

6

Functions and an Introduction to Recursion



Form ever follows function.

— Louis Henri Sullivan

E pluribus unum. (One composed of many.)

— Virgil

O! call back yesterday, bid time return.

— William Shakespeare

Call me Ishmael.

— Herman Melville



When you call me that, smile!

— Owen Wister

Answer me in one word.

— William Shakespeare

There is a point at which methods devour themselves.

— Frantz Fanon

Life can only be understood backwards; but it must be lived forwards.

— Soren Kierkegaard



OBJECTIVES

In this chapter you will learn:

- To construct programs modularly from functions.
- To use common math functions available in the C++ Standard Library.
- To create functions with multiple parameters.
- The mechanisms for passing information between functions and returning results.
- How the function call/return mechanism is supported by the function call stack and activation records.
- To use random number generation to implement game-playing applications.
- How the visibility of identifiers is limited to specific regions of programs.
- To write and use recursive functions, i.e., functions that call themselves.



Outline

- 6.1 Introduction**
- 6.2 Program Components in C++**
- 6.3 Math Library Functions**
- 6.4 Function Definitions with Multiple Parameters**
- 6.5 Function Prototypes and Argument Coercion**
- 6.6 C++ Standard Library Header Files**
- 6.7 Case Study: Random Number Generation**
- 6.8 Case Study: Game of Chance and Introducing enum**
- 6.9 Storage Classes**
- 6.10 Scope Rules**
- 6.11 Function Call Stack and Activation Records**
- 6.12 Functions with Empty Parameter Lists**



Outline

- 6.13 **Inline Functions**
- 6.14 **References and Reference Parameters**
- 6.15 **Default Arguments**
- 6.16 **Unary Scope Resolution Operator**
- 6.17 **Function Overloading**
- 6.18 **Function Templates**
- 6.19 **Recursion**
- 6.20 **Example Using Recursion: Fibonacci Series**
- 6.21 **Recursion vs. Iteration**
- 6.22 **(Optional) Software Engineering Case Study:
Identifying Class Operations in the ATM System**
- 6.23 **Wrap-Up**



6.1 Introduction

- **Divide and conquer technique**
 - Construct a large program from small, simple pieces (e.g., components)
- **Functions**
 - Facilitate the design, implementation, operation and maintenance of large programs
- **C++ Standard Library math functions**



6.2 Program Components in C++

- **C++ Standard Library**
 - Rich collection of functions for performing common operations, such as:
 - Mathematical calculations
 - String manipulations
 - Character manipulations
 - Input/Output
 - Error checking
 - Provided as part of the C++ programming environment



Software Engineering Observation 6.1

Read the documentation for your compiler to familiarize yourself with the functions and classes in the C++ Standard Library.



6.2 Program Components in C++ (Cont.)

- **Functions**

- Called methods or procedures in other languages
- Allow programmers to modularize a program by separating its tasks into self-contained units
 - Statements in function bodies are written only once
 - Reused from perhaps several locations in a program
 - Hidden from other functions
 - Avoid repeating code
 - Enable the divide-and-conquer approach
 - Reusable in other programs
 - User-defined or programmer-defined functions



Software Engineering Observation 6.2

To promote software reusability, every function should be limited to performing a single, well-defined task, and the name of the function should express that task effectively. Such functions make programs easier to write, test, debug and maintain.



Error-Prevention Tip 6.1

A small function that performs one task is easier to test and debug than a larger function that performs many tasks.



Software Engineering Observation 6.3

If you cannot choose a concise name that expresses a function's task, your function might be attempting to perform too many diverse tasks. It is usually best to break such a function into several smaller functions.



6.2 Program Components in C++ (cont.)

- **Function (Cont.)**
 - A function is invoked by a function call
 - Called function either returns a result or simply returns to the caller
 - Function calls form hierarchical relationships



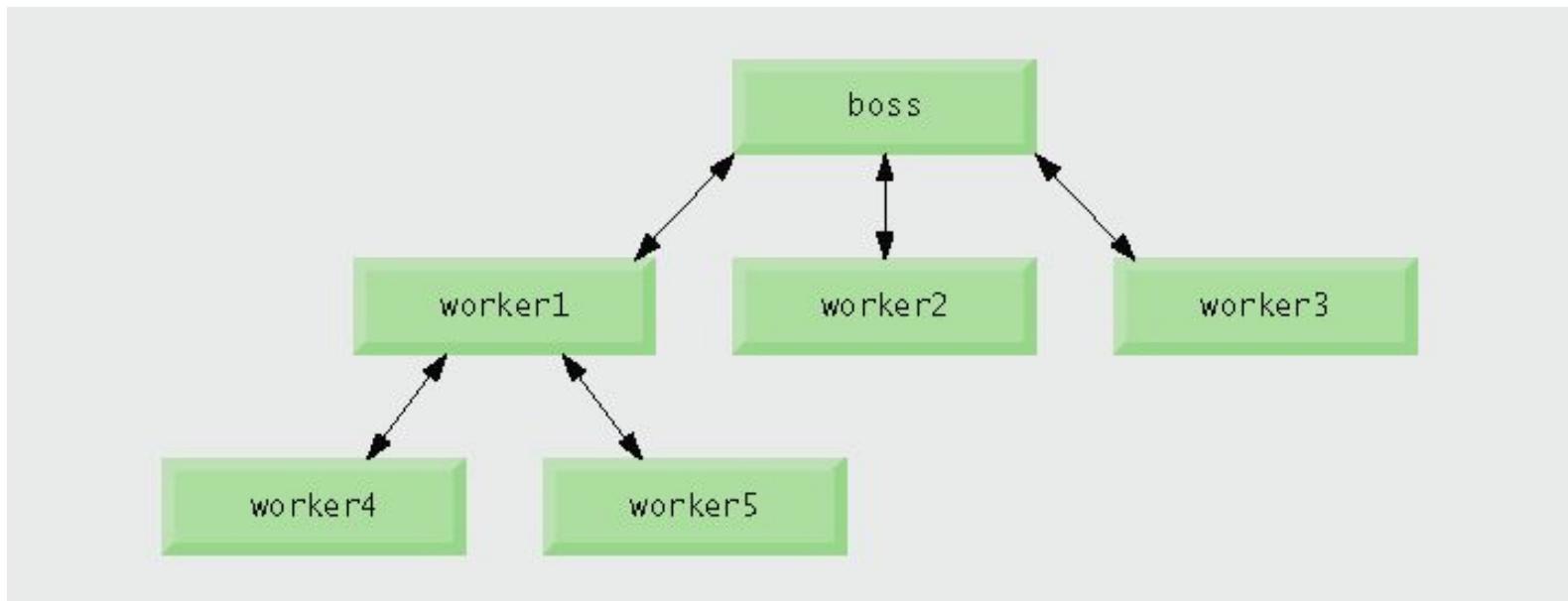


Fig. 6.1 | Hierarchical boss function/worker function relationship.



6.3 Math Library Functions

- **Global functions**

- Do not belong to a particular class
- Have function prototypes placed in header files
 - Can be reused in any program that includes the header file and that can link to the function's object code
- Example: `sqrt` in `<cmath>` header file
 - `sqrt(900.0)`
 - All functions in `<cmath>` are global functions



Function	Description	Example
ceil(x)	rounds x to the smallest integer not less than x	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
cos(x)	trigonometric cosine of x (x in radians)	cos(0.0) is 1.0
exp(x)	exponential function e^x	exp(1.0) is 2.71828 exp(2.0) is 7.38906
fabs(x)	absolute value of x	fabs(5.1) is 5.1 fabs(0.0) is 0.0 fabs(-8.76) is 8.76
floor(x)	rounds x to the largest integer not greater than x	floor(9.2) is 9.0 floor(-9.8) is -10.0
fmod(x, y)	remainder of x/y as a floating-point number	fmod(2.6, 1.2) is 0.2
log(x)	natural logarithm of x (base e)	log(2.718282) is 1.0 log(7.389056) is 2.0
log10(x)	logarithm of x (base 10)	log10(10.0) is 1.0 log10(100.0) is 2.0
pow(x, y)	x raised to power y (x^y)	pow(2, 7) is 128 pow(9, .5) is 3
sin(x)	trigonometric sine of x (x in radians)	sin(0.0) is 0
sqrt(x)	square root of x (where x is a nonnegative value)	sqrt(9.0) is 3.0
tan(x)	trigonometric tangent of x (x in radians)	tan(0.0) is 0

Fig. 6.2 | Math library functions.



6.4 Function Definitions with Multiple Parameters

- **Multiple parameters**

- Functions often require more than one piece of information to perform their tasks
- Specified in both the function prototype and the function header as a comma-separated list of parameters



```
1 // Fig. 6.3: GradeBook.h
2 // Definition of class GradeBook that finds the maximum of three grades.
3 // Member functions are defined in GradeBook.cpp
4 #include <string> // program uses C++ standard string class
5 using std::string;
6
7 // GradeBook class definition
8 class GradeBook
9 {
10 public:
11     GradeBook( string ); // constructor initializes course name
12     void setCourseName( string ); // function to set the course name
13     string getCourseName(); // function to retrieve the course name
14     void displayMessage(); // display a welcome message
15     void inputGrades(); // input three grades from user
16     void displayGradeReport(); // display a report based on the grades
17     int max( int, int, int ); // determine max of 3 values
18 private:
19     string courseName; // course name for this GradeBook
20     int maxGrade; // maximum of three grades
21 }; // end class GradeBook
```

GradeBook.h

(1 of 1)

Prototype for a member function
that takes three arguments

Data member to store maximum grade



```
1 // Fig. 6.4: GradeBook.cpp
2 // Member-function definitions for class GradeBook that
3 // determines the maximum of three grades.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "GradeBook.h" // include definition of class GradeBook
10
11 // constructor initializes courseName with string supplied as argument;
12 // initializes studentMaximum to 0
13 GradeBook::GradeBook( string name )
14 {
15     setCourseName( name ); // validate and store courseName
16     maximumGrade = 0; // this value will be replaced by the maximum grade
17 } // end GradeBook constructor
18
19 // function to set the course name; limits name to 25 or fewer characters
20 void GradeBook::setCourseName( string name )
21 {
22     if ( name.length() <= 25 ) // if name has 25 or fewer characters
23         courseName = name; // store the course name in the object
24     else // if name is longer than 25 characters
25     { // set courseName to first 25 characters of parameter name
26         courseName = name.substr( 0, 25 ); // select first 25 characters
27         cout << "Name '" << name << "' exceeds maximum length (25).\\n"
28             << "Limiting courseName to first 25 characters.\\n" << endl;
29     } // end if...else
30 } // end function setCourseName
```

GradeBook.cpp

(1 of 3)



```
31
32 // function to retrieve the course name
33 string GradeBook::getCourseName()
34 {
35     return courseName;
36 } // end function getCourseName
37
38 // display a welcome message to the GradeBook user
39 void GradeBook::displayMessage()
40 {
41     // this statement calls getCourseName to get the
42     // name of the course this GradeBook represents
43     cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
44         << endl;
45 } // end function displayMessage
46
47 // input three grades from user; determine maximum
48 void GradeBook::inputGrades()
49 {
50     int grade1; // first grade entered by user
51     int grade2; // second grade entered by user
52     int grade3; // third grade entered by user
53
54     cout << "Enter three integer grades: ";
55     cin >> grade1 >> grade2 >> grade3;
56
57     // store maximum in member studentMaxGrade
58     maxGrade = maximum( grade1, grade2, grade3 );
59 } // end function inputGrades
```

Call to function **maximum**
passes three arguments



```
60
61 // returns the maximum of its three integer parameters
62 int GradeBook::maximum( int x, int y, int z ) ← maximum member function header
63 {
64     int maximumValue = x; // assume x is the largest to start
65
66     // determine whether y is greater than maximumValue ← Comma-separated parameter list
67     if ( y > maximumValue )
68         maximumValue = y; // make y the new maximumValue
69
70     // determine whether z is greater than maximumValue
71     if ( z > maximumValue )
72         maximumValue = z; // make z the new maximumValue
73
74     return maximumValue; ← Returning a value to the caller
75 } // end function maximum
76
77 // display a report based on the grades entered by user
78 void GradeBook::displayGradeReport()
79 {
80     // output maximum of grades entered
81     cout << "Maximum of grades entered: " << maximumGrade << endl;
82 } // end function displayGradeReport
```



```
1 // Fig. 6.5: fig06_05.cpp
2 // Create GradeBook object, input grades and display grade report.
3 #include "GradeBook.h" // include definition of class GradeBook
4
5 int main()
6 {
7     // create GradeBook object
8     GradeBook myGradeBook( "CS101 C++ Programming" );
9
10    myGradeBook.displayMessage(); // display welcome message
11    myGradeBook.inputGrades(); // read grades from user
12    myGradeBook.displayGradeReport(); // display report based on grades
13    return 0; // indicate successful termination
14 } // end main
```

fig06_05.cpp

(1 of 1)

Welcome to the grade book for
CS101 C++ Programming!

Enter three integer grades: 86 67 75
Maximum of grades entered: 86

Welcome to the grade book for
CS101 C++ Programming!

Enter three integer grades: 67 86 75
Maximum of grades entered: 86

Welcome to the grade book for
CS101 C++ Programming!

Enter three integer grades: 67 75 86
Maximum of grades entered: 86



Software Engineering Observation 6.4

The commas used in line 58 of Fig. 6.4 to separate the arguments to function maximum are not comma operators as discussed in Section 5.3. The comma operator guarantees that its operands are evaluated left to right. The order of evaluation of a function's arguments, however, is not specified by the C++ standard. Thus, different compilers can evaluate function arguments in different orders. The C++ standard does guarantee that all arguments in a function call are evaluated before the called function executes.



6.4 Function Definitions with Multiple Parameters (Cont.)²⁵

- **Compiler uses a function prototype to:**
 - Check that calls to the function contain the correct number and types of arguments in the correct order
 - Ensure that the value returned by the function is used correctly in the expression that called the function
- **Each argument must be consistent with the type of the corresponding parameter**
 - Parameters are also called formal parameters



Portability Tip 6.1

Sometimes when a function's arguments are more involved expressions, such as those with calls to other functions, the order in which the compiler evaluates the arguments could affect the values of one or more of the arguments. If the evaluation order changes between compilers, the argument values passed to the function could vary, causing subtle logic errors.



Error-Prevention Tip 6.2

If you have doubts about the order of evaluation of a function's arguments and whether the order would affect the values passed to the function, evaluate the arguments in separate assignment statements before the function call, assign the result of each expression to a local variable, then pass those variables as arguments to the function.



Common Programming Error 6.1

Declaring method parameters of the same type as double x, y instead of double x, double y is a syntax error—an explicit type is required for each parameter in the parameter list.



Common Programming Error 6.2

Compilation errors occur if the function prototype, function header and function calls do not all agree in the number, type and order of arguments and parameters, and in the return type.



Software Engineering Observation 6.5

A function that has many parameters may be performing too many tasks. Consider dividing the function into smaller functions that perform the separate tasks. Limit the function header to one line if possible.



6.4 Function Definitions with Multiple Parameters (Cont.)³¹

- Three ways to return control to the calling statement:
 - If the function does not return a result:
 - Program flow reaches the function-ending right brace or
 - Program executes the statement `return;`
 - If the function does return a result:
 - Program executes the statement `return expression;`
 - *expression* is evaluated and its value is returned to the caller



6.5 Function Prototypes and Argument Coercion

- **Function prototype**
 - Also called a function declaration
 - Indicates to the compiler:
 - Name of the function
 - Type of data returned by the function
 - Parameters the function expects to receive
 - Number of parameters
 - Types of those parameters
 - Order of those parameters



Software Engineering Observation 6.6

Function prototypes are required in C++. Use #include preprocessor directives to obtain function prototypes for the C++ Standard Library functions from the header files for the appropriate libraries (e.g., the prototype for math function sqrt is in header file <cmath>; a partial list of C++ Standard Library header files appears in Section 6.6). Also use #include to obtain header files containing function prototypes written by you or your group members.



Common Programming Error 6.3

If a function is defined before it is invoked, then the function's definition also serves as the function's prototype, so a separate prototype is unnecessary. If a function is invoked before it is defined, and that function does not have a function prototype, a compilation error occurs.



Software Engineering Observation 6.7

Always provide function prototypes, even though it is possible to omit them when functions are defined before they are used (in which case the function header acts as the function prototype as well). Providing the prototypes avoids tying the code to the order in which functions are defined (which can easily change as a program evolves).



6.5 Function Prototypes and Argument Coercion (Cont.)

- **Function signature (or simply signature)**
 - The portion of a function prototype that includes the name of the function and the types of its arguments
 - Does not specify the function's return type
 - Functions in the same scope must have unique signatures
 - The scope of a function is the region of a program in which the function is known and accessible



Common Programming Error 6.4

It is a compilation error if two functions in the same scope have the same signature but different return types.



6.5 Function Prototypes and Argument Coercion (Cont.)

- **Argument Coercion**

- Forcing arguments to the appropriate types specified by the corresponding parameters
 - For example, calling a function with an integer argument, even though the function prototype specifies a double argument
 - The function will still work correctly



6.5 Function Prototypes and Argument Coercion (Cont.)

- **C++ Promotion Rules**

- Indicate how to convert between types without losing data
- Apply to expressions containing values of two or more data types
 - Such expressions are also referred to as mixed-type expressions
 - Each value in the expression is promoted to the “highest” type in the expression
 - Temporary version of each value is created and used for the expression
 - Original values remain unchanged



6.5 Function Prototypes and Argument Coercion (Cont.)

- **C++ Promotion Rules (Cont.)**

- **Promotion also occurs when the type of a function argument does not match the specified parameter type**
 - Promotion is as if the argument value were being assigned directly to the parameter variable
- **Converting a value to a lower fundamental type**
 - Will likely result in the loss of data or incorrect values
 - Can only be performed explicitly
 - By assigning the value to a variable of lower type (some compilers will issue a warning in this case) or
 - By using a cast operator



Data types

long double

double

float

unsigned long int

(synonymous with **unsigned long**)

long int

(synonymous with **long**)

unsigned int

(synonymous with **unsigned**)

int

unsigned short int

(synonymous with **unsigned short**)

short int

(synonymous with **short**)

unsigned char

char

bool

Fig. 6.6 | Promotion hierarchy for fundamental data types.



Common Programming Error 6.5

Converting from a higher data type in the promotion hierarchy to a lower type, or between signed and unsigned, can corrupt the data value, causing a loss of information.



Common Programming Error 6.6

It is a compilation error if the arguments in a function call do not match the number and types of the parameters declared in the corresponding function prototype. It is also an error if the number of arguments in the call matches, but the arguments cannot be implicitly converted to the expected types.



6.6 C++ Standard Library Header Files

- **C++ Standard Library header files**
 - Each contains a portion of the Standard Library
 - Function prototypes for the related functions
 - Definitions of various class types and functions
 - Constants needed by those functions
 - “Instruct” the compiler on how to interface with library and user-written components
 - Header file names ending in .h
 - Are “old-style” header files
 - Superseded by the C++ Standard Library header files



C++ Standard Library header file	Explanation
<code><iostream></code>	Contains function prototypes for the C++ standard input and standard output functions, introduced in Chapter 2, and is covered in more detail in Chapter 15, Stream Input/Output. This header file replaces header file <code><iostream.h></code> .
<code><iomanip></code>	Contains function prototypes for stream manipulators that format streams of data. This header file is first used in Section 4.9 and is discussed in more detail in Chapter 15, Stream Input/Output. This header file replaces header file <code><iomanip.h></code> .
<code><cmath></code>	Contains function prototypes for math library functions (discussed in Section 6.3). This header file replaces header file <code><math.h></code> .
<code><stdlib.h></code>	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header file are covered in Section 6.7; Chapter 11, Operator Overloading; String and Array Objects; Chapter 16, Exception Handling; Chapter 19, Web Programming; Chapter 22, Bits, Characters, C-Strings and structs ; and Appendix E, C Legacy Code Topics. This header file replaces header file <code><stdlib.h></code> .

Fig. 6.7 | C++ Standard Library header files. (Part 1 of 4)



C++ Standard Library header file	Explanation
<code><ctime></code>	Contains function prototypes and types for manipulating the time and date. This header file replaces header file <code><time.h></code> . This header file is used in Section 6.7.
<code><vector>, <list>, <deque>, <queue>, <stack>, <map>, <set>, <bitset></code>	These header files contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The <code><vector></code> header is first introduced in Chapter 7, Arrays and Vectors. We discuss all these header files in Chapter 23, Standard Template Library (STL).
<code><cctype></code>	Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. This header file replaces header file <code><ctype.h></code> . These topics are discussed in Chapter 8, Pointers and Pointer-Based Strings, and Chapter 22, Bits, Characters, C-Strings and <code>structs</code> .
<code><cstring></code>	Contains function prototypes for C-style string-processing functions. This header file replaces header file <code><string.h></code> . This header file is used in Chapter 11, Operator Overloading; String and Array Objects.

Fig. 6.7 | C++ Standard Library header files. (Part 2 of 4)



C++ Standard Library header file	Explanation
<code><typeinfo></code>	Contains classes for runtime type identification (determining data types at execution time). This header file is discussed in Section 13.8.
<code><exception>, <stdexcept></code>	These header files contain classes that are used for exception handling (discussed in Chapter 16).
<code><memory></code>	Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 16, Exception Handling.
<code><iostream></code>	Contains function prototypes for functions that perform input from files on disk and output to files on disk (discussed in Chapter 17, File Processing). This header file replaces header file <code><iostream.h></code> .
<code><string></code>	Contains the definition of class <code>string</code> from the C++ Standard Library (discussed in Chapter 18).
<code><iomanip></code>	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 18, Class <code>string</code> and String Stream Processing).
<code><functional></code>	Contains classes and functions used by C++ Standard Library algorithms. This header file is used in Chapter 23.

Fig. 6.7 | C++ Standard Library header files. (Part 3 of 4)



C++ Standard Library header file

<i t e r a t o r >	Contains classes for accessing data in the C++ Standard Library containers. This header file is used in Chapter 23, Standard Template Library (STL).
<a l g o r i t h m >	Contains functions for manipulating data in C++ Standard Library containers. This header file is used in Chapter 23.
<c a s s e r t >	Contains macros for adding diagnostics that aid program debugging. This replaces header file <assert . h> from pre-standard C++. This header file is used in Appendix F, Preprocessor.
<c f l o a t >	Contains the floating-point size limits of the system. This header file replaces header file <f l o a t . h> .
<c l i n i t s >	Contains the integral size limits of the system. This header file replaces header file < i n i t s . h> .
<c s t d i o >	Contains function prototypes for the C-style standard input/output library functions and information used by them. This header file replaces header file <s t d i o . h> .
<l o c a l e >	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.).
<i n i t s >	Contains classes for defining the numerical data type limits on each computer platform.
<u t i l i t y >	Contains classes and functions that are used by many C++ Standard Library header files.

Fig. 6.7 | C++ Standard Library header files. (Part 4 of 4)



6.7 Case Study: Random Number Generation

- C++ Standard Library function **rand**
 - Introduces the element of chance into computer applications
 - Example
 - **i = rand();**
 - Generates an unsigned integer between 0 and **RAND_MAX** (a symbolic constant defined in header file `<cstdlib>`)
 - Function prototype for the **rand** function is in `<cstdlib>`



6.7 Case Study: Random Number Generation (Cont.)

- To produce integers in a specific range, use the modulus operator (%) with rand
 - Example
 - `rand() % 6;`
 - Produces numbers in the range 0 to 5
 - This is called scaling, 6 is the scaling factor
 - Shifting can move the range to 1 to 6
 - `1 + rand() % 6;`



```
1 // Fig. 6.8: fig06_08.cpp
2 // Shifted and scaled random integers.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstdlib> // contains function prototype for rand
11 using std::rand;
12
13 int main()
14 {
15     // Loop 20 times
16     for (int counter = 1; counter <= 20; counter++)
17     {
18         // pick random number from 1 to 6 and output it
19         cout << setw(10) << (1 + rand() % 6);
20     }
21 }
```

#include and using for function rand

Calling function rand



```
20
21     // if counter is divisible by 5, start a new line of output
22     if ( counter %5 == 0 )
23         cout << endl ;
24 } // end for
25
26 return 0; // indicates successful termination
27 } // end main
```

fig06_08.cpp

(2 of 2)

6	6	5	5	6
5	1	1	5	3
6	6	2	4	2
6	2	3	4	1



```
1 // Fig. 6.9: fig06_09.cpp
2 // Roll a six-sided die 6,000,000 times.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstdlib> // contains function prototype for rand
11 using std::rand;
12
13 int main()
14 {
15     int frequency1 = 0; // count of 1s rolled
16     int frequency2 = 0; // count of 2s rolled
17     int frequency3 = 0; // count of 3s rolled
18     int frequency4 = 0; // count of 4s rolled
19     int frequency5 = 0; // count of 5s rolled
20     int frequency6 = 0; // count of 6s rolled
21
22     int face; // stores most recently rolled value
23
24     // summarize results of 6,000,000 rolls of a die
25     for (int roll = 1; roll <= 6000000; roll++)
26     {
27         face = 1 + rand() % 6; // random number from 1 to 6
```

Scaling and shifting the value produced by function `rand`



```
28 // determine roll value 1-6 and increment appropriate counter
29 switch ( face )
30 {
31     case 1:
32         ++frequency1; // increment the 1s counter
33         break;
34     case 2:
35         ++frequency2; // increment the 2s counter
36         break;
37     case 3:
38         ++frequency3; // increment the 3s counter
39         break;
40     case 4:
41         ++frequency4; // increment the 4s counter
42         break;
43     case 5:
44         ++frequency5; // increment the 5s counter
45         break;
46     case 6:
47         ++frequency6; // increment the 6s counter
48         break;
49     default: // invalid value
50         cout << "Program should never get here!";
51     } // end switch
52 } // end for
```

fig06_09.cpp

(2 of 3)



```
54
55 cout << "Face" << setw( 13 ) << "Frequency" << endl ; // output headers
56 cout << "    1" << setw( 13 ) << frequency1
57     << "\n    2" << setw( 13 ) << frequency2
58     << "\n    3" << setw( 13 ) << frequency3
59     << "\n    4" << setw( 13 ) << frequency4
60     << "\n    5" << setw( 13 ) << frequency5
61     << "\n    6" << setw( 13 ) << frequency6 << endl ;
62 return 0; // indicates successful termination
63 } // main
```

Face	Frequency
1	999702
2	1000823
3	999378
4	998898
5	1000777
6	1000422

Each face value appears approximately 1,000,000 times



Error-Prevention Tip 6.3

Provide a default case in a switch to catch errors even if you are absolutely, positively certain that you have no bugs!



6.7 Case Study: Random Number Generation (Cont.)

- **Function rand**
 - Generates pseudorandom numbers
 - The same sequence of numbers repeats itself each time the program executes
- **Randomizing**
 - Conditioning a program to produce a different sequence of random numbers for each execution
- **C++ Standard Library function srand**
 - Takes an unsigned integer argument
 - Seeds the rand function to produce a different sequence of random numbers



```
1 // Fig. 6.10: fig06_10.cpp
2 // Randomizing die rolling program
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <random>
9 using std::setw;
10
11 #include <cstdlib> // contains prototypes for functions srand and rand
12 using std::rand;
13 using std::srand;
14
15 int main()
16 {
17     unsigned seed; // stores the seed entered by the user
18
19     cout << "Enter seed: ";
20     cin >> seed;
21     srand( seed ); // seed random number generator
22 }
```

using statement for function **srand**

Data type **unsigned** is short for **unsigned int**

Passing **seed** to **srand** to randomize the program



```
23 // Loop 10 times
24 for ( int counter = 1; counter <= 10; counter++ )
25 {
26     // pick random number from 1 to 6 and output it
27     cout << setw( 10 ) << ( 1 + rand() %6 );
28
29     // if counter is divisible by 5, start a new line of output
30     if ( counter %5 == 0 )
31         cout << endl;
32 } // end for
33
34 return 0; // indicates successful termination
35 } // end main
```

fig06_10.cpp

(2 of 2)

Enter seed: 67

6	1	4	6	2
1	6	1	6	4

Enter seed: 432

4	6	3	1	6
3	1	5	4	2

Program outputs show that each unique seed value produces a different sequence of random numbers

Enter seed: 67

6	1	4	6	2
1	6	1	6	4



6.7 Case Study: Random Number Generation (Cont.)

- To randomize without having to enter a seed each time
 - `srand(time(0));`
 - This causes the computer to read its clock to obtain the seed value
 - Function `time` (with the argument `0`)
 - Returns the current time as the number of seconds since January 1, 1970 at midnight Greenwich Mean Time (GMT)
 - Function prototype for `time` is in `<ctime>`



Common Programming Error 6.7

Calling function `strand` more than once in a program restarts the pseudorandom number sequence and can affect the randomness of the numbers produced by `rand`.



Common Programming Error 6.8

Using `srand` in place of `rand` to attempt to generate random numbers is a compilation error—function `srand` does not return a value.



6.7 Case Study: Random Number Generation (Cont.)

- **Scaling and shifting random numbers**

- To obtain random numbers in a desired range, use a statement like

`number = shiftingValue + rand() % scalingFactor;`

- *shiftingValue* is equal to the first number in the desired range of consecutive integers
 - *scalingFactor* is equal to the width of the desired range of consecutive integers
 - number of consecutive integers in the range



6.8 Case Study: Game of Chance and Introducing enum

- **Enumeration**

- A set of integer constants represented by identifiers
 - The values of enumeration constants start at 0, unless specified otherwise, and increment by 1
 - The identifiers in an enum must be unique, but separate enumeration constants can have the same integer value
- Defining an enumeration
 - Keyword enum
 - A type name
 - Comma-separated list of identifier names enclosed in braces
 - Example
 - `enum Months { JAN = 1, FEB, MAR, APR };`



```
1 // Fig. 6.11: fig06_11.cpp
2 // Craps simulation.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // contains prototypes for functions srand and rand
8 using std::rand;
9 using std::srand;
10
11 #include <ctime> // contains prototype for function time
12 using std::time;
13
14 int rollDice(); // rolls dice, calculates and displays sum
15
16 int main()
17 {
18     // enumeration with constants that represent the game status
19     enum Status { CONTINUE, WON, LOST }; // all caps in constants
20
21     int myPoint; // point if no win or loss on first roll
22     Status gameStatus; // can contain CONTINUE, WON or LOST
23
24     // randomize random number generator
25     srand( time( 0 ) );
26
27     int sumOfDice = rollDice();
```

#include and using for function time

Enumeration to keep track of the game status

Declaring a variable of the user-defined enumeration type

Seeding the random number generator with the current time



```
28
29 // determine game status and point (if needed) based on first roll
30 switch ( sumOfDice )
31 {
32     case 7: // win with 7 on first roll
33     case 11: // win with 11 on first roll
34         gameStatus = WON;
35         break;
36     case 2: // lose with 2 on first roll
37     case 3: // lose with 3 on first roll
38     case 12: // lose with 12 on first roll
39         gameStatus = LOST;
40         break;
41     default: // did not win or lose, so remember point
42         gameStatus = CONTINUE; // game is not over
43         myPoint = sumOfDice; // remember the point
44         cout << "Point is " << myPoint << endl;
45         break; // optional at end of switch
46 } // end switch
47
48 // while game is not complete
49 while ( gameStatus == CONTINUE ) // not WON or LOST
50 {
51     sumOfDice = rollDice(); // roll dice again
52 }
```

Assigning an enumeration constant to `gameStatus`

Comparing a variable of an enumeration type to an enumeration constant



```
33 // determine game status
34 if ( sumOfDice == myPoint ) // win by making point
35     gameStatus = WON;
36 else
37     if ( sumOfDice == 7 ) // lose by rolling 7 before point
38         gameStatus = LOST;
39 } // end while
40
41 // display won or lost message
42 if ( gameStatus == WON )
43     cout << "Player wins" << endl;
44 else
45     cout << "Player loses" << endl;
46
47 return 0; // indicates successful termination
48 } // end main
49
50 // roll dice, calculate sum and display results
51 int rollDice() {
52     // pick random die values
53     int die1 = 1 + rand() % 6; // first die roll
54     int die2 = 1 + rand() % 6; // second die roll
55
56     int sum = die1 + die2; // compute sum of die values
```

Function that performs the task of rolling the dice



```
78
79 // display results of this roll
80 cout << "Player rolled " << die1 << " + " << die2
81     << " = " << sum << endl;
82 return sum // end function rollDice
83 } // end function rollDice
```

Player rolled 2 + 5 = 7
Player wins

fig06_11.cpp

(4 of 4)

Player rolled 6 + 6 = 12
Player loses

Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins

Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses



Good Programming Practice 6.1

Capitalize the first letter of an identifier used as a user-defined type name.



Good Programming Practice 6.2

Use only uppercase letters in the names of enumeration constants. This makes these constants stand out in a program and reminds the programmer that enumeration constants are not variables.



Good Programming Practice 6.3

Using enumerations rather than integer constants can make programs clearer and more maintainable. You can set the value of an enumeration constant once in the enumeration declaration.



Common Programming Error 6.9

Assigning the integer equivalent of an enumeration constant to a variable of the enumeration type is a compilation error.



Common Programming Error 6.10

**After an enumeration constant has been defined,
attempting to assign another value to the
enumeration constant is a compilation error.**



6.9 Storage Classes

- **Each identifier has several attributes**
 - Name, type, size and value
 - Also storage class, scope and linkage
- **C++ provides five storage-class specifiers:**
 - auto, register, extern, mutable and static
- **Identifier's storage class**
 - Determines the period during which that identifier exists in memory
- **Identifier's scope**
 - Determines where the identifier can be referenced in a program



6.9 Storage Classes (Cont.)

- **Identifier's linkage**
 - Determines whether an identifier is known only in the source file where it is declared or across multiple files that are compiled, then linked together
- **An identifier's storage-class specifier helps determine its storage class and linkage**



6.9 Storage Classes (Cont.)

- **Automatic storage class**
 - Declared with keywords **auto** and **register**
 - **Automatic variables**
 - Created when program execution enters block in which they are defined
 - Exist while the block is active
 - Destroyed when the program exits the block
 - Only local variables and parameters can be of automatic storage class
 - Such variables normally are of automatic storage class



Performance Tip 6.1

Automatic storage is a means of conserving memory, because automatic storage class variables exist in memory only when the block in which they are defined is executing.



Software Engineering Observation 6.8

Automatic storage is an example of the principle of least privilege, which is fundamental to good software engineering. In the context of an application, the principle states that code should be granted only the amount of privilege and access that it needs to accomplish its designated task, but no more. Why should we have variables stored in memory and accessible when they are not needed?



Performance Tip 6.2

The storage-class specifier `register` can be placed before an automatic variable declaration to suggest that the compiler maintain the variable in one of the computer's high-speed hardware registers rather than in memory. If intensely used variables such as counters or totals are maintained in hardware registers, the overhead of repeatedly loading the variables from memory into the registers and storing the results back into memory is eliminated.



6.9 Storage Classes (Cont.)

- **Storage-class specifier auto**
 - Explicitly declares variables of automatic storage class
 - Local variables are of automatic storage class by default
 - So keyword `auto` rarely is used
- **Storage-class specifier register**
 - Data in the machine-language version of a program is normally loaded into registers for calculations and other processing
 - Compiler tries to store register storage class variables in a register
 - The compiler might ignore register declarations
 - May not be sufficient registers for the compiler to use



Common Programming Error 6.11

Using multiple storage-class specifiers for an identifier is a syntax error. Only one storage class specifier can be applied to an identifier. For example, if you include register, do not also include auto.



Performance Tip 6.3

Often, register is unnecessary. Today's optimizing compilers are capable of recognizing frequently used variables and can decide to place them in registers without needing a register declaration from the programmer.



6.9 Storage Classes (Cont.)

- **Static storage class**
 - Declared with keywords `extern` and `static`
 - **Static-storage-class variables**
 - Exist from the point at which the program begins execution
 - Initialized once when their declarations are encountered
 - Last for the duration of the program
 - **Static-storage-class functions**
 - The name of the function exists when the program begins execution, just as for all other functions
 - **However, even though the variables and the function names exist from the start of program execution, this does not mean that these identifiers can be used throughout the program.**



6.9 Storage Classes (Cont.)

- **Two types of identifiers with static storage class**
 - External identifiers
 - Such as global variables and global function names
 - Local variables declared with the storage class specifier **static**
- **Global variables**
 - Created by placing variable declarations outside any class or function definition
 - Retain their values throughout the execution of the program
 - Can be referenced by any function that follows their declarations or definitions in the source file



Software Engineering Observation 6.9

Declaring a variable as global rather than local allows unintended side effects to occur when a function that does not need access to the variable accidentally or maliciously modifies it. This is another example of the principle of least privilege. In general, except for truly global resources such as cin and cout, the use of global variables should be avoided except in certain situations with unique performance requirements.



Software Engineering Observation 6.10

Variables used only in a particular function should be declared as local variables in that function rather than as global variables.



6.9 Storage Classes (Cont.)

- **Local variables declared with keyword static**
 - Known only in the function in which they are declared
 - Retain their values when the function returns to its caller
 - Next time the function is called, the static local variables contain the values they had when the function last completed
 - If numeric variables of the static storage class are not explicitly initialized by the programmer
 - They are initialized to zero



6.10 Scope Rules

- **Scope**

- Portion of the program where an identifier can be used
- Four scopes for an identifier
 - Function scope
 - File scope
 - Block scope
 - Function-prototype scope



6.10 Scope Rules (Cont.)

- **File scope**
 - **For an identifier declared outside any function or class**
 - Such an identifier is “known” in all functions from the point at which it is declared until the end of the file
 - **Global variables, function definitions and function prototypes placed outside a function all have file scope**
- **Function scope**
 - **Labels (identifiers followed by a colon such as start:) are the only identifiers with function scope**
 - Can be used anywhere in the function in which they appear
 - Cannot be referenced outside the function body
 - Labels are implementation details that functions hide from one another



6.10 Scope Rules (Cont.)

- **Block scope**
 - **Identifiers declared inside a block have block scope**
 - Block scope begins at the identifier's declaration
 - Block scope ends at the terminating right brace {} of the block in which the identifier is declared
 - **Local variables and function parameters have block scope**
 - The function body is their block
 - **Any block can contain variable declarations**
 - **Identifiers in an outer block can be “hidden” when a nested block has a local identifier with the same name**
 - **Local variables declared static still have block scope, even though they exist from the time the program begins execution**
 - Storage duration does not affect the scope of an identifier



6.10 Scope Rules (Cont.)

- **Function-prototype scope**
 - Only identifiers used in the parameter list of a function prototype have function-prototype scope
 - Parameter names appearing in a function prototype are ignored by the compiler
 - Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity
 - However, in a single prototype, a particular identifier can be used only once



Common Programming Error 6.12

Accidentally using the same name for an identifier in an inner block that is used for an identifier in an outer block, when in fact the programmer wants the identifier in the outer block to be active for the duration of the inner block, is normally a logic error.



Good Programming Practice 6.4

Avoid variable names that hide names in outer scopes. This can be accomplished by avoiding the use of duplicate identifiers in a program.



```
1 // Fig. 6.12: fig06_12.cpp
2 // A scoping example.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void useLocal( void ); // function prototype
8 void useStaticLocal( void ); // function prototype
9 void useGlobal( void ); // function prototype
10
11 int x = 1; // global variable
12
13 int main()
14 {
15     int x = 5; // local variable to main
16
17     cout << "Local x in main's outer scope is "
18
19     { // start new scope
20         int x = 7; // hides x in outer scope
21
22         cout << "Local x in main's inner scope is "
23     } // end new scope
24
25     cout << "Local x in main's outer scope is " << x << endl;
```

Declaring a global variable outside any class or function definition

Local variable x that hides global variable x

Local variable x in a block that hides local variable x in outer scope



```
26
27 useLocal(); // useLocal has local x
28 useStaticLocal(); // useStaticLocal has static local x
29 useGlobal(); // useGlobal uses global x
30 useLocal(); // useLocal reinitializes its local x
31 useStaticLocal(); // static local x retains its prior value
32 useGlobal(); // global x also retains its value
33
34 cout << "\nlocal x in main is " << x << endl;
35 return 0; // indicates successful termination
36 } // end main
37
38 // useLocal reinitializes local variable x during each call
39 void useLocal( void )
40 {
41     int x = 25; // initialized
42
43     cout << "\nlocal x is " << x << " on entering useLocal" << endl;
44     x++;
45     cout << "local x is " << x << " on exiting useLocal" << endl;
46 } // end function useLocal
```

Local variable that gets recreated and
reinitialized each time **useLocal** is called

fig06_12.cpp

(2 of 4)



```
47
48 // useStaticLocal initializes static local variable x only the
49 // first time the function is called; value of x is saved
50 // between calls to this function
51 void useStaticLocal( void )
52 {
53     static int x = 50; // initialized first time useStaticLocal is called
54
55     cout << "\nlocal static x is " << x << " on entering useStaticLocal "
56         << endl ;
57
58     x++;
59
60     cout << "local static x is " << x << " on exiting useStaticLocal "
61         << endl ;
62 } // end function useStaticLocal
63
64 // useGlobal modifies global variable x during each call
65 void useGlobal( void )
66 {
67     cout << "\nglobal x is " << x << " on entering useGlobal " << endl ;
68     x *= 10;
69     cout << "global x is " << x << " on exiting useGlobal " << endl ;
70 } // end function useGlobal
```

static local variable that gets initialized only once

ng06_12.cpp

(3 of 4)

Statement refers to global variable x because no local variable named x exists



fig06_12.cpp

(4 of 4)

```
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5
```

```
local x is 25 on entering useLocal
local x is 26 on exiting useLocal
```

```
local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal
```

```
global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal
```

```
local x is 25 on entering useLocal
local x is 26 on exiting useLocal
```

```
local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal
```

```
global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal
```

```
local x in main is 5
```



6.11 Function Call Stack and Activation Records

- **Data structure: collection of related data items**
- **Stack data structure**
 - Analogous to a pile of dishes
 - When a dish is placed on the pile, it is normally placed at the top
 - Referred to as pushing the dish onto the stack
 - Similarly, when a dish is removed from the pile, it is normally removed from the top
 - Referred to as popping the dish off the stack
 - A last-in, first-out (LIFO) data structure
 - The last item pushed (inserted) on the stack is the first item popped (removed) from the stack



6.11 Function Call Stack and Activation Records (Cont.)

- **Function Call Stack**

- Sometimes called the program execution stack
- Supports the function call/return mechanism
 - Each time a function calls another function, a stack frame (also known as an activation record) is pushed onto the stack
 - Maintains the return address that the called function needs to return to the calling function
 - Contains automatic variables—parameters and any local variables the function declares



6.11 Function Call Stack and Activation Records (Cont.)

- **Function Call Stack (Cont.)**
 - When the called function returns
 - Stack frame for the function call is popped
 - Control transfers to the return address in the popped stack frame
 - If a function makes a call to another function
 - Stack frame for the new function call is simply pushed onto the call stack
 - Return address required by the newly called function to return to its caller is now located at the top of the stack.
- **Stack overflow**
 - Error that occurs when more function calls occur than can have their activation records stored on the function call stack (due to memory limitations)



```
1 // Fig. 6.13: fig06_13.cpp
2 // square function used to demonstrate the function
3 // call stack and activation records.
4 #include <iostream>
5 using std::cin;
6 using std::cout;
7 using std::endl;
8
9 int square( int ); // prototype for function square
10
11 int main()
12 {
13     int a = 10; // value to square (local automatic variable in main)
14
15     cout << a << " squared: " << square( a ) << endl; // display a squared
16     return 0; // indicate successful termination
17 } // end main
18
19 // returns the square of an integer
20 int square( int x ) // x is a local variable
21 {
22     return x * x; // calculate square and return result
23 } // end function square
```

Calling function **square**

10 squared: 100



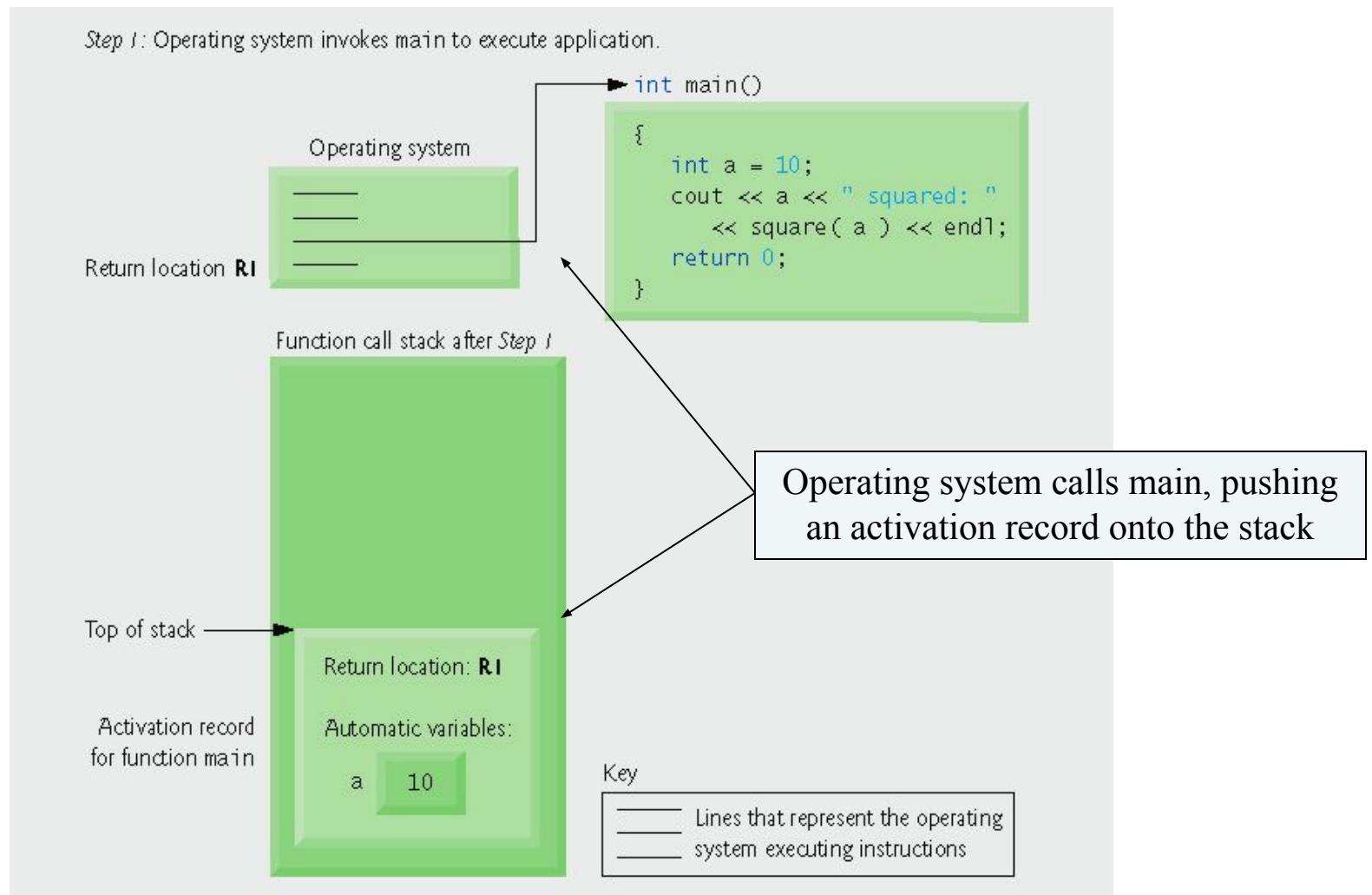


Fig. 6.14 | Function call stack after the operating system invokes main to execute the application.



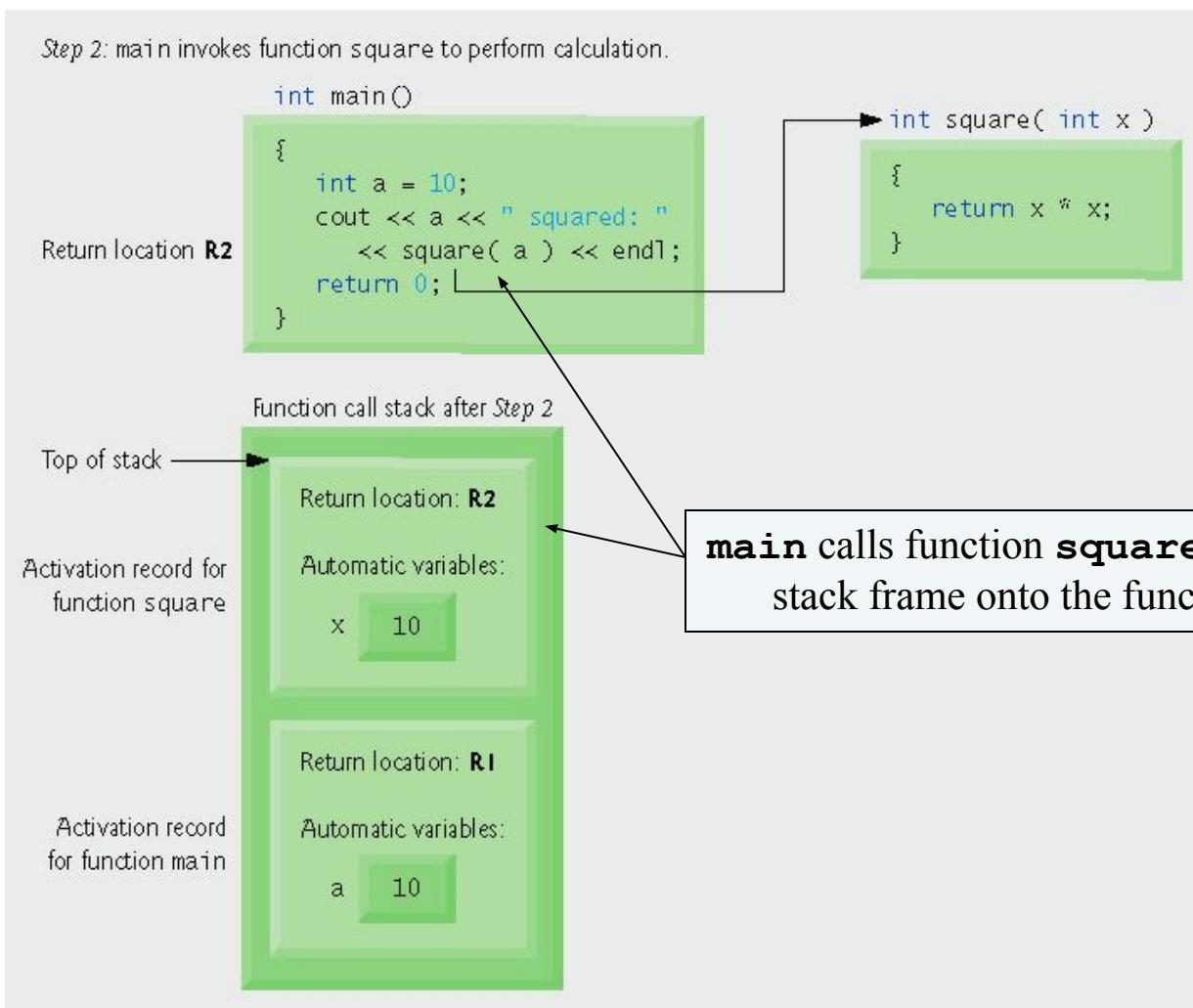


Fig. 6.15 | Function call stack after main invokes function square to perform the calculation.



Step 3: square returns its result to main.

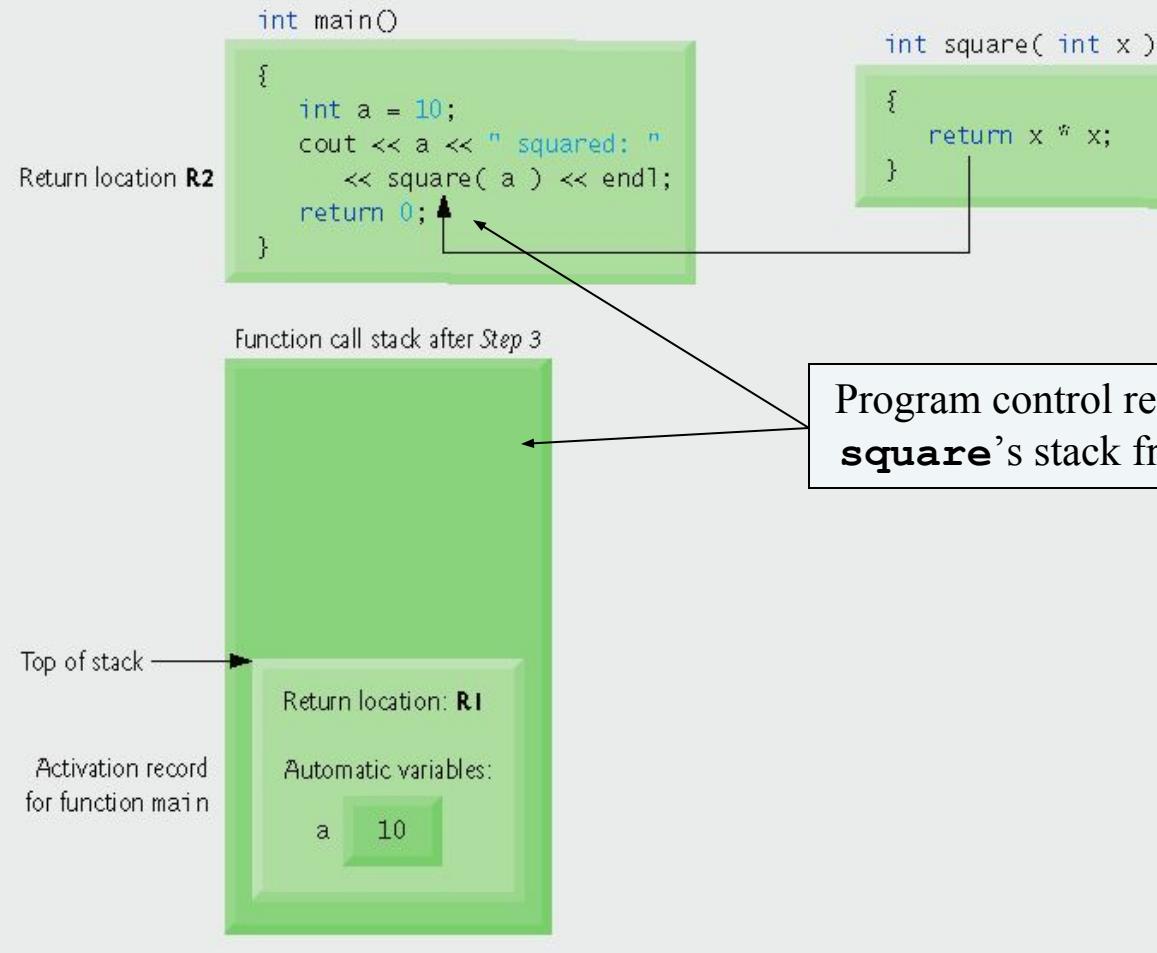


Fig. 6.16 | Function call stack after function square returns to main.



6.12 Functions with Empty Parameter Lists

105

- **Empty parameter list**

- Specified by writing either **void** or **nothing at all** in parentheses
 - For example,

void print();

specifies that function print does not take arguments and does not return a value



Portability Tip 6.2

The meaning of an empty function parameter list in C++ is dramatically different than in C. In C, it means all argument checking is disabled (i.e., the function call can pass any arguments it wants). In C++, it means that the function explicitly takes no arguments. Thus, C programs using this feature might cause compilation errors when compiled in C++.



```
1 // Fig. 6.17: fig06_17.cpp
2 // Functions that take no arguments.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void function1(); // function that takes no arguments
8 void function2( void ); // function that takes no arguments
9
10 int main()
11 {
12     function1(); // call function1 with no arguments
13     function2(); // call function2 with no arguments
14     return 0; // indicates successful termination
15 } // end main
16
17 // function1 uses an empty parameter list to specify that
18 // the function receives no arguments
19 void function1()
20 {
21     cout << "function1 takes no arguments" << endl;
22 } // end function1
```

Specify an empty parameter list by putting nothing in the parentheses

fig06_17.cpp
(1 of 2)

Specify an empty parameter list by putting **void** in the parentheses



```
24 // function2 uses a void parameter list to specify that  
25 // the function receives no arguments  
26 void function2( void )  
27 {  
28     cout << "function2 also takes no arguments" << endl;  
29 } // end function2
```

function1 takes no arguments
function2 also takes no arguments



Common Programming Error 6.13

C++ programs do not compile unless function prototypes are provided for every function or each function is defined before it is called.



6.13 Inline Functions

- **Inline functions**
 - Reduce function call overhead—especially for small functions
 - Qualifier inline before a function's return type in the function definition
 - “Advises” the compiler to generate a copy of the function's code in place (when appropriate) to avoid a function call
 - Trade-off of inline functions
 - Multiple copies of the function code are inserted in the program (often making the program larger)
 - The compiler can ignore the inline qualifier and typically does so for all but the smallest functions



Software Engineering Observation 6.11

Any change to an inline function could require all clients of the function to be recompiled. This can be significant in some program development and maintenance situations.



Good Programming Practice 6.5

The inline qualifier should be used only with small, frequently used functions.



Performance Tip 6.4

**Using inline functions can reduce execution time
but may increase program size.**



Software Engineering Observation 6.12

The `const` qualifier should be used to enforce the principle of least privilege. Using the principle of least privilege to properly design software can greatly reduce debugging time and improper side effects and can make a program easier to modify and maintain.



```
1 // Fig. 6.18: fig06_18.cpp
2 // Using an inline function to calculate the volume of a cube.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 // Definition of inline function cube. Definition of function appears
9 // before function is called, so a function prototype is not required.
10 // First line of function definition acts as the prototype.
11 inline double cube( const double side )
12 {
13     return side * side * side; // calculate cube
14 } // end function cube
15
16 int main()
17 {
18     double sideValue; // stores value entered by user
19     cout << "Enter the side length of your cube: ";
20     cin >> sideValue; // read value from user
21
22     // calculate cube of sideValue and display result
23     cout << "Volume of cube with side "
24         << sideValue << " is " << cube( sideValue ) << endl;
25     return 0; // indicates successful termination
26 } // end main
```

Complete function definition so the compiler knows how to expand a **cube** function call into its inlined code.

cube function call that could be inlined

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```



6.14 References and Reference Parameters

- Two ways to pass arguments to functions
 - Pass-by-value
 - A *copy* of the argument's value is passed to the called function
 - Changes to the copy do not affect the original variable's value in the caller
 - Prevents accidental side effects of functions
 - Pass-by-reference
 - Gives called function the ability to access and modify the caller's argument data directly



Performance Tip 6.5

One disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.



6.14 References and Reference Parameters (Cont.)

- **Reference Parameter**

- An alias for its corresponding argument in a function call
- & placed after the parameter type in the function prototype and function header
- Example
 - int &count in a function header
 - Pronounced as “count is a reference to an int”
 - Parameter name in the body of the called function actually refers to the original variable in the calling function



Performance Tip 6.6

Pass-by-reference is good for performance reasons, because it can eliminate the pass-by-value overhead of copying large amounts of data.



Software Engineering Observation 6.13

Pass-by-reference can weaken security, because the called function can corrupt the caller's data.



```
1 // Fig. 6.19: fig06_19.cpp
2 // Comparing pass-by-value and pass-by-reference with references.
```

```
3 #include <iostream>
4 using std::cout;
5 using std::endl;
```

```
6
7 int squareByValue( int ); // function prototype (value pass)
8 void squareByReference( int & ); // function prototype (reference pass)
```

```
9
10 int main()
11 {
```

```
12     int x = 2; // value to square using squareByValue
13     int z = 4; // value to square using squareByReference
```

```
14
```

```
15     // demonstrate squareByValue
16     cout << "x = " << x << " before squareByValue\n";
17     cout << "Value returned by squareByValue: "
18         << squareByValue( x ) << endl;
19     cout << "x = " << x << " after squareByValue\n" << endl;
```

```
20
21     // demonstrate squareByReference
22     cout << "z = " << z << " before squareByReference" << endl;
23     squareByReference( z );
24     cout << "z = " << z << " after squareByReference" << endl;
25
26 } // end main
27
```

Function illustrating pass-by-value

fig06_19.cpp

(1 of 2)

Function illustrating pass-by-reference

Variable is simply mentioned
by name in both function calls



```
28 // squareByValue multiplies number by itself, stores the
29 // result in number and returns the new value of number
30 int squareByValue( int number )
31 {
32     return number *= number; // caller's argument not modified
33 } // end function squareByValue
34
35 // squareByReference multiplies numberRef by itself and stores the result
36 // in the variable to which numberRef refers in function main
37 void squareByReference( int &numberRef )
38 {
39     numberRef *= numberRef; // caller's argument modified
40 } // end function squareByReference
```

Receives copy of argument in main

fig06_19.cpp

(2 of 2)

Receives reference to argument in main

x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

Modifies variable in main

z = 4 before squareByReference
z = 16 after squareByReference



Common Programming Error 6.14

Because reference parameters are mentioned only by name in the body of the called function, the programmer might inadvertently treat reference parameters as pass-by-value parameters. This can cause unexpected side effects if the original copies of the variables are changed by the function.



Performance Tip 6.7

For passing large objects, use a constant reference parameter to simulate the appearance and security of pass-by-value and avoid the overhead of passing a copy of the large object.



Software Engineering Observation 6.14

Many programmers do not bother to declare parameters passed by value as const, even though the called function should not be modifying the passed argument. Keyword const in this context would protect only a copy of the original argument, not the original argument itself, which when passed by value is safe from modification by the called function.



Software Engineering Observation 6.15

For the combined reasons of clarity and performance, many C++ programmers prefer that modifiable arguments be passed to functions by using pointers (which we study in Chapter 8), small nonmodifiable arguments be passed by value and large nonmodifiable arguments be passed to functions by using references to constants.



6.14 References and Reference Parameters (Cont.)

- **References**

- Can also be used as aliases for other variables within a function
 - All operations supposedly performed on the alias (i.e., the reference) are actually performed on the original variable
 - An alias is simply another name for the original variable
 - Must be initialized in their declarations
 - Cannot be reassigned afterward
- Example
 - `int count = 1;
int &cRef = count;
cRef++;`
 - Increments count through alias cRef



```
1 // Fig. 6.20: fig06_20.cpp
2 // References must be initialized.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int x = 3;
10    int &y = x; // y refers to (is an alias for) x
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7; // actually modifies x
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indicates successful termination
16 } // end main
```

Creating a reference as an alias to another variable in the function

Assign 7 to **x** through alias **y**

```
x = 3
y = 3
x = 7
y = 7
```

fig06_20.cpp
(1 of 1)



```
1 // Fig. 6.21: fig06_21.cpp
2 // References must be initialized.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int x = 3;           // Uninitialized reference
10    int &y; // Error: y must be initialized
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7;
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indicates successful termination
16 } // end main
```

fig06_21.cpp

(1 of 2)

Borland C++ command-line compiler error message:

```
Error E2304 C:\cpphtp5_examples\ch06\Fig06_21\fig06_21.cpp 10:
Reference variable 'y' must be initialized in function main()
```

Microsoft Visual C++ compiler error message:

```
C:\cpphtp5_examples\ch06\Fig06_21\fig06_21.cpp(10) : error C2530: 'y' :
references must be initialized
```

GNU C++ compiler error message:

```
fig06_21.cpp:10: error: 'y' declared as a reference but not initialized
```



6.14 References and Reference Parameters (Cont.)

- Returning a reference from a function
 - Functions can return references to variables
 - Should only be used when the variable is static
 - Dangling reference
 - Returning a reference to an automatic variable
 - That variable no longer exists after the function ends



Common Programming Error 6.15

Not initializing a reference variable when it is declared is a compilation error, unless the declaration is part of a function's parameter list. Reference parameters are initialized when the function in which they are declared is called.



Common Programming Error 6.16

Attempting to reassign a previously declared reference to be an alias to another variable is a logic error. The value of the other variable is simply assigned to the variable for which the reference is already an alias.



Common Programming Error 6.17

Returning a reference to an automatic variable in a called function is a logic error. Some compilers issue a warning when this occurs.



6.15 Default Arguments

- **Default argument**

- A default value to be passed to a parameter
 - Used when the function call does not specify an argument for that parameter
- Must be the rightmost argument(s) in a function's parameter list
- Should be specified with the first occurrence of the function name
 - Typically the function prototype



Common Programming Error 6.18

It is a compilation error to specify default arguments in both a function's prototype and header.



```
1 // Fig. 6.22: fig06_22.cpp
2 // Using default arguments.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // function prototype that specifies default arguments
8 int boxVolume( int length = 1, int width = 1, int height = 1 );
9
10 int main()
11 {
12     // no arguments--use default values for all dimensions
13     cout << "The default box volume is: " << boxVolume();
14
15     // specify length; default width and height
16     cout << "\n\nThe volume of a box with length 10,\n"
17         << "width 1 and height 1 is: " << boxVolume( 10 );
18
19     // specify length and width; default height
20     cout << "\n\nThe volume of a box with length 10,\n"
21         << "width 5 and height 1 is: " << boxVolume( 10, 5 );
22
23     // specify all arguments
24     cout << "\n\nThe volume of a box with length 10,\n"
25         << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
26         << endl;
27     return 0; // indicates successful termination
28 } // end main
```

fig06_22.cpp

(1 of 2)

Default arguments

Calling function with no arguments

Calling function with one argument

Calling function with two arguments

Calling function with three arguments



```
30 // function boxVolume calculates the volume of a box
31 int boxVolume( int length, int width, int height )
32 {
33     return length * width * height;
34 } // end function boxVolume
```

The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100

Note that default arguments were specified in the function prototype, so they are not specified in the function header



Good Programming Practice 6.6

Using default arguments can simplify writing function calls. However, some programmers feel that explicitly specifying all arguments is clearer.



Software Engineering Observation 6.16

If the default values for a function change, all client code using the function must be recompiled.



Common Programming Error 6.19

Specifying and attempting to use a default argument that is not a rightmost (trailing) argument (while not simultaneously defaulting all the rightmost arguments) is a syntax error.



6.16 Unary Scope Resolution Operator

- **Unary scope resolution operator (::)**
 - Used to access a global variable when a local variable of the same name is in scope
 - Cannot be used to access a local variable of the same name in an outer block



Common Programming Error 6.20

It is an error to attempt to use the unary scope resolution operator (:) to access a nonglobal variable in an outer block. If no global variable with that name exists, a compilation error occurs. If a global variable with that name exists, this is a logic error, because the program will refer to the global variable when you intended to access the nonglobal variable in the outer block.



Good Programming Practice 6.7

Always using the unary scope resolution operator (::) to refer to global variables makes programs easier to read and understand, because it makes it clear that you are intending to access a global variable rather than a nonglobal variable.



```
1 // Fig. 6.23: fig06_23.cpp
2 // Using the unary scope resolution operator.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int number = 7; // global variable named number
8
9 int main()
10 {
11     double number = 10.5; // local variable named number
12
13     // display values of local and global variables
14     cout << "Local double value of number = " << number
15     << "\nGlobal int value of number = " << ::number << endl;
16     return 0; // indicates successful termination
17 } // end main
```

fig06_23.cpp

(1 of 1)

Unary scope resolution operator used
to access global variable **number**

```
Local double value of number = 10.5
Global int value of number = 7
```



Software Engineering Observation 6.17

Always using the unary scope resolution operator (::) to refer to global variables makes programs easier to modify by reducing the risk of name collisions with nonglobal variables.



Error-Prevention Tip 6.4

Always using the unary scope resolution operator (::) to refer to a global variable eliminates possible logic errors that might occur if a nonglobal variable hides the global variable.



Error-Prevention Tip 6.5

Avoid using variables of the same name for different purposes in a program. Although this is allowed in various circumstances, it can lead to errors.



6.17 Function Overloading

- **Overloaded functions**

- Overloaded functions have
 - Same name
 - Different sets of parameters
- Compiler selects proper function to execute based on number, types and order of arguments in the function call
- Commonly used to create several functions of the same name that perform similar tasks, but on different data types



Good Programming Practice 6.8

Overloading functions that perform closely related tasks can make programs more readable and understandable.



```
1 // Fig. 6.24: fig06_24.cpp
2 // Overloaded functions.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // function square for int values
8 int square( int x ) ← Defining a square function for ints
9 {
10    cout << "square of integer " << x << " is ";
11    return x * x;
12 } // end function square with int argument
13
14 // function square for double values
15 double square( double y ) ← Defining a square function for doubles
16 {
17    cout << "square of double " << y << " is ";
18    return y * y;
19 } // end function square with double argument
```



```
20
21 int main()
22 {
23     cout << square( 7 ); // calls int version
24     cout << endl ;
25     cout << square( 7.5 ); // calls double version
26     cout << endl ;
27     return 0; // indicates successful termination
28 } // end main
```

fig06_24.cpp

(2 of 2)

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

Output confirms that the proper function was called in each case



6.17 Function Overloading (Cont.)

- **How the compiler differentiates overloaded functions**
 - Overloaded functions are distinguished by their signatures
 - Name mangling or name decoration
 - Compiler encodes each function identifier with the number and types of its parameters to enable type-safe linkage
 - Type-safe linkage ensures that
 - Proper overloaded function is called
 - Types of the arguments conform to types of the parameters



```
1 // Fig. 6.25: fig06_25.cpp
2 // Name mangling.
3
4 // function square for int values
5 int square( int x )
6 {
7     return x * x;
8 } // end function square
9
10 // function square for double values
11 double square( double y )
12 {
13     return y * y;
14 } // end function square
15
16 // function that receives arguments of types
17 // int, float, char and int &
18 void nothing1( int a, float b, char c, int &d )
19 {
20     // empty function body
21 } // end function nothing1
```

Overloaded **square** functions

fig06_25.cpp

(1 of 2)



```
23 // function that receives arguments of types
24 // char, int, float & and double &
25 int nothing2( char a, int b, float &c, double &d )
26 {
27     return 0;
28 } // end function nothing2
29
30 int main()
31 {
32     return 0; // indicates successful termination
33 } // end main
```

@square\$qi
@square\$qd
@nothing1\$qi fcri
@nothing2\$qci rfrd
main

Mangled names of overloaded functions

main is not mangled because it cannot be overloaded



Common Programming Error 6.21

Creating overloaded functions with identical parameter lists and different return types is a compilation error.



Common Programming Error 6.22

A function with default arguments omitted might be called identically to another overloaded function; this is a compilation error. For example, having in a program both a function that explicitly takes no arguments and a function of the same name that contains all default arguments results in a compilation error when an attempt is made to use that function name in a call passing no arguments. The compiler does not know which version of the function to choose.



6.18 Function Templates

- **Function templates**
 - More compact and convenient form of overloading
 - Identical program logic and operations for each data type
 - Function template definition
 - Written by programmer once
 - Essentially defines a whole family of overloaded functions
 - Begins with the template keyword
 - Contains template parameter list of formal type parameters for the function template enclosed in angle brackets (<>)
 - Formal type parameters
 - Preceded by keyword typename or keyword class
 - Placeholders for fundamental types or user-defined types



6.18 Function Templates (Cont.)

- **Function-template specializations**
 - Generated automatically by the compiler to handle each type of call to the function template
 - Example for function template **max** with type parameter **T** called with **int** arguments
 - Compiler detects a **max** invocation in the program code
 - **int** is substituted for **T** throughout the template definition
 - This produces function-template specialization **max< int >**



```
1 // Fig. 6.26: maxi num.h
2 // Definition of function template maxi num
3
4 template < class T > // or template< typename T >
5 T maxi num( T value1, T value2, T value3 )
6 {
7     T maxi numVal ue = value1; // assume value1 is maxi num
8
9     // determine whether value2 is greater than maxi numVal ue
10    if ( value2 > maxi numVal ue )
11        maxi numVal ue = value2;
12
13    // determine whether value3 is greater than maxi numVal ue
14    if ( value3 > maxi numVal ue )
15        maxi numVal ue = value3;
16
17    return maxi numVal ue;
18 } // end function template maxi num
```

Using formal type parameter **T** in place of data type

(1 of 1)



Common Programming Error 6.23

Not placing keyword `class` or keyword `typename` before every formal type parameter of a function template (e.g., writing `< class S, T >` instead of `< class S, class T >`) is a syntax error.



```
1 // Fig. 6.27: fig06_27.cpp
2 // Function template maxi num test program
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include "maxi num.h" // include definition of function template maxi num
9
10 int main()
11 {
12     // demonstrate maxi num with int values
13     int int1, int2, int3;
14
15     cout << "Input three integer values: ";
16     cin >> int1 >> int2 >> int3;
17
18     // invoke int version of maxi num
19     cout << "The maxi num integer value is: "
20     << maxi num( int1, int2, int3 );
21
22     // demonstrate maxi num with double values
23     double double1, double2, double3;
24
25     cout << "\n\nInput three double values: ";
26     cin >> double1 >> double2 >> double3;
27
28     // invoke double version of maxi num
29     cout << "The maxi num double value is: "
30     << maxi num( double1, double2, double3 );
```

Invoking **maximum** with **int** arguments

Invoking **maximum** with **double** arguments



```
31
32 // demonstrate maximum with char values
33 char char1, char2, char3;
34
35 cout << "\n\n input three characters: ";
36 cin >> char1 >> char2 >> char3;
37
38 // invoke char version of maximum
39 cout << "The maximum character value is: "
40     << maximum( char1, char2, char3 ) << endl ;
41 return 0; // indicates successful termination
42 } // end main
```

fig06_27.cpp

(2 of 2)

Invoking **maximum** with **char** arguments

I input three integer values: 1 2 3
The maximum integer value is: 3

I input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

I input three characters: A C B
The maximum character value is: C



6.19 Recursion

- **Recursive function**

- A function that calls itself, either directly, or indirectly (through another function)

- **Recursion**

- **Base case(s)**
 - The simplest case(s), which the function knows how to handle
 - For all other cases, the function typically divides the problem into two conceptual pieces
 - A piece that the function knows how to do
 - A piece that it does not know how to do
 - Slightly simpler or smaller version of the original problem



6.19 Recursion (Cont.)

- **Recursion (Cont.)**

- **Recursive call (also called the recursion step)**
 - The function launches (calls) a fresh copy of itself to work on the smaller problem
 - Can result in many more recursive calls, as the function keeps dividing each new problem into two conceptual pieces
 - This sequence of smaller and smaller problems must eventually converge on the base case
 - Otherwise the recursion will continue forever

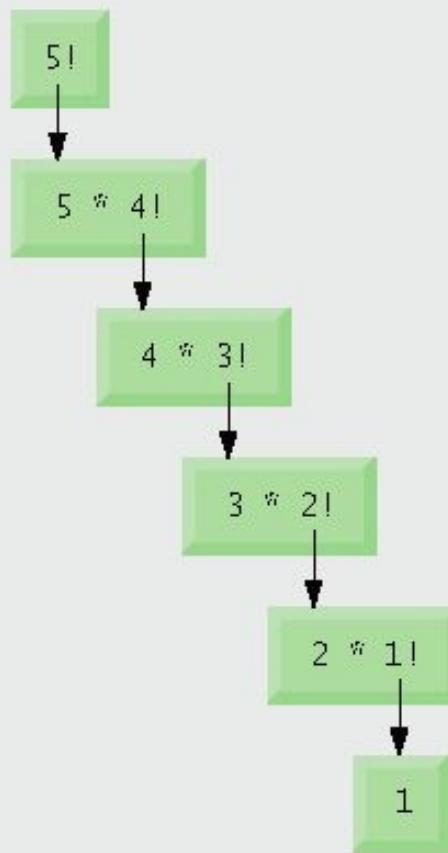


6.19 Recursion (Cont.)

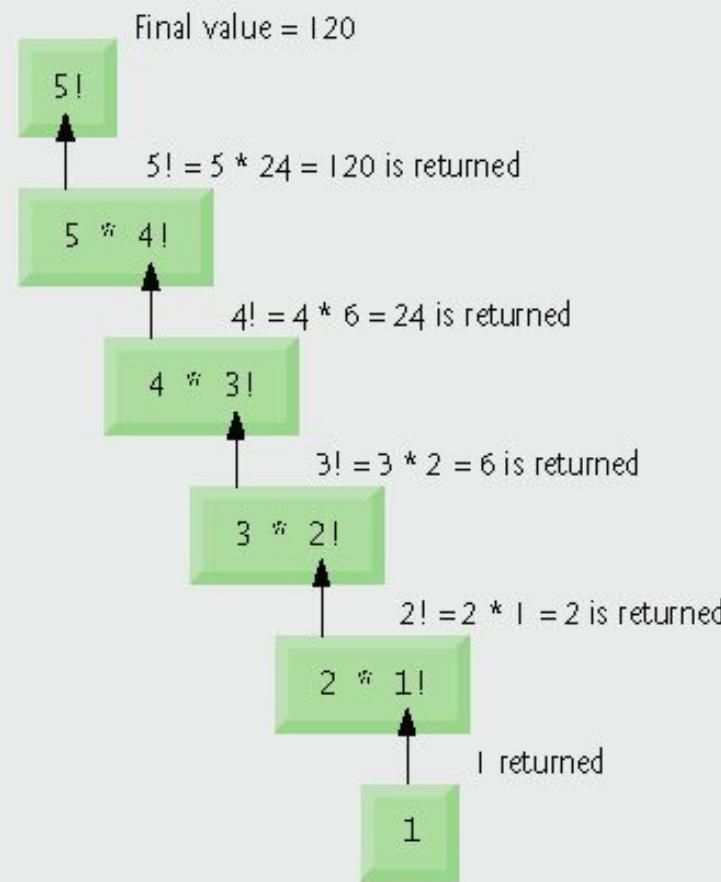
- **Factorial**

- The factorial of a nonnegative integer n , written $n!$ (and pronounced “ n factorial”), is the product
 - $n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$
- Recursive definition of the factorial function
 - $n! = n \cdot (n - 1)!$
 - Example
 - $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$
 $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$
 $5! = 5 \cdot (4!)$





(a) Procession of recursive calls.



(b) Values returned from each recursive call.

Fig. 6.28 | Recursive evaluation of 5!.

```
1 // Fig. 6.29: fig06_29.cpp
2 // Testing the recursive factorial function.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 unsigned long factorial( unsigned long ); // function prototype
11
12 int main()
13 {
14     // calculate the factorials of 0 through 10
15     for ( int counter = 0; counter <= 10; counter++ )
16         cout << setw( 2 ) << counter << "!" << factorial( counter )
17             << endl;
18
19     return 0; // indicates successful termination
20 } // end main
```

fig06_29.cpp

(1 of 2)

First call to **factorial** function



```
21
22 // recursive definition of function factorial
23 unsigned long factorial( unsigned long number )
24 {
25     if ( number <= 1 ) // test for base case
26         return 1; // base cases: 0! = 1 and 1! = 1
27     else // recursion step
28         return number * factorial( number - 1 );
29 } // end function factorial
```

Base cases simply **return 1**

fig06_29.cpp

(2 of 2)

$0! = 1$
 $1! = 1$
 $2! = 2$
 $3! = 6$
 $4! = 24$
 $5! = 120$
 $6! = 720$
 $7! = 5040$
 $8! = 40320$
 $9! = 362880$
 $10! = 3628800$

Recursive call to **factorial** function
with a slightly smaller problem



Common Programming Error 6.24

Either omitting the base case, or writing the recursion step incorrectly so that it does not converge on the base case, causes “infinite” recursion, eventually exhausting memory. This is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.



6.20 Example Using Recursion: Fibonacci Series

- The Fibonacci series
 - 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
 - Begins with 0 and 1
 - Each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers
 - can be defined recursively as follows:
 - $\text{fibonacci}(0) = 0$
 - $\text{fibonacci}(1) = 1$
 - $\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$



```
1 // Fig. 6.30: fig06_30.cpp
2 // Testing the recursive fibonacci function.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 unsigned long fibonacci( unsigned long ); // function prototype
9
10 int main()
11 {
12     // calculate the fibonacci values of 0 through 10
13     for ( int counter = 0; counter <= 10; counter++ )
14         cout << "fibonacci( " << counter << " ) = "
15             << fibonacci( counter ) << endl;
16
17     // display higher fibonacci values
18     cout << "fibonacci( 20 ) = " << fibonacci( 20 ) << endl;
19     cout << "fibonacci( 30 ) = " << fibonacci( 30 ) << endl;
20     cout << "fibonacci( 35 ) = " << fibonacci( 35 ) << endl;
21     return 0; // indicates successful termination
22 } // end main
23
```

fig06_30.cpp

(1 of 2)



```
24 // recursive method fibonacci
25 unsigned long fibonacci( unsigned long number )
26 {
27     if ( ( number == 0 ) || ( number == 1 ) ) // base cases
28         return number;
29     else // recursion step
30         return fibonacci( number - 1 ) + fibonacci( number - 2 );
31 } // end function fibonacci
```

```
fibonacci( 0 ) = 0
fibonacci( 1 ) = 1
fibonacci( 2 ) = 1
fibonacci( 3 ) = 2
fibonacci( 4 ) = 3
fibonacci( 5 ) = 5
fibonacci( 6 ) = 8
fibonacci( 7 ) = 13
fibonacci( 8 ) = 21
fibonacci( 9 ) = 34
fibonacci( 10 ) = 55
fibonacci( 20 ) = 6765
fibonacci( 30 ) = 832040
fibonacci( 35 ) = 9227465
```

Base cases

Recursive calls to **fibonacci** function

fig06_30.cpp

(2 of 2)



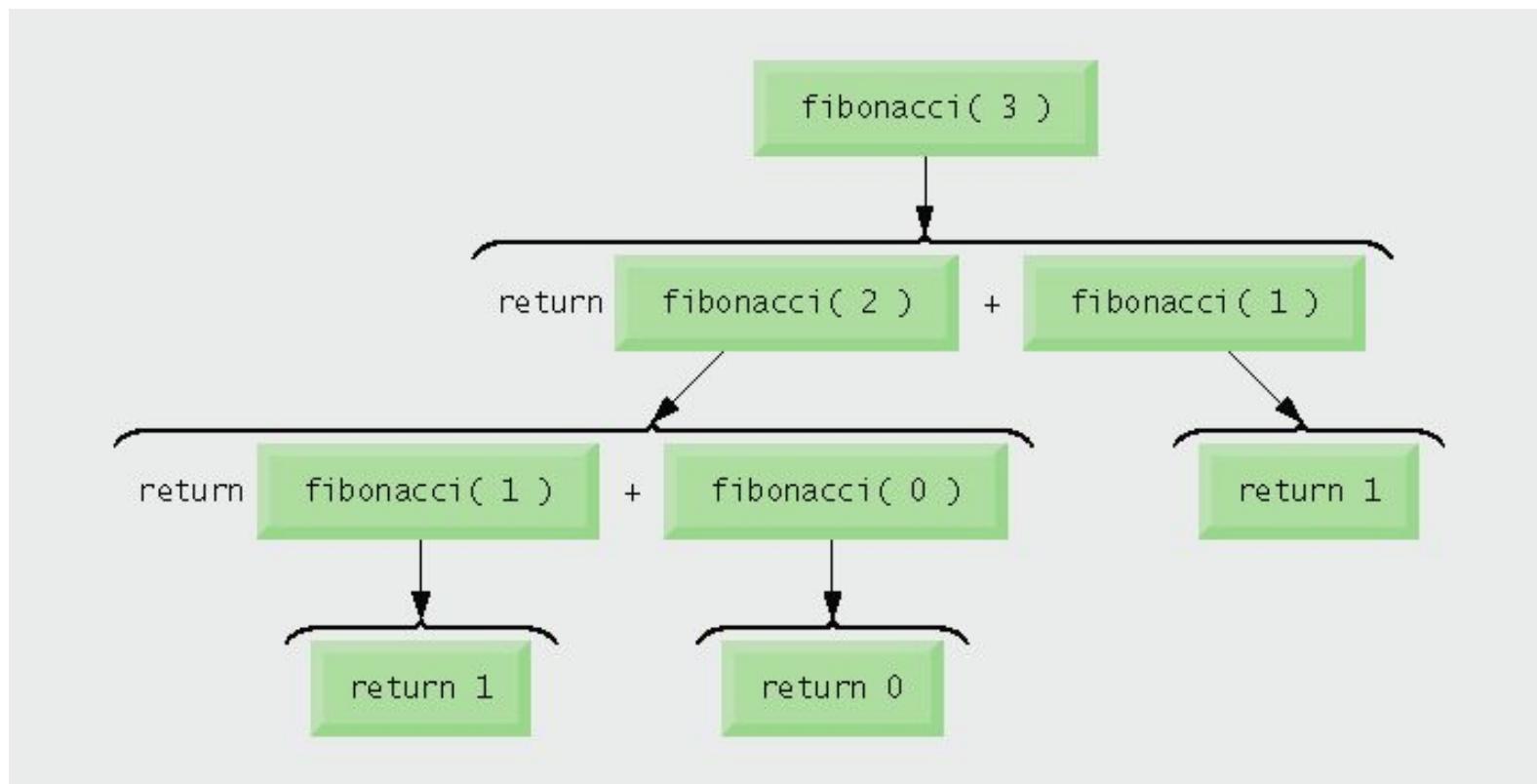


Fig. 6.31 | Set of recursive calls to function `fibonacci`.



Common Programming Error 6.25

Writing programs that depend on the order of evaluation of the operands of operators other than &&, ||, ?: and the comma (,) operator can lead to logic errors.



Portability Tip 6.3

Programs that depend on the order of evaluation of the operands of operators other than &&, ||, ?: and the comma (,) operator can function differently on systems with different compilers.



6.20 Example Using Recursion: Fibonacci Series (Cont.)

- **Caution about recursive programs**
 - Each level of recursion in function fibonacci has a doubling effect on the number of function calls
 - i.e., the number of recursive calls that are required to calculate the n th Fibonacci number is on the order of 2^n
 - 20th Fibonacci number would require on the order of 2^{20} or about a million calls
 - 30th Fibonacci number would require on the order of 2^{30} or about a billion calls.
 - Exponential complexity
 - Can humble even the world's most powerful computers



Performance Tip 6.8

Avoid Fibonacci-style recursive programs that result in an exponential “explosion” of calls.



6.21 Recursion vs. Iteration

- Both are based on a control statement
 - Iteration – repetition structure
 - Recursion – selection structure
- Both involve repetition
 - Iteration – explicitly uses repetition structure
 - Recursion – repeated function calls
- Both involve a termination test
 - Iteration – loop-termination test
 - Recursion – base case



6.21 Recursion vs. Iteration (Cont.)

- Both gradually approach termination
 - Iteration modifies counter until loop-termination test fails
 - Recursion produces progressively simpler versions of problem
- Both can occur infinitely
 - Iteration – if loop-continuation condition never fails
 - Recursion – if recursion step does not simplify the problem



```
1 // Fig. 6.32: fig06_32.cpp
2 // Testing the iterative factorial function.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 unsigned long factorial( unsigned long ); // function prototype
11
12 int main()
13 {
14     // calculate the factorials of 0 through 10
15     for ( int counter = 0; counter <= 10; counter++ )
16         cout << setw( 2 ) << counter << "!" = " << factorial( counter )
17             << endl;
18
19     return 0;
20 } // end main
21
22 // iterative function factorial
23 unsigned long factorial( unsigned long number )
24 {
25     unsigned long result = 1;
```

fig06_32.cpp

(1 of 2)



```
26
27 // iterative declaration of function factorial
28 for ( unsigned long i = number; i >= 1; i-- )
29     result *= i;
30
31 return result;
32 } // end function factorial
```

Iterative approach to finding a factorial

ng06_32.cpp

(2 of 2)

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800



6.21 Recursion vs. Iteration (Cont.)

- **Negatives of recursion**
 - Overhead of repeated function calls
 - Can be expensive in both processor time and memory space
 - Each recursive call causes another copy of the function (actually only the function's variables) to be created
 - Can consume considerable memory
- **Iteration**
 - Normally occurs within a function
 - Overhead of repeated function calls and extra memory assignment is omitted



Software Engineering Observation 6.18

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution is not apparent.



Performance Tip 6.9

**Avoid using recursion in performance situations.
Recursive calls take time and consume additional
memory.**



Common Programming Error 6.26

Accidentally having a nonrecursive function call itself, either directly or indirectly (through another function), is a logic error.



Location in Text	Recursion Examples and Exercises
<i>Chapter 6</i>	
Section 6.19, Fig. 6.29	Factorial function
Section 6.19, Fig. 6.30	Fibonacci function
Exercise 6.7	Sum of two integers
Exercise 6.40	Raising an integer to an integer power
Exercise 6.42	Towers of Hanoi
Exercise 6.44	Visualizing recursion
Exercise 6.45	Greatest common divisor
Exercise 6.50, Exercise 6.51	Mystery “What does this program do?” exercise

Fig. 6.33 | Summary of recursion examples and exercises in the text. (Part 1 of 3)



Location in Text	Recursion Examples and Exercises
<i>Chapter 7</i>	
Exercise 7.18	Mystery “What does this program do?” exercise
Exercise 7.21	Mystery “What does this program do?” exercise
Exercise 7.31	Selection sort
Exercise 7.32	Determine whether a string is a palindrome
Exercise 7.33	Linear search
Exercise 7.34	Binary search
Exercise 7.35	Eight Queens
Exercise 7.36	Print an array
Exercise 7.37	Print a string backward
Exercise 7.38	Minimum value in an array
<i>Chapter 8</i>	
Exercise 8.24	Quicksort
Exercise 8.25	Maze traversal
Exercise 8.26	Generating Mazes Randomly
Exercise 8.27	Mazes of Any Size

Fig. 6.33 | Summary of recursion examples and exercises in the text. (Part 2 of 3)



Location in Text	Recursion Examples and Exercises
<i>Chapter 20</i>	
Section 20.3.3, Figs. 20.5–20.7	Mergesort
Exercise 20.8	Linear search
Exercise 20.9	Binary search
Exercise 20.10	Quicksort
<i>Chapter 21</i>	
Section 21.7, Figs. 21.20–21.22	Binary tree insert
Section 21.7, Figs. 21.20–21.22	Preorder traversal of a binary tree
Section 21.7, Figs. 21.20–21.22	Inorder traversal of a binary tree
Section 21.7, Figs. 21.20–21.22	Postorder traversal of a binary tree
Exercise 21.20	Print a linked list backward
Exercise 21.21	Search a linked list
Exercise 21.22	Binary tree delete
Exercise 21.25	Printing tree

Fig. 6.33 | Summary of recursion examples and exercises in the text. (Part 3 of 3)



6.22 (Optional) Software Engineering Case Study: Identifying Class Operations in the ATM System

- **Operation**

- A service that objects of a class provide to their clients
 - For example, a radio's operations include setting its station and volume
- Implemented as a member function in C++
- Identifying operations
 - Examine key verbs and verb phrases in the requirements document



Class	Verbs and verb phrases
ATM	executes financial transactions
Balance inquiry	[none in the requirements document]
Withdrawal	[none in the requirements document]
Deposit	[none in the requirements document]
BankDatabase	authenticates a user, retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
Account	retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
Screen	displays a message to the user
Keypad	receives numeric input from the user
CashDispenser	dispenses cash, indicates whether it contains enough cash to satisfy a withdrawal request
DepositSlot	receives a deposit envelope

Fig. 6.34 | Verbs and verb phrases for each class in the ATM system.



6.22 (Optional) Software Engineering Case Study: Identifying Class Operations in the ATM System (Cont.)

- **Modeling operations in UML**
 - Each operation is given an operation name, a parameter list and a return type:
 - $\text{operationName}(\text{parameter}1, \dots, \text{parameter}N) : \text{return type}$
 - Each parameter has a parameter name and a parameter type
 - $\text{parameterName} : \text{parameterType}$
 - Some operations may not have return types yet
 - Remaining return types will be added as design and implementation proceed



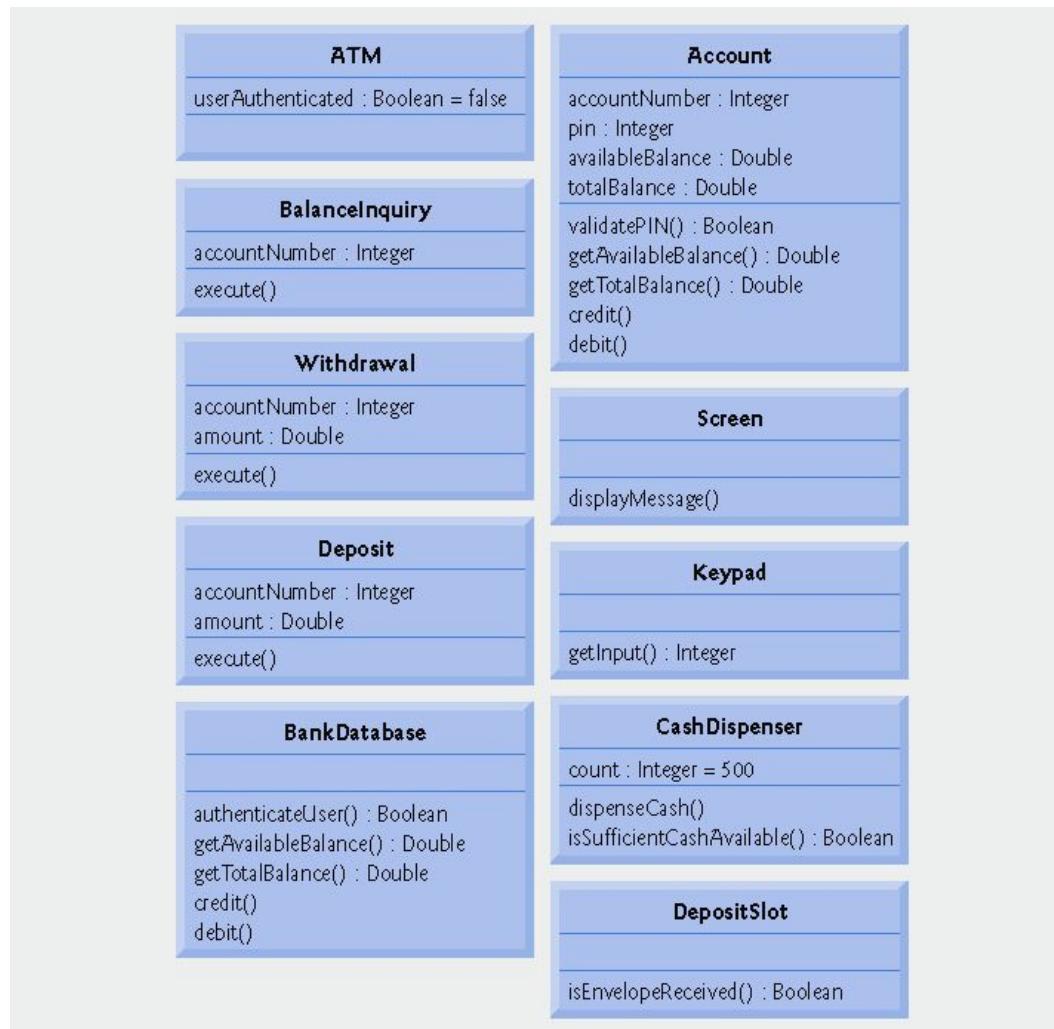


Fig. 6.35 | Classes in the ATM system with attributes and operations.



6.22 (Optional) Software Engineering Case Study: Identifying Class Operations in the ATM System (Cont.)

- **Identifying and modeling operation parameters**
 - Examine what data the operation requires to perform its assigned task
 - Additional parameters may be added later on



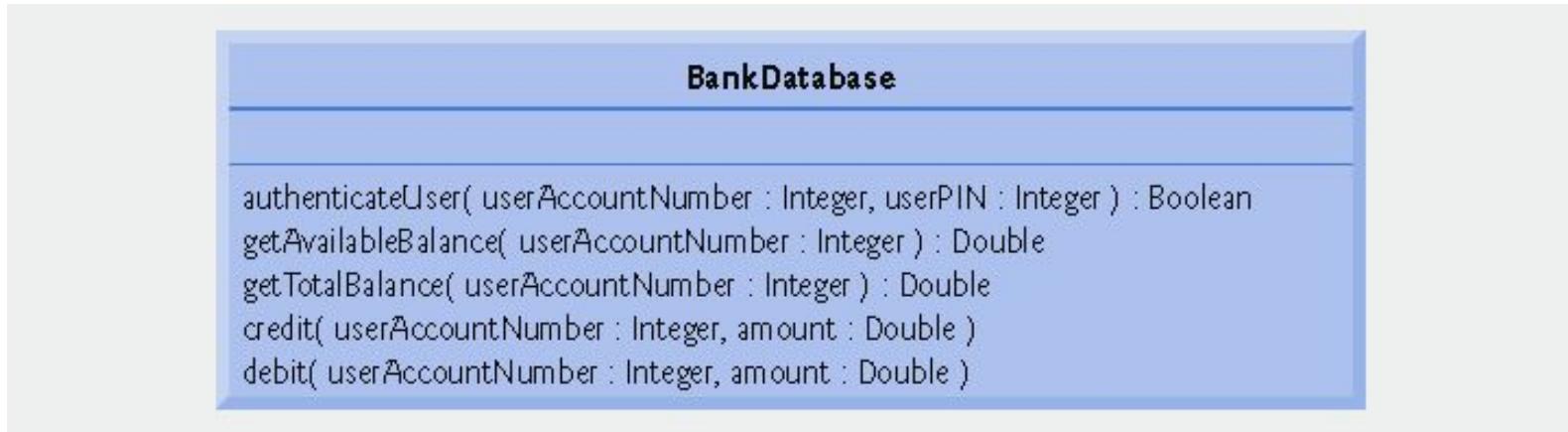


Fig. 6.36 | Class BankDatabase with operation parameters.



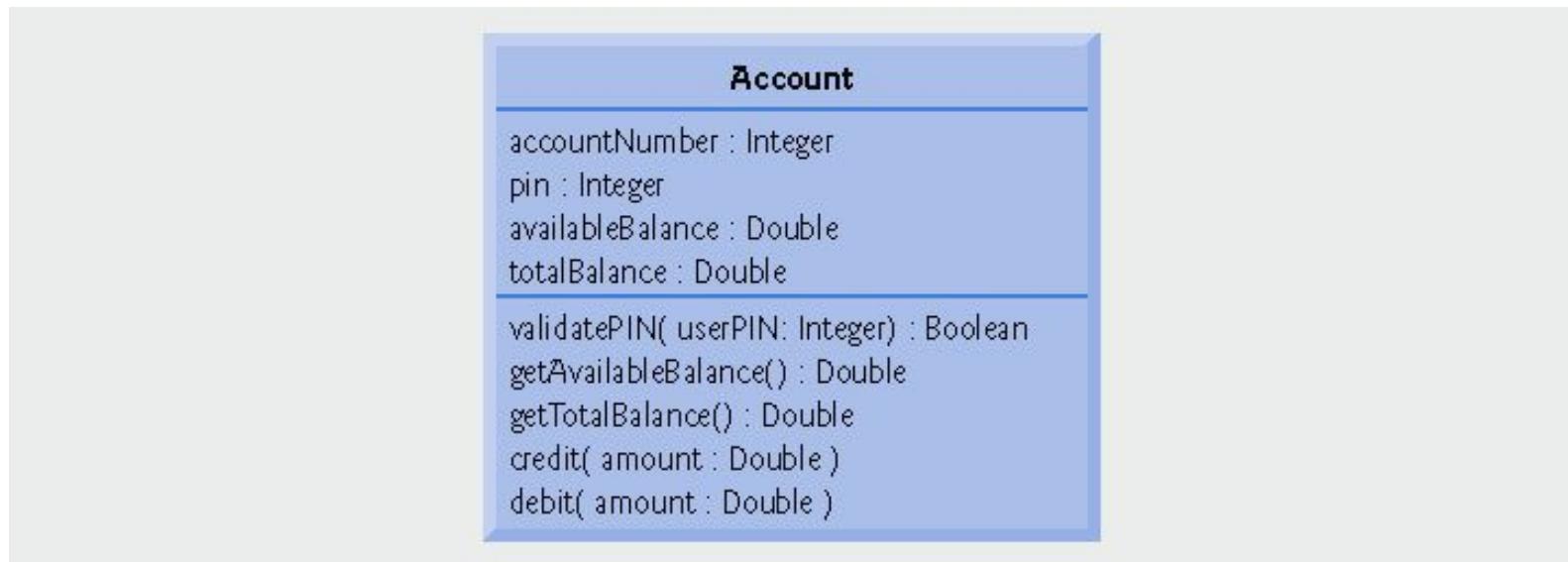


Fig. 6.37 | Class Account with operation parameters.





Fig. 6.38 | Class Screen with operation parameters.





Fig. 6.39 | Class CashDispenser with operation parameters.

