 6.15 Iterative factorial function

```
// Pre: n is >= 0
// Returns: The product 1 * 2 * 3 * . . . * n for n > 1;
//          otherwise 1.
int factorial(int n)
{

   int productSoFar;        // output: accumulated product

   // Initialize accumulated product
   productSoFar = 1;

   // Perform the repeated multiplication for i > 1.
   for (int i = n; i > 1; i--)
      productSoFar = productSoFar * i;

      return productSoFar;
}
```

EXERCISES FOR SECTION 6.6

Self-Check

1. Trace the execution of function `factorial` when `n` is 5.
2. Show the chain of recursive calls to function `mystery` when `m` is 5 and `n` is 3. What do you think `mystery` does?

```
int mystery(int m, int n)
{
   if (n == 1) then
      return m;
   else
      return m * mystery(m, n-1);
}
```

3. Complete the recursive definition of function `gcd` which computes the greatest common divisor of two integers (`gcd(20, 15)` is 5).

```
// precondition: m is greater than or equal to n
int gcd(int m, int n)
{
   if (n > m)
      return gcd(__, __);        //exchange arguments
   else if (m % n == 0)
      return __;                 //the result is known
   else
      return gcd(__, __);        //find gcd of n and
                                 //remainder of m / n
}
```

Programming

1. Write a recursive function that, given an input value of `n`, computes:

   `n + n-1 + . . . + 2 + 1.`

2. Write a function `c(n, r)` that returns the number of different ways *r* items can be selected from a group of *n* items. The mathematical formula for `c(n, r)` follows. Test `c(n, r)` using both the recursive and the nonrecursive versions of function `factorial`.

$$c(n, r) = \frac{n!}{r!(n - r)!}$$

6.7 Common Programming Errors

Many opportunities for error arise when you use functions with parameter lists, so be extremely careful. Proper use of parameters is difficult for beginning programmers to master, but it's an essential skill. One obvious pitfall is not ensuring that the actual argument list has the same number of items as the formal parameter list. Each actual input argument must be of a type that can be assigned to its corresponding formal parameter. An actual output argument or inout argument should be the same data type as the corresponding formal parameter. Let's now look at two specific errors that can occur with reference parameters.

- *Parameter inconsistencies with call-by-reference parameters:* In Chapter 3, we indicated that argument/parameter inconsistencies involving character, integer, and floating-point data usually don't cause even a warning message from the compiler. However, such inconsistencies will cause a warning message if they occur when using reference parameters.

  For example, if `initiate` has the prototype

  ```
  void initiate(char &);
  ```

  and `sum` is type `float`, the call

  ```
  initiate(sum);
  ```

  will result in a compiler warning such as

  ```
  "Temporary used for parameter 1 in call to 'initiate
  (char &)'"
  ```

  Even though this is only a warning, it does indicate a possible error. Check the function call carefully to ensure that you're passing the correct argument.

- *Forgetting to use an `&` with call-by-reference parameters:* Make sure you remember to use an `&` in the function prototype and function header (but not the call) before each output or inout parameter. Forgetting the `&` doesn't cause an error message; however, C++ will treat the formal parameter as call-by-value instead of call-by-reference. Therefore, any change made to the parameter by the called function will not be returned to the calling function.

Chapter Review

1. Functions provide a mechanism for separating and hiding the details for each component of a program system. The use of functions makes it possible to carry this separation through from the problem analysis and program design stages to the implementation of a program in C++.
2. Parameters enable a programmer to pass data to functions and to return multiple results from functions. The parameter list provides a highly visible communication path between a function and its calling program. Using parameters enables a function to process different data each time it executes, thereby making it easier to reuse the function in other programs.
3. Parameters may be used for input to a function, for output or sending back results, and for both input and output. An input parameter is used only for passing data into a function. The actual argument corresponding

to an input parameter may be an expression or a constant. The parameter's declared type should be the same as the type of the argument.

4. C++ uses call-by-value for input parameters and call-by-reference for output and inout (input/output) parameters. For call-by-value, the actual argument's value is passed to the function when it's called; for call-by-reference, the actual argument's address is passed to the function when it's called. C++ uses the symbol `&` after the parameter type to indicate a reference parameter in a function prototype and in a function definition but not in a function call.

5. Output and inout parameters must be able to access variables in the calling function's data area, so the actual argument corresponding to an output or inout parameter must be a variable. The parameter's declared type should be the same as the type of the data.

Summary of New C++ Constructs

Construct	Effect
Function Prototype `void doIt (float, char, float&, char&);`	Prototype for a function having two input and two output parameters.

Function Definition
`void doIt`
`  (float x,        // IN`
`   char op,        // IN`
`   float& y,       // OUT`
`   char& sign)     // OUT`
`{`
`    switch(op)`
`    {`
`       case '+':`
`          y = x + x;`
`          break;`
`       case '*':`
`          y = x * x;`
`          break;`
`    }`
`    if (x >= 0.0)`
`       sign = '+';`
`    else`
`       sign = '-';`
`} // end doIt`
	x and op are input parameters that contain valid values passed in from the calling function. The memory cells corresponding to the output parameters y and `sign` are undefined upon entrance to function `doIt`, but they contain valid values that are passed back to the calling function upon exit.

Function Call Statement `doIt(-5.0, '*', p, mySign);`	Calls function `doIt`. `-5.0` is passed into x and `'*'` is passed into op. `25.0` is returned to p, and `'-'` is returned to `mySign`.

Quick-Check Exercises

1. Actual arguments appear in a function ________; formal parameters appear in a function ________ and function ________. Formal parameters are optional in a function ________.

2. Constants and expressions may be used as actual arguments corresponding to formal parameters that are ________ parameters.

3. In a function header, ________ parameters used for function output are designated by using the symbol ________ placed ________ the parameter type.

4. A ________ must be used as an actual argument corresponding to a call-by-reference formal parameter. You can use an ________ as an actual argument corresponding to a call-by-value parameter.

5. For ________ parameters, the argument's address is stored in the called function data area for the corresponding formal parameter. For ________ parameters, the argument's value is stored in the called function data area for the corresponding formal parameter.

6. If a function returns all its results (type `double`) through its formal parameters, what would the type of the function be?

7. Is a driver or a stub used to allow an upper-level function to be tested before all lower-level functions are complete?

8. Does a testing team use white-box or black-box testing?

9. What output lines are displayed by the program below?

```
void silly(int);

int main()
{
    int x,  y;
    // Do something silly.
    x = 8;  y = 5;
    silly(x);
    cout << x << ", " << y << endl;
    silly(y); // values here
    cout << x << ", " << y << endl;

    return 0;

}

void silly(int x)
{
    int y;

    y = x;
    x *= 2;
    cout << x << ", " << y << endl;
} // end silly
```

10. Answer Quick-Check Exercise 9 if `silly`'s parameter is a reference parameter.
11. In what ways can a function return values to its caller?

Review Questions

1. Write the prototype for a function named `pass` that has two integer parameters. The first parameter should be a value parameter and the second a reference parameter.
2. Write a function called `letterGrade` with a type `int` input parameter called `score`. The function should return through an output parameter (grade) the appropriate letter grade using a straight scale (90 to 100 is an A; 80 to 89 is a B; 70 to 79 is a C; 60 to 69 is a D; and 0 to 59 is an F). Return through a second output parameter a plus symbol if the student just missed the next higher grade by one or two points (for example, if a student got an 88 or 89, the second output parameter should be '+'). If the student just made the grade (for example, got a grade of 80 or 81), return a '–' for the second output parameter. The second output parameter should be blank if the student did not just miss or just make the grade.
3. Would you write a function that computes a single numeric value as a non-void function that returns a result through a return statement or as a void function with one output parameter? Explain your choice.
4. Explain the allocation of memory cells when a function is called. What is stored in the called function's data area for an input parameter? What is stored for an output parameter? What is stored for an inout parameter?
5. Write a driver program to test the function that you wrote for Review Question 2.
6. What are the two kinds of reference parameters? Explain the differences between them.
7. Sketch the data areas of functions `main` and `silly` as they appear immediately before the return from the first call to `silly` in Quick-Check Exercise 9 as modified in 10.
8. Present arguments against these statements:
   a. It's foolish to use function subprograms because a program written with functions has many more lines than the same program written without functions.
   b. The use of function subprograms leads to more errors because of mistakes in using argument lists.

Programming Projects

1. A hospital supply company wants to market a program to assist with the calculation of intravenous rates. Design and implement a program that interacts with the user as follows:

```
INTRAVENOUS RATE ASSISTANT

Enter the number of the problem you wish to solve.
    GIVEN A MEDICAL ORDER IN            CALCULATE RATE IN
(1) ml/hr & tubing drop factor            drops / min
(2) 1 L for n hr                          ml / hr
(3) mg/kg/hr & concentration in mg/ml     ml / hr
(4) units/hr & concentration in units/ml  ml / hr
(5) QUIT

Problem=> 1
Enter rate in ml/hr=> 150
Enter tubing's drop factor(drops/ml)=> 15
The drop rate per minute is 38.

Enter the number of the problem you wish to solve.
    GIVEN A MEDICAL ORDER IN            CALCULATE RATE IN
(1) ml/hr & tubing drop factor            drops / min
(2) 1 L for n hr                          ml / hr
(3) mg/kg/hr & concentration in mg/ml     ml / hr
(4) units/hr & concentration in units/ml  ml / hr
(5) QUIT

Problem=> 2
Enter number of hours=> 8
The rate in milliliters per hour is 125.

Enter the number of the problem you wish to solve.
    GIVEN A MEDICAL ORDER IN            CALCULATE RATE IN
(1) ml/hr & tubing drop factor            drops / min
(2) 1 L for n hr                          ml / hr
(3) mg/kg/hr & concentration in mg/ml     ml / hr
(4) units/hr & concentration in units/ml  ml / hr
(5) QUIT

Problem=> 3
Enter rate in mg/kg/hr=> 0.6
Enter patient weight in kg=> 70
Enter concentration in mg/ml=> 1
The rate in milliliters per hour is 42.
```

```
Enter the number of the problem you wish to solve.
     GIVEN A MEDICAL ORDER IN            CALCULATE RATE IN
(1) ml/hr & tubing drop factor             drops / min
(2) 1 L for n hr                           ml / hr
(3) mg/kg/hr & concentration in mg/ml      ml / hr
(4) units/hr & concentration in units/ml   ml / hr
(5) QUIT

Problem=> 4
Enter rate in units/hr=> 1000
Enter concentration in units/ml=> 25
The rate in milliliters per hour is 40.

Enter the number of the problem you wish to solve.
    GIVEN A MEDICAL ORDER IN             CALCULATE RATE IN
(1) ml/hr & tubing drop factor           drops / min
(2) 1 L for n hr                         ml / hr
(3) mg/kg/hr & concentration in mg/ml    ml / hr
(4) units/hr & concentration in units/ml ml / hr
(5) QUIT

Problem=> 5
```

Implement the following functions:

`getProblem`—Displays the user menu, then inputs and returns as the function value the problem number selected.

`getRateDropFactor`—Prompts the user to enter the data required for problem 1, and sends this data back to the calling module via output parameters.

`getKgRateConc`—Prompts the user to enter the data required for problem 3, and sends this data back to the calling module via output parameters.

`getUnitsConc`—Prompts the user to enter the data required for problem 4, and sends this data back to the calling module via output parameters.

`figDropsMin`—Takes rate and drop factor as input parameters and returns drops/min (rounded to the nearest whole drop) as function value.

`figMlHr`—Takes as an input parameter the number of hours over which one liter is to be delivered and returns ml/hr (rounded) as function value.

`byWeight`—Takes as input parameters rate in mg/kg/hr, patient weight in kg, and concentration of drug in mg/ml and returns ml/hr (rounded) as function value.

`byUnits`—Takes as input parameters rate in units/hr and concentration in units/ml, and returns ml/hr(rounded) as function value.

(Hint: Use a sentinel-controlled loop. Call `getProblem` once before the loop to initialize the problem number and once again at the end of the loop body to update it.)

2. The table below summarizes three commonly used mathematical models of nonvertical straight lines.

Mode	Equation	Given
Two-point form	$m = \dfrac{y_2 - y_1}{x_2 - x_1}$	$(x_1, y_1), (x_2, y_2)$
Point-slope form	$y - y_1 = m(x - x_1)$	$m, (x_1, y_1)$
Slope-intercept form	$y = mx + b$	m, b

Design and implement a program that permits the user to convert either two-point form or point-slope form into slope-intercept form. Your program should interact with the user as follows:

```
Select the form that you would like to convert to slope-
intercept form:
1) Two-point form (you know two points on the line)
2) Point-slope form (you know the line's slope and one point)
=> 2

Enter the slope=> 4.2
Enter the x-y coordinates of the point separated by a
space=> 1 1

Point-slope form
  y - 1.00 = 4.20(x - 1.00)

Slope-intercept form
  y = 4.20x - 3.20

Do another conversion (Y or N) => Y

Select the form that you would like to convert to slope-
intercept form:
1) Two-point form (you know two points on the line)
2) Point-slope form (you know the line's slope and one
point)
=> 1

Enter the x-y coordinates of the first point separated by a
space=> 4 3
```

```
Enter the x-y coordinates of the second point separated by a
space=> -2 1

Two-point form
       (1.00 - 3.00)
   m = -------------
      (-2.00 - 4.00)

Slope-intercept form
   y = 0.33x + 1.66

Do another conversion (Y or N)=> N
```

Implement the following functions:

`getProblem`—Displays the user menu, then inputs and returns as the function value the problem number selected.

`get2Pt`—Prompts the user for the x-y coordinates of both points, inputs the four coordinates, and returns them to the calling function through output parameters.

`getPtSlope`—Prompts the user for the slope and x-y coordinates of the point, inputs the three values, and returns them to the calling function through output parameters.

`slopeIntcptFrom2Pt`—Takes four input parameters, the x-y coordinates of two points, and returns through output parameters the slope (*m*) and y-intercept (*b*).

`intcptFromPtSlope`—Takes three input parameters, the x-y coordinates of one point and the slope, and returns as the function value the y-intercept.

`display2Pt`—Takes four input parameters, the x-y coordinates of two points, and displays the two-point line equation with a heading.

`displayPtSlope`—Takes three input parameters, the x-y coordinates of one point and the slope, and displays the point-slope line equation with a heading.

`displaySlopeIntcpt`—Takes two input parameters, the slope and y-intercept, and displays the slope-intercept line equation with a heading.

3. The trustees of a small college are considering voting a pay raise for their 12 faculty members. They want to grant a 2.5-percent pay raise; however, before doing so, they want to know how much this will cost. Write a program that will print the pay raise for each faculty member and the total amount of the raises. Also, print the total faculty payroll before and after the raise. Test your function for the salaries:

\$52,500	\$64,029.50	\$56,000	\$53,250
\$65,500	\$42,800	\$45,000.50	\$68,900
\$53,780	\$77,300	\$84,120.25	\$64,100

4. Redo Programming Project 3 assuming that faculty members earning less than \$50,000 receive a 3-percent raise, those earning more than \$70,000 receive a 3.5-percent raise, and all others receive a 2.5-percent raise. For each faculty member, print the raise percentage as well as the amount.

5. Patients required to take many kinds of medication often have difficulty in remembering when to take their medicine. Given the following set of medications, write a function that prints an hourly table indicating what medication to take at any given hour. Use a counter variable `clock` to go through a 24-hour day. Print the table based on the following prescriptions:

Medication	Frequency
Iron pill	0800, 1200, 1800
Antibiotic	Every 4 hours starting at 0400
Aspirin	0800, 2100
Decongestant	1100, 2000

6. After studying gross annual revenues of Broadway shows over a 20-year period, you model the revenue as a function of time:

$$R(t) = 203.265 \times (1.071)^t$$

where R is in millions of dollars and t is the years since 1984. Create the following C++ functions to implement this model:

`revenue`—calculates and returns R for an input parameter of t.

`predict`—predicts the year in which revenues (in millions) will first equal or exceed the value of the input parameter. For example, `predict(200)` would return 1984.

Write a main function that calls `predict` to determine when revenues will likely exceed \$1 trillion (that is, 1,000 million). Then display a table of estimated revenues (in millions of dollars) for all the years from 1984 through the year when revenues should exceed \$1 trillion. Round revenue estimates to three decimal places.

7. The square root of a number N can be approximated by repeated calculation using the formula

$$NG = 0.5(LG + N/LG)$$

where NG stands for *next guess* and LG stands for *last guess*. Write a function that implements this computation. The first parameter will be a positive real number, the second will be an initial guess of the square root of that number, and the third will be the computed result.

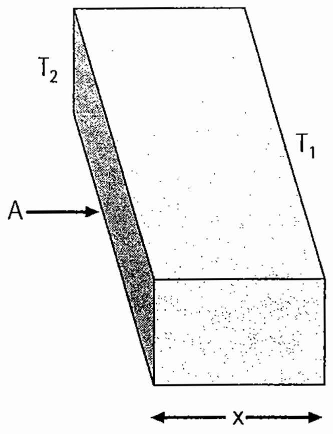
The initial guess will be the starting value of *LG*. The function will compute a value for *NG* using the formula above. To control the computation, we can use a `while` loop. Each time through the loop, the difference between *NG* and *LG* is checked to see whether these two guesses are almost identical. If so, the function returns *NG* as the square root; otherwise, the next guess (*NG*) becomes the last guess (*LG*) and the process is repeated (that is, another value is computed for *NG*, the difference is checked, and so forth).

For this problem, the loop should be repeated until the magnitude of the difference between *LG* and *NG* is less than 0.005. Use an initial guess of 1.0 and test the function for the numbers 4.0, 120.5, 88.0, 36.01, and 10,000.0.

8. Develop a collection of functions to solve simple conduction problems using various forms of the formula

$$H = \frac{kA(T_2 - T_1)}{X}$$

where H is the rate of heat transfer in watts, k is the coefficient of thermal conductivity for the particular substance, A is the cross-sectional area in m^2 (square meters), T_2 and T_1 are the Kelvin temperatures on the two sides of the conductor, and X is the thickness of the conductor in meters.



Define a function for each variable in the formula. For example, function `calcH` would compute the rate of heat transfer, `calcK` would figure the coefficient of thermal conductivity, `calcA` would find the cross-sectional area, and so on.

Develop a driver function that interacts with the user in the following way:

```
Respond to the prompts with the data known. For the
unknown quantity, enter -999
```

```
Rate of heat transfer (watts) : 755.0
Coefficient of thermal conductivity (W/m-K) : 0.8
Cross-sectional area of conductor (m^2) : 0.12
Temperature on one side (K) : 298
Temperature on other side (K) : -999
Thickness of conductor (m) : 0.003
```

$$H = \frac{kA(T_2 - T_1)}{X}$$

```
Temperature on the other side is 274 K.
H = 755.0 W                  T2 = 298 K
k = 0.800 W/m-K              T1 = 274 K
A = 0.120 m^2                X = 0.0003 m
```

9. Write a program to model a simple calculator. Each data line should consist of the next operation to be performed from the list below and the right operand. Assume the left operand is the accumulator value (initial value of 0). You need a function `scan_data` with two output parameters that returns the operator and right operand scanned from a data line. You need a function `do_next_op` that performs the required operation. `do_next_op` has two input parameters (the operator and operand) and one input/output parameter (the accumulator). The valid operators are:

+	add
−	subtract
*	multiply
/	divide
^	power (raise left operand to power of right operand)
q	quit

Your calculator should display the accumulator value after each operation. A sample run follows.

```
+ 5.0
result so far is 5.0
^2
result so far is 25.0
/ 2.0
result so far is 12.5
q 0
final result is 12.5
```

10. Write a function that reads a problem involving two common fractions such as 2/4 + 5/6. After reading the common fractions problem, call a function to perform the indicated operation (call `addFrac` for +, call

`multiplyFrac` for `*`, and so on). Pass the numerator and denominator of both fractions to the function that performs the operation; the function should return the numerator and denominator of the result through its output parameters. Then display the result as a common fraction. (Hint: Use functions `readFracProblem` and `getFrac`; see Listings 6.3 and 6.4.)

11. Write a function to calculate the day number for a given day represented by three type `int` values (the function input). The day number should be between 1 and 365 (366 if the year is a leap year). Also, write a function that returns true if its first date (first three `int` arguments) comes after its second date (last three `int` arguments). Then write a recursive function that determines the number of days between two dates, each represented by three integers. Use the following algorithm:

```
If the first date comes after the second date
    Reverse the dates and calculate the days between them
Else if the year of both dates is the same
    The result is the day number of the second date minus
    the day number of the first date
Else
    The result is the day number of the second date +
    the number of days between the first date and the last
    day of the previous year
```

The recursive function requires the second date to follow the first. If it doesn't, the task under the first condition should return the result of calling the function with the arguments reversed, so the second date will follow the first in this next call.

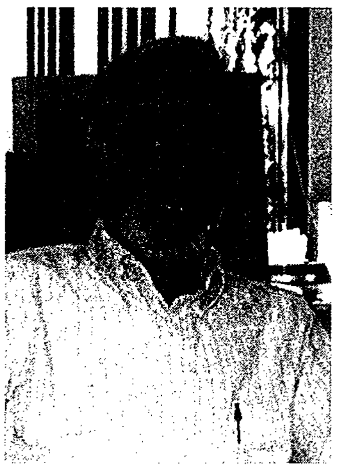
If the dates are in the correct order and the year is the same (tested by the second condition), the result is just the difference in the day numbers for each date. Otherwise, the last task calculates the result by adding the day number of the second date to the number of days between the first date and the end of the year just before the year of the second date. For example, if the second date is January 5, 2003, the result is 5 + the number of days between the first date and December 31, 2002. The next call to the function will calculate the latter value either by executing the second task (if 2002 is the year of the first date) or by adding 365 (the number of days in 2002) to the difference between the first date and December 31, 2001, and so on.

Answers to Quick-Check Exercises

1. call; definition, prototype; prototype
2. value (or call-by-value)
3. reference (or call-by-reference), `&`, after
4. variable, expression
5. call-by-reference; call-by-value
6. void
7. stub
8. black-box testing
9.
```
16, 8
8, 5
10, 5
8, 5
```
10.
```
16, 8
16, 5
10, 5
16, 10
```
11. A function can return values by using a `return` statement or by assigning the values to be returned to reference parameters.

Robert Sebesta

Dr. Sebesta received his Ph.D. in Computer Science from Penn State University. He retired from the faculty of the University of Colorado at Colorado Springs in 2005, after teaching computer science for 35 years. Dr. Sebesta currently works on revisions of his books and teaches programming languages and Web programming part-time. He is the author of several books, including Concepts of Programming Languages *and* Programming the World Wide Web.



What is your educational background, and how did you decide to study computer science?
I have a B.S. in Applied Mathematics (University of Colorado at Boulder), an M.S. in Computer Science (Penn State), and a Ph.D. in Computer Science (Penn State). I decided to study computer science for two reasons: (1) I minored in computer science as an undergraduate and enjoyed programming, and (2) most of the jobs available for someone with a B.S. in applied math were in software development. I decided that if I was going to be a software developer, I needed more education in computer science.

What was your first job in the computer industry?
I worked half-time in a government research lab during my last two years of undergraduate school. I was a programmer for a small group of scientists who were doing research on wave propagation.

Which person in the field has inspired you most?
One early inspiration for me was the researcher for whom I worked as an undergraduate, Bob Lawrence. He was very interested in computer applications in his research. I learned much about scientific computing and in the process discovered that I really enjoyed doing that kind of work.

Do you have any advice about programming in C++?
Because of its widespread use in industry, all undergraduate students in computer science must learn it. However, by most reasonable measures, C++ is a large, complex language that includes more insecurities than some other languages, such as Java and C#. Because of its complexity and insecurities, C++ programs are more difficult to maintain than programs written in some other languages.

What advice do you have for students entering the computer science field?
First, computer science is not an easy discipline, so plan on spending large amounts of time studying and working on programming projects. Although the job market is certainly not as strong as it was in the 1990s, there are many jobs available in software development.

What is the most challenging part of teaching programming?
Because only a small percentage of people can easily become skilled programmers, the challenge for educators is to find ways to teach as many of the others as we can.

How has C++ evolved in the last few years?
Although new class libraries are always being created, the language itself has not changed much in the last few years. The changes that would be required to make the language less complex and more reliable are substantial and unlikely to happen.

What is your vision for the future of the C++ programming language?
I think C++ will remain a very popular language in industry, in part because there is a great deal of legacy software that is written in C++. However, the popularity of C++ has been eroded by the widespread use of Java and, more recently, C#.

chapter seven

Simple Data Types

Learning Objectives

- To learn how to define constants using the #define compiler directive
- To understand the differences between fixed-point and floating-point representations of numbers
- To learn about additional data types for representing very large integers and real numbers, and the range of numbers that can be represented by these different data types
- To learn how to manipulate character data using functions in library cctype
- To understand the encoding of character data using the ASCII character set
- To learn how to write and use bool functions in programming
- To understand how to use enumeration types and enumerators to define a data type and its associated values

SO FAR WE'VE used four *predefined data types* of C++: `int`, `float`, `char`, and `bool`. In this chapter, we look more closely at these data types and introduce some variants of them. We examine the forms in which these types of values are stored in memory, and we introduce some new operators that can be applied to these types. We also introduce the *enumeration type*. A characteristic shared by all of these data types is that only a single value can be stored in a variable of the type. Such types are often referred to as *simple* or *scalar data types*.