# Programming Fundamentals Section B10
# Session: Spring 2025

**Assignment no ONE:**

**😊 DON'T PANIC 😊**

**Part One: Due Handwritten on TUE April 08, 2025 in Class**

**AND Portal Submission at 23:59 PM SUN April 06, 2025**

**The Use of ChatGPT is Allowed for Understanding**

**Copy (Even One line) from ChatGPT or Anyone or Anywhere Else is Zero Marks**

**100 Marks**

**Use of string results in no Grading of Assignment**

**Use of cstring library (strlen, strcpy etc.) results in no Grading of Assignment**

**Use of getline and single character reading (get, getc, getch etc.) results in no Grading of Assignment**

**Final Marks are based on Assignment Evaluation and Formal Quiz One**

| | |
|---|---|
| **Topic** | Variables, If-else, loop, 1D Array, File handling and Functions |
| **Objective/ Outcome** | • Use of arrays with text files<br>• Converting Standalone Code to Function-Based Code |
| **# of Problems** | **Part One (no coding): 20 problems for description in English**<br>**Part Two (coding): Set One: 07 Problems; Set Two: 10 Problems ; Set Three: 07 Problems** |

## General Instructions

- **😊 DON'T PANIC 😊**
- Assignment may seem long but it contains tasks that will give you good coding practice
- The codes done in this assignment will be used in subsequent assignments
- Make sure to read the **whole assignment document** carefully and take **notes**
- **DON'T DISREGARD INSTRUCTIONS**
- Do as many tasks as you can by the **DEADLINE** as there is no extension in the deadline and **ABSOLUTELY** no Submission will be accepted after the deadline
- Recommended to submit work at least one hour before the deadline to avoid any internet or portal issues

- **CORRECTION** of any **FOUND ERRORS** or **MISTAKES** is **part of the assignment**

## Portal Submission Instructions

- Each Set must be in a separate CPP file that should compile and run
- **NO ZIP OR RAR FILES (ZERO MARKS)**
- Name each CPP file for each problem must be as per the following convention:
  - F24_PF_**section**_Registration_A1 _set#.cpp (25 % marks deduction if this is not followed)
  - For example .CPP file for Set 1 of registration L1S24BSCS001 of Section B10 **MUST** have this name:
    **F24_PF_B10_ L1S24BSCS001_A1_Set1.cpp**
- In each CPP file: (25 % marks deduction if not followed)
  - The first lines in comments must contain: your name, registration, assignment number and problem number
  - AND copy the problem statement
- **Must use the provided following template or ZERO Marks**

Here is a template CPP file for problem one

```
/*
Ahmad Usman L1F23BSCS0012
PF Section B10 Assignment One Set One

Problem One (example only)
Write complete C++ code that keeps reading (positive) prices from a user until user enters a negative price and then displays sum, average, maximum and minimum price value
The program must produce output as per the following sample output screen:

Enter prices (positive) values or a negative value to stop: 234.24 349.65 789.25 298.12 456.23 98.7 123.34 56.96 -12.3
You have entered 8 prices
Sum is 2406.49 and Average is 300.811
Maximum Price is 789.25 and Minimum Price is 56.96

Problem two
……..
*/
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
        // write code here
```

```
return 0;
}
```

## Marking Scheme:

- Each Day Assignment is Late is 25 % marks deduction per day
- Each Problem marks are Listed with the Problem:
  - Correct Logic: 30 % marks
  - Correct Code: 20 % marks
  - Correct Syntax: 20 % Marks
  - Correct Output: 30 % marks
  - Not Obeying any instruction in **red is ZERO Marks**
- Marks Deductions:
  - Not Following Instructions (up to 100 % marks deductions)
  - Not Following Good Code Writing Practices as Listed below (2 marks per violation)
    - Meaningful names of variables
    - Each variable in a separate line
    - Each variable must be initialized
    - Use of Indentation and White spaces to make code readable
      Each if, else, while, for, do-while etc. must have brackets { }
- Handwritten on Time Submission 60 %
- **Use of Advance C++ than Covered in Class is zero Marks**
- **Copy is ZERO for All Parties**
- **Copy from ChatGPT or anywhere elase iz ZERO marks**
- Code does not **compile or run**: **ZERO MARKS**

## PART ONE [5 Marks per Question; Do as many as you Can]: Handwritten only (Marks are based on attempt, completeness and details and not on the correct answer)

### Reading and Description in Own Words

### (Don't Use any Code or Diagrams, Explain Everything in Pure English)

1. Read PF TextBook Chapter Six (**PDF is appended at the End of this assignment after page 10**) on User-Defined Function and Describe in your own words
2. Explain in your own words, **in pure English,** how to write code for the following tasks (**the coding part consists of some of these tasks**) **[Any code will result in zero marks]**
   i. Shift Right N Times (**User Provided**) an Array of Any Data Type **(including cstring)** from a given index (**User Provided)**
   ii. Shift Left N Times (**User Provided**) an Array of Any Data Type **(including cstring)** from a given index (**User Provided)**

iii. Rotate Right N Times (**User Provided**) an Array of Any Data Type **(including cstring)** from a given index (**User Provided)**
iv. Rotate Left N Times (**User Provided**) an Array of Any Data Type **(including cstring)** from a given index (**User Provided)**
v. Delete N values from an array at a given position (**User Provided**)
vi. Insert a Given Array (**User Provided**) in another array at a given position (**User Provided**)
vii. Extract N (**User Provided**) values as an array from another array
viii. Extract values as an array from another array from a given start and end index (Extract N (**both User Provided**)
ix. Given an integer, count the number of digits e.g. 12345 has 5 digits
x. Given decimal, count the number of digits e.g. 123.45 has 5 digits
xi. Extract a digit from an integer (from right) from a user given point e.g. digit at position 3 of 123456 is 3
xii. Separate digits of an integer and store each individual digit in an array
xiii. Separate digits of a decimal and store each individual digit of mantissa and fraction in 2 separate arrays
xiv. Convert an Integer to an equivalent cstring e.g. 12456 to "12456"
xv. Convert a decimal to an equivalent cstring e.g. 124.567 to "124.567"
xvi. Convert a cstring to an equivalent integer e.g. "123567" to 123567
xvii. Convert a cstring to an equivalent decimal e.g. "123.567" to 123.567
xviii. Given array with many repeated values, save values only once in an array e.g. 1.1 2.2 1.1 3.3 2.2 4.4 5.5 6.6 1.1 3.3 has repeated values, so considering each value once will have 1.1 2.2 3.3 4.4 5.5 6.6
xix. Given an array, count the frequency (occurrence) of each value and save the frequency in an array
xx. Read an English paragraph from a file and display the frequency of each of its characters

# PART TWO: CODING[Each Task of each Set is of 5 marks and Total marks are based on number of tasks done]:

## Problem Set One

Write a complete C++ Program that does the following:

1. Read Integers from a CSV File **(file path is provided by the user)** in an array **named values** (**maximum 100 values in the file**)and display on screen
2. Calculate and display on screen the sum (**ignore sign of values**), average, maximum and minimum values
3. Count and display number of values that are zero, positive, negative, odd and even
4. Count and display number of values that are odd & positive, odd & negative, positive & even and negative & even

5. Count and display number of values that are perfect squares and prime
6. Count and display number of values that are palindromes e.g. 1221 and 3456543 (**single digits are palindromes)**
7. Take each pair of values as coordinates (x, y) of a point in 2D, display which quadrant and the count the number of points falling in each quadrant (**see sample output below**)

## Sample Output Screen of Problem Set One (user inputs are shown in bold and blue)

1. Please enter path for the input file: **values.csv**
   Read the following integers from file "values.txt":
   -1 2 3 -4 5 6 -7 8 9 -10
2. After calculation of sum, average, maximum and minimum values, following is the results:
   Sum Average Maximum Minimum
   55 5.5 9 -10
3. After counting number of values that are zero, positive, negative, odd and even, following is the results:
   Zeros Positives Negatives Odds Evens
   0 6 4 5 5
4. After counting number of values that are odd & positive, odd & negative, positive & odd and positive & even, following is the results:
   odd & positive odd & negative positive & even negative & even
   3 2 3 2
5. After counting number of values that are perfect squares and prime, following is the results:
   perfect squares prime
   2 5
6. After counting number of values that are palindromes and that are not, following is the results:
   Palindrome Not Palindrome
   9 1
7. Taking pair of values as point (x, y) and following is the result of checking and counting points in each quadrant:
   Point 1st Quadrant 2nd Quadrant 3rd Quadrant 4th Quadrant
   (-1 2) x
   (3 -4) x
   (5 6) x
   (-7 8) x
   (9 -10) x
   1st Quadrant 2nd Quadrant 3rd Quadrant 4th Quadrant
   1 2 0 2

## Problem Set Two

Write a complete C++ Program that does the following:

1. Read Prices (Decimal Values) from a CSV File **(file path is provided by the user)** in an array **named prices** and display on screen **as shown in the following example output.**
2. Read Costs (Decimal Values) from a CSV File **(file path is provided by the user)** in an array **named costs** and display on screen
3. Calculates % profits (% profit = (price – cost)/cost * 100) and save in an array **named profits** and display price, cost and profit on screen **in the format shown in the following example output.**
4. Combines prices, costs and profits in a single array **named values** and display on screen
5. Copy array values in an array **named single** such that each value is only copied once, display array single on screen
6. Count frequency of each values and save in an array **named frequencies** and display frequencies on screen **in the format shown in the following example output.**
7. Read number of values to delete and the index from the user and delete from array **prices then display on screen**
8. Read integer values from a file "ints.txt" and save in an **array named ints** and display on screen
9. Insert array **ints** at a given position provided by the user in the array **costs** at the user provided index and then display **values on screen**
10. First **copy** the current array **profit** to **another array** to save, then read index and amount of rotation from user and then
    a. **rotate left** array **profit** from given **index** and as per the **rotation amount**, display on screen
    b. **rotate right** array **profit** from given **index** and as per the **rotation amount**, display on screen (**must restore the original array**)
    c. Now **compare** the array **profit** with the **saved copy** to **verify** that both **arrays** are **same**, hence the array is restored

## Sample Output Screen of Problem Set Two (user inputs are shown in bold and blue)

1. Please enter path for the input file containing prices: **prices.csv**
   The following Prices are read from File **"prices.csv":**
   12.23 14.45 16.75 9.25 19.9 12.23 16.75 14.45 9.25 19.9 14.45 16.75
2. Please enter path for the input file containing costs: **costs.csv**
   The following Costs are read from File **"costs.csv":**
   1.1 2.2 3.3 4.4 5.5 1.1 2.2 2.2 3.3 3.3 4.4 5.5 1.1 2.210.5 9.5 14.45 8.5 14.5 10.25 14.94 12.56 7.96 14.87 10.75 12.87
3. After % profits calculation, the results are as following:
   Price Cost Profit

| Price | Cost | Profit |
|---|---|---|
| 12.23 | 10.5 | 16.48 |
| 14.45 | 9.5 | 52.11 |
| 16.75 | 14.45 | 15.91 |

| | | |
|---|---|---|
| 9.25 | 8.5 | 8.82 |
| 19.9 | 14.5 | 37.24 |
| 12.23 | 10.25 | 19.31 |
| 16.75 | 14.94 | 12.11 |
| 14.45 | 12.56 | 15.04 |
| 9.25 | 7.96 | 16.20 |
| 19.9 | 14.87 | 33.82 |
| 14.45 | 10.75 | 34.41 |
| 16.75 | 12.87 | 30.14 |

4. After combining prices, costs and profits in a single array **named values**, the results are as following:
   12.23 14.45 16.75 9.25 19.9 12.23 16.75 14.45 9.25 19.9 14.45 16.75 10.5 9.5 14.45 8.5 14.5 10.25 14.94 12.56 7.96 14.87 10.75 12.87 16.48 52.11 15.92 8.82 37.24 19.32 12.12 15.05 16.21 33.83 34.42 30.15
5. After copying array values in an array **named single**, the results are as following:
   12.23 14.45 16.75 9.25 19.90 10.50 9.50 8.50 14.50 10.25 14.94 12.56 7.96 14.87 10.75 12.87 16.48 52.11 15.92 8.82 37.24 19.32 12.12 15.05 16.21 33.83 34.42 30.15
6. After counting frequency of each values, the results are as following:
   Value Frequency

| Value | Frequency |
|---|---|
| 12.23 | 2 |
| 14.45 | 4 |
| 16.75 | 3 |
| 9.25 | 2 |
| 19.9 | 2 |
| 19.9 | 2 |
| 10.5 | 1 |
| 9.5 | 1 |
| 8.5 | 1 |
| 14.5 | 1 |
| 10.25 | 1 |
| 14.94 | 1 |
| 12.56 | 1 |
| 7.96 | 1 |
| 14.87 | 1 |
| 10.75 | 1 |
| 12.87 | 1 |
| 16.48 | 1 |
| 52.11 | 1 |
| 15.92 | 1 |
| 8.82 | 1 |
| 37.24 | 1 |
| 19.32 | 1 |
| 12.12 | 1 |
| 15.05 | 1 |

16.21 1
33.83 1
34.42 1
30.15 1

7. Please enter the number of values that you want to delete and the index: **4 3**
   After deletion of 4 values from index 3 of array prices, the results are as following:
   12.23 14.45 16.75 14.45 9.25 19.9 14.45 16.75
8. After read integer values from a file "ints.txt" , the values are as following:
   1 2 3 4 5
9. Please enter position (index) in array where you want to insert: **5**
   After Insertion of array **ints** at position 5 in array costs, the results are as following:
   1.1 2.2 3.3 4.4 5.5 **1 2 3 4 5** 1.1 2.2 2.2 3.3 3.3 4.4 5.5 1.1 2.210.5 9.5 14.45 8.5 14.5 10.25 14.94 12.56 7.96 14.87 10.75 12.87
10. Copying array profit to save current values
    Please Enter index and amount of rotation: 4 5
    a. **After rotating left** array **profit** from **index** 4 and **rotation amount 5**, the array is as following:
       12.23 16.75 14.45 9.25 19.9 14.45 16.75 19.9 9.25 16.75 14.45 12.23
    b. **After rotating right** array **profit** from **index** 4 and **rotation amount 5**, the array is as following:
       12.23 14.45 16.75 9.25 19.9 12.23 16.75 14.45 9.25 19.9 14.45 16.75
    c. **Comparing with the saved copy of array profit conforms restoration of the array**

## Problem Set Three

Write a complete C++ Program that does the following:

1. Read an English paragraph from a file (path provided by the user) in an array named **para** and display para on screen (**maximum characters in the paragraph are 200)**
2. In array para, count and display number of lowercase, UPPERCASE, digits e.g. 0 to 9, punctuations e.g. , . ; : and Non-English e.g. & % $ # \ characters
3. In array para, convert all lowercase characters to uppercase and all uppercase characters to lowercase and display on screen as well as write to file (path provided by the user)
4. In array para, count the frequency of each character in array **para, save in an array named frequencies** and display on screen as well as write to a file "**freq.csv" in the format shown in the following example freq.csv file:**

| Character | Frequency |
|---|---|
| T | 12 |
| h | 15 |

5. Copy array **para** in (**English**) **word reverse** to another array **reverse and** display arrays **para a**nd **reverse** on screen as well as write to a file "**reverse.csv" (see the format in the following**

**example). word-reverse mean that first word of array para will be the last word on reverse and so on**

| Para Word | Word Reverse Para Word |
|---|---|
| The | example |
| work | good |
| is | a |
| good | is |
| This | This |
| is | good |
| a | is |
| good | work |
| example | The |

6. Copy **words** from array **para** to an array named **single such that single contains a word only once (**or alternatively copy para to single and then remove duplicates from single and display on screen
7. Count frequency of each word of **para** (**ignore case)** and save in an array **wordFreq** and display on screen as well as write to a file "**word_freq.csv" in the format shown in the following example word_freq.csv file:**

| Para Word | Frequency |
|---|---|
| This | 2 |
| work | 1 |
| is | 2 |
| good | 2 |
| a | 1 |
| example | 1 |

## Sample Output Screen of Problem Set Three (user inputs are shown in bold and blue)

1. Please enter path for the input file containing paragraph: **para.txt**
   After reading an English paragraph from file **para.txt**
   This work is good. 4 5 6 & % ; : This is a good example. 1 4 5 & $ # @ ' :
2. After character count, following is the result
   Lowercase UPPERCASE digits punctuations Non-English
   30 2 6 6 6
3. Please enter path for the output file: **converted.txt**
   After conversion of all lowercase characters to uppercase and all uppercase characters to lowercase, results are as following:
   tHIS WORK IS GOOD. 4 5 6 & % ; : tHIS IS A GOOD EXAMPLE. 1 4 5 & $ # @ ' :
4. After counting the frequency of each character in array **para**, results are as following:

Character: t H I S W O R K G D . 4 5 6 & % ; : E X A M P L E 1 $ # @ '
Frequency: 2 2 4 4 1 5 1 1 2 2 2 2 1 2 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1

5. After copy of array **para** in **word reverse** to another array **reverse, following is the result:**

| Para Word | Word Reverse Para Word |
|---|---|
| The | example |
| work | good |
| is | a |
| good | is |
| This | This |
| is | good |
| a | is |
| good | work |
| example | The |

6. After Copy **words** from array **para** removing duplicates, the array single is as following:
   tHIS WORK IS GOOD A EXAMPLE
7. After frequency count of each word of **para, the result is as following**

| Para Word: | this | work | is | good | a | example |
|---|---|---|---|---|---|---|
| Frequency: | 2 | 1 | 2 | 2 | 1 | 1 |

**Chapter 6 from PF text Book on Next Pages**

CHAPTER 6



# User-Defined Functions

IN THIS CHAPTER, YOU WILL:

1. Learn about standard (predefined) functions and discover how to use them in a program
2. Learn about user-defined functions
3. Examine value-returning functions, including actual and formal parameters
4. Explore how to construct and use a value-returning, user-defined function in a program
5. Learn about function prototypes
6. Learn how to construct and use void functions in a program
7. Discover the difference between value and reference parameters
8. Explore reference parameters and value-returning functions
9. Learn about the scope of an identifier
10. Examine the difference between local and global identifiers
11. Discover static variables
12. Learn how to debug programs using drivers and stubs
13. Learn function overloading
14. Explore functions with default parameters

In Chapter 2, you learned that a C++ program is a collection of functions. One such function is **main**. The programs in Chapters 1 through 5 use only the function **main**; the programming instructions are packed into one function. This technique, however, is good only for short programs. For large programs, it is not practical (although it is possible) to put the entire programming instructions into one function, as you will soon discover. You must learn to break the problem into manageable pieces. This chapter first discusses the functions previously defined and then discusses user-defined functions.

Let us imagine an automobile factory. When an automobile is manufactured, it is not made from basic raw materials; it is put together from previously manufactured parts. Some parts are made by the company itself; others, by different companies.

Functions are like building blocks. They let you divide complicated programs into manageable pieces. They have other advantages, too:

- While working on one function, you can focus on just that part of the program and construct it, debug it, and perfect it.
- Different people can work on different functions simultaneously.
- If a function is needed in more than one place in a program or in different programs, you can write it once and use it many times.
- Using functions greatly enhances the program's readability because it reduces the complexity of the function **main**.

Functions are often called **modules**. They are like miniature programs; you can put them together to form a larger program. When user-defined functions are discussed, you will see that this is the case. This ability is less apparent with predefined functions because their programming code is not available to us. However, because predefined functions are already written for us, you will learn these first so that you can use them when needed.

## Predefined Functions

Before formally discussing predefined functions in C++, let us review a concept from a college algebra course. In algebra, a function can be considered a rule or correspondence between values, called the function's arguments, and the unique values of the function associated with the arguments. Thus, if $f(x) = 2x + 5$, then $f(1) = 7$, $f(2) = 9$, and $f(3) = 11$, where 1, 2, and 3 are the arguments of the function *f*, and 7, 9, and 11 are the corresponding values of *f*.

In C++, the concept of a function, either predefined or user-defined, is similar to that of a function in algebra. For example, every function has a name and, depending on the values specified by the user, it does some computation. This section discusses various predefined functions.

Some of the predefined mathematical functions are `pow(x, y)`, `sqrt(x)`, and `floor(x)`.

The *power* function, `pow(x, y)`, calculates $x^y$; that is, the value of `pow(x,y)` = $x^y$. For example, `pow(2.0,3)` = $2.0^3$ = `8.0` and `pow(2.5,3)` = $2.5^3$ = `15.625`. Because the value of `pow(x, y)` is of type `double`, we say that the function `pow` is of type `double` or that the function `pow` returns a value of type `double`. Moreover, `x` and `y` are called the parameters (or arguments) of the function `pow`. Function `pow` has two parameters.

The *square root* function, `sqrt(x)`, calculates the nonnegative square root of `x` for `x >= 0.0`. For example, `sqrt(2.25)` is `1.5`. The function `sqrt` is of type `double` and has only one parameter.

The *floor* function, `floor(x)`, calculates the largest whole number that is less than or equal to `x`. For example, `floor(48.79)` is `48.0`. The function `floor` is of type `double` and has only one parameter.

In C++, predefined functions are organized into separate libraries. For example, the header file `iostream` contains I/O functions, and the header file `cmath` contains math functions. Table 6-1 lists some of the more commonly used predefined functions, the name of the header file in which each function's specification can be found, the data type of the parameters, and the function type. The function type is the data type of the value returned by the function. (For a list of additional predefined functions, see Appendix F.)



**TABLE 6-1** Predefined Functions

| Function | Header File | Purpose | Parameter(s) Type | Result |
|---|---|---|---|---|
| `abs(x)` | `<cmath>` | Returns the absolute value of its argument: `abs(-7) = 7` | `int (double)` | `int (double)` |
| `ceil(x)` | `<cmath>` | Returns the smallest whole number that is not less than `x`: `ceil(56.34) = 57.0` | `double` | `double` |
| `cos(x)` | `<cmath>` | Returns the cosine of angle `x`: `cos(0.0) = 1.0` | `double (radians)` | `double` |
| `exp(x)` | `<cmath>` | Returns $e^x$, where `e = 2.718: exp(1.0) = 2.71828` | `double` | `double` |

**TABLE 6-1** Predefined Functions (continued)

| Function | Header File | Purpose | Parameter(s) Type | Result |
|---|---|---|---|---|
| `fabs(x)` | `<cmath>` | Returns the absolute value of its argument: `fabs(-5.67) = 5.67` | `double` | `double` |
| `floor(x)` | `<cmath>` | Returns the largest whole number that is not greater than `x`: `floor(45.67) = 45.00` | `double` | `double` |
| `islower(x)` | `<cctype>` | Returns `true` if `x` is a lowercase letter; otherwise it returns `false`; `islower('h')` is `true` | `int` | `int` |
| `isupper(x)` | `<cctype>` | Returns `true` if `x` is a uppercase letter; otherwise it returns `false`; `isupper('K')` is `true` | `int` | `int` |
| `pow(x, y)` | `<cmath>` | Returns $x^y$; If x is negative, `y` must be a whole number: `pow(0.16, 0.5) = 0.4` | `double` | `double` |
| `sqrt(x)` | `<cmath>` | Returns the nonnegative square root of `x`, `x` must be nonnegative: `sqrt(4.0) = 2.0` | `double` | `double` |
| `tolower(x)` | `<cctype>` | Returns the lowercase value of `x` if `x` is uppercase; otherwise, returns `x` | `int` | `int` |
| `toupper(x)` | `<cctype>` | Returns the uppercase value of x if x is lowercase; otherwise, returns `x` | `int` | `int` |

To use predefined functions in a program, you must include the header file that contains the function's specification via the include statement. For example, to use the function `pow`, the program must include:

```
#include <cmath>
```

Example 6-1 shows you how to use some of the predefined functions.

## EXAMPLE 6-1

```
// How to use predefined functions.
#include <iostream>                                          //Line 1
#include <cmath>                                             //Line 2
#include <cctype>                                            //Line 3
#include <iomanip>                                           //Line 4

using namespace std;                                         //Line 5

int main()                                                   //Line 6
{                                                            //Line 7
    int num;                                                 //Line 8
    double firstNum, secondNum;                              //Line 9
    char ch = 'T';                                           //Line 10

    cout << fixed << showpoint << setprecision (2)
         << endl;                                            //Line 11

    cout << "Line 12: Is " << ch
         << " a lowercase letter? "
         << islower(ch) << endl;                             //Line 12
    cout << "Line 13: Uppercase a is "
         << static_cast<char>(toupper('a')) << endl;         //Line 13

    cout << "Line 14: 4.5 to the power 6.0 = "
         << pow(4.5, 6.0) << endl;                           //Line 14

    cout << "Line 15: Enter two decimal numbers: ";          //Line 15
    cin >> firstNum >> secondNum;                            //Line 16
    cout << endl;                                            //Line 17

    cout << "Line 18: " << firstNum
         << " to the power of " << secondNum
         << " = " << pow(firstNum, secondNum) << endl;       //Line 18

    cout << "Line 19: 5.0 to the power of 4 = "
         << pow(5.0, 4) << endl;                             //Line 19

    firstNum = firstNum + pow(3.5, 7.2);                     //Line 20
    cout << "Line 21: firstNum = " << firstNum << endl;      //Line 21

    num = -32;                                               //Line 22
    cout << "Line 23: Absolute value of " << num
         << " = " << abs(num) << endl;                       //Line 23

    cout << "Line 24: Square root of 28.00 = "
         << sqrt(28.00) << endl;                             //Line 24

     return 0;                                               //Line 25
}                                                            //Line 26
```

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

```
Line 12: Is T a lowercase letter? 0
Line 13: Uppercase a is A
Line 14: 4.5 to the power 6.0 = 8303.77
Line 15: Enter two decimal numbers: 24.7 3.8

Line 18: 24.70 to the power of 3.80 = 195996.55
Line 19: 5.0 to the power of 4 = 625.00
Line 21: firstNum = 8290.60
Line 23: Absolute value of -32 = 32
Line 24: Square root of 28.00 = 5.29
```

This program works as follows. The statements in Lines 1 to 4 include the header files that are necessary to use the functions used in the program. The statements in Lines 8 to 10 declare the variables used in the program. The statement in Line 11 sets the output of decimal numbers in fixed decimal format with two decimal places. The statement in Line 12 uses the function **islower** to determine and output whether **ch** is a lowercase letter. The statement in Line 13 uses the function **toupper** to output the uppercase letter that corresponds to **'a'**, which is **A**. Note that the function **toupper** returns an **int** value. Therefore, the value of the expression **toupper('a')** is **65**, which is the ASCII value of **'A'**. To print the character **A** rather than the value **65**, you need to apply the cast operator as shown in the statement in Line 13. The statement in Line 14 uses the function **pow** to output $4.5^{6.0}$. In C++ terminology, it is said that the function **pow** is called with the parameters **4.5** and **6.0**. The statements in Lines 15 to 17 prompt the user to enter two decimal numbers and store the numbers entered by the user in the variables **firstNum** and **secondNum**. In the statement in Line 18, the function **pow** is used to output $\text{firstNum}^{\text{secondNum}}$. In this case, the function **pow** is called with the parameters **firstNum** and **secondNum** and the values of **firstNum** and **secondNum** are passed to the function **pow**. The other statements have similar meanings. Once again, note that the program includes the header files **cctype** and **cmath**, because it uses the functions **islower**, **toupper**, **pow**, **abs**, and **sqrt** from these header files.

## User-Defined Functions

As Example 6-1 illustrates, using functions in a program greatly enhances the program's readability because it reduces the complexity of the function **main**. Also, once you write and properly debug a function, you can use it in the program (or different programs) again and again without having to rewrite the same code repeatedly. For instance, in Example 6-1, the function **pow** is used more than once.

Because C++ does not provide every function that you will ever need and designers cannot possibly know a user's specific needs, you must learn to write your own functions.

User-defined functions in C++ are classified into two categories:

- **Value-returning functions**—functions that have a return type. These functions return a value of a specific data type using the `return` statement, which we will explain shortly. Note that the function `main` has used a `return` statement to return the value `0` in every program we've seen so far.
- **Void functions**—functions that do not have a return type. These functions *do not* use a `return` statement to return a value.

We will first discuss value-returning functions. Many of the concepts discussed in regard to value-returning functions also apply to void functions.

## Value-Returning Functions



The previous section introduced some predefined C++ functions such as `pow`, `abs`, `islower`, and `toupper`. These are examples of value-returning functions. To use these functions in your programs, you must know the name of the header file that contains the functions' specification. You need to include this header file in your program using the include statement and know the following items:

1. The name of the function
2. The **parameters**, if any
3. The data type of each parameter
4. The data type of the value computed (that is, the value returned) by the function, called the type of the function

Because a value-returning function returns only one value, the natural thing for you to do is to use the value in one of three ways:

- Save the value for further calculation. For example, `x = pow(3.0, 2.5);`
- Use the value in some calculation. For example,
  `area = PI * pow(radius, 2.0);` or
- Print the value. For example, `cout << abs(-5) << endl;`

This suggests that a value-returning function is used

- In an assignment statement.
- As a parameter in a function call.
- In an output statement.

That is, a value-returning function is used (called) in an expression. Before we look at the syntax of a user-defined, value-returning function, let us consider the things associated with such functions. In addition to the four properties described previously, one more thing is associated with functions (both value-returning and void):

5. The code required to accomplish the task

The first four properties form what is called the **heading** of the function (also called the **function header**); the fifth property is called the **body** of the function. Together, these five properties form what is called the **definition** of the function. For example, for the function `abs`, the heading might look like:

```
int abs(int number)
```

Similarly, the function `abs` might have the following definition:

```
int abs(int number)
{
    if (number < 0)
        number = -number;

    return number;
}
```

The variable declared in the heading of the function `abs` is called a **formal parameter** of the function `abs`. Thus, the formal parameter of `abs` is `number`.

The program in Example 6-1 contains several statements that use the function `pow`. That is, in C++ terminology, the function `pow` is called several times. Later in this chapter, we discuss what happens when a function is called.

Suppose that the heading of the function `pow` is:

```
double pow(double base, double exponent)
```

From the heading of the function `pow`, it follows that the formal parameters of `pow` are `base` and `exponent`. Consider the following statements:

```
double u = 2.5;
double v = 3.0;
double x, y;

x = pow(u, v);                                          //Line 1
y = pow(2.0, 3.2) + 5.1;                                //Line 2
cout << u << " to the power of 7 = " <<  pow(u, 7) << endl;  //Line 3
```

In Line 1, the function `pow` is called with the parameters `u` and `v`. In this case, the values of `u` and `v` are passed to the function `pow`. In fact, the value of `u` is copied into `base`, and the value of `v` is copied into `exponent`. The variables `u` and `v` that appear in the call to the function `pow` in Line 1 are called the **actual parameters** of that call. In Line 2, the function `pow` is called with the parameters `2.0` and `3.2`. In this call, the value `2.0` is copied into `base`, and `3.2` is copied into `exponent`. Moreover, in this call of the function `pow`, the actual parameters are `2.0` and `3.2`, respectively. Similarly, in Line 3, the actual parameters of the function `pow` are `u` and `7`; the value of `u` is copied into `base`, and `7.0` is copied into `exponent`.

We can now formally present two definitions:

**Formal Parameter:** A variable declared in the function heading.

**Actual Parameter:** A variable or expression listed in a call to a function.

For predefined functions, you only need to be concerned with the first four properties. Software companies, typically, do not give out the actual source code, which is the body of the function.

## Syntax: Value-Returning Function

The syntax of a value-returning function is:

```
functionType functionName(formal parameter list)
{
    statements
}
```

in which statements are usually declaration statements and/or executable statements. In this syntax, `functionType` is the type of the value that the function returns. The `functionType` is also called the **data type** or the **return type** of the value-returning function. Moreover, statements enclosed between curly braces form the body of the function.



## Syntax: Formal Parameter List

The syntax of the formal parameter list is:

```
dataType identifier, dataType identifier, ...
```

Consider the definition of the function `abs` given earlier in this chapter. Figure 6-1 identifies various parts of this function.

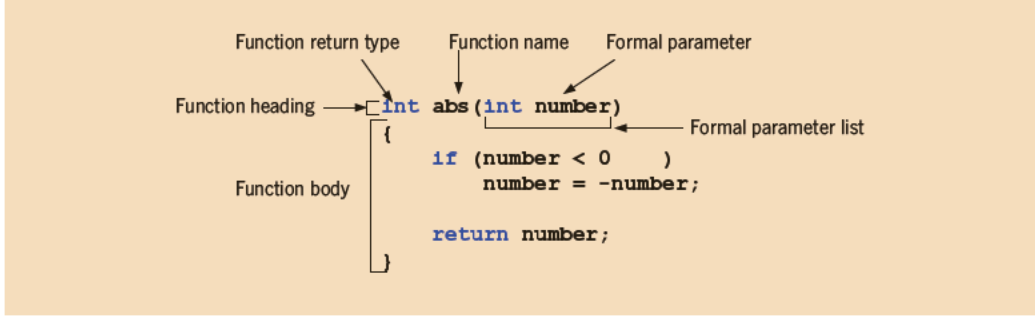



FIGURE 6-1 Various parts of the function `abs`

## Function Call

The syntax to call a value-returning function is:

```
functionName(actual parameter list)
```

For example, in the expression `x = abs(-5);`, the function `abs` is called.

## Syntax: Actual Parameter List

The syntax of the actual parameter list is:

```
expression or variable, expression or variable, ...
```

(In this syntax, **expression** can be a single constant value.) Thus, to call a value-returning function, you use its name, with the actual parameters (if any) in parentheses.

A function's formal parameter list *can* be empty. However, if the formal parameter list is empty, the parentheses are still needed. The function heading of the value-returning function thus takes, if the formal parameter list is empty, the following form:

```
functionType functionName()
```

If the formal parameter list of a value-returning function is empty, the actual parameter is also empty in a function call. In this case (that is, an empty formal parameter list), in a function call, the empty parentheses are still needed. Thus, a call to a value-returning function with an empty formal parameter list is:

```
functionName()
```

In a function call, the number of actual parameters, together with their data types, must match with the formal parameters in the order given. That is, actual and formal parameters have a one-to-one correspondence. (Later in this chapter, we discuss functions with default parameters.)

As stated previously, a value-returning function is called in an expression. The expression can be part of either an assignment statement or an output statement, or a parameter in a function call. A function call in a program causes the body of the called function to execute.

## `return` Statement

Once a value-returning function computes the value, the function returns this value via the `return` statement. In other words, it passes this value outside the function via the `return` statement.

## Syntax: `return` Statement

The **return** statement has the following syntax:

```
return expr;
```

in which **expr** is a variable, constant value, or expression. The **expr** is evaluated, and its value is returned. The data type of the value that **expr** computes must match the function type.

In C++, `return` is a reserved word.

When a `return` statement executes in a function, the function immediately terminates and the control goes back to the calling function. Moreover, the function call statement is replaced by the value returned by the `return` statement. When a `return` statement executes in the function `main`, the program terminates.

To put the ideas in this discussion to work, let us write a function that determines the larger of two numbers. Because the function compares two numbers, it follows that this function has two parameters and that both parameters are numbers. Let us assume that the data type of these numbers is floating-point (decimal)—say, `double`. Because the larger number is of type `double`, the function's data type is also `double`. Let us name this function `larger`. The only thing you need to complete this function is the body of the function. Thus, following the syntax of a function, you can write this function as follows:



```
double larger(double x, double y)
{
    double max;

    if (x >= y)
        max = x;
    else
        max = y;

    return max;
}
```

Note that the function `larger` uses an additional variable `max` (called a **local declaration**, in which `max` is a variable local to the function `larger`). Figure 6-2 describes various parts of the function `larger`.

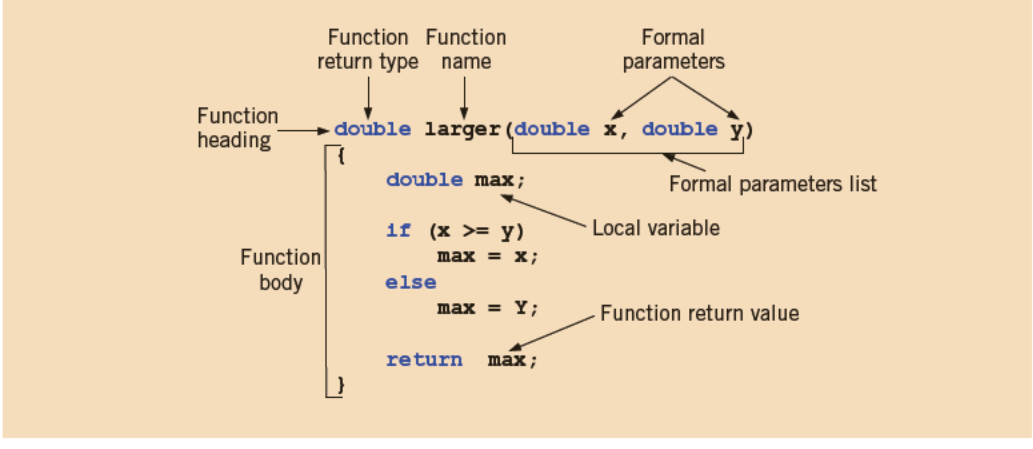



FIGURE 6-2 Various parts of the function `larger`

Suppose that **num**, **num1**, and **num2** are **double** variables. Also suppose that **num1 = 45.75** and **num2 = 35.50**. Figure 6-3 shows various ways the function **larger** can be called.

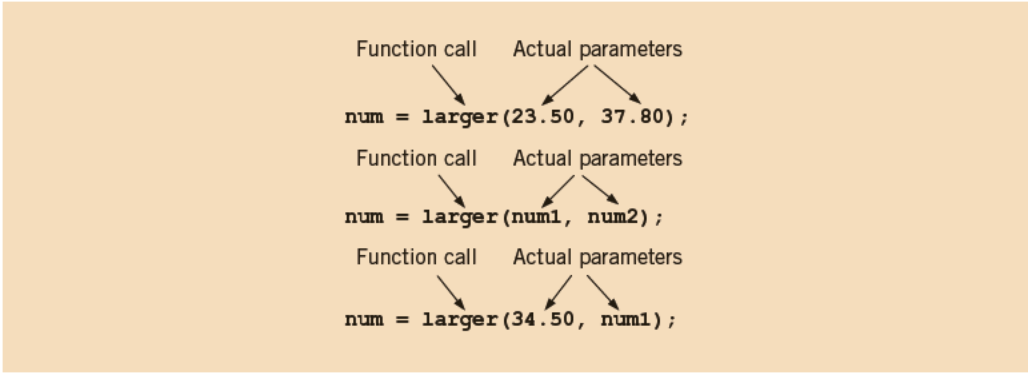



**FIGURE 6-3** Function calls

In Figure 6-3, in the first statement, the function **larger** determines the **larger** of **23.50** and **37.80**, and the assignment statement stores the result in **num**. The meaning of the other two statements is similar.

You can also write the definition of the function **larger** as follows:

```
double larger(double x, double y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

Because the execution of a **return** statement in a function terminates the function, the preceding function **larger** can also be written (without the word **else**) as:

```
double larger(double x, double y)
{
    if (x >= y)
        return x;

    return y;
}
```

Note that these forms of the function **larger** do not require you to declare any local variable.

**NOTE**

1. In the definition of the function **larger**, **x** and **y** are formal parameters.
2. The **return** statement can appear anywhere in the function. Recall that once a **return** statement executes, all subsequent statements are skipped. Thus, it's a good idea to return the value as soon as it is computed.

## EXAMPLE 6-2

Now that the function `larger` is written, the following C++ code illustrates how to use it:

```
double num1 = 13.00;
double num2 = 36.53;
double maxNum;
```

Consider the following statements:

```
cout << "The larger of 5 and 6 is " << larger(5, 6)
     << endl;                                          //Line 1

cout << "The larger of " << num1 << " and " << num2
     << " is " << larger(num1, num2) << endl;          //Line 2

cout << "The larger of " << num1 << " and 29 is "
     << larger(num1, 29) << endl;                      //Line 3

maxNum = larger(38.45, 56.78);                         //Line 4
```

- The expression `larger(5, 6)` in Line 1 is a function call, and `5` and `6` are actual parameters. When the expression `larger(5, 6)` executes, `5` is copied into `x`, and `6` is copied into `y`. Therefore, the statement in Line 1 outputs the larger of `5` and `6`.
- The expression `larger(num1, num2)` in Line 2 is a function call. Here, `num1` and `num2` are actual parameters. When the expression `larger(num1, num2)` executes, the value of `num1` is copied into `x`, and the value of `num2` is copied into `y`. Therefore, the statement in Line 2 outputs the larger of `num1` and `num2`.
- The expression `larger(num1, 29)` in Line 3 is also a function call. When the expression `larger(num1, 29)` executes, the value of `num1` is copied into `x`, and `29` is copied into `y`. Therefore, the statement in Line 3 outputs the larger of `num1` and `29`. Note that the first parameter, `num1`, is a variable, while the second parameter, `29`, is a constant value.
- The expression `larger(38.45, 56.78)` in Line 4 is a function call. In this call, the actual parameters are `38.45` and `56.78`. In this statement, the value returned by the function `larger` is assigned to the variable `maxNum`.

**NOTE**

In a function call, you specify only the actual parameter, not its data type. For example, in Example 6-2, the statements in Lines 1, 2, 3, and 4 show how to call the function `larger` with the actual parameters. However, the following statements contain incorrect calls to the function `larger` and would result in syntax errors. (Assume that all variables are properly declared.)

```
x = larger (int one, 29);            //illegal
y = larger (int one, int 29);        //illegal
cout << larger (int one, int two);   //illegal
```

Once a function is written, you can use it anywhere in the program. The function **larger** compares two numbers and returns the larger of the two. Let us now write another function that uses this function to determine the largest of three numbers. We call this function **compareThree**.

```
double compareThree(double x, double y, double z)
{
    return larger(x, larger(y, z));
}
```

In the function heading, **x**, **y**, and **z** are formal parameters.

Let us take a look at the expression:

```
larger(x, larger(y, z))
```

in the definition of the function **compareThree**. This expression has two calls to the function **larger**. The actual parameters to the outer call are **x** and **larger(y, z)**; the actual parameters to the inner call are **y** and **z**. It follows that, first, the expression **larger(y,z)** is evaluated; that is, the inner call executes first, which gives the larger of **y** and **z**. Suppose that **larger(y, z)** evaluates to, say, **t**. (Notice that **t** is either **y** or **z**.) Next, the outer call determines the larger of **x** and **t**. Finally, the **return** statement returns the largest number. It thus follows that to execute a function call, the parameters must be evaluated first. For example, the actual parameter **larger(y,z)** of the outer call is evaluated first to render a resulting value that is sent with **x** to the outer call to **larger**.

Note that the function **larger** is much more general purpose than the function **compareThree**. Here, we are merely illustrating that once you have written a function, you can use it to write other functions. Later in this chapter, we will show how to use the function **larger** to determine the largest number from a set of numbers.

## Function Prototype

Now that you have some idea of how to write and use functions in a program, the next question relates to the order in which user-defined functions should appear in a program. For example, do you place the function **larger** before or after the function **main**? Should **larger** be placed before **compareThree** or after it? Following the rule that you must declare an identifier before you can use it and knowing that the function **main** uses the identifier **larger**, logically you must place **larger** before **main**.

In reality, C++ programmers customarily place the function **main** before all other user-defined functions. However, this organization could produce a compilation error because functions are compiled in the order in which they appear in the program. For example, if the function **main** is placed before the function **larger**, the identifier **larger** will be undefined when the function **main** is compiled. To work around this problem of undeclared identifiers, we place **function prototypes** before any function definition (including the definition of **main**).

The function prototype is *not* a definition. It gives the program the name of the function, the number and data types of the parameters, and the data type of the returned value: just enough information to let C++ use the function. It is also a promise that the full definition will appear later in the program. If you neglect to write the definition of the function, the program may compile, but it will not execute.

**Function Prototype:** The function heading, terminated by a semicolon, `;`, without the body of the function.

## Syntax: Function Prototype

The general syntax of the function prototype of a value-returning function is:

```
functionType functionName(parameter list);
```

(Note that the function prototype ends with a semicolon.)

For the function `larger`, the prototype is:

```
double larger(double x, double y);    //function prototype
```


When writing the function prototype, you do not have to specify the variable name in the parameter list. However, you must specify the data type of each parameter.

You can rewrite the function prototype of the function `larger` as follows:

```
double larger(double, double);  //function prototype
```

### FINAL PROGRAM

You now know enough to write the entire program, compile it, and run it. The following program uses the functions `larger`, `compareThree`, and `main` to determine the larger/largest of two or three numbers.

```
//Program: Largest of three numbers

#include <iostream>                                          //Line 1

using namespace std;                                         //Line 2

//Function prototype
double larger(double x, double y);                           //Line 3
double compareThree(double x, double y, double z);           //Line 4

int main()                                                   //Line 5
{                                                            //Line 6
    double one, two;                                         //Line 7

    cout << "Line 8: The larger of 5 and 10 is "
         << larger(5, 10) << endl;                           //Line 8
```

```
    cout << "Line 9: Enter two numbers: ";          //Line 9
    cin >> one >> two;                              //Line 10
    cout << endl;                                   //Line 11

    cout << "Line 12: The larger of " << one
         << " and " << two << " is "
         << larger(one, two) << endl;               //Line 12

    cout << "Line 13: The largest of 43.48, 34.00, "
         << "and 12.65 is "
         << compareThree(43.48, 34.00, 12.65)
         << endl;                                   //Line 13

   return 0;                                        //Line 14
}                                                   //Line 15

double larger(double x, double y)
{
    double max;

    if (x >= y)
        max = x;
    else
        max = y;

    return max;
}

double compareThree (double x, double y, double z)
{
    return larger(x, larger(y, z));
}
```

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

```
Line 8: The larger of 5 and 10 is 10
Line 9: Enter two numbers: 37.93 26.785

Line 12: The larger of 37.93 and 26.785 is 37.93
Line 13: The largest of 43.48, 34.00, and 12.65 is 43.48
```


In the previous program, the function prototypes of the functions `larger` and `compareThree` appear before their function definitions. Therefore, the definition of the functions `larger` and `compareThree` can appear in any order.

## Value-Returning Functions: Some Peculiarities

A value-returning function must return a value. Consider the following function, `secret`, that takes as a parameter an `int` value. If the value of the parameter, `x`, is greater than `5`, it returns twice the value of `x`.

```
int secret(int x)
{
    if (x > 5)              //Line 1
        return 2 * x;       //Line 2
}
```

Because this is a value-returning function of type `int`, it must return a value of type `int`. Suppose the value of `x` is `10`. Then the expression `x > 5` in Line 1 evaluates to `true`. So the `return` statement in Line 2 returns the value `20`. Now suppose that `x` is `3`. The expression `x > 5` in Line 1 now evaluates to `false`. The `if` statement therefore fails, and the `return` statement in Line 2 *does not* execute. However, there are no more statements to be executed in the body of the function. In this case, the function returns a strange value. It thus follows that if the value of `x` is less than or equal to `5`, the function does not contain any valid `return` statements to return a value of type `int`.

A correct definition of the function `secret` is:

```
int secret(int x)
{
    if (x > 5)              //Line 1
        return 2 * x;       //Line 2

    return x;               //Line 3
}
```

Here, if the value of `x` is less than or equal to `5`, the `return` statement in Line 3 executes, which returns the value of `x`. On the other hand, if the value of `x` is, say `10`, the `return` statement in Line 2 executes, which returns the value `20` and also terminates the function.

Recall that in a value-returning function, the `return` statement returns the value. Consider the following `return` statement:

```
return x, y;  //only the value of y will be returned
```

This is a legal `return` statement. You might think that this `return` statement is returning the values of `x` and `y`. However, this is not the case. Remember, a `return` statement returns only *one* value, even if the `return` statement contains more than one expression. If a `return` statement contains more than one expression, *only the value of the last expression is returned*. Therefore, in the case of the above `return` statement, the value of `y` is returned. The following program further illustrates this concept:

```
// This program illustrates that a value-returning function
// returns only one value, even if the return statement
// contains more than one expression. This is a legal, but not
// a recommended code.

#include <iostream>

using namespace std;
```

```
int funcRet1();
int funcRet2(int z);

int main()
{
    int num = 4;

    cout << "Line 1: The value returned by funcRet1: "
         << funcRet1() << endl;                          // Line 1
    cout << "Line 2: The value returned by funcRet2: "
         << funcRet2(num) << endl;                       // Line 2

    return 0;
}

int funcRet1()
{
    int x = 45;

    return 23, x;  //only the value of x is returned
}

int funcRet2(int z)
{
    int a = 2;
    int b = 3;

    return 2 * a + b, z + b; //only the value of z + b is returned
}
```

**Sample Run:**

```
Line 1: The value returned by funcRet1: 45
Line 2: The value returned by funcRet2: 7
```

Even though a `return` statement can contain more than one expression, a return statement in your program should contain only one expression. Having more than one expression in a `return` statement may result in redundancy, wasted code, and a confusing syntax.

## More Examples of Value-Returning Functions

### EXAMPLE 6-3

In this example, we write the definition of the function `courseGrade`. This function takes as a parameter an `int` value specifying the score for a course and returns the grade, a value of type `char`, for the course. (We assume that the test score is a value between `0` and `100` inclusive.)

```
char courseGrade(int score)
{
    switch (score / 10)
    {
    case 0:
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        return 'F';
    case 6:
        return 'D';
    case 7:
        return 'C';
    case 8:
        return 'B';
    case 9:
    case 10:
        return 'A';
    }
}
```

You can also write an equivalent definition of the function **courseGrade** that uses an **if...else** structure to determine the course grade.



## EXAMPLE 6-4 (ROLLING A PAIR OF DICE)

In this example, we write a function that rolls a pair of dice until the sum of the numbers rolled is a specific number. We also want to know the number of times the dice are rolled to get the desired sum.

The smallest number on each die is 1, and the largest number is 6. So the smallest sum of the numbers rolled is 2, and the largest sum of the numbers rolled is 12. Suppose that we have the following declarations:

```
int die1;
int die2;
int sum;
int rollCount = 0;
```

We use the random number generator, discussed in Chapter 5, to randomly generate a number between 1 and 6. Then, the following statement randomly generates a number between 1 and 6 and stores that number into **die1,** which becomes the number rolled by **die1**.

```
die1 = rand() % 6 + 1;
```

Similarly, the following statement randomly generates a number between 1 and 6 and stores that number into **die2**, which becomes the number rolled by **die2**.

```
die2 = rand() % 6 + 1;
```

The sum of the numbers rolled by two dice is:

```
sum = die1 + die2;
```

Next, we determine whether **sum** contains the desired sum of the numbers rolled by the dice. (Assume that the `int` variable **num** contains the desired sum to be rolled.) If **sum** is not equal to **num**, then we roll the dice again. This can be accomplished by the following `do...while` loop.

```
do
{
    die1 = rand() % 6 + 1;
    die2 = rand() % 6 + 1;
    sum = die1 + die2;
    rollCount++;
}
while (sum != num);
```

We can now write the function **rollDice** that takes as a parameter the desired sum of the numbers to be rolled and returns the number of times the dice are rolled to roll the desired sum.

```
int rollDice(int num)
{
    int die1;
    int die2;
    int sum;
    int rollCount = 0;

    srand(time(0));

    do
    {
        die1 = rand() % 6 + 1;
        die2 = rand() % 6 + 1;
        sum = die1 + die2;
        rollCount++;
    }
    while (sum != num);

    return rollCount;
}
```

The following program shows how to use the function **rollDice** in a program:

```
//Program: Roll dice

#include <iostream>
#include <cstdlib>
```

```
#include <ctime>

using namespace std;

int rollDice(int num);

int main()
{
    cout << "The number of times the dice are rolled to "
         << "get the sum 10 = " << rollDice(10) << endl;
    cout << "The number of times the dice are rolled to "
         << "get the sum 6 = " << rollDice(6) << endl;

    return 0;
}

int rollDice(int num)
{
    int die1;
    int die2;
    int sum;
    int rollCount = 0;

    srand(time(0));

    do
    {
        die1 = rand() % 6 + 1;
        die2 = rand() % 6 + 1;
        sum = die1 + die2;
        rollCount++;
    }
    while (sum != num);

    return rollCount;
}
```



**Sample Run:**

```
The number of times the dice are rolled to get the sum 10 = 11
The number of times the dice are rolled to get the sum 6 = 7
```

We leave it as an exercise for you to modify this program so that it allows the user to enter the desired sum of the numbers to be rolled. (See Programming Exercise 10 at the end of this chapter.)

## EXAMPLE 6-5 (FIBONACCI NUMBER)

In the first programming example in Chapter 5, we designed and implemented an algorithm to find the number of a Fibonacci sequence. In this example, we modify the

main program by writing a function that computes and returns the desired number of a Fobinacci sequence. Because we have already designed and discussed how to determine a specific number of a Fibonacci sequence, next, we give the definition of the function to implement the algorithm.

Given the first number, the second number, and the position of the desired Fibonacci number, the following function returns the Fibonacci number:

```
int nthFibonacciNum(int first, int second, int nthFibNum)
{
    int current;
    int counter;

    if (nthFibNum == 1)
        current = first;
    else if (nthFibNum == 2)
        current = second;
    else
    {
        counter = 3;

        while (counter <= nthFibNum)
        {
            current = second + first;
            first = second;
            second = current;
            counter++;
        }//end while
    }//end else

    return current;
}
```

The following shows how to use this function in a program.

```
//Program: Fibonacci number

#include <iostream>

using namespace std;

int nthFibonacciNum(int first, int second, int position);

int main()
{
    int firstFibonacciNum;
    int secondFibonacciNum;
    int nthFibonacci;

    cout << "Enter the first two Fibonacci "
         << "numbers: ";
    cin >> firstFibonacciNum >> secondFibonacciNum;
    cout << endl;
```

```
    cout << "The first two Fibonacci numbers are "
         << firstFibonacciNum << " and "
         << secondFibonacciNum
         << endl;

    cout << "Enter the position of the desired "
         << "Fibonacci number: ";
    cin >> nthFibonacci;
    cout << endl;

    cout << "The Fibonacci number at position "
         << nthFibonacci << " is "
         << nthFibonacciNum(firstFibonacciNum, secondFibonacciNum,
                            nthFibonacci)
         << endl;

    return 0;
}

int nthFibonacciNum(int first, int second, int nthFibNum)
{
    int current;
    int counter;

    if (nthFibNum == 1)
        current = first;
    else if (nthFibNum == 2)
        current = second;
    else
    {
        counter = 3;

        while (counter <= nthFibNum)
        {
            current = second + first;
            first = second;
            second = current;
            counter++;
        }//end while
    }//end else

    return current;
}
```



**Sample Runs:** In these sample runs, the user input is shaded.

**Sample Run 1:**

```
Enter the first two Fibonacci numbers: 12 16

The first two Fibonacci numbers are 12 and 16
Enter the position of the desired Fibonacci number: 10

The Fibonacci number at position 10 is 796
```

**Sample Run 2:**

```
Enter the first two Fibonacci numbers: 1 1

The first two Fibonacci numbers are 1 and 1
Enter the position of the desired Fibonacci number: 15

The Fibonacci number at position 15 is 610
```

The following is an example of a function that returns a Boolean value.

## EXAMPLE 6-6 (PALINDROME)

In this example, a function, `isPalindrome`, is designed that returns `true` if a string is a palindrome and `false` otherwise. A string is a palindrome if it reads forward and backward in the same way. For example, the strings `"madamimadam"`, `"5"`, `"434"`, and `"789656987"` are all palindromes.

Suppose `str` is a string. To determine whether `str` is a palindrome, first compare the first and the last characters of `str`. If they are not the same, `str` is not a palindrome and so the function should return `false`. If the first and the last characters of `str` are the same, then we compare the second character with the second character from the end, and so on.

Note that if `length = str.length()`, the number of characters in `str`, then we need to compare `str[0]` with `str[length - 1]`, `str[1]` with `str[length - 2]`, and in general `str[i]` with `str[length - 1 - i]`, where `0 <= i <= length / 2`.

The following algorithm implements this discussion:

1. `int length = str.length();`
2. `for (int i = 0; i < length / 2; i++)`
   `if (str[i] != str[length - 1 - i])`
   `return false;`
   `return true;`

The following function implements this algorithm:

```
bool isPalindrome(string str)
{
    int length = str.length();                    //Step 1

    for (int i = 0; i < length / 2; i++)          //Step 2
        if (str[i] != str[length - 1 - i])
            return false;

    return true;
}
```

## EXAMPLE 6-7 (CABLE COMPANY)

Chapter 4 contains a program to calculate the bill for a cable company. In that program, all of the programming instructions are packed in the function **main**. Here, we rewrite the same program using user-defined functions, further illustrating structured programming.

Because there are two types of customers, residential and business, the program contains two separate functions: one to calculate the bill for residential customers and one to calculate the bill for business customers. Both functions calculate the billing amount and then return the billing amount to the function **main**. The function **main** prints the amount due. Let us call the function that calculates the residential bill **residential** and the function that calculates the business bill **business**. The formulas to calculate the bills are the same as before.



Function **residential**: To compute the residential bill, you need to know the number of premium channels to which the customer subscribes. Based on the number of premium channels, you can calculate the billing amount. After calculating the billing amount, the function returns the billing amount using the `return` statement. The following four steps describe this function:

a. Prompt the user for the number of premium channels.
b. Read the number of premium channels.
c. Calculate the amount due.
d. Return the amount due.

This function contains a statement to prompt the user to enter the number of premium channels (Step a) and a statement to read the number of premium channels (Step b). Other items needed to calculate the billing amount, such as the cost of basic service connection and bill processing fees, are defined as named constants (before the definition of the function **main**). Therefore, to calculate the billing amount, this function does not need to get any value from the function **main**. This function, therefore, has no parameters.

From the previous discussion, it follows that the function **residential** requires local variables to store both the number of premium channels and the billing amount. This function needs only these two local variables to calculate the billing amount:

```
int noOfPChannels;  //number of premium channels
double bAmount;     //billing amount
```

The definition of the function **residential** can now be written as follows:

```
double residential()
{
    int noOfPChannels;  //number of premium channels
    double bAmount;     //billing amount
```

```
    cout << "Enter the number of premium "
         << "channels used: ";
    cin >> noOfPChannels;
    cout << endl;

    bAmount= RES_BILL_PROC_FEES +
             RES_BASIC_SERV_COST +
             noOfPChannels * RES_COST_PREM_CHANNEL;

    return bAmount;
}
```

Function **business**: To compute the business bill, you need to know the number of both the basic service connections and the premium channels to which the customer subscribes. Then, based on these numbers, you can calculate the billing amount. The billing amount is then returned using the `return` statement. The following six steps describe this function:

a. Prompt the user for the number of basic service connections.
b. Read the number of basic service connections.
c. Prompt the user for the number of premium channels.
d. Read the number of premium channels.
e. Calculate the amount due.
f. Return the amount due.

This function contains the statements to prompt the user to enter the number of basic service connections and premium channels (Steps a and c). The function also contains statements to input the number of basic service connections and premium channels (Steps b and d). Other items needed to calculate the billing amount, such as the cost of basic service connections and bill processing fees, are defined as named constants (before the definition of the function **main**). It follows that to calculate the billing amount this function does not need to get any values from the function **main**. Therefore, it has no parameters.

From the preceding discussion, it follows that the function **business** requires variables to store the number of basic service connections and the number of premium channels, as well as the billing amount. In fact, this function needs only these three local variables to calculate the billing amount:

```
int noOfBasicServiceConnections;
int noOfPChannels; //number of premium channels
double bAmount;    //billing amount
```

The definition of the function **business** can now be written as follows:

```
double business()
{
    int noOfBasicServiceConnections;
    int noOfPChannels; //number of premium channels
    double bAmount;    //billing amount
```

```
    cout << "Enter the number of basic "
         << "service connections: ";
    cin >> noOfBasicServiceConnections;
    cout << endl;

    cout << "Enter the number of premium "
         << "channels used: ";
    cin >> noOfPChannels;
    cout << endl;

    if (noOfBasicServiceConnections <= 10)
        bAmount = BUS_BILL_PROC_FEES + BUS_BASIC_SERV_COST +
                  noOfPChannels * BUS_COST_PREM_CHANNEL;
    else
        bAmount = BUS_BILL_PROC_FEES + BUS_BASIC_SERV_COST +
                  (noOfBasicServiceConnections - 10) *
                  BUS_BASIC_CONN_COST +
                  noOfPChannels * BUS_COST_PREM_CHANNEL;

    return bAmount;
}
```

The algorithm for the main program is as follows:

1. To output floating-point numbers in a fixed decimal format with the decimal point and trailing zeros, set the manipulators **`fixed`** and **`showpoint`**.
2. To output floating-point numbers to two decimal places, set the precision to two decimal places.
3. Prompt the user for the account number.
4. Get the account number.
5. Prompt the user to enter the customer type.
6. Get the customer type.
7. a. If the customer type is **`R`** or **`r`**,
    i. Call the function **`residential`** to calculate the bill.
    ii. Print the bill.
   b. If the customer type is **`B`** or **`b`**,
    i. Call the function **`business`** to calculate the bill.
    ii. Print the bill.
   c. If the customer type is other than **`R`**, **`r`**, **`B`**, or **`b`**, it is an invalid customer type.

## PROGRAM LISTING

```
//***********************************************************
// Author: D. S. Malik
//
// Program: Cable Company Billing
// This program calculates and prints a customer's bill for
// a local cable company. The program processes two types of
// customers: residential and business.
//***********************************************************

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

      //Named constants - residential customers
const double RES_BILL_PROC_FEES = 4.50;
const double RES_BASIC_SERV_COST = 20.50;
const double RES_COST_PREM_CHANNEL = 7.50;

      //Named constants - business customers
const double BUS_BILL_PROC_FEES = 15.00;
const double BUS_BASIC_SERV_COST = 75.00;
const double BUS_BASIC_CONN_COST = 5.00;
const double BUS_COST_PREM_CHANNEL = 50.00;

double residential();  //Function prototype
double business();     //Function prototype

int main()
{
        //declare variables
    int accountNumber;
    char customerType;
    double amountDue;

    cout << fixed << showpoint;                    //Step 1
    cout << setprecision(2);                       //Step 2

    cout << "This program computes a cable bill."
         << endl;
    cout << "Enter account number: ";              //Step 3
    cin >> accountNumber;                          //Step 4
    cout << endl;

    cout << "Enter customer type: R, r "
         << "(Residential), B, b (Business): ";    //Step 5
    cin >> customerType;                           //Step 6
    cout << endl;
```

```
    switch (customerType)                          //Step 7
    {
    case 'r':                                      //Step 7a
    case 'R':
        amountDue = residential();                 //Step 7a.i
        cout << "Account number = "
             << accountNumber << endl;             //Step 7a.ii
        cout << "Amount due = $"
             << amountDue << endl;                 //Step 7a.ii
        break;
    case 'b':                                      //Step 7b
    case 'B':
        amountDue = business();                    //Step 7b.i
        cout << "Account number = "
             << accountNumber << endl;             //Step 7b.ii
        cout << "Amount due = $"
             << amountDue << endl;                 //Step 7b.ii
        break;
    default:
        cout << "Invalid customer type." << endl; //Step 7c
   }

   return 0;
}
//Place the definitions of the functions residential and business here.
```

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

```
This program computes a cable bill.
Enter account number: 21341

Enter customer type: R, r (Residential), B, b (Business): B

Enter the number of basic service connections: 25

Enter the number of premium channels used: 9

Account number = 21341
Amount due = $615.00
```

## Flow of Compilation and Execution

As stated earlier, a C++ program is a collection of functions. Recall that functions can appear in any order. The only thing that you have to remember is that you must declare an identifier before you can use it. The program is compiled by the compiler sequentially from beginning to end. Thus, if the function **main** appears before any other user-defined functions, it is compiled first. However, if **main** appears at the end (or middle) of the program, all functions whose definitions (not prototypes) appear before the function **main** are compiled before the function **main**, in the order they are placed.

Function prototypes appear before any function definition, so the compiler complies these first. The compiler can then correctly translate a function call. *However, when the program executes, the first statement in the function* **main** *always executes first, regardless of where in the program the function* **main** *is placed*. Other functions execute only when they are called.

A function call transfers control to the first statement in the body of the function. In general, after the last statement of the called function executes, control is passed back to the point immediately following the function call. A value-returning function returns a value. Therefore, after executing the value-returning function, when the control goes back to the caller, the value that the function returns replaces the function call statement. The execution continues at the point immediately following the function call.

Suppose that a program contains functions **funcA** and **funcB**, and **funcA** contains a statement that calls **funcB**. Suppose that the program calls **funcA**. When the statement that contains a call to **funcB** executes, **funcB** executes, and while **funcB** is executing, the execution of the current call of **funcA** is on hold until **funcB** is done.

## PROGRAMMING EXAMPLE: Largest Number

In this programming example, the function **larger** is used to determine the largest number from a set of numbers. For the purpose of illustration, this program determines the largest number from a set of 10 numbers. You can easily enhance this program to accommodate any set of numbers.

**Input** A set of 10 numbers.

**Output** The largest of 10 numbers.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

Suppose that the input data is:

```
10 56 73 42 22 67 88 26 62 11
```

Read the first number of the data set. Because this is the only number read to this point, you may assume that it is the largest number so far and call it **max**. Read the second number and call it **num**. Now compare **max** and **num** and store the larger number into **max**. Now **max** contains the larger of the first two numbers. Read the third number. Compare it with **max** and store the larger number into **max**. At this point, **max** contains the largest of the first three numbers. Read the next number, compare it with **max**, and store the larger number into **max**. Repeat this process for each remaining number in the data set. Eventually, **max** will contain the largest number in the data set. This discussion translates into the following algorithm:

1. Read the first number. Because this is the only number that you have read so far, it is the largest number so far. Save it in a variable called **max**.

2. For each remaining number in the list:
   a. Read the next number. Store it in a variable called `num`.
   b. Compare `num` and `max`. If `max < num`, then `num` is the new largest number, so update the value of `max` by copying `num` into `max`. If `max >= num`, discard `num`; that is, do nothing.
3. Because `max` now contains the largest number, print it.

To find the larger of two numbers, the program uses the function `larger`.

## COMPLETE PROGRAM LISTING

```
//*********************************************************
// Author: D.S. Malik
//
// This program find the largest number of a set of 10
// numbers.
//*********************************************************

#include <iostream>

using namespace std;

double larger(double x, double y);

int main()
{
    double num; //variable to hold the current number
    double max; //variable to hold the larger number
    int count;  //loop control variable

    cout << "Enter 10 numbers." << endl;
    cin >> num;                                    //Step 1
    max = num;                                     //Step 1

    for (count = 1; count < 10; count++)           //Step 2
    {
        cin >> num;                                //Step 2a
        max = larger(max, num);                    //Step 2b
    }

    cout << "The largest number is " << max
         << endl;                                  //Step 3

    return 0;
} //end main
```

```
double larger(double x, double y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

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

```
Enter 10 numbers.
20 36 71 89 11 65 55 90 44 65
The largest number is 90
```

# Void Functions

Earlier in this chapter, you learned how to use value-returning functions. In this section, you will explore user-defined functions in general and, in particular, those C++ functions that do not have a data type, called **void functions**.

Void functions and value-returning functions have similar structures. Both have a heading and a body. Like value-returning functions, you can place user-defined void functions either before or after the function `main`. However, the program execution always begins with the first statement in the function `main`. If you place a user-defined void function after the function `main`, you should place the function prototype before the function `main`. A void function does not have a data type. Therefore, `functionType`—that is, the return type—in the heading part and the return statement in the body of the void functions are meaningless. However, in a void function, you can use the return statement without any value; it is typically used to exit the function early. Like value-returning functions, void functions may or may not have formal parameters.

Because void functions do not have a data type, they are not used (called) in an expression. A call to a void function is a stand-alone statement. Thus, to call a void function, you use the function name together with the actual parameters (if any) in a stand-alone statement. Before giving examples of void functions, next we give the syntax of a void function.

## FUNCTION DEFINITION

The function definition of void functions with parameters has the following syntax:

```
void functionName(formal parameter list)
{
    statements
}
```

in which **statements** are usually declaration and/or executable statements. The formal parameter list may be empty, in which case, in the function heading, the empty parentheses are still needed.

### FORMAL PARAMETER LIST

The formal parameter list has the following syntax:

```
dataType& variable, dataType& variable, ...
```

You must specify both the data type and the variable name in the formal parameter list. The symbol **&** after **dataType** has a special meaning; *some* parameters will have **&** and some will not, and we will explain *why* later in this chapter.

### FUNCTION CALL

The function call has the following syntax:

```
functionName(actual parameter list);
```

### ACTUAL PARAMETER LIST

The actual parameter list has the following syntax:

```
expression or variable, expression or variable, ...
```

in which **expression** can consist of a single constant value. As with value-returning functions, in a function call, the number of actual parameters together with their data types must match the formal parameters in the order given. Actual and formal parameters have a one-to-one correspondence. (Functions with default parameters are discussed at the end of this chapter.) A function call results in the execution of the body of the called function.

Example 6-8 shows a void function with parameters.

## EXAMPLE 6-8

```
void funExp(int a, double b, char c, int x)
{
    .
    .
    .
}
```

The function **funExp** has four parameters.

### PARAMETER TYPES

Parameters provide a communication link between the calling function (such as `main`) and the called function. They enable functions to manipulate different data each time they are called. In general, there are two types of formal parameters: **value parameters** and **reference parameters**.

**Value parameter:** A formal parameter that receives a copy of the content of the corresponding actual parameter.

**Reference parameter:** A formal parameter that receives the location (memory address) of the corresponding actual parameter.

When you attach `&` after the `dataType` in the formal parameter list of a function, the variable following that `dataType` becomes a reference parameter.

Example 6-9 shows a void function with value and reference parameters.

## EXAMPLE 6-9

Consider the following function definition:

```
void areaAndPerimeter(double length, double width,
                      double& area, double& perimeter)
{
    area = length * width;
    perimeter = 2 * (length + width);
}
```

The function `areaAndPerimeter` has four parameters: `length` and `width` are value parameters of type `double`; and `area` and `perimeter` are reference parameters of type `double`.

Figure 6-4 describes various parts of the function `areaAndPerimeter`.

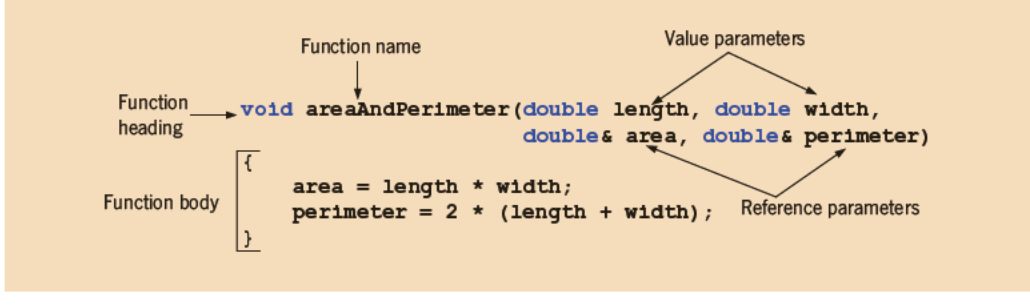



FIGURE 6-4 Various parts of the function `areaAndPerimeter`

## EXAMPLE 6-10

Consider the following definition:

```
void averageAndGrade(int testScore, int progScore,
                     double& average, char& grade)
{
    average = (testScore + progScore) / 2.0;

    if (average >= 90.00)
        grade = 'A';
    else if (average >= 80.00)
        grade = 'B';
    else if (average >= 70.00)
        grade = 'C';
    else if (average >= 60.00)
        grade = 'D';
    else
        grade = 'F';
}
```

The function **averageAndGrade** has four parameters: **testScore** and **progScore** are value parameters of type **int**, **average** is a reference parameter of type **double**, and **grade** is a reference parameter of type **char**. Using visual diagrams, Examples 6-13 to 6-15 explicitly show how value and reference parameters work.



## EXAMPLE 6-11

We write a program to print a pattern (a triangle of stars) similar to the following:

```
   *
  * *
 * * *
* * * *
```

The first line has one star with some blanks before the star, the second line has two stars, some blanks before the stars, and a blank between the stars, and so on. Let's write the function **printStars** that has two parameters, a parameter to specify the number of blanks before the stars in a line and a second parameter to specify the number of stars in a line. To be specific, the definition of the function **printStars** is:

```
void printStars(int blanks, int starsInLine)
{
    int count;

      //print the number of blanks before the stars in a line
    for (count = 1; count <= blanks; count++)
        cout << ' ';

      //print the number of stars with a blank between stars
    for (count = 1; count <= starsInLine; count++)
        cout << " *";

    cout << endl;
} //end printStars
```

The first parameter, **blanks**, determines how many blanks to print preceding the star(s); the second parameter, **starsInLine**, determines how many stars to print in a line. If the value of the parameter **blanks** is **30**, for instance, then the first **for** loop in the function **printStars** executes 30 times and prints 30 blanks. Also, because you want to print a space between the stars, every iteration of the second **for** loop in the function **printStars** prints the string **" * "**—a blank followed by a star.

Next, consider the following statements:

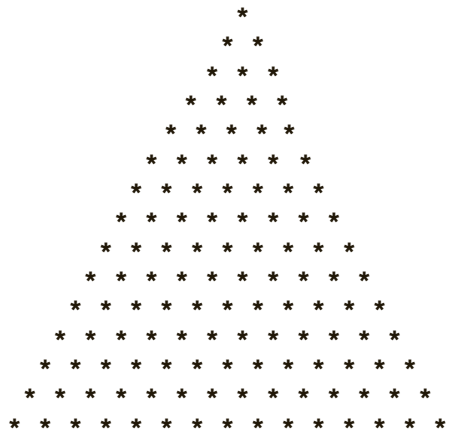
```
int numberOfLines = 15;
int numberOfBlanks = 30;

for (counter = 1; counter <= numberOfLines; counter++)
{
    printStars(numberOfBlanks, counter);
    numberOfBlanks--;
}
```

The **for** loop calls the function **printStars**. Every iteration of this **for** loop specifies the number of blanks followed by the number of stars to print in a line, using the variables **numberOfBlanks** and **counter**. Every invocation of the function **printStars** receives one fewer blank and one more star than the previous call. For example, the first iteration of the **for** loop in the function **main** specifies **30** blanks and **1** star (which are passed as the parameters **numberOfBlanks** and **counter** to the function **printStars**). The **for** loop then decrements the number of blanks by **1** by executing the statement, **numberOfBlanks--;**. At the end of the **for** loop, the number of stars is incremented by **1** for the next iteration. This is done by executing the update statement **counter++** in the **for** statement, which increments the value of the variable **counter** by **1**. In other words, the second call of the function **printStars** receives **29** blanks and **2** stars as parameters. Thus, the previous statements will print a triangle of stars consisting of 15 lines.

```
//Program: Print a triangle of stars

#include <iostream>                                          //Line 1

using namespace std;                                         //Line 2

void printStars(int blanks, int starsInLine);                //Line 3

int main()                                                    //Line 4
{                                                             //Line 5
    int noOfLines;  //var to store the number of lines        Line 6
    int counter;    //for loop control variable               Line 7
    int noOfBlanks; //var to store the number of blanks      Line 8

    cout << "Enter the number of star lines (1 to 20) "
         << "to be printed: ";                                //Line 9
    cin >> noOfLines;                                         //Line 10

    while (noOfLines < 0 || noOfLines > 20)                   //Line 11
    {                                                         //Line 12
        cout << "Enter the number of star lines "
             << "(1 to 20) to be printed: ";                  //Line 13
        cin  >>  noOfLines;                                   //Line 14
    }                                                         //Line 15

    cout << endl << endl;                                     //Line 16
    noOfBlanks = 30;                                          //Line 17

    for (counter = 1; counter <= noOfLines; counter++)        //Line 18
    {                                                         //Line 19
        printStars(noOfBlanks, counter);                      //Line 20
        noOfBlanks--;                                         //Line 21
    }                                                         //Line 22

    return 0;                                                 //Line 23
}                                                             //Line 24

void printStars(int blanks, int starsInLine)                  //Line 25
{                                                             //Line 26
    int count;                                                //Line 27

    for (count = 1; count <= blanks; count++)                 //Line 28
        cout << ' ';                                          //Line 29
    for (count = 1; count <= starsInLine; count++)            //Line 30
        cout << " *";                                         //Line 31
    cout << endl;                                             //Line 32
}                                                             //Line 33
```

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

```
Enter the number of star lines (1 to 20) to be printed: 15
```



In the function **main**, the user is first asked to specify how many lines of stars to print (Line 9). (In this program, the user is restricted to 20 lines because a triangular grid of up to 20 lines fits nicely on the screen.) Because the program is restricted to only 20 lines, the **while** loop at Lines 11 through 15 ensures that the program prints only the triangular grid of stars if the number of lines is between 1 and 20.

## Value Parameters

The previous section defined two types of parameters—value parameters and reference parameters. Example 6-10 showed a program that uses a function with parameters. Before considering more examples of void functions with parameters, let us make the following observation about value parameters. When a function is called, for a value parameter, the value of the actual parameter is copied into the corresponding formal parameter. Therefore, if the formal parameter is a value parameter, then after copying the value of the actual parameter, there is no connection between the formal parameter and actual parameter; that is, the formal parameter is a separate variable with its own copy of the data. Therefore, during program execution, the formal parameter manipulates the data stored in its own memory space. The program in Example 6-12 further illustrates how a value parameter works.

### EXAMPLE 6-12

The following program shows how a formal parameter of a simple data type works.

```
//Example 6-12: Program illustrating how a value parameter works.

#include <iostream>                                    //Line 1

using namespace std;                                   //Line 2
```

```
void funcValueParam(int num);                                   //Line 3

int main()                                                      //Line 4
{                                                               //Line 5
    int number = 6;                                             //Line 6

    cout << "Line 7: Before calling the function "
         << "funcValueParam, number = " << number
         << endl;                                               //Line 7

    funcValueParam(number);                                     //Line 8

    cout << "Line 9: After calling the function "
         << "funcValueParam, number = " << number
         << endl;                                               //Line 9

    return 0;                                                   //Line 10
}                                                               //Line 11

void funcValueParam(int num)                                    //Line 12
{                                                               //Line 13
    cout << "Line 14: In the function funcValueParam, "
         << "before changing, num = " << num
         << endl;                                               //Line 14

    num = 15;                                                   //Line 15

    cout << "Line 16: In the function funcValueParam, "
         << "after changing, num = " << num
         << endl;                                               //Line 16
}                                                               //Line 17
```



**Sample Run:**

```
Line 7: Before calling the function funcValueParam, number = 6
Line 14: In the function funcValueParam, before changing, num = 6
Line 16: In the function funcValueParam, after changing, num = 15
Line 9: After calling the function funcValueParam, number = 6
```

This program works as follows. The execution begins at the function **main**. The statement in Line 6 declares and initializes the `int` variable **number**. The statement in Line 7 outputs the value of **number** before calling the function **funcValueParam**; the statement in Line 8 calls the function **funcValueParam**. The value of the variable **number** is then passed to the formal parameter **num**. Control now transfers to the function **funcValueParam**.

The statement in Line 14 outputs the value of **num** before changing its value. The statement in Line 15 changes the value of **num** to **15**; the statement in Line 16 outputs the value of **num**. After this statement executes, the function **funcValueParam** exits and control goes back to the function **main**.

The statement in Line 9 outputs the value of `number` after calling the function `funcValueParam`. The sample run shows that the value of `number` (Lines 7 and 9) remains the same even though the value of its corresponding formal parameter `num` was changed within the function `funcValueParam`.

The output shows the sequence in which the statements execute.

After copying data, a value parameter has no connection with the actual parameter, so a value parameter cannot pass any result back to the calling function. When the function executes, any changes made to the formal parameters do not in any way affect the actual parameters. The actual parameters have no knowledge of what is happening to the formal parameters. Thus, value parameters cannot pass information outside of the function. Value parameters provide only a one-way link from the actual parameters to the formal parameters. Hence, functions with only value parameters have limitations.

## Reference Variables as Parameters

The program in Example 6-12 illustrates how a value parameter works. On the other hand, suppose that a formal parameter is a reference parameter. Because a reference parameter receives the address (memory location) of the actual parameter, reference parameters can pass one or more values from a function and can change the value of the actual parameter.

Reference parameters are useful in three situations:

- When the value of the actual parameter needs to be changed
- When you want to return more than one value from a function (recall that the return statement can return only one value)
- When passing the address would save memory space and time relative to copying a large amount of data

The first two situations are illustrated throughout this book. Chapters 8 and 10 discuss the third situation, when arrays and classes are introduced.

Recall that when you attach `&` after the `dataType` in the formal parameter list of a function, the variable following that `dataType` becomes a reference parameter.


You can declare a reference (formal) parameter as a constant by using the keyword `const`. This will prevent the formal parameter from being able to change the value of the corresponding actual parameter. Chapters 9 and 10 discuss constant reference parameters. Until then, the reference parameters that you use will be nonconstant as defined in this chapter. From the definition of a reference parameter, it follows that a constant value or an expression cannot be passed to a nonconstant reference parameter. If a formal parameter is a nonconstant reference parameter, during a function call, its corresponding actual parameter must be a variable.

**EXAMPLE 6-13**

## Calculate Grade

The following program takes a course score (a value between `0` and `100`) and determines a student's course grade. This program has three functions: **`main`**, **`getScore`**, and **`printGrade`**, as follows:

1. **`main`**
   a. Get the course score.
   b. Print the course grade.
2. **`getScore`**
   a. Prompt the user for the input.
   b. Get the input.
   c. Print the course score.
3. **`printGrade`**
   a. Calculate the course grade.
   b. Print the course grade.

The complete program is as follows:

```
//This program reads a course score and prints the
//associated course grade.

#include <iostream>                                    //Line 1

using namespace std;                                   //Line 2

void getScore(int& score);                             //Line 3
void printGrade(int score);                            //Line 4

int main ()                                            //Line 5
{                                                      //Line 6
    int courseScore;                                   //Line 7

    cout << "Line 8: Based on the course score, \n"
         << "        this program computes the "
         << "course grade." << endl;                   //Line 8

    getScore(courseScore);                             //Line 9

    printGrade(courseScore);                           //Line 10

    return 0;                                          //Line 11
}                                                      //Line 12
```

```
void getScore(int& score)                                   //Line 13
{                                                           //Line 14
    cout << "Line 15: Enter course score: ";                //Line 15
    cin >> score;                                           //Line 16
    cout << endl << "Line 17: The course score is "
         << score << endl;                                  //Line 17
}                                                           //Line 18

void printGrade(int cScore)                                 //Line 19
{                                                           //Line 20
    cout << "Line 21: The course grade is: ";               //Line 21

    if (cScore >= 90)                                       //Line 22
        cout << "A." << endl;                               //Line 23
    else if (cScore >= 80)                                  //Line 24
        cout << "B." << endl;                               //Line 25
    else if(cScore >= 70)                                   //Line 26
        cout << "C." << endl;                               //Line 27
    else if (cScore >= 60)                                  //Line 28
        cout << "D." << endl;                               //Line 29
    else                                                    //Line 30
        cout << "F." << endl;                               //Line 31
 }                                                          //Line 32
```

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

```
Line 8: Based on the course score,
        this program computes the course grade.
Line 15: Enter course score: 85

Line 17: The course score is 85
Line 21: The course grade is: B.
```

This program works as follows. The program starts to execute at Line 8, which prints the first line of the output (see the sample run). The statement in Line 9 calls the function `getScore` with the actual parameter `courseScore` (a variable declared in `main`). Because the formal parameter `score` of the function `getScore` is a reference parameter, the address (that is, the memory location of the variable `courseScore`) passes to `score`. Thus, both `score` and `courseScore` now refer to the same memory location, which is `courseScore` (see Figure 6-5).

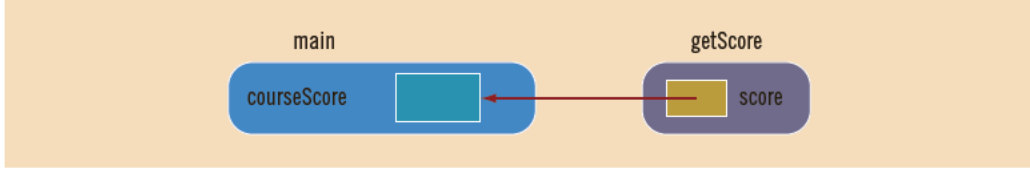



FIGURE 6-5 Variable `courseScore` and the parameter `score`

Any changes made to `score` immediately change the value of `courseScore`.

Control is then transferred to the function `getScore`, and the statement in Line 15 executes, printing the second line of output. This statement prompts the user to enter the course `score`. The statement in Line 16 reads and stores the value entered by the user (`85` in the sample run) in `score`, which is actually `courseScore` (because `score` is a reference parameter). Thus, at this point, the value of both variables `score` and `courseScore` is `85` (see Figure 6-6).

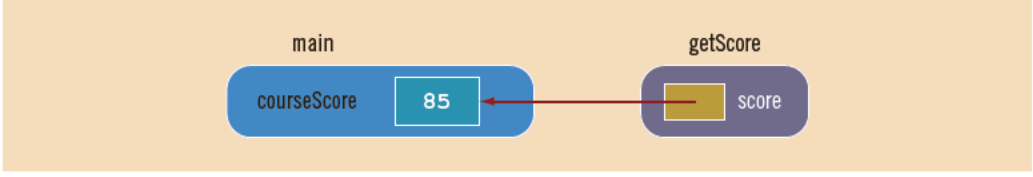



FIGURE 6-6 Variable `courseScore` and the parameter `score` after the statement in Line 16 executes



Next, the statement in Line 17 outputs the value of `score` as shown by the third line of the sample run. After Line 17 executes, control goes back to the function `main` (see Figure 6-7).

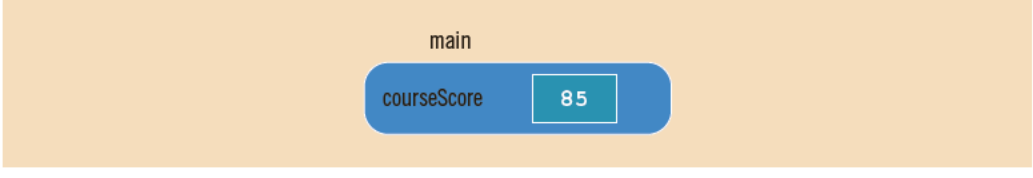



FIGURE 6-7 Variable `courseScore` after the statement in Line 17 is executed and control goes back to `main`

The statement in Line 10 executes next. It is a function call to the function `printGrade` with the actual parameter `courseScore`. Because the formal parameter `cScore` of the function `printGrade` is a value parameter, the parameter `cScore` receives the value of the corresponding actual parameter `courseScore`. Thus, the value of `cScore` is `85`. After copying the value of `courseScore` into `cScore`, no communication exists between `cScore` and `courseScore` (see Figure 6-8).

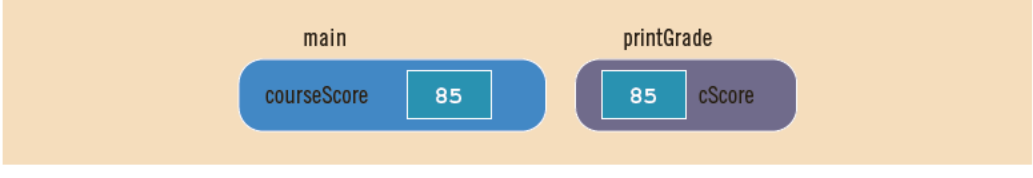



FIGURE 6-8 Variable `courseScore` and the parameter `cScore`

The program then executes the statement in Line 21, which outputs the fourth line. The `if. . .else` statement in Line 22 to 31 determines and outputs the grade for the course. Because the output statement in Line 7 does not contain the newline character or the manipulator **`endl`**, the output of the `if. . .else` statement is part of the fourth line of the output. After the `if. . .else` statement executes, control goes back to the function **`main`**. Because the next statement to execute in the function **`main`** is the last statement of the function **`main`**, the program terminates.

In this program, the function **`main`** first calls the function **`getScore`** to obtain the course score from the user. The function **`main`** then calls the function **`printGrade`** to calculate and print the grade based on this course score. The course score is retrieved by the function **`getScore`**; later, this course score is used by the function **`printGrade`**. Because the value retrieved by the **`getScore`** function is used later in the program, the function **`getScore`** must pass this value outside. Because **`getScore`** is written as a void function, the formal parameter that holds this value must be a reference parameter.

## Value and Reference Parameters and Memory Allocation

When a function is called, memory for its formal parameters and variables declared in the body of the function (called **local variables**) is allocated in the function data area. Recall that in the case of a value parameter, the value of the actual parameter is copied into the memory cell of its corresponding formal parameter. In the case of a reference parameter, the address of the actual parameter passes to the formal parameter. That is, the content of the formal parameter is an address. During data manipulation, the address stored in the formal parameter directs the computer to manipulate the data of the memory cell at that address. Thus, in the case of a reference parameter, both the actual and formal parameters refer to the same memory location. Consequently, during program execution, changes made by the formal parameter permanently change the value of the actual parameter.


Stream variables (for example, **`ifstream`** and **`ofstream`**) should be passed by reference to a function. After opening the input/output file or after reading and/or outputting data, the state of the input and/or output stream can then be passed outside the function.

Because parameter passing is fundamental to any programming language, Examples 6-14 and 6-15 further illustrate this concept. Each covers a different scenario.

## EXAMPLE 6-14

The following program shows how reference and value parameters work.

```
//Example 6-14: Reference and value parameters

#include <iostream>                                          //Line 1

using namespace std;                                         //Line 2

void funOne(int a, int& b, char v);                          //Line 3
void funTwo(int& x, int y, char& w);                         //Line 4

int main()                                                   //Line 5
{                                                            //Line 6
    int num1, num2;                                          //Line 7
    char ch;                                                 //Line 8

    num1 = 10;                                               //Line 9
    num2 = 15;                                               //Line 10
    ch = 'A';                                                //Line 11

    cout << "Line 12: Inside main: num1 = " << num1
         << ", num2 = " << num2 << ", and ch = "
         << ch << endl;                                      //Line 12

    funOne(num1, num2, ch);                                  //Line 13

    cout << "Line 14: After funOne: num1 = " << num1
         << ", num2 = " << num2 << ", and ch = "
         << ch << endl;                                      //Line 14

    funTwo(num2, 25, ch);                                    //Line 15

    cout << "Line 16: After funTwo: num1 = " << num1
         << ", num2 = " << num2 << ", and ch = "
         << ch << endl;                                      //Line 16

    return 0;                                                //Line 17
}                                                            //Line 18

void funOne(int a, int& b, char v)                           //Line 19
{                                                            //Line 20
    int one;                                                 //Line 21

    one = a;                                                 //Line 22
    a++;                                                     //Line 23
    b = b * 2;                                               //Line 24
    v = 'B';                                                 //Line 25
```

```
    cout << "Line 26: Inside funOne: a = " << a
         << ", b = " << b << ", v = " << v
         << ", and one = " << one << endl;          //Line 26
}                                                   //Line 27

void funTwo(int& x, int y, char& w)                 //Line 28
{                                                   //Line 29
    x++;                                            //Line 30
    y = y * 2;                                      //Line 31
    w = 'G';                                        //Line 32

    cout << "Line 33: Inside funTwo: x = " << x
         << ", y = " << y << ", and w = " << w
         << endl;                                   //Line 33
}                                                   //Line 34
```

**Sample Run:**

```
Line 12: Inside main: num1 = 10, num2 = 15, and ch = A
Line 26: Inside funOne: a = 11, b = 30, v = B, and one = 10
Line 14: After funOne: num1 = 10, num2 = 30, and ch = A
Line 33: Inside funTwo: x = 31, y = 50, and w = G
Line 16: After funTwo: num1 = 10, num2 = 31, and ch = G
```

Let us walk through this program. The values of the variables are shown before and/or after each statement executes.

Just before the statement in Line 9 executes, memory is allocated only for the variables of the function `main`; this memory is not initialized. After the statement in Line 11 executes, the variables are as shown in Figure 6-9.

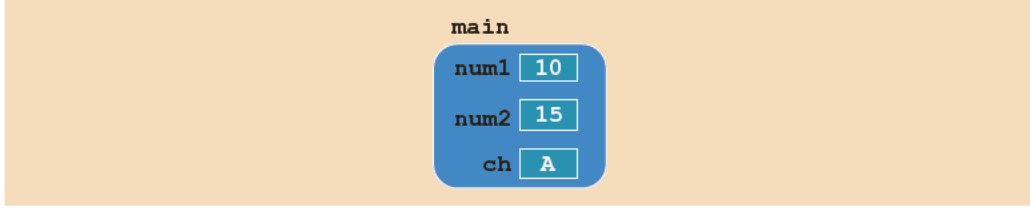



FIGURE 6-9 Values of the variables after the statement in Line 11 executes

The statement in Line 12 produces the following output:

```
Line 12: Inside main: num1 = 10, num2 = 15, and ch = A
```

The statement in Line 13 is a function call to the function `funOne`. Now function `funOne` has three parameters (`a`, `b`, and `v`) and one local variable (`one`). Memory for the parameters and the local variable of function `funOne` is allocated. Because the formal parameter `b` is a reference parameter, it receives the address (memory location) of the corresponding actual parameter, which is `num2`. The other two formal parameters are value parameters, so they copy the values of their corresponding actual parameters. Just before the statement in Line 22 executes, the variables are as shown in Figure 6-10.

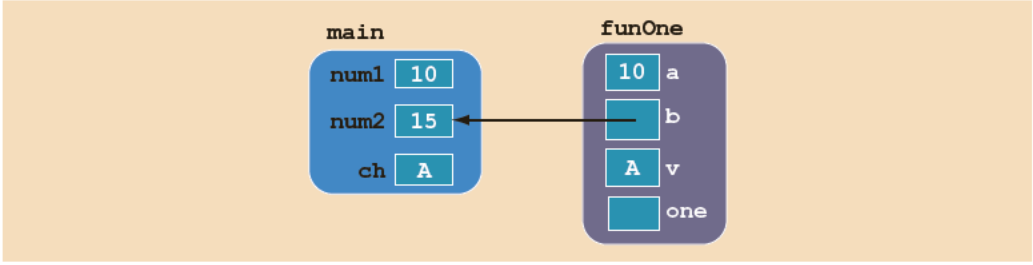
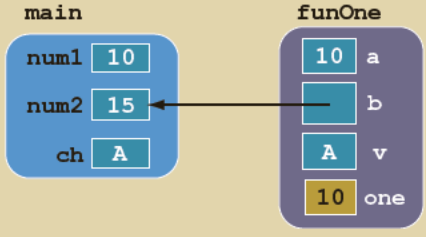
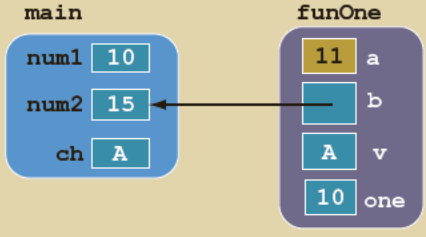
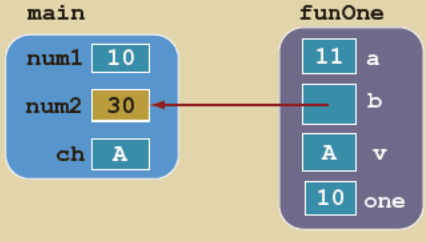
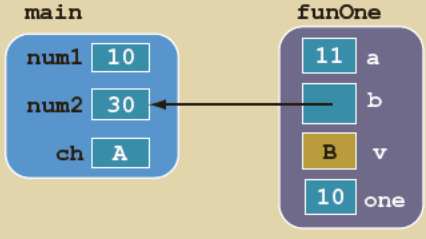



FIGURE 6-10 Values of the variables just before the statement in Line 22 executes

The following shows how the variables are manipulated after each statement from Lines 22 to 25 executes.

| St. in Line | Value of the Variables | Statement and Effect |
|---|---|---|
| 22 |  | `one = a;` Copy the value of `a` into one. |
| 23 |  | `a++;` Increment the value of `a` by `1`. |
| 24 |  | `b = b * 2;` Multiply the value of `b` by `2` and store the result in `b`. Because `b` is the reference parameter and contains the address of `num`, the value of `num` is updated. |
| 25 |  | `v = 'B';` Store `'B'` into `v`. |

The statement in Line 26 produces the following output:

```
Line 26: Inside funOne: a = 11, b = 30, v = B, and one = 10
```

After the statement in Line 26 executes, control goes back to Line 14 in `main` and the memory allocated for the variables of function `funOne` is deallocated. Figure 6-11 shows the values of the variables of the function `main`.

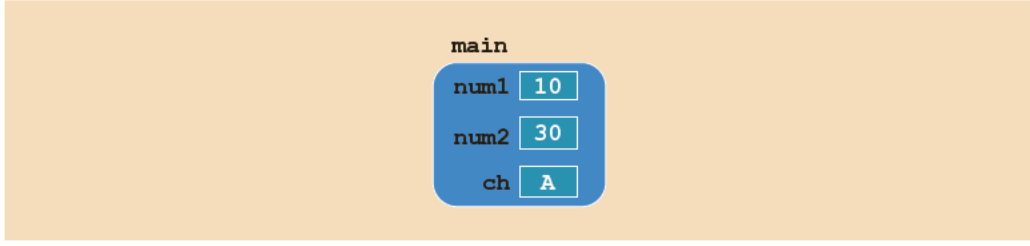



FIGURE 6-11 Values of the variables after the statement in Line 14

Line 14 produces the following output:

```
Line 14: After funOne: num1 = 10, num2 = 30, and ch = A
```

The statement in Line 15 is a function call to the function `funTwo`. Now `funTwo` has three parameters: `x`, `y`, and `w`. Also, `x` and `w` are reference parameters, and `y` is a value parameter. Thus, `x` receives the address of its corresponding actual parameter, which is `num2`, and `w` receives the address of its corresponding actual parameter, which is `ch`. The variable `y` copies the value `25` into its memory cell. Figure 6-12 shows the values before the statement in Line 30 executes.

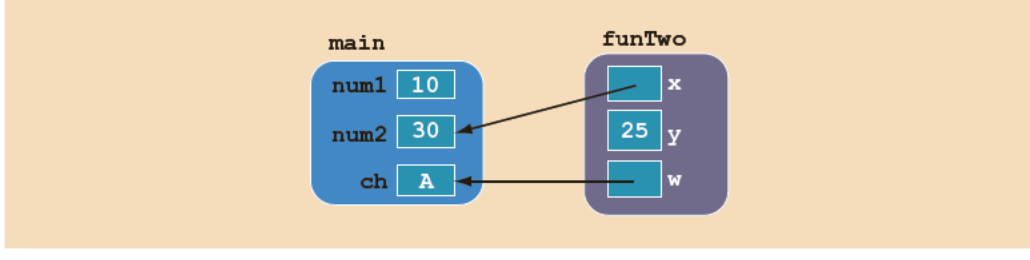
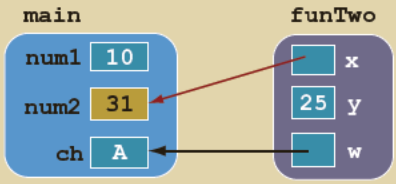
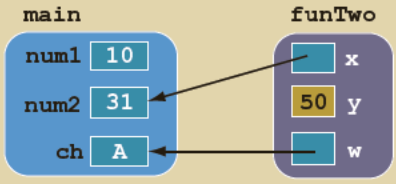
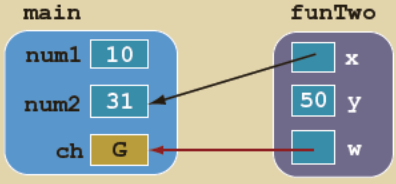



FIGURE 6-12 Values of the variables before the statement in Line 30 executes

The following shows how the variables are manipulated after each statement from Lines 30 to 32 executes.

| St. in Line | Value of the Variables | Statement and Effect |
|---|---|---|
| 30 |  | `x++;`<br>Increment the value of `x` by `1`. Because `x` is the reference parameter and contains the address of `num2`, the value of `num2` is incremented by `1`. |
| 31 |  | `y = y * 2;`<br>Multiply the value of `y` by `2` and store the result in `y`. |
| 32 |  | `w = 'G';`<br>Store `'G'` in `w`. Because `w` is the reference parameter and contains the address of `ch`, the value of `ch` is updated. |

Line 33 produces the following output:

```
Line 33: Inside funTwo: x = 31, y = 50, and w = G
```

After the statement in Line 33 executes, control goes to Line 16. The memory allocated for the variables of function `funTwo` is deallocated. The values of the variables of the function `main` are as shown in Figure 6-13.

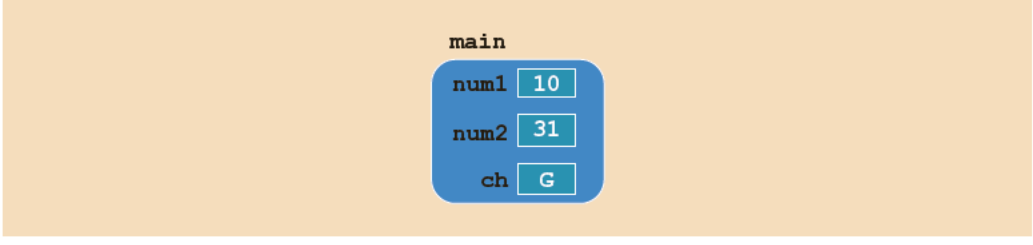



FIGURE 6-13 Values of the variables after the statement in Line 16

The statement in Line 16 produces the following output:

```
Line 16: After funTwo: num1 = 10, num2 = 31, and ch = G
```

After the statement in Line 16 executes, the program terminates.

## EXAMPLE 6-15

This example also shows how reference parameters manipulate actual parameters.

```
//Example 6-15: Reference and value parameters.
//Program: Makes you think.

#include <iostream>                                          //Line 1

using namespace std;                                         //Line 2

void addFirst(int& first, int& second);                      //Line 3
void doubleFirst(int one, int two);                          //Line 4
void squareFirst(int& ref, int val);                         //Line 5

int main()                                                   //Line 6
{                                                            //Line 7
    int num = 5;                                             //Line 8

    cout << "Line 9: Inside main: num = " << num
         << endl;                                            //Line 9

    addFirst(num, num);                                      //Line 10
    cout << "Line 11: Inside main after addFirst:"
         << " num = " << num << endl;                        //Line 11

    doubleFirst(num, num);                                   //Line 12
    cout << "Line 13: Inside main after "
         << "doubleFirst: num = " << num << endl;            //Line 13

    squareFirst(num, num);                                   //Line 14
    cout << "Line 15: Inside main after "
         << "squareFirst: num = " << num << endl;            //Line 15

    return 0;                                                //Line 16
}                                                            //Line 17

void addFirst(int& first, int& second)                       //Line 18
{                                                            //Line 19
    cout << "Line 20: Inside addFirst:  first = "
         << first << ", second = " << second << endl;        //Line 20

    first = first + 2;                                       //Line 21

    cout << "Line 22: Inside addFirst:  first = "
         << first << ", second = " << second << endl;        //Line 22

    second = second * 2;                                     //Line 23

    cout << "Line 24: Inside addFirst:  first = "
         << first << ", second = " << second << endl;        //Line 24
}                                                            //Line 25
```

```
void doubleFirst(int one, int two)                          //Line 26
{                                                           //Line 27
    cout << "Line 28: Inside doubleFirst:  one = "
         << one << ", two = " << two << endl;               //Line 28

    one = one * 2;                                          //Line 29

    cout << "Line 30: Inside doubleFirst:  one = "
         << one << ", two = " << two << endl;               //Line 30

    two = two + 2;                                          //Line 31

    cout << "Line 32: Inside doubleFirst:  one = "
         << one << ", two = " << two << endl;               //Line 32
}                                                           //Line 33

void squareFirst(int& ref, int val)                         //Line 34
{                                                           //Line 35
    cout << "Line 36: Inside squareFirst: ref = "
         << ref << ", val = " << val << endl;               //Line 36

    ref = ref * ref;                                        //Line 37

    cout << "Line 38: Inside squareFirst: ref = "
         << ref << ", val = " << val << endl;               //Line 38

    val = val + 2;                                          //Line 39

    cout << "Line 40: Inside squareFirst: ref = "
         << ref << ", val = " << val << endl;               //Line 40
}                                                           //Line 41
```

**Sample Run:**

```
Line 9: Inside main:  num = 5
Line 20: Inside addFirst:  first = 5, second = 5
Line 22: Inside addFirst:  first = 7, second = 7
Line 24: Inside addFirst:  first = 14, second = 14
Line 11: Inside main after addFirst: num = 14
Line 28: Inside doubleFirst:  one = 14, two = 14
Line 30: Inside doubleFirst:  one = 28, two = 14
Line 32: Inside doubleFirst:  one = 28, two = 16
Line 13: Inside main after doubleFirst: num = 14
Line 36: Inside squareFirst: ref = 14, val = 14
Line 38: Inside squareFirst: ref = 196, val = 14
Line 40: Inside squareFirst: ref = 196, val = 16
Line 15: Inside main after squareFirst: num = 196
```

Both parameters of the function `addFirst` are reference parameters, and both parameters of the function `doubleFirst` are value parameters. The statement:

```
addFirst(num, num);
```

in the function **main** (Line 10) passes the reference of **num** to both formal parameters **first** and **second** of the function **addFirst**, because the corresponding actual parameters for both formal parameters are the same. That is, the variables **first** and **second** refer to the same memory location, which is **num**. Figure 6-14 illustrates this situation.

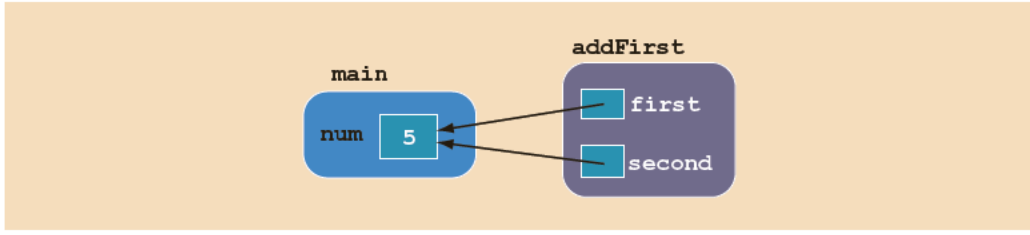



FIGURE 6-14 Parameters of the function **addFirst**

Any changes that **first** makes to its value immediately change the value of **second** and **num**. Similarly, any changes that **second** makes to its value immediately change **first** and **num**, because all three variables refer to the same memory location. (Note that **num** was initialized to **5**.)

The formal parameters of the function **doubleFirst** are value parameters. So the statement:

```
doubleFirst(num, num);
```

in the function **main** (Line 12) copies the value of **num** into **one** and **two** because the corresponding actual parameters for both formal parameters are the same. Figure 6-15 illustrates this scenario.

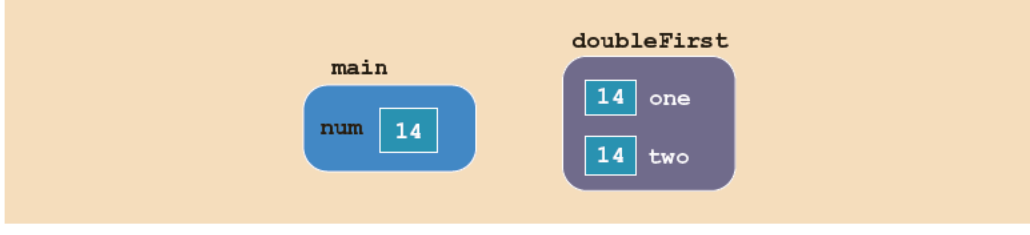



FIGURE 6-15 Parameters of the function **doubleFirst**

Because both **one** and **two** are value parameters, any changes that **one** makes to its value do not affect the values of **two** and **num**. Similarly, any changes that **two** makes to its value do not affect **one** and **num**. (Note that the value of **num** before the function **doubleFirst** executes is **14**.)

The formal parameter **ref** of the function **squareFirst** is a reference parameter, and the formal parameter **val** is a value parameter. The variable **ref** receives the address of its corresponding actual parameter, which is **num**, and the variable **val** copies the

value of its corresponding actual parameter, which is also **num**. Thus, both **num** and **ref** refer to the same memory location, which is **num**. Figure 6-16 illustrates this situation.

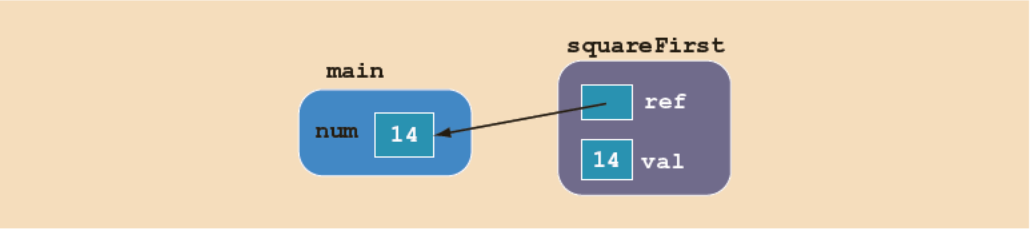



FIGURE 6-16 Parameters of the function **squareFirst**

Any changes that **ref** makes immediately change **num**. Any changes made by **val** do not affect **num**. (Note that the value of **num** before the function **squareFirst** executes is **14**.)

We recommend that you walk through the program in Example 6-15. The output shows the order in which the statements execute.



## Reference Parameters and Value-Returning Functions

Earlier in this chapter, in the discussion of value-returning functions, you learned how to use value parameters only. You can also use reference parameters in a value-returning function, although this approach is not recommended. By definition, a value-returning function returns a single value; this value is returned via the `return` statement. If a function needs to return more than one value, as a rule of good programming style, you should change it to a void function and use the appropriate reference parameters to return the values.

## Scope of an Identifier

The previous sections presented several examples of programs with user-defined functions. Identifiers are declared in a function heading, within a block, or outside a block. A question naturally arises: Are you allowed to access any identifier anywhere in the program? The answer is no. You must follow certain rules to access an identifier. The **scope** of an identifier refers to where in the program an identifier is accessible (visible). Recall that an identifier is the name of something in C++, such as a variable or function name.

This section examines the scope of an identifier. First, we define the following two terms:

**Local identifier:** Identifiers declared within a function (or block).

Local identifiers are not accessible outside of the function (block).

**Global identifier:** Identifiers declared outside of every function definition.

Also, C++ does not allow the nesting of functions. That is, you cannot include the definition of one function in the body of another function.

In general, the following rules apply when an identifier is accessed:

1. Global identifiers (such as variables) are accessible by a function or a block if
   a. The identifier is declared before the function definition (block),
   b. The function name is different than the identifier,
   c. All parameters of the function have names different than the name of the identifier, and
   d. All local identifiers (such as local variables) have names different than the name of the identifier.
2. (**Nested Block**) An identifier declared within a block is accessible
   a. Only within the block from the point at which it is declared until the end of the block, and
   b. By those blocks that are nested within that block if the nested block does not have an identifier with the same name as that of the outside block (the block that encloses the nested block).
3. The scope of a function name is similar to the scope of an identifier declared outside any block. That is, the scope of a function name is the same as the scope of a global variable.

Before considering an example to explain these scope rules, first note the scope of the identifier declared in the `for` statement. C++ allows the programmer to declare a variable in the initialization statement of the `for` statement. For example, the following `for` statement:

```
for (int count = 1; count < 10; count++)
     cout << count << endl;
```

declares the variable `count` and initializes it to `1`. The scope of the variable `count` is limited to only the body of the `for` loop.


This scope rule for the variable declared in a `for` statement may not apply to Standard C++, that is, non–ANSI/ISO Standard C++. In Standard C++, the scope of the variable declared in the `initialize` statement may extend from the point at which it is declared until the end of the block that immediately surrounds the `for` statement. (To be absolutely sure, check your compiler's documentation.)

The following C++ programming code helps illustrate the scope rules:

```
#include <iostream>

using namespace std;
```

```
const double RATE = 10.50;
int z;
double t;

void one(int x, char y);
void two(int a, int b, char x);
void three(int one, double y, int z);

int main()
{
    int num, first;
    double x, y, z;
    char name, last;
      .
      .
      .
    return 0;
}

void one(int x, char y)
{
      .
      .
      .
}

int w;

void two(int a, int b, char x)
{
   int count;
      .
      .
      .
}

void three(int one, double y, int z)
{
   char ch;
   int a;
     .
     .
     .
   //Block four
   {
       int x;
       char a;
          .
          .
   }//end Block four
      .
      .
      .
}
```

Table 6-2 summarizes the scope (visibility) of the identifiers.

**TABLE 6-2** Scope (Visibility) of the Identifiers

| Identifier | Visibility in `one` | Visibility in `two` | Visibility in `three` | Visibility in Block `four` | Visibility in `main` |
|---|---|---|---|---|---|
| **`RATE`** (before **`main`**) | Y | Y | Y | Y | Y |
| **`z`** (before **`main`**) | Y | Y | N | N | N |
| **`t`** (before **`main`**) | Y | Y | Y | Y | Y |
| **`main`** | Y | Y | Y | Y | Y |
| local variables of **`main`** | N | N | N | N | Y |
| **`one`** (function name) | Y | Y | N | N | Y |
| **`x`** (one's formal parameter) | Y | N | N | N | N |
| **`y`** (one's formal parameter) | Y | N | N | N | N |
| **`w`** (before function **`two`**) | N | Y | Y | Y | N |
| **`two`** (function name) | Y | Y | Y | Y | Y |
| **`a`** (**`two`**'s formal parameter) | N | Y | N | N | N |
| **`b`** (**`two`**'s formal parameter) | N | Y | N | N | N |
| **`x`** (**`two`**'s formal parameter) | N | Y | N | N | N |
| local variables of **`two`** | N | Y | N | N | N |
| **`three`** (function name) | Y | Y | Y | Y | Y |
| **`one`** (**`three`**'s formal parameter) | N | N | Y | Y | N |
| **`y`** (**`three`**'s formal parameter) | N | N | Y | Y | N |
| **`z`** (**`three`**'s formal parameter) | N | N | Y | Y | N |
| **`ch`** (**`three`**'s local variable) | N | N | Y | Y | N |
| **`a`** (**`three`**'s local variable) | N | N | Y | N | N |
| **`x`** (Block four's local variable) | N | N | N | Y | N |
| **`a`** (Block four's local variable) | N | N | N | Y | N |

Note that function **three** cannot call function **one**, because function **three** has a formal parameter named **one**. Similarly, the block marked **four** in function **three** cannot use the **int** variable **a**, which is declared in function **three**, because block **four** has an identifier named **a**.

Before closing this section, let us note the following about global variables:

1. Chapter 2 stated that C++ does not automatically initialize variables. However, some compilers initialize *global* variables to their default values. For example, if a global variable is of type **int**, **char**, or **double**, it is initialized to zero.
2. In C++, **::** is called the **scope resolution operator**. By using the scope resolution operator, a global variable declared before the definition of a function (block) can be accessed by the function (or block) even if the function (or block) has an identifier with the same name as the variable. In the preceding program, by using the scope resolution operator, the function **main** can refer to the global variable **z** as **::z**. Similarly, suppose that a global variable **t** is declared before the definition of the function—say, **funExample**. Then, **funExample** can access the variable **t** using the scope resolution operator even if **funExample** has an identifier **t**. Using the scope resolution operator, **funExample** refers to the variable **t** as **::t**. Also, in the preceding program, using the scope resolution operator, function **three** can call function **one**.
3. C++ provides a way to access a global variable declared after the definition of a function. In this case, the function must not contain any identifier with the same name as the global variable. In the preceding program, the global variable **w** is declared after the definition of function **one**. The function **one** does not contain any identifier named **w**; therefore, **w** can be accessed by function **one** only if you declare **w** as an **external variable** inside **one**. To declare **w** as an external variable inside function **one**, the function **one** must contain the following statement:

   ```
   extern int w;
   ```

   In C++, **extern** is a reserved word. The word **extern** in the above statement announces that **w** is a global variable declared elsewhere. Thus, when function **one** is called, no memory for **w**, as declared inside **one**, is allocated. In C++, external declaration also has another use, but it is not discussed in this book.

## Global Variables, Named Constants, and Side Effects

A C++ program can contain global variables and you might be tempted to make all of the variables in a program global variables so that you do not have to worry about what a function knows about which variable. Using global variables, however, has

side effects. If more than one function uses the same global variable and something goes wrong, it is difficult to discover what went wrong and where. Problems caused by global variables in one area of a program might be misunderstood as problems caused in another area.

For example, consider the following program:

```
//Global variable

#include <iostream>                                          //Line 1

using namespace std;                                         //Line 2

int t;    //global variable                                  Line 3

void funOne(int& a);                                         //Line 4

int main()                                                   //Line 5
{                                                            //Line 6
    t = 15;                                                  //Line 7

    cout << "Line 8: In main: t = " << t << endl;            //Line 8

    funOne(t);                                               //Line 9

    cout << "Line 10: In main after funOne: "
         << " t = " << t << endl;                            //Line 10

    return 0;                                                //Line 11
}                                                            //Line 12

void funOne(int& a)                                          //Line 13
{                                                            //Line 14
    cout << "Line 15: In funOne: a = " << a
         << " and t = " << t << endl;                        //Line 15

    a = a + 12;                                              //Line 16
    cout << "Line 17: In funOne: a = " << a
         << " and t = " << t << endl;                        //Line 17

    t = t + 13;                                              //Line 18

    cout << "Line 19: In funOne: a = " << a
         << " and t = " << t << endl;                        //Line 19
}                                                            //Line 20
```

**Sample Run:**

```
Line 8: In main: t = 15
Line 15: In funOne: a = 15 and t = 15
Line 17: In funOne: a = 27 and t = 27
Line 19: In funOne: a = 40 and t = 40
Line 10: In main after funOne: t = 40
```

This program has a variable `t` that is declared before the definition of any function. Because none of the functions has an identifier `t`, the variable `t` is accessible anywhere in the program. Also, the program consists of a void function with a reference parameter.

In Line 9, the function `main` calls the function `funOne`, and the actual parameter passed to `funOne` is `t`. So, `a`, the formal parameter of `funOne`, receives the address of `t`. Any changes that `a` makes to its value immediately change `t`. Because `t` can be directly accessed anywhere in the program, in Line 18, the function `funOne` changes the value of `t` by using `t` itself. Thus, you can manipulate the value of `t` by using either a reference parameter or `t` itself.

In the previous program, if the last value of `t` is incorrect, it would be difficult to determine what went wrong and in which part of the program. We strongly recommend that *you do not use global variables*; instead, use the appropriate parameters.

In the programs given in this book, we typically placed named constants before the function `main`, outside of every function definition. That is, the named constants we used are *global named constants*. Unlike global variables, global named constants have no side effects because their values cannot be changed during program execution. Moreover, placing a named constant in the beginning of the program can increase readability, even if it is used only in one function. If you need to later modify the program and change the value of a named constant, it will be easier to find if it is placed in the beginning of the program.



## EXAMPLE 6-16 (FACTORING A SECOND DEGREE POLYNOMIAL)

In an algebra course, one learns how to factor a polynomial by using various techniques. In this example, we write a program to factor a second-degree polynomial of the form $x^2 + bx + c$, that is, write $x^2 + bx + c = (x - u)(x - v)$. For simplicity, we restrict this program to factor polynomials, where $b$, $c$, $u$, and $v$ are integers. For example, $x^2 + 5x + 6 = (x + 2)(x + 3)$, $x^2 + 10x - 24 = (x + 12)(x - 2)$ and $x^2 - 25 = (x + 5)(x - 5)$.

It can be shown that the values of $u$ and $v$ are given by

$$u = \frac{-b + \sqrt{b^2 - 4c}}{2}$$

$$v = \frac{-b - \sqrt{b^2 - 4c}}{2}$$

If $b^2 - 4c < 0$, then $u$ and $v$ are complex numbers; if $b^2 - 4c > 0$ and $b^2 - 4c$ is not the square of an integer, then $u$ and $v$ are not integers. Also, if $b^2 - 4c$ is the square of an integer and $-b + \sqrt{b^2 - 4c}$ and $-b - \sqrt{b^2 - 4c}$ are not divisible by 2, then $u$ and $v$ are not integers. It follows that for $u$ and $v$ to be integers, $-b + \sqrt{b^2 - 4c}$ and

$-b - \sqrt{b^2 - 4c}$ must be divisible by 2. The following function takes as a parameter, the values of $b$ and $c$, and returns the values of $u$ and $v$ as well as indicating whether the polynomial is factorable.

```
void factorization(int b, int c, int& u1, int& v1, bool& isFactorable)
{
    double discriminant;
    int temp;

    isFactorable = true;

    discriminant = b * b - 4 * c;

    if (discriminant < 0)
        isFactorable = false;
    else
    {
        temp = static_cast<int>(sqrt(discriminant));

        if (temp * temp != discriminant)
            isFactorable = false;
        else
        {
            if (((-b + temp) % 2 != 0) || ((-b - temp) % 2 != 0))
                isFactorable = false;
            else
            {
                u1 = (-b + temp) / 2;
                v1 = (-b - temp) / 2;
            }
        }
    }
}
```

The following program shows how to use the function `factorization` in a program.

```
//Program: Second degree polynomial factorization

#include <iostream>
#include <cmath>

using namespace std;

void factorization(int b, int c, int& u1, int& v1, bool& isFactorable);

int main()
{
    int coeffOfX;
    int constantTerm;
    int u;
    int v;
    bool isPolynomialFactorable;
```

```
    cout << "Enter the coefficient of x: ";
    cin >> coeffOfX;
    cout << endl;

    cout << "Enter the constant term: ";
    cin >> constantTerm;
    cout << endl;

    factorization(coeffOfX, constantTerm, u, v,
                  isPolynomialFactorable);

    if (isPolynomialFactorable)
    {
        cout << "x^2";

        if (coeffOfX > 0)
            cout << " + " << coeffOfX << "x";
        else if (coeffOfX < 0)
            cout << " - " << abs(coeffOfX) << "x";

        if (constantTerm > 0)
            cout << " + " << constantTerm;
        else if (constantTerm < 0)
            cout << " - " << abs(constantTerm);

        cout << " = (x";
        if (u > 0)
            cout << " - " << u << ")(x";
        else if (u < 0)
            cout << " + " << abs(u) << ")(x";

        if (v > 0)
            cout << " - " << v << ")" << endl;
        else if (v < 0)
            cout << " + " << abs(v) << ")" << endl;
    }
    else
        cout << "The polynomial is not factorable." << endl;

    return 0;
}

//Place the definition of the function factorization here.
```

**Sample Runs:** In these sample runs, the user input is shaded.

**Sample Run 1:**

```
Enter the coefficient of x: 5
Enter the constant term: 6
x^2 + 5x + 6 = (x + 3)(x + 2)
```

**Sample Run 2:**

```
Enter the coefficient of x: 0
Enter the constant term: -25
x^2 - 25 = (x - 5)(x + 5)
```

**Sample Run 3:**

```
Enter the coefficient of x: 8
Enter the constant term: 16
x^2 + 8x + 16 = (x + 4)(x + 4)
```

**Sample Run 4:**

```
Enter the coefficient of x: -13
Enter the constant term: 20
The polynomial is not factorable.
```

## EXAMPLE 6-17 (MENU-DRIVEN PROGRAM)

The following is an example of a menu-driven program. When the program executes, it gives the user a list of choices to choose from. This program further illustrates how value and reference parameters work. It converts length from feet and inches to meters and centimeters and vice versa. The program contains three functions: `showChoices`, `feetAndInchesToMetersAndCent`, and `metersAndCentToFeetAndInches`. The function `showChoices` informs the user how to use the program. The user has the choice to run the program as long as the user wishes.

```
//Menu driven program.

#include <iostream>

using namespace std;

const double CONVERSION = 2.54;
const int INCHES_IN_FOOT = 12;
const int CENTIMETERS_IN_METER = 100;

void showChoices();

void feetAndInchesToMetersAndCent(int f, int in,
                                  int& mt, int& ct);
void metersAndCentTofeetAndInches(int mt, int ct,
                                  int& f, int& in);

int main()
{
    int feet, inches;
    int meters, centimeters;
    int choice;

    do
    {
        showChoices();
        cin >> choice;
        cout << endl;
```

```
        switch (choice)
        {
        case 1:
            cout << "Enter feet and inches: ";
            cin >> feet >> inches;
            cout << endl;
            feetAndInchesToMetersAndCent(feet, inches,
                                         meters, centimeters);
            cout << feet << " feet(foot), "
                 << inches << " inch(es) = "
                 << meters << " meter(s), "
                 << centimeters << " centimeter(s)." << endl;
            break;

        case 2:
            cout << "Enter meters and centimeters: ";
            cin >> meters >> centimeters;
            cout << endl;
            metersAndCentTofeetAndInches(meters, centimeters,
                                         feet, inches);
            cout << meters << " meter(s), "
                 << centimeters << " centimeter(s) = "
                 << feet << " feet(foot), "
                 << inches << " inch(es)."
                 << endl;
            break;

        case 99:
            break;

        default:
            cout << "Invalid input." << endl;
        }
    }
    while (choice != 99);

    return 0;
 }

void showChoices()
{
    cout << "Enter--" << endl;
    cout << "1: To convert from feet and inches to meters "
         << "and centimeters." << endl;
    cout << "2: To convert from meters and centimeters to feet "
         << "and inches." << endl;
    cout << "99: To quit the program." << endl;
}

void feetAndInchesToMetersAndCent(int f, int in,
                                  int& mt, int& ct)
```

```
{
    int inches;

    inches = f * INCHES_IN_FOOT + in;
    ct = static_cast<int>(inches * CONVERSION);
    mt = ct / CENTIMETERS_IN_METER;
    ct = ct % CENTIMETERS_IN_METER;
}

void metersAndCentTofeetAndInches(int mt, int ct,
                                  int& f, int& in)
{
    int centimeters;

    centimeters = mt * CENTIMETERS_IN_METER + ct;
    in = static_cast<int>(centimeters / CONVERSION);
    f = in / INCHES_IN_FOOT;
    in = in % INCHES_IN_FOOT;
}
```

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

```
Enter--
1: To convert from feet and inches to meters and centimeters.
2: To convert from meters and centimeters to feet and inches.
99: To quit the program.
2

Enter meters and centimeters: 6 28

6 meter(s), 28 centimeter(s) = 20 feet(foot), 7 inch(es).
Enter--
1: To convert from feet and inches to meters and centimeters.
2: To convert from meters and centimeters to feet and inches.
99: To quit the program.
1

Enter feet and inches: 18 7

18 feet(foot), 7 inch(es) = 5 meter(s), 66 centimeter(s).
Enter--
1: To convert from feet and inches to meters and centimeters.
2: To convert from meters and centimeters to feet and inches.
99: To quit the program.
99
```

The `do...while` loop in the function `main` continues to execute as long as the user has not entered `99`, which allows the user to run the program as long as the user wishes. The preceding output is self-explanatory.

# Static and Automatic Variables

The variables discussed so far have followed two simple rules:

1. Memory for global variables remains allocated as long as the program executes.
2. Memory for a variable declared within a block is allocated at block entry and deallocated at block exit. For example, memory for the formal parameters and local variables of a function is allocated when the function is called and deallocated when the function exits.

A variable for which memory is allocated at block entry and deallocated at block exit is called an **automatic variable**. A variable for which memory remains allocated as long as the program executes is called a **static variable**. Global variables are static variables, and by default, variables declared within a block are automatic variables. You can declare a static variable within a block by using the reserved word `static`. The syntax for declaring a static variable is:

```
static dataType identifier;
```



The statement:

```
static int x;
```

declares `x` to be a static variable of type `int`.

Static variables declared within a block are local to the block, and their scope is the same as that of any other local identifier of that block.

Most compilers initialize `static` variables to their default values. For example, `static int` variables are initialized to `0`. However, it is a good practice to initialize `static` variables yourself, especially if the initial value is not the default value. In this case, `static` variables are initialized when they are declared. The statement:

```
static int x = 0;
```

declares `x` to be a static variable of type `int` and initializes `x` to `0`, the first time the function is called.

## EXAMPLE 6-18

The following program shows how static and automatic variables behave.

```
//Program: Static and automatic variables
#include <iostream>

using namespace std;

void test();
```

```
int main()
{
    int count;

    for (count = 1; count <= 5; count++)
        test();

    return 0;
}

void test()
{
    static int x = 0;
    int y = 10;

    x = x + 2;
    y = y + 1;

    cout << "Inside test x = " << x << " and y = "
         << y << endl;
}
```

**Sample Run:**

```
Inside test x = 2 and y = 11
Inside test x = 4 and y = 11
Inside test x = 6 and y = 11
Inside test x = 8 and y = 11
Inside test x = 10 and y = 11
```

In the function `test`, `x` is a `static` variable initialized to `0`, and `y` is an automatic variable initialized to `10`. The function `main` calls the function `test` five times. Memory for the variable `y` is allocated every time the function `test` is called and deallocated when the function exits. Thus, every time the function `test` is called, it prints the same value for `y`. However, because `x` is a static variable, memory for `x` remains allocated as long as the program executes. The variable `x` is initialized once to `0`, the first time the function is called. The subsequent calls of the function `test` use the value `x` had when the program last left (executed) the function `test`.

---

Because memory for static variables remains allocated between function calls, static variables allow you to use the value of a variable from one function call to another function call. Even though you can use global variables if you want to use certain values from one function call to another, the local scope of a static variable prevents other functions from manipulating its value.

# Debugging: Using Drivers and Stubs

In this and the previous chapters, you learned how to write functions to divide a problem into subproblems, solve each subproblem, and then combine the functions to form the complete program to get a solution of the problem. A program may contain a number of functions. In a complex program, usually, when a function is written, it is tested and debugged alone. You can write a separate program to test the function. The program that tests a function is called a **driver** program. For example, the program in Example 6-15 contains functions to convert the length from feet and inches to meters and centimeters and vice versa. Before writing the complete program, you could write separate driver programs to make sure that each function is working properly.

Sometimes, the results calculated by one function are needed in another function. In that case, the function that depends on another function cannot be tested alone. For example, consider the following program that determines the time needed to fill a swimming pool.



```
#include <iostream>
#include <iomanip>

using namespace std;

const double GALLONS_IN_A_CUBIC_FOOT = 7.48;

double poolCapacity(double len, double wid, double dep);
void poolFillTime(double len, double wid, double dep,
                  double fRate, int& fTime);
void print(int fTime);

int main()
{
    double length, width, depth;
    double fillRate;
    int fillTime;

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

    cout << "Enter the length, width, and the depth of the "
         << "pool, (in feet): ";
    cin >> length >> width >> depth;
    cout << endl;

    cout << "Enter the rate of the water, (in gallons per minute): ";
    cin >> fillRate;
    cout << endl;

    poolFillTime(length, width, depth, fillRate, fillTime);
    print(fillTime);

   return 0;
}
```

```
double poolCapacity(double len, double wid, double dep)
{
    double volume;
    double poolWaterCapacity;

    volume = len * wid * dep;
    poolWaterCapacity = volume * GALLONS_IN_A_CUBIC_FOOT;

    return poolWaterCapacity;
}

void poolFillTime(double len, double wid, double dep,
                  double fRate, int& fTime)
{
    double poolWaterCapacity;

    poolWaterCapacity = poolCapacity(len, wid, dep);
    fTime = static_cast<int> (poolWaterCapacity / fRate + 0.5);
}

void print(int fTime)
{
    cout << "The time to fill the pool is approximately: "
         << fTime / 60 << " hour(s) and " << fTime % 60
         << " minute(s)." << endl;
}
```

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

```
Enter the length, width, and the depth of the pool, (in feet): 30 15 10

Enter the rate of the water, (in gallons per minute): 100

The time to fill the pool is approximately: 5 hour(s) and 37 minute(s).
```

As you can see, the program contains the function `poolCapacity` to find the amount of water needed to fill the pool, the function `poolFillTime` to find the time to fill the pool, and some other functions. Now, to calculate the time to fill the pool, you must know the amount of the water needed and the rate at which the water is released in the pool. Because the results of the function `poolCapacity` are needed in the function `poolFillTime`, the function `poolFillTime` cannot be tested alone. Does this mean that we must write the functions in a specific order? Not necessarily, especially when different people are working on different parts of the program. In situations such as these, we use function stubs. A function **stub** is a function that is not fully coded. For a void function, a function stub might consist of only a function header and a set of empty braces, `{}`, and for a value-returning function it might contain only a return statement with a plausible and easy to use return value. For example, the function stub for the function `poolCapacity` can be:

```
double poolCapacity(double len, double wid, double dep)
{
    return 1000.00;
}
```

This allows the function **poolCapacity** to be called while the program is being coded. Ultimately, the stub for function **poolCapacity** is replaced with a function that properly calculates the amount of water needed to fill the pool based on the values of the parameters. In the meantime, the function stub allows work to continue on other parts of the program that call the function **poolCapacity**.

Because a stub looks a lot like a viable function, it must be properly documented in a way that would remind you to replace it with the actual definition. If you forget to replace a stub with the actual definition, the program will generate erroneous results, which sometimes might be embarrassing.

Before we look at some programming examples, another concept about functions is worth mentioning: function overloading.



## Function Overloading: An Introduction

In a C++ program, several functions can have the same name. This is called **function overloading**, or **overloading a function name**. Before we state the rules to overloading a function, let us define the following:

Two functions are said to have **different formal parameter lists** if both functions have

- A different number of formal parameters or
- The same number of formal parameters and the data types of the formal parameters, in the order listed, differ in at least one position.

For example, consider the following function headings:

```
void functionOne(int x)
void functionTwo(int x, double y)
void functionThree(double y, int x)
int functionFour(char ch, int x, double y)
int functionFive(char ch, int x, string name)
```

These functions all have different formal parameter lists.

Now consider the following function headings:

```
void functionSix(int x, double y, char ch)
void functionSeven(int one, double u, char firstCh)
```

The functions **functionSix** and **functionSeven** both have three formal parameters, and the data type of the corresponding parameters is the same. Therefore, these functions have the same formal parameter list. Note that it is the *data types* and not the *parameter names* or the *return type* that are examined.

To overload a function name, any two definitions of the function must have different formal parameter lists.

**Function overloading:** Creating several functions with the same name.

The **signature** of a function consists of the function name and its formal parameter list. Two functions have different signatures if they have either different names or different formal parameter lists. (Note that the signature of a function does not include the return type of the function.)

If a function's name is overloaded, then all of the functions in the set have the same name. Therefore, all the functions in the overloaded set must have different formal parameter lists. Thus, the following function headings correctly overload the function `functionXYZ`:

```
void functionXYZ()
void functionXYZ(int x, double y)
void functionXYZ(double one, int y)
void functionXYZ(int x, double y, char ch)
```

Consider the following function headings to overload the function `functionABC`:

```
void functionABC(int x, double y)
int functionABC(int x, double y)
```

Both of these function headings have the same name and same formal parameter list. Therefore, these function headings to overload the function `functionABC` are incorrect. In this case, the compiler will generate a syntax error. (Notice that the return types of these function headings are different.)

If a function is overloaded, then in a call to that function the formal parameter list of the function determines which function to execute.

Some authors define the signature of a function as the formal parameter list, and some consider the entire heading of the function as its signature. However, in this book, the signature of a function consists of the function's name and its formal parameter list. If the function's names are different, then, of course, the compiler would have no problem in identifying which function is called, and it will correctly translate the code. However, if a function's name is overloaded, then, as noted, the function's formal parameter list determines which function's body executes.

Suppose you need to write a function that determines the larger of two items. Both items can be integers, floating-point numbers, characters, or strings. You could write several functions as follows:

```
int largerInt(int x, int y);
char largerChar(char first, char second);
double largerDouble(double u, double v);
string largerString(string first, string second);
```

The function `largerInt` determines the larger of two integers; the function `largerChar` determines the larger of two characters, and so on. All of these functions perform similar operations. Instead of giving different names to these functions, you can use the

same name—say, `larger`—for each function; that is, you can overload the function `larger`. Thus, you can write the previous function prototypes simply as follows:

```
int larger(int x, int y);
char larger(char first, char second);
double larger(double u, double v);
string larger(string first, string second);
```

If the call is `larger(5, 3)`, for example, the version having `int` parameters is executed. If the call is `larger('A', '9')`, the version having `char` parameters is executed, and so on.

Function overloading is used when you have the same action for different sets of data. Of course, for function overloading to work, you must give a separate definition for each function.

## Functions with Default Parameters




This section is not needed until Chapter 10.

This section discusses functions with default parameters. Recall that when a function is called, the number of actual and formal parameters must be the same. C++ relaxes this condition for functions with default parameters. You specify the value of a default parameter when the function name appears for the first time, usually in the prototype. In general, the following rules apply for functions with default parameters:

- If you do not specify the value of a default parameter, the default value is used for that parameter.
- All of the default parameters must be the far-right parameters of the function.
- Suppose a function has more than one default parameter. In a function call, if a value to a default parameter is not specified, then you must omit all of the arguments to its right.
- Default values can be constants, global variables, or function calls.
- The caller has the option of specifying a value other than the default for any default parameter.
- You cannot assign a constant value as a default value to a reference parameter.

Consider the following function prototype:

```
void funcExp(int t, int u, double v, char w = 'A', int x = 67,
             char y = 'G', double z = 78.34);
```

The function **funcExp** has seven parameters. The parameters **w**, **x**, **y**, and **z** are default parameters. If no values are specified for **w**, **x**, **y**, and **z** in a call to the function **funcExp**, their default values are used.

Suppose you have the following statements:

```
int a, b;
char ch;
double d;
```

The following function calls are legal:

1. `funcExp(a, b, d);`
2. `funcExp(a, 15, 34.6, 'B', 87, ch);`
3. `funcExp(b, a, 14.56, 'D');`

In statement 1, the default values of **w**, **x**, **y**, and **z** are used. In statement 2, the default value of **w** is replaced by `'B'`, the default value of **x** is replaced by `87`, the default value of **y** is replaced by the value of **ch**, and the default value of **z** is used. In statement 3, the default value of **w** is replaced by `'D'`, and the default values of **x**, **y**, and **z** are used.

The following function calls are illegal:

1. `funcExp(a, 15, 34.6, 46.7);`
2. `funcExp(b, 25, 48.76, 'D', 4567, 78.34);`

In statement 1, because the value of **w** is omitted, all other default values must be omitted.

In statement 2, because the value of **y** is omitted, the value of **z** should be omitted, too.

The following are illegal function prototypes with default parameters:

1. `void funcOne(int x, double z = 23.45, char ch, int u = 45);`
2. `int funcTwo(int length = 1, int width, int height = 1);`
3. `void funcThree(int x, int& y = 16, double z = 34);`

In statement 1, because the second parameter **z** is a default parameter, all other parameters after **z** must also be default parameters. In statement 2, because the first parameter is a default parameter, all parameters must be default parameters. In statement 3, a constant value cannot be assigned to **y** because **y** is a reference parameter.

Example 6-19 further illustrates functions with default parameters.

## EXAMPLE 6-19

```
#include <iostream>                                    //Line 1
#include <iomanip>                                     //Line 2

using namespace std;                                   //Line 3
```

```
int volume(int l = 1, int w = 1, int h = 1);          //Line 4
void funcOne(int& x, double y = 12.34, char z = 'B'); //Line 5

int main()                                            //Line 6
{                                                     //Line 7
    int a = 23;                                       //Line 8
    double b = 48.78;                                 //Line 9
    char ch = 'M';                                    //Line 10

    cout << fixed << showpoint << setprecision(2);    //Line 11

    cout << "Line 12: a = " << a << ", b = "
         << b  << ", ch = " << ch << endl;            //Line 12
    cout << "Line 13: Volume = " << volume()
         << endl;                                     //Line 13
    cout << "Line 14: Volume = " << volume(5, 4)
         << endl;                                     //Line 14
    cout << "Line 15: Volume = " << volume(34)
         << endl;                                     //Line 15
    cout << "Line 16: Volume = "
         << volume(6, 4, 5) << endl;                  //Line 16
    funcOne(a);                                       //Line 17
    funcOne(a, 42.68);                                //Line 18
    funcOne(a, 34.65, 'Q');                           //Line 19
    cout << "Line 20: a = " << a << ", b = "
         << b << ", ch = " << ch << endl;             //Line 20

    return 0;                                         //Line 21
}                                                     //Line 22

int volume(int l, int w, int h)                       //Line 23
{                                                     //Line 24
    return l * w * h;                                 //Line 25
}                                                     //Line 26

void funcOne(int& x, double y, char z)                //Line 27
{                                                     //Line 28
    x = 2 * x;                                        //Line 29
    cout << "Line 30: x = " << x << ", y = "
         << y << ", z = " << z << endl;               //Line 30
}                                                     //Line 31
```

**Sample Run:**

```
Line 12: a = 23, b = 48.78, ch = M
Line 13: Volume = 1
Line 14: Volume = 20
Line 15: Volume = 34
Line 16: Volume = 120
Line 30: x = 46, y = 12.34, z = B
Line 30: x = 92, y = 42.68, z = B
Line 30: x = 184, y = 34.65, z = Q
Line 20: a = 184, b = 48.78, ch = M
```

NOTE

In programs in this book, and as is recommended, the definition of the function **main** is placed before the definition of any user-defined functions. You must, therefore, specify the default value for a parameter in the function prototype and in the function prototype only, *not* in the function definition because this must occur at the first appearance of the function name.

## PROGRAMMING EXAMPLE: Classify Numbers

In this example, we use functions to rewrite the program that determines the number of odds and evens from a given list of integers. This program was first written in Chapter 5.

The main algorithm remains the same:

1. Initialize the variables, **zeros**, **odds**, and **evens** to **0**.
2. Read a number.
3. If the number is even, increment the even count, and if the number is also zero, increment the zero count; otherwise, increment the odd count.
4. Repeat Steps 2 and 3 for each number in the list.

The main parts of the program are: initialize the variables, read and classify the numbers, and then output the results. To simplify the function **main** and further illustrate parameter passing, the program includes:

- A function **initialize** to initialize the variables, such as **zeros**, **odds**, and **evens**.
- A function **getNumber** to get the number.
- A function **classifyNumber** to determine whether the number is odd or even (and whether it is also zero). This function also increments the appropriate count.
- A function **printResults** to print the results.

Let us now describe each of these functions.

**initialize**

The function **initialize** initializes variables to their initial values. The variables that we need to initialize are **zeros**, **odds**, and **evens**. As before, their initial values are all zero. Clearly, this function has three parameters. Because the values of the formal parameters initializing these variables must be passed outside of the function, these formal parameters must be reference parameters. Essentially, this function is:

```
void initialize(int& zeroCount, int& oddCount, int& evenCount)
{
    zeroCount = 0;
    oddCount = 0;
    evenCount = 0;
}
```

**getNumber**

The function **getNumber** reads a number and then passes this number to the function **main**. Because you need to pass only one number, this function has only one parameter. The formal parameter of this (void) function must be a reference parameter because the number read is passed outside of the function. Essentially, this function is:

```
void getNumber(int& num)
{
    cin >> num;
}
```

You can also write the function **getNumber** as a value-returning function. See the note at the end of this programming example.

**classify Number**

The function **classifyNumber** determines whether the number is odd or even, and if the number is even, it also checks whether the number is zero. It also updates the values of some of the variables, **zeros**, **odds**, and **evens**. This function needs to know the number to be analyzed; therefore, the number must be passed as a parameter. Because this function also increments the appropriate count, the variables (that is, **zeros**, **odds**, and **evens** declared in **main**) holding the counts must be passed as parameters to this function. Thus, this function has four parameters.

Because the number will only be analyzed and not altered, you need to pass only its value. Thus, the formal parameter corresponding to this variable is a value parameter. After analyzing the number, this function increments the values of some of the variables, **zeros**, **odds**, and **evens**. Therefore, the formal parameters corresponding to these variables must be reference parameters. The algorithm to analyze the number and increment the appropriate count is the same as before. The definition of this function is:

```
void classifyNumber(int num, int& zeroCount, int& oddCount,
                    int& evenCount)
{
    switch (num % 2)
    {
    case 0:
        evenCount++;
        if (num == 0)
            zeroCount++;
        break;
   case 1:
   case -1:
       oddCount++;
   } //end switch
} //end classifyNumber
```

**print Results**

The function **printResults** prints the final results. To print the results (that is, the number of zeros, odds, and evens), this function must have access to the values of the variables, **zeros**, **odds**, and **evens** declared in the function **main**. Therefore, this function has three parameters. Because this function doesn't change the values

of the variables but only prints them, the formal parameters are value parameters. The definition of this function is:

```
void printResults(int zeroCount, int oddCount, int evenCount)
{
    cout << "There are " << evenCount << " evens, "
         << "which includes " << zeroCount << " zeros"
         << endl;

    cout << "The number of odd numbers is: " << oddCount
         << endl;
} //end printResults
```

MAIN ALGORITHM

We now give the main algorithm and show how the function `main` calls these functions.

1. Call the function `initialize` to initialize the variables.
2. Prompt the user to enter 20 numbers.
3. For each number in the list:
   a. Call the function `getNumber` to read a number.
   b. Output the number.
   c. Call the function `classifyNumber` to classify the number and increment the appropriate count.
4. Call the function `printResults` to print the final results.

## COMPLETE PROGRAM LISTING

```
//*************************************************************
// Author: D.S. Malik
//
// Program: Classify Numbers
// This program reads 20 numbers and outputs the number of
// zeros, odd, and even numbers.
//*************************************************************

#include <iostream>
#include <iomanip>

using namespace std;

const int N = 20;

    //Function prototypes
void initialize(int& zeroCount, int& oddCount, int& evenCount);
void getNumber(int& num);
void classifyNumber(int num, int& zeroCount, int& oddCount,
                    int& evenCount);
void printResults(int zeroCount, int oddCount, int evenCount);
```

```
int main ()
{
        //Variable declaration
    int counter; //loop control variable
    int number;  //variable to store the new number
    int zeros;   //variable to store the number of zeros
    int odds;    //variable to store the number of odd integers
    int evens;   //variable to store the number of even integers

    initialize(zeros, odds, evens);                      //Step 1

    cout << "Please enter " << N << " integers."
         << endl;                                        //Step 2
    cout << "The numbers you entered are: " << endl;

    for (counter = 1; counter <= N; counter++)           //Step 3
    {
        getNumber(number);                               //Step 3a
        cout << number << " ";                           //Step 3b
        classifyNumber(number, zeros, odds, evens);      //Step 3c
    } // end for loop

    cout << endl;

    printResults(zeros, odds, evens);                    //Step 4

    return 0;
}

void initialize(int& zeroCount, int& oddCount, int& evenCount)
{
    zeroCount = 0;
    oddCount = 0;
    evenCount = 0;
}

void getNumber(int& num)
{
    cin >> num;
}

void classifyNumber(int num, int& zeroCount, int& oddCount,
                    int& evenCount)
{
    switch (num % 2)
    {
    case 0:
        evenCount++;
        if (num == 0)
            zeroCount++;
        break;
```

```
    case 1:
    case -1:
        oddCount++;
    } //end switch
} //end classifyNumber

void printResults(int zeroCount, int oddCount, int evenCount)
{
    cout << "There are " << evenCount << " evens, "
         << "which includes " << zeroCount << " zeros"
         << endl;
    cout << "The number of odd numbers is: " << oddCount
         << endl;
} //end printResults
```

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

```
Please enter 20 integers.
The numbers you entered are:
18 0 7 5 17 0 0 24 88 36 0 9 31 26 62 0 1 92 55 0
18 0 7 5 17 0 0 24 88 36 0 9 31 26 62 0 1 92 55 0
There are 13 evens, which includes 6 zeros
The number of odd numbers is: 7
```

NOTE

In the previous program, because the data is assumed to be input from the standard input device (the keyboard) and the function `getNumber` returns only one value, you can also write the function `getNumber` as a value-returning function. If written as a value-returning function, the definition of the function `getNumber` is:

```
int getNumber()
{
    int num;

    cin >> num;

    return num;
}
```

In this case, the statement (function call):

```
getNumber(number);
```

in the function `main` should be replaced by the statement:

```
number = getNumber();
```

Of course, you also need to change the function prototype.

## PROGRAMMING EXAMPLE: Data Comparison


This programming example illustrates the following:

- How to read data from more than one file in the same program.
- How to send output to a file.
- How to generate bar graphs.
- With the help of functions and parameter passing, how to use the same program segment on different (but similar) sets of data.
- How to use structured design to solve a problem and how to perform parameter passing.

This program is broken into two parts. First, you learn how to read data from more than one file. Second, you learn how to generate bar graphs.



Two groups of students at a local university are enrolled in certain special courses during the summer semester. The courses are offered for the first time and are taught by different teachers. At the end of the semester, both groups are given the same tests for the same courses, and their scores are recorded in separate files. The data in each file is in the following form:

```
courseNo score1, score2, ..., scoreN -999
courseNo score1, score2, ..., scoreM -999
.
.
.
```

Let us write a program that finds the average course score for each course for each group. The output is of the following form:

```
Course No   Group No      Course Average
  CSC           1              83.71
                2              80.82

  ENG           1              82.00
                2              78.20

    .
    .
    .
Avg for group 1: 82.04
Avg for group 2: 82.01
```

**Input** Because the data for the two groups are recorded in separate files, the input data appears in two separate files.

**Output** As shown above.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

Reading input data from both files is straightforward. Suppose the data is stored in the file **group1.txt** for group 1 and file **group2.txt** for group 2. After processing the data for one group, we can process the data for the second group for the same course and continue until we run out of data. Processing data for each course is similar and is a two-step process:

1. a. Sum the scores for the course.
   b. Count the number of students in the course.
   c. Divide the total score by the number of students to find the course average.
2. Output the results.

We are comparing only the averages of the corresponding courses in each group, and the data in each file is ordered according to course ID. To ensure that only the averages of the corresponding courses are compared, we compare the course IDs for each group. If the corresponding course IDs are not the same, we output an error message and terminate the program.

This discussion suggests that we should write a function, **calculateAverage**, to find the course average. We should also write another function, **printResult**, to output the data in the form given. By passing the appropriate parameters, we can use the same functions, **calculateAverage** and **printResult**, to process each course's data for both groups. (In the second part of the program, we modify the function **printResult**.)

The preceding discussion translates into the following algorithm:

1. Initialize the variables.
2. Get the course IDs for group 1 and group 2.
3. If the course IDs are different, print an error message and exit the program.
4. Calculate the course averages for group 1 and group 2.
5. Print the results in the form given above.
6. Repeat Steps 2 through 5 for each course.
7. Print the final results.

Variables (Function **main**)

The preceding discussion suggests that the program needs the following variables for data manipulation in the function **main**:

```
string courseId1;      //course ID for group 1
string courseId2;      //course ID for group 2
int numberOfCourses;
double avg1;           //average for a course in group 1
double avg2;           //average for a course in group 2
double avgGroup1;      //average group 1
double avgGroup2;      //average group 2
ifstream group1;       //input stream variable for group 1
ifstream group2;       //input stream variable for group 2

ofstream outfile;      //output stream variable
```

Next, we discuss the functions **calculateAverage** and **printResult**. Then, we will put the function **main** together.

### calculateAverage

This function calculates the average for a course. Because the input is stored in a file and the input file is opened in the function **main**, we must pass the **ifstream** variable associated with the input file to this function. Furthermore, after calculating the course average, this function must pass the course average to the function **main**. Therefore, this function has two parameters, and both parameters must be reference parameters.

To find the course average, we must first find the sum of all scores for the course and the number of students who took the course and then divide the sum by the number of students. Thus, we need a variable to find the sum of the scores, a variable to count the number of students, and a variable to read and store a score. Of course, we must initialize the variable to find the sum and the variable to count the number of students to zero.

### Local Variables: (Function calculateAverage)

In the previous discussion of data manipulation, we identified three variables for the function **calculateAverage**:

```
double totalScore = 0.0;
int numberOfStudents = 0;
int score;
```

The above discussion translates into the following algorithm for the function **calculateAverage**:

1. Declare and initialize variables.
2. Get the (next) course score, **score**.
3. **while** the **score** is not **-999**
   a. Update **totalScore** by adding the course score.
   b. Increment **numberOfStudents** by 1.
   c. Get the (next) course score, **score**.
4. **courseAvg = totalScore / numberOfStudents;**

We are now ready to write the definition of the function **calculateAverage**.

```
void calculateAverage(ifstream& inp, double& courseAvg)
{
    double totalScore = 0.0;
    int numberOfStudents = 0;
    int score;

    inp >> score;
    while (score != -999)
    {
        totalScore = totalScore + score;
        numberOfStudents++;
        inp >> score;
    } //end while
```

```
    courseAvg = totalScore / numberOfStudents;
} //end calculate Average
```

**print Result**

The function **printResult** prints the group's course ID, group number, and course average. The output is stored in a file. So we must pass four parameters to this function: the **ofstream** variable associated with the output file, the group number, the course ID, and the course average for the group. The **ofstream** variable must be passed by reference. Because the function uses only the values of the other variables, the remaining three parameters should be value parameters. Also, from the output, it is clear that we print the course ID only before the group number.

In pseudocode, the algorithm is:

```
if (group number == 1)
    print course ID
else
    print a blank

print group number and course average
```

The definition of the function **printResult** follows:

```
void printResult(ofstream& outp, string courseID, int groupNo,
                 double avg)
{
    if (groupNo == 1)
       outp << "  " << courseID << "   ";
    else
       outp << "        ";

    outp << setw(8) << groupNo << setw(17) << avg << endl;
}//end printResult
```

Now that we have designed and defined the functions **calculateAverage** and **printResult**, we can describe the algorithm for the function **main**. Before outlining the algorithm, however, we note the following: It is quite possible that in both input files, the data is ordered according to the course IDs, but one file might have one or more additional courses that are not in the other file. We do not discover this error until after we have processed both files and discovered that one file has unprocessed data. Make sure to check for this error before printing the final answer—that is, the averages for group 1 and group 2.

**MAIN ALGORITHM: Function main**

1. Declare the variables (local declaration).
2. Open the input files.
3. Print a message if you are unable to open a file and terminate the program.
4. Open the output file.

5. To output floating-point numbers in a fixed decimal format with the decimal point and trailing zeros, set the manipulators `fixed` and `showpoint`. Also, to output floating-point numbers to two decimal places, set the precision to two decimal places.
6. Initialize the course average for group 1 to `0.0`.
7. Initialize the course average for group 2 to `0.0`.
8. Initialize the number of courses to `0`.
9. Print the heading.
10. Get the course ID, `courseId1`, for group 1.
11. Get the course ID, `courseId2`, for group 2.
12. For each course in group 1 and group 2,
    a.
    ```
    if (courseId1 != courseId2)
    {
        cout << "Data error: Course IDs do not match.\n";
        return 1;
    }
    ```
    b.
    ```
    else
    {
    ```
        i. Calculate the course average for group 1 (call the function `calculateAverage` and pass the appropriate parameters).
        ii. Calculate the course average for group 2 (call the function `calculateAverage` and pass the appropriate parameters).
        iii. Print the results for group 1 (call the function `printResult` and pass the appropriate parameters).
        iv. Print the results for group 2 (call the function `printResult` and pass the appropriate parameters).
        v. Update the average for group 1.
        vi. Update the average for group 2.
        vii. Increment the number of courses.
    ```
    }
    ```
    c. Get the course ID, `courseId1`, for group 1.
    d. Get the course ID, `courseId2`, for group 2.
13. a. if not_end_of_file on group 1 and end_of_file on group 2 print "Ran out of data for group 2 before group 1"
    b. else if end_of_file on group 1 and not_end_of_file on group 2 print "Ran out of data for group 1 before group 2"
    c. else print the average of group 1 and group 2.
14. Close the input and output files.

## COMPLETE PROGRAM LISTING

```
//***********************************************************
// Author: D.S. Malik
//
// Program: Comparison of Class Averages
// This program computes and compares the class averages of
// two groups of students.
//***********************************************************

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

    //function prototypes
void calculateAverage(ifstream& inp, double& courseAvg);
void printResult(ofstream& outp, string courseId,
                 int groupNo, double avg);

int main()
{
        //Step 1
    string courseId1;       //course ID for group 1
    string courseId2;       //course ID for group 2
    int numberOfCourses;
    double avg1;            //average for a course in group 1
    double avg2;            //average for a course in group 2
    double avgGroup1;       //average group 1
    double avgGroup2;       //average group 2
    ifstream group1;        //input stream variable for group 1
    ifstream group2;        //input stream variable for group 2
    ofstream outfile;       //output stream variable

    group1.open("group1.txt");                  //Step 2
    group2.open("group2.txt");                  //Step 2

    if (!group1 || !group2)                     //Step 3
    {
        cout << "Unable to open files." << endl;
        cout << "Program terminates." << endl;
        return 1;
    }

    outfile.open("student.out");                //Step 4
    outfile << fixed << showpoint;              //Step 5
    outfile << setprecision(2);                 //Step 5

    avgGroup1 = 0.0;                            //Step 6
    avgGroup2 = 0.0;                            //Step 7
```

```
numberOfCourses = 0;                          //Step 8

outfile << "Course No   Group No    "
        << "Course Average" << endl;          //Step 9


group1 >> courseId1;                          //Step 10
group2 >> courseId2;                          //Step 11
while  (group1 && group2)                     //Step 12
{
    if (courseId1 != courseId2)               //Step 12a
    {
        cout << "Data error: Course IDs "
             << "do not match." << endl;
        cout << "Program terminates." << endl;
        return 1;
    }
    else                                      //Step 12b
    {
        calculateAverage(group1, avg1);       //Step 12b.i
        calculateAverage(group2, avg2);       //Step 12b.ii
        printResult(outfile, courseId1,
                    1, avg1);                 //Step 12b.iii
        printResult(outfile, courseId2,
                    2, avg2);                 //Step 12b.iv
        avgGroup1 = avgGroup1 + avg1;         //Step 12b.v
        avgGroup2 = avgGroup2 + avg2;         //Step 12b.vi
        outfile << endl;
        numberOfCourses++;                    //Step 12b.vii
    }

    group1 >> courseId1;                      //Step 12c
    group2 >> courseId2;                      //Step 12d
} // end while

if (group1 && !group2)                        //Step 13a
    cout << "Ran out of data for group 2 "
         << "before group 1." << endl;
else if (!group1 && group2)                   //Step 13b
    cout << "Ran out of data for group 1 "
         << "before group 2." << endl;
else                                          //Step 13c
{
    outfile << "Avg for group 1: "
            << avgGroup1 / numberOfCourses
            << endl;
    outfile << "Avg for group 2: "
            << avgGroup2 / numberOfCourses
            << endl;
}
```

```
    group1.close();                                    //Step 14
    group2.close();                                    //Step 14
    outfile.close();                                   //Step 14

    return 0;
}

void calculateAverage(ifstream& inp, double& courseAvg)
{
    double totalScore = 0.0;
    int numberOfStudents = 0;
    int score;

    inp >> score;
    while (score != -999)
    {
        totalScore = totalScore + score;
        numberOfStudents++;
        inp >> score;
    } //end while

    courseAvg = totalScore / numberOfStudents;
} //end calculate Average

void printResult(ofstream& outp, string courseID, int groupNo,
                 double avg)
{
    if (groupNo == 1)
       outp << "  " << courseID << "   ";
    else
       outp << "        ";

    outp << setw(8) << groupNo << setw(17) << avg << endl;
} //end printResult
```

**Sample Run:**

```
Course No    Group No     Course Average
  CSC            1            83.71
                 2            80.82

  ENG            1            82.00
                 2            78.20

  HIS            1            77.69
                 2            84.15

  MTH            1            83.57
                 2            84.29

  PHY            1            83.22
                 2            82.60
```

```
Avg for group 1: 82.04
Avg for group 2: 82.01
```

### Input Data Group 1

```
CSC 80 100 70 80 72 90 89 100 83 70 90 73 85 90 -999
ENG 80 90 80 94 90 74 78 63 83 80 90 -999
HIS 90 70 80 70 90 50 89 83 90 68 90 60 80 -999
MTH 74 80 75 89 90 73 90 82 74 90 84 100 90 79 -999
PHY 100 83 93 80 63 78 88 89 75 -999
```

### Input Data Group 2

```
CSC 90 75 90 75 80 89 100 60 80 70 80 -999
ENG 80 80 70 68 70 78 80 90 90 76 -999
HIS 100 80 80 70 90 76 88 90 90 75 90 85 80 -999
MTH 80 85 85 92 90 90 74 90 83 65 72 90 84 100 -999
PHY 90 93 73 85 68 75 67 100 87 88 -999
```



BAR GRAPH

In the business world, company executives often like to see results in some visual form, such as bar graphs. Many currently available software packages can analyze data in several forms and then display the results in a visual form, such as bar graphs or pie charts. The second part of this program aims to display the results found earlier in the form of bar graphs, as shown below:

```
Course          Course Average
  ID     0   10   20   30   40   50   60   70   80   90   100
         |....|....|....|....|....|....|....|....|....|....|
 CSC     ****************************************
         ########################################
 ENG     ****************************************
         ########################################
   .
   .
   .
Group 1 -- ****
Group 2 -- ####

Avg for group 1: 82.04
Avg for group 2: 82.01
```

Each symbol (`*` or `#`) in the bar graph represents two points. If a course average is less than 2, no symbol is printed.

Because the output is in the form of a bar graph, we need to modify the function `printResult`.

Print Bars

The function `printResult` prints the course ID and the bar graph representing the average for a course. The output is stored in a file. So we must pass four parameters to this function: the `ofstream` variable associated with the output file, the group

number (to print * or #), the course ID, and the course average for the department. To print the bar graph, we can use a loop to print one symbol for every two points. To find the number of symbols to print, we can use integer division as follows:

```
numberOfSymbols = static_cast<int>(average) / 2;
```

For example, `static_cast<int> / 2 = 78 /2 = 39`. If the average is `78.45`, we must print `39` symbols to represent this average.

Following this discussion, the definition of the function `printResult` is:

```
void printResult(ofstream& outp, string courseID,
                 int groupNo, double avg)
{
    int noOfSymbols;
    int count;

    if (groupNo == 1)
        outp << setw(4) << courseID << "    ";
    else
        outp << "        ";

    noOfSymbols = static_cast<int>(avg)/2;

    if (groupNo == 1)
        for (count = 1; count <= noOfSymbols; count++)
            outp << '*';
    else
        for (count = 1; count <= noOfSymbols; count++)
            outp << '#';
    outp << endl;
}//end printResult
```

We also include a function `printHeading` to print the first two lines of the output. The definition of this function is:

```
void printHeading(ofstream& outp)
{
    outp << "Course          Course Average" << endl;
    outp << "  ID    0   10   20   30   40   50   60   70"
         << "   80   90   100" << endl;
    outp << "        |....|....|....|....|....|....|....|"
         << "....|....|....|"<<endl;
}//end printHeading
```

Replace the function `printResult` in the preceding program, include the function `printHeading`, include the statements to output—`Group 1 --****` and `Group 2 --####` —, and rerun the program. Your program should generate a bar graph similar to the bar graph shown earlier. (The complete program listing is available on the website accompanying this book.)

## QUICK REVIEW

1. Functions, also called modules, are like miniature programs.
2. Functions enable you to divide a program into manageable tasks.
3. The C++ system provides the standard (predefined) functions.
4. To use a standard function, you must
   i. Know the name of the header file that contains the function's specification,
   ii. Include that header file in the program, and
   iii. Know the name and type of the function and number and types of the parameters (arguments).
5. There are two types of user-defined functions: value-returning functions and void functions.
6. Variables defined in a function heading are called formal parameters.
7. Expressions, variables, or constant values used in a function call are called actual parameters.
8. In a function call, the number of actual parameters and their types must match with the formal parameters in the order given.
9. To call a function, use its name together with the actual parameter list.
10. A value-returning function returns a value. Therefore, a value-returning function is used (called) in either an expression or an output statement or as a parameter in a function call.
11. The general syntax of a user-defined function is:

```
functionType functionName(formal parameter list)
{
    statements
}
```

12. The line `functionType functionName(formal parameter list)` is called the function heading (or function header). Statements enclosed between braces ({ and }) are called the body of the function.
13. The function heading and the body of the function are called the definition of the function.
14. If a function has no parameters, you still need the empty parentheses in both the function heading and the function call.
15. A value-returning function returns its value via the `return` statement.
16. A function can have more than one `return` statement. However, whenever a `return` statement executes in a function, the remaining statements are skipped and the function exits.
17. A `return` statement returns only one value.

18. A function prototype is the function heading without the body of the function; the function prototype ends with the semicolon.
19. A function prototype announces the function type, as well as the type and number of parameters, used in the function.
20. In a function prototype, the names of the variables in the formal parameter list are optional.
21. Function prototypes help the compiler correctly translate each function call.
22. In a program, function prototypes are placed before every function definition, including the definition of the function **`main`**.
23. When you use function prototypes, user-defined functions can appear in any order in the program.
24. When the program executes, the execution always begins with the first statement in the function **`main`**.
25. Functions execute only when they are called.
26. A call to a function transfers control from the caller to the called function.
27. In a function call statement, you specify only the actual parameters, not their data type or the function type.
28. When a function exits, control goes back to the caller.
29. A function that does not have a data type is called a void function.
30. A return statement without any value can be used in a void function. If a return statement is used in a void function, it is typically used to exit the function early.
31. The heading of a void function starts with the word `void`.
32. In C++, `void` is a reserved word.
33. A void function may or may not have parameters.
34. A call to a void function is a stand-alone statement.
35. To call a void function, you use the function name together with the actual parameters in a stand-alone statement.
36. There are two types of formal parameters: value parameters and reference parameters.
37. A value parameter receives a copy of its corresponding actual parameter.
38. A reference parameter receives the address (memory location) of its corresponding actual parameter.
39. The corresponding actual parameter of a value parameter is an expression, a variable, or a constant value.
40. A constant value cannot be passed to a reference parameter.
41. The corresponding actual parameter of a reference parameter must be a variable.

42. When you include & after the data type of a formal parameter, the formal parameter becomes a reference parameter.
43. The stream variables should be passed by reference to a function.
44. If a formal parameter needs to change the value of an actual parameter, in the function heading, you must declare this formal parameter as a reference parameter.
45. The scope of an identifier refers to those parts of the program where it is accessible.
46. Variables declared within a function (or block) are called local variables.
47. Variables declared outside of every function definition (and block) are called global variables.
48. The scope of a function name is the same as the scope of an identifier declared outside of any block.
49. See the scope rules in this chapter (section, Scope of an Identifier).
50. C++ does not allow the nesting of function definitions.
51. An automatic variable is a variable for which memory is allocated on function (or block) entry and deallocated on function (or block) exit.
52. A static variable is a variable for which memory remains allocated throughout the execution of the program.
53. By default, global variables are static variables.
54. In C++, a function can be overloaded.
55. Two functions are said to have different formal parameter lists if both functions have
    - A different number of formal parameters, or
    - The same number of formal parameters and the data types of the formal parameters, in the order listed, differ in at least one position.
56. The signature of a function consists of the function name and its formal parameter list. Two functions have different signatures if they have either different names or different formal parameter lists.
57. If a function is overloaded, then in a call to that function the formal parameter list of the function determines which function to execute.
58. C++ allows functions to have default parameters.
59. If you do not specify the value of a default parameter, the default value is used for that parameter.
60. All of the default parameters must be the far-right parameters of the function.
61. Suppose a function has more than one default parameter. In a function call, if a value to a default parameter is not specified, then you must omit all arguments to its right.

62. Default values can be constants, global variables, or function calls.
63. The calling function has the option of specifying a value other than the default for any default parameter.
64. You cannot assign a constant value as a default value to a reference parameter.

## EXERCISES

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

1. Mark the following statements as true or false:
   a. To use a predefined function in a program, you need to know only the name of the function and how to use it. (1)
   b. A value-returning function returns only one value. (2, 3)
   c. Parameters allow you to use different values each time the function is called. (2, 7, 9)
   d. When a `return` statement executes in a user-defined function, the function immediately exits. (3, 4)
   e. A value-returning function returns only integer values. (4)
   f. A variable name cannot be passed to a value parameter. (3, 6)
   g. If a C++ function does not use parameters, parentheses around the empty parameter list are still required. (2, 3, 6)
   h. In C++, the names of the corresponding formal and actual parameters must be the same. (3, 4, 6)
   i. A function that changes the value of a reference parameter also changes the value of the actual parameter. (7)
   j. Whenever the value of a reference parameter changes, the value of the actual parameter changes. (7)
   k. In C++, function definitions can be nested; that is, the definition of one function can be enclosed in the body of another function. (9)
   l. Using global variables in a program is a better programming style than using local variables, because extra variables can be avoided. (10)
   m. In a program, global constants are as dangerous as global variables. (10)
   n. The memory for a static variable remains allocated between function calls. (11)

2. Determine the value of each of the following expressions. (1)
   a. `static_cast<char>(toupper('#'))`
   b. `static_cast<char>(toupper('k'))`

c. `static_cast<char>(toupper('3'))`

d. `static_cast<char>(toupper('-'))`

e. `static_cast<char>(tolower('T'))`

f. `static_cast<char>(tolower(':'))`

g. `static_cast<char>(tolower('u'))`

h. `static_cast<char>(tolower('{'))`

3. Determine the value of each of the following expressions. (For decimal numbers, round your answer to two decimal places.) (1)

a. `abs(-18)` b. `fabs(20.5)` c. `fabs(-87.2)` d. `pow(4, 2.0)`

e. `pow(8.4, 3.5)` f. `sqrt(7.84)` g. `sqrt(196.0)`

h. `sqrt(38.44)* pow(2.4, 2) / fabs(-3.2)` i. `floor(27.37)`

j. `ceil(19.2)` k. `floor(12.45) + ceil(6.7)`

l. `floor(-8.9) + ceil(3.45)` m. `floor(9.6) / ceil(3.7)`

n. `pow(-4.0, 6.0)` o. `pow(10, -2.0)` p. `pow(9.2, 1.0 / 2)`

4. Using the functions described in Table 6-1, write each of the following as a C++ expression. (The expression in (e), denotes the absolute value of $3x^2 - 2y$.) (1)

a. $9.2^{4.0}$ b. $\sqrt{5x - 3xy}$ c. $\sqrt[3]{a + b}$

d. $\dfrac{-b + \sqrt{b^2 - 4ac}}{2a}$ e. $|3x^2 - 2y|$

5. Consider the following function definition. (4, 6)

```
double funcTest(string name, char u, int num, double y)
{
}
```

Which of the following are correct function prototypes of the function `funcTest`?

a. `double funcTest(string, char, int, double);`

b. `double funcTest(string n, char ch, int x, double t);`

c. `double funcTest(name, u, num, y);`

d. `int funcTest(string, char, int, double)`

6. Consider the following program. (1)

```
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;
```

```
int main()
{
    double num1;
    double num2;

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

    cout << "Enter two decimal numbers: ";
    cin >> num1 >> num2;
    cout << endl;

    if (num1 > 0 && num2 > 0)
        cout << sqrt(num1 * num2) << endl;
    else if (num1 < 0 && num2 >= 0)
        cout << floor(num1) + floor(num2) << endl;
    else if (num2 < 0 && num1 >= 0)
        cout << ceil(num1) + ceil(num2) << endl;
    else
        cout << floor(num1) + ceil(num2) << endl;

    return 0;
}
```

a. What is the output if the input is `12.5 8.0`?

b. What is the output if the input is `15.5 -6.5`?

c. What is the output if the input is `-18.35 34.6`?

d. What is the output if the input is `-20.90 -10.4`?

7. Consider the following statements:

```
int num1, num2, num3;
double length, width, height;
double volume;

num1 = 6; num2 = 7; num3 = 4;
length = 6.2; width = 2.3; height = 3.4;
```

and the function prototype:

```
double box(double, double, double);
```

Which of the following statements are valid? If they are invalid, explain why. (4)

a. `volume = box(length, width, height);`

b. `volume = box(length, 3.8, height);`

c. `cout << box(num1, num3, num2) << endl;`

d. `cout << box(length, width, 7.0) << endl;`

e. `volume = box(length, num1, height);`

f. `cout << box(6.2, , height) << endl;`

g. `volume = box(length + width, height);`

h. `volume = box(num1, num2 + num3);`

8. Consider the following functions: (4)

```
int hidden(int num1, int num2)
{
    if (num1 > 20)
        num1 = num2 / 10;
    else if (num2 > 20)
        num2 = num1 / 20;
    else
        return num1 - num2;

    return num1 * num2;
}

int compute(int one, int two)
{
    int secret = one;

    for (int i = one + 1; i <= two % 2; i++)
        secret = secret + i * i;

    return secret;
}
```

What is the output of each of the following program segments?

a. `cout << hidden(15, 10) << endl;`

b. `cout << compute(3, 9) << endl;`

c.
```
cout << hidden(30, 20) << " "
     << compute(10, hidden(30, 20)) << endl;
```

d.
```
x = 2; y = 8;
cout << compute(y, x) << endl;
```

9. Consider the following function prototypes: (3, 4)

```
double first(double, double);
int second(char, double, double);
string third(string, string, int, double);
char grade(double, double);
```

Answer the following questions.

a. How many parameters does the function `first` have? What is the type of function `first`?

b. How many parameters does function `second` have? What is the type of function `second`?

c. How many parameters does function `third` have? What is the type of function `third`?

d. How many parameters does function `grade` have? What is the type of function `grade`?

e. How many actual parameters are needed to call the function `third`? What is the type of each actual parameter, and in what order should you use these parameters in a call to the function `third`?

f. Write a C++ statement that prints the value returned by function `first` with the actual parameters `2.5` and `7.8`.

g. Write a C++ statement that prints the value returned by function `grade` with the actual parameters `82.50` and `92.50`, respectively.

h. Write a C++ statement that prints the `string` returned by function `third`. (Use your own actual parameters.)

10. Why do you need to include function prototypes in a program that contains user-defined functions? (5)

11. Write the definition of a function that takes as input a `char` value, and returns `true` if the character is a whitespace character; otherwise it returns `false`. (4)

12. Consider the following function: (1, 4)

```
double mystery(int x, double y, char ch)
{
    if (x > 0 && isupper(ch))
        return(sqrt(x * 1.0) + (static_cast<int>(ch) - 65));
    else
        return(pow(y,3) + x + static_cast<int>(ch));
}
```

What is the output of the following C++ statements? (Assume that the output of decimal numbers is set to two decimal places.)

a. `cout << mystery(16, 2.0, 'A') << endl;`

b. `cout << mystery(5, 3.0, '*') << endl;`

c. `cout << mystery(-7, 2.5, 'T') << endl;`

13. Consider the following function: (4)

```
int secret(int m, int n)
{
    int temp = n;

    for (int i = 1; i < abs(m); i++)
        temp = temp + n;
    if (m < 0)
        temp = -temp;

    return temp;
}
```

a. What is the output of the following C++ statements?

   i. `cout << secret(18, 4) << endl;`

   ii. `cout << secret(-10, 20) << endl;`

b. What does the function `secret` do?

14. Write the definition of a function that takes as input three numbers. The function returns `true` if the floor of the product of the first two numbers equals the floor of the third number; otherwise it returns `false`. (Assume that the three numbers are of type `double`.) (4)

15. Consider the following C++ program: (1, 4)

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    int temp = 0;

    for (int counter = 1; counter <= 100; counter++)
        if (pow(floor(sqrt(counter / 1.0)), 2.0) == counter)
            temp = temp + counter;

    cout << temp << endl;

    return 0;
}
```

a. What is the output of this program?

b. What does this program do?

16. What is the output of the following program? (4)

```
#include <iostream>

using namespace std;

int mystery(int x, int y, int z);

int main()
{
    cout << mystery(4, 18, 12) << endl;
    cout << mystery(6, -3, 45) << endl;
    cout << mystery(-5, 12, -7) << endl;
    cout << mystery(1, 0, 0) << endl;
    cout << mystery(10, 10, 10) << endl;

    return 0;
 }
```

```
int mystery(int x, int y, int z)
{
    if (x <= y && x <= z && x != 0)
        return (y + z) / x;
    else if (y <= z && y <= x && y != 0)
        return (z + x) / y;
    else if (z <= x && z <= y && z != 0)
        return (x + y) / z;
    else
        return x + y + z;
}
```

17. Write the definition of a function that takes as input two decimal numbers and returns first number to the power of the second number plus second number to the power of the first number. (4)

18. Consider the following C++ function. (4)

```
int mystery(int num1, int num2)
{
    if (num1 > 0)
    {
        for (int i = 1; i <= num1; i++)
            num2 = num2 * i;
        return num2;
    }
    else if (num2 > 0)
    {
        for (int i = 0; i <= num2; i++)
            num1 = num1 + i;
        return num1;
    }

    return 0;
}
```

What is the output of the following statements?

a. `cout << mystery(4, -5) << endl;`
b. `cout << mystery(-8, 9) << endl;`
c. `cout << mystery(2, 3) << endl;`
d. `cout << mystery(-2, -4) << endl;`

19. a. How would you use a return statement in a void function? (6)
    b. Why would you want to use a return statement in a void function? (6)

20. Identify the following items in the programming code shown below: (5, 6)
    a. Function prototype, function heading, function body, and function definitions.

b. Function call statements, formal parameters, and actual parameters.

c. Value parameters and reference parameters.

d. Local variables and global variables.

e. Named constants.

```
#include <iostream>                                          //Line 1

using namespace std;                                         //Line 2

const double RATE = 15.50;                                   //Line 3
const char STAR = '*';                                       //Line 4

double temp;                                                 //Line 5

void func(int, int, double&, char&);                         //Line 6

int main()                                                   //Line 7
{                                                            //Line 8
    int s = 50;                                              //Line 9
    int t = 6;                                               //Line 10
    double d;                                                //Line 11
    char ch = STAR;                                          //Line 12

    func(s, t, d, ch);                                       //Line 13
    cout << "d = " << d << ", ch = " << ch << endl;          //Line 14
    func(75, 8, d, ch);                                      //Line 15
    cout << "d = " << d << ", ch = " << ch << endl;          //Line 16

    return 0;                                                //Line 17
}                                                            //Line 18

void func(int speed, int time,                               //Line 19
          double& distance, char& c)                         //Line 20
{                                                            //Line 21
    double num;                                              //Line 22

    distance = speed * time;                                 //Line 23
    num = static_cast<int> (c);                              //Line 24
    c = static_cast<char>(num + 1);                          //Line 25
}                                                            //Line 26
```

What is the output of this program?

21. a. Explain the difference between an actual and a formal parameter. (4, 6, 7, 10)

b. Explain the difference between a value and a reference parameter.

c. Explain the difference between a local and a global variable.

22. What is the output of the following program? (6)

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

void func1();
void func2();

int main()
{
    int num;

    cout << "Enter 1 or 2: ";
    cin >> num;
    cout << endl;

    cout << "Take ";

    if (num == 1)
        func1();
    else if (num == 2)
        func2();
    else
       cout << "Invalid input. You must enter a 1 or 2" << endl;

    return 0;
 }

void func1()
{
    cout << "Programming I." <<endl;
}

void func2()
{
    cout << "Programming II." << endl;
}
```

a. What is the output if the input is `1`?

b. What is the output if the input is `2`?

c. What is the output if the input is `3`?

d. What is the output if the input is `-1`?

23. Write the definition of a `void` function that takes as input an integer and outputs two times the number if it is even; otherwise it outputs five times the number. (6, 7, 8)

24. Write the definition of a `void` function that takes as input three decimal numbers. The function returns the sum and average of the three numbers. If the average is greater than or equal to 70, it returns `"Pass"`; otherwise it returns `"Fail"`. (6, 7, 8)

25. Write the definition of a `void` function with three reference parameters of type `int`, `double`, and `string`. The function sets the values of the `int` and `double` variables to `0` and the value of the `string` variable to an empty string. (6, 7, 8)

26. Write the definition of a `void` function that takes as input two integer values, say `n` and `m`. The function returns the sum and average of all the numbers between `n` and `m` (inclusive) . (6, 7, 8)

27. What is the output of the following program? (6, 7, 8)

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

void compute(int x, int& y, int& z);

int main()
{
    int one = 7;
    int two = 5;
    int three = 6;

    compute(one, two, three);
    cout << one << ", " << two << ", " << three << endl;

    compute(two, one, three);
    cout << one << ", " << two << ", " << three << endl;

    compute(three, two, one);
    cout << one << ", " << two << ", " << three << endl;

    compute(two, three, one);
    cout << one << ", " << two << ", " << three << endl;

    return 0;
}

void compute(int x, int& y, int& z)
{
    int w = x + y - z;

    y = w % 2;
    x = y + w;
    z = x * y;
}
```

28. What is the output of the following program? (6, 7, 8)

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

int temp;
```

```
void sunny(int&, int);
void cloudy(int, int&);

int main()
{
    int num1 = 6;
    int num2 = 10;
    temp = 20;

    cout << num1 << " " << num2 << " " << temp << endl;

    sunny(num1, num2);
    cout << num1 << " " << num2 << " " << temp << endl;

    cloudy(num1, num2);
    cout << num1 << " " << num2 << " " << temp << endl;
    return 0;
}

void sunny(int& a, int b)
{
    int w;
    temp = (a + b) / 2;
    w = a / temp;
    b = a + w;
    a = temp - b;
}

void cloudy(int u, int& v)
{
    temp = 2 * u + v;
    v = u;
    u = v - temp;
}
```

29. In the following program, number the marked statements to show the order in which they will execute (the logical order of execution). Also what is the output if the input is `10`? (6, 7, 8)

```
#include <iostream>

using namespace std;

int secret(int, int);

void func(int x, int& y);

int main()
{
        int num1, num2;

_____   num1 = 6;
```

```
_____    cout << "Enter a positive integer: ";
_____    cin >> num2;
_____    cout << endl;
_____    cout << secret(num1, num2) << endl;
_____    num2 = num2 - num1;
_____    cout << num1 << " " << num2 << endl;
_____    func(num2, num1);
_____    cout << num1 << " " << num2 << endl;

_____    return 0;
   }

   int secret(int a, int b)
   {
         int d;

_____    d = a + b;
_____    b = a * d;

_____    return b;
   }

   void func (int x, int& y)
   {
         int val1, val2;

_____    val1 = x + y;
_____    val2 = x * y;
_____    y = val1 + val2;
_____    cout << val1 << " " << val2 << endl;
   }
```

30. Consider the following program. (6, 7, 8)

```
#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;

void trackVar(double& x, double y);

int main()
{
    double one, two;

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

    cout << "Enter two numbers: ";
    cin >> one >> two;
    cout << endl;
```

```
    trackVar(one, two);
    cout << "one = " << one << ", two = " << two << endl;
    trackVar(two, one);
    cout << "one = " << one << ", two = " << two << endl;

    return 0;
}

void trackVar(double& x, double y)
{
    double z;

    z = floor(x) + ceil(y);
    x = x + z;
    y = y - z;

    cout << "z = " << z << ", ";
}
```

a. What is the output if the input is `3 5`?

b. What is the output if the input is `4 11`?

c. What is the output if the input is `10 10`?

31. The function `trackVar` in Exercise 30, outputs the value of `z`. Modify the definition of this function so that rather than printing the value of `z` it sends its values back to the calling environment and the calling environment prints the values of `z`. (6, 7, 8)

32. In Exercise 30, determine the scope of each identifier. (9)

33. What is the output of the following code fragment? (9)

```
int num1 = 3;
int num2 = 5;

cout << num1 << " " << num2 << endl;

if (num1 + num2 > 7)
{
    int num2 = 12;
    int num3;

    num3 = num2 - num1;
    num1 = num2   * num3;
    num2 = num3 / num1;
    cout << num1 << " " << num2 << endl;
}
cout << num1 << " " << num2 << endl;
```

34. Consider the following program. What is its exact output? Show the values of the variables after each line executes, as in Example 6-14. (6, 7, 8)

```
#include <iostream>                                          //Line 1
using namespace std;                                        //Line 2

void funOne(int& a);                                        //Line 3

int main()                                                  //Line 4
{                                                           //Line 5
    int num1, num2;                                         //Line 6

    num1 = 10;                                              //Line 7

    num2 = 20;                                              //Line 8

    cout << "Line 9: In main: num1 = " << num1
         << ", num2 = " << num2 << endl;                    //Line 9

    funOne(num1);                                           //Line 10
    cout << "Line 11: In main after funOne: num1 = "
         << num1 << ", num2 = " << num2 << endl;            //Line 11

    return 0;                                               //Line 12
}                                                           //Line 13

void funOne(int& a)                                         //Line 14
{                                                           //Line 15
    int x = 12;                                             //Line 16
    int z;                                                  //Line 17

    z = a + x;                                              //Line 18

    cout << "Line 19: In funOne: a = " << a
         << ", x = " << x
         << ", and z = " << z << endl;                      //Line 19

    x = x + 5;                                              //Line 20

    cout << "Line 21: In funOne: a = " << a
         << ", x = " << x
         << ", and z = " << z << endl;                      //Line 21

    a = a + 8;                                              //Line 22

    cout << "Line 23: In funOne: a = " << a
         << ", x = " << x
         << ", and z = " << z << endl;                      //Line 23
}                                                           //Line 24
```

35. What is the output of the following program? (11)

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

void trackStaticVar(int& x);
```

```
int main()
{
    int temp = 1;

    for (int count = 1; count < 5; count++)
        trackStaticVar(temp);

    return 0;
}

void trackStaticVar(int& x)
{
    static int stVar = 1;
    int u = 3;

    if (x >= u)
        stVar = 2 * stVar;
    else
        stVar = 3 * stVar;
    x++;

    cout << "stVar = " << stVar << ", u = " << u << ", x = "
         << x << endl;
}
```

36. What is the signature of a function? (13)

37. Consider the following function prototype: (14)

```
void funcDefaultParam(int num, char ch = '*', double y = 2.5,
                      string z = "*");
```

Which of the following function calls is correct?

a. `funcDefaultParam();`

b. `funcDefaultParam(10);`

c. `funcDefaultParam(10, 3.8, 'u', "*");`

d. `funcDefaultParam(20, 'a', 2.8);`

e. `funcDefaultParam(28, '**');`

38. Consider the following function definition: (14)

```
void defaultParam(char ch, int num, double x)
{
    int temp;

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

    temp = num + static_cast<int>(ch);
    x = x + temp;

    cout << temp << ", " << x << endl;
}
```

What is the output of the following function calls?

a. `defaultParam('A');`

b. `defaultParam('*', 9);`

c. `defaultParam('+', 10, 7.5);`

d. `defaultParam('8', -5, 6.5);`

## PROGRAMMING EXERCISES

1. a. Write a program that uses the function `isPalindrome` given in Example 6-6 (Palindrome). Test your program on the following strings: `"madam"`, `"abba"`, `"22"`, `"67876"`, `"444244"`, and `"trymeuemyrt"`

   b. Modify the function `isPalindrome` of Example 6-6 so that when determining whether a string is a palindrome, cases are ignored, that is, uppercase and lowercase letters are considered the same.

2. Write a value-returning function, `isVowel`, that returns the value `true` if a given character is a vowel and otherwise returns `false`.

3. Write a program that prompts the user to input a sequence of characters and outputs the number of vowels. (Use the function `isVowel` written in Programming Exercise 2.)

4. Write a program that defines the named constant `PI`, `const double PI = 3.14159;`, which stores the value of π. The program should use `PI` and the functions listed in Table 6-1 to accomplish the following:

   a. Output the value of $\sqrt{\pi}$.

   b. Prompt the user to input the value of a `double` variable $r$, which stores the radius of a sphere. The program then outputs the following:

      i. The value of $4.0\pi r^2$, which is the surface area of the sphere.

      ii. The value of $(4.0/3.0)\pi r^3$, which is the volume of the sphere.

5. Write a program to test the functions described in Exercises 11 and 14 of this chapter.

6. Write a program to test the functions described in Exercises 23 and 26 of this chapter.

7. Write a program that prompts the user to enter two positive integers less than 1,000,000,000 and the program outputs the sum of all the prime numbers between the two integers. Two prime numbers are called twin primes, if the difference between the two primes is 2 or –2. Have the program output all the twin primes and the number of twin primes between the two integers.

8. The following program is designed to find the area of a rectangle, the area of a circle, or the volume of a cylinder. However (a) the statements