Chapter 5 Functions and an Introduction to Recursion

C++ How to Program, Late Objects Version, 7/e

Chapter 5

by Windows User

OBJECTIVES

In this chapter you'll learn:

- To construct programs modularly from functions.
- To use common math library functions.
- The mechanisms for passing data to functions and returning results.
- The function call mechanism 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 recursive functions.

5.1 Introduction

- Construct programs from small, simple pieces, or components.
  - divide and conquer
- This chapter emphasizes how to declare and use functions to facilitate the design, implementation, operation and maintenance of large programs.
- We'll overview several C++ Standard Library math functions.
- You'll learn how to declare your own functions.
- We'll discuss function prototypes and how the compiler uses them to ensure that functions are called properly.
- We'll take a brief diversion into simulation techniques with random number generation and develop a version of the casino dice game called craps that uses most of the programming techniques you've learned.

5.1 Introduction (cont.)

- We'll discuss C++'s storage classes and scope rules.
- You'll learn how C++ keeps track of which function is currently executing, how parameters and other local variables of functions are maintained in memory and how a function knows where to return after it completes execution.
- We discuss topics that help improve program performance
  - inline functions that can eliminate the overhead of a function call
  - reference parameters that can be used to pass large data items to functions efficiently.

5.1 Introduction (cont.)

- Many of the applications you develop will have more than one function of the same name.
  - Function overloading
  - Used to implement functions that perform similar tasks for arguments of different types or possibly for different numbers of arguments.
- We consider function templates—a mechanism for defining a family of overloaded functions.
- The chapter concludes with a discussion of functions that call themselves, either directly, or indirectly through another function—a topic called recursion.

5.2 Program Components in C++

- C++ programs are typically written by combining new functions and classes you write with “prepackaged” functions and classes available in the C++ Standard Library.
- The C++ Standard Library provides a rich collection of functions for
  - common mathematical calculations,
  - string manipulations,
  - character manipulations,
  - input/output,
  - error checking and
  - many other useful operations.

5.2 Program Components in C++ (cont.)

- Functions allow you to modularize a program by separating its tasks into self-contained units.
- We've used library functions in many of the programs you've seen so far in this book.
- Functions you write are referred to as user-defined functions or programmer-defined functions.
- The statements in function bodies are written only once, are reused from perhaps several locations in a program and are hidden from other functions.

5.2 Program Components in C++ (cont.)

- Motivations for "functionalizing" a program.
  - Divide-and-conquer makes program development more manageable.
  - Software reusability—using existing functions as building blocks to create new programs.
    - With good function naming and definition, programs can be created from standardized functions that accomplish specific tasks, rather than being built by using customized code.
  - Avoid repeating code in a program.
    - Packaging code as a function allows the code to be executed from different locations in a program simply by calling the function.

Software Engineering Observation 5.1

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.

Software Engineering Observation 5.2

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

5.2 Program Components in C++ (cont.)

- A function is invoked by a function call, and when the called function completes its task, it either returns a result or simply returns control to the caller.
- An analogy to this program structure is the hierarchical form of management (Figure 5.1).

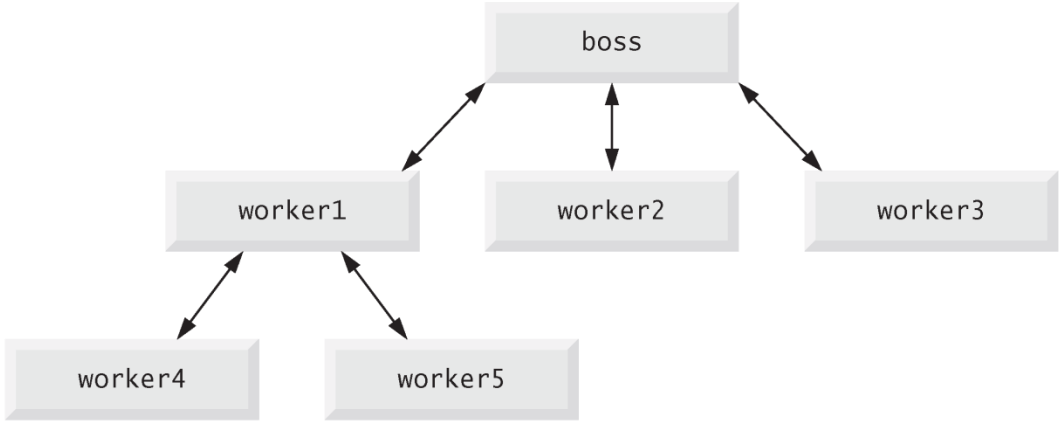


Fig. 5.1 | Hierarchical boss function/worker function relationship.

5.3 Math Library Functions

- Sometimes functions are *not* members of a class.
  - Such functions are called global functions.
- Throughout the book, we'll use a combination of global functions (such as `main`) and classes with member functions to implement our example programs.
- The `<cmath>` header file provides a collection of functions that enable you to perform common mathematical calculations.
- All functions in the `<cmath>` header file are global functions—each is called simply by specifying the name of the function followed by parentheses containing the function's arguments.
- Function arguments may be constants, variables or more complex expressions.
- Some math library functions are summarized in Fig. 5.2.
  - In the figure, the variables `x` and `y` are of type `double`.

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 `2.718282` `exp( 2.0 )` is `7.389056`

`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`

Fig. 5.2 | Math library functions. (Part 1 of 2.)

Function	Description	Example
`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. 5.2 | Math library functions. (Part 2 of 2.)

5.4 Function Definitions

- Each program we've presented consisted of function `main` calling standard library functions to accomplish its tasks.
- Now you'll learn how to write your own customized functions.
- Consider a program with a programmer-defined function `square` that calculates and displays the squares of the integers from 1 to 10 (Fig. 5.3; lines 19–22).

```
1  // Fig. 5.3: fig05_03.cpp
2  // Creating and using a programmer-defined function.
3  #include <iostream>
4  using namespace std;
5
6  int square( int ); // function prototype
7
8  int main()
9  {
10    // loop 10 times and calculate and output the
11    // square of x each time
12    for ( int x = 1; x <= 10; x++ )
13       cout << square( x ) << "  ";  // function call
14
15    cout << endl;
16 } // end main
17
18 // square function definition returns square of an integer
19 int square( int y )  // y is a copy of argument to function
20 {
21    return y * y;     // returns square of y as an int
22 } // end function square
```

```
1  4  9  16  25  36  49  64  81  100
```

Fig. 5.3 | Programmer-defined function `square`.

5.4 Program Components in C++ (cont.)

- The format of a function definition is as follows:

```
return-value-type  function-name( parameter-list )
{
    declarations and statements
}
```

- The *function-name* is any valid identifier.
- The *return-value-type* is the data type of the result returned from the function to the caller.
- The *return-value-type* `void` indicates that a function does not return a value.
- All variables defined in function definitions are local variables—they're known only in the function in which they're defined.
- Most functions have a list of parameters that provide the means for communicating information between functions.
- A function's parameters are also local variables of that function.

5.4 Function Definitions (cont.)

- Function `square` is invoked or called in `main` with the expression `square( x )` in line 13.
- The parentheses `()` in the function call are an operator in C++ that causes the function to be called.
- Function `square` (lines 19–22) receives a copy of the value of argument `x` from line 13 and stores it in the parameter `y`.
- Then `square` calculates `y * y` (line 21) and passes the result back to the point in `main` where `square` was invoked (line 13).
- The result is displayed.
- The function call does not change the value of `x`.
- The `for` repetition structure repeats this process for each of the values 1 through 10.

5.4 Function Definitions (cont.)

- The definition of `square` (lines 19–22) shows that it uses integer parameter `y`.
- Keyword `int` preceding the function name indicates that `square` returns an integer result.
- The `return` statement in `square` (line 21) passes the result of the calculation back to the calling function.

5.4 Function Definitions (cont.)

- Line 6 is a function prototype.
- The data type `int` in parentheses informs the compiler that function `square` expects to receive an integer value from the caller.
- The data type `int` to the left of the function name `square` informs the compiler that `square` returns an integer result to the caller.
- The compiler refers to the function prototype to check that calls to `square` contain the correct number and types of arguments and that the arguments are in the correct order.
- Also uses the prototype to ensure that the data returned by the function can be used correctly in the expression that called the function.
- If the arguments passed to a function do not match the types specified in the function's prototype, the compiler attempts to convert the arguments to the types specified in the prototype.
- Section 5.6 discusses the rules for these conversions.

5.4 Function Definitions (cont.)

- The function prototype is not required if the definition of the function appears *before* the function's first use in the program.
- In such a case, the function header also serves as the function prototype.
- Thus, if lines 19–22 in Fig. 5.3 appeared before `main`, the function prototype on line 6 would be unnecessary.
- Function prototypes are discussed in more detail in Section 5.6.

5.4 Function Definitions (cont.)

- The *parameter-list is* a comma-separated list containing the declarations of the parameters received by the function when it's called.
- If a function does not receive any values, *parameter-list* is `void` or simply left empty.
- A type must be listed for each parameter in the parameter list of a function.

5.4 Function Definitions (cont.)

- The *declarations and statements* in braces form the function body, which is also called a block or compound statement.
- Variables can be declared in any block, and blocks can be nested.
- There are three ways to return control to the point at which a function was invoked.
- If the function does not return a result, control returns when the program reaches the function-ending right brace, or by executing the statement

  ```
  return;
  ```

- If the function does return a result, the statement

  ```
  return expression;
  ```

  evaluates *expression* and returns the value of expression to the caller.

Common Programming Error 5.1

Forgetting to return a value from a function that's supposed to return a value is a compilation error.

Common Programming Error 5.2

Returning a value from a function whose return type has been declared `void` *is a compilation error.*

Common Programming Error 5.3

Placing a semicolon after the right parenthesis enclosing the parameter list of a function definition is a syntax error.

Common Programming Error 5.4

Defining a function parameter again as a local variable in the function is a compilation error.

Good Programming Practice 5.1

To avoid ambiguity, do not use the same names for the arguments passed to a function and the corresponding parameters in the function definition.

Common Programming Error 5.5

Defining a function inside another function is a syntax error.

Good Programming Practice 5.2

Choosing meaningful function names and meaningful parameter names makes programs more readable and helps avoid excessive use of comments.

Software Engineering Observation 5.3

Try to keep functions small. Regardless of how long a function is, it should perform one task well. Small functions promote software reusability.

Common Programming Error 5.6

It's a compilation error 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-value type.

5.5 Functions with Multiple Parameters

- Second example (Fig. 5.4) uses a function `maximum` to determine and return the largest of three floating-point numbers.
- The program prompts the user to input three floating-point numbers (line 14), then inputs the numbers (line 15).
- Next, the program calls function `maximum` (line 20), passing the numbers as arguments.
- Function `maximum` determines the largest value, then the `return` statement (line 35) returns that value to the point at which function `main` invoked `maximum` (line 20).
- Lines 19–20 output the returned value.
- The commas used in line 20 to separate the arguments to function `maximum` are not comma operators.
  - The comma operator guarantees that its operands are evaluated left to right; however, the order of evaluation of a function's arguments is not defined.

```
1  // Fig. 5.4: fig05_04.cpp
2  // Finding the maximum of three floating-point numbers.
3  #include <iostream>
4  using namespace std;
5
6  double maximum( double, double, double ); // function prototype
7
8  int main()
9  {
10    double number1;
11    double number2;
12    double number3;
13
14    cout << "Enter three floating-point numbers: ";
15    cin >> number1 >> number2 >> number3;
16
17    // number1, number2 and number3 are arguments to
18    // the maximum function call
19    cout << "Maximum is: "
20         << maximum( number1, number2, number3 ) << endl;
21 } // end main
22
```

Fig. 5.4 | Programmer-defined `maximum` function. (Part 1 of 3.)

```
23 // function maximum definition;
24 // x, y and z are parameters
25 double maximum( double x, double y, double z )
26 {
27    double max = x; // assume x is largest
28
29    if ( y > max ) // if y is larger,
30       max = y; // assign y to max
31
32    if ( z > max ) // if z is larger,
33       max = z; // assign z to max
34
35    return max; // max is largest value
36 } // end function maximum
```

Fig. 5.4 | Programmer-defined `maximum` function. (Part 2 of 3.)

```
Enter three floating-point numbers: 99.32 37.3 27.1928
Maximum is: 99.32
```

```
Enter three floating-point numbers: 1.1 3.333 2.22
Maximum is: 3.333
```

```
Enter three floating-point numbers: 27.9 14.31 88.99
Maximum is: 88.99
```

Fig. 5.4 | Programmer-defined `maximum` function. (Part 3 of 3.)

Portability Tip 5.1

The commas used to separate the arguments in a function call are not comma operators (discussed in Section 4.3). 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.*

Portability Tip 5.2

Sometimes when a function's arguments are 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 5.1

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.

5.5 Functions with Multiple Parameters (cont.)

- Function `maximum`'s prototype (Fig. 5.4, line 6) indicates that the function returns an integer value, has the name `maximum` and requires three integer parameters to perform its task.
- The function header (line 25) matches the function prototype and indicates that the parameter names are `x`, `y` and `z`.
- When `maximum` is called (line 20), the parameter `x` is initialized with the value of the argument `grade1`, the parameter `y` is initialized with the value of the argument `grade2` and the parameter `z` is initialized with the value of the argument `grade3`.
- There must be one argument in the function call for each parameter (also called a formal parameter) in the function definition.

5.5 Functions with Multiple Parameters (cont.)

- Notice that multiple parameters are specified in both the function prototype and the function header as a comma-separated list.
- The compiler refers to the function prototype to check that calls to `maximum` contain the correct number and types of arguments and that the types of the arguments are in the correct order.
- In addition, the compiler uses the prototype to ensure that the value returned by the function can be used correctly in the expression that called the function (e.g., a function call that returns `void` cannot be used as the right side of an assignment statement).

5.5 Functions with Multiple Parameters (cont.)

- Each argument must be consistent with the type of the corresponding parameter.
  - For example, a parameter of type `double` can receive values like 7.35, 22 or –0.03456, but not a string like `"hello"`.
- If the arguments passed to a function do not match the types specified in the function's prototype, the compiler attempts to convert the arguments to those types.
- Section 5.6 discusses this conversion.

Common Programming Error 5.7

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

Software Engineering Observation 5.4

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.

5.6 Function Prototypes and Argument Coercion

- A function prototype (also called a function declaration) tells the compiler the name of a function, the type of data returned by the function, the number of parameters the function expects to receive, the types of those parameters and the order in which the parameters of those types are expected.

Software Engineering Observation 5.5

Function prototypes are required. Use `#include` *preprocessor directives to obtain function prototypes for the C++ Standard Library functions from the header files of the appropriate libraries (e.g., the prototype for* `sqrt` *is in header file* `<cmath>`*; a partial list of C++ Standard Library header files appears in Section 5.7). Also use* `#include` *to obtain header files containing function prototypes written by you or other programmers.*

Common Programming Error 5.8

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

Software Engineering Observation 5.6

Always provide function prototypes, even though it's possible to omit them when functions are defined before they're 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).

5.6 Function Prototypes and Argument Coercion (cont.)

- The portion of a function prototype that includes the name of the function and the types of its arguments is called the function signature or simply the signature.
- The function signature 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.
- We'll say more about scope in Section 5.11.

Common Programming Error 5.9

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

5.6 Function Prototypes and Argument Coercion (cont.)

- An important feature of function prototypes is argument coercion—i.e., forcing arguments to the appropriate types specified by the parameter declarations.
- For example, a program can call a function with an integer argument, even though the function prototype specifies a `double` argument—the function will still work correctly.

5.6 Function Prototypes and Argument Coercion (cont.)

- Sometimes, argument values that do not correspond precisely to the parameter types in the function prototype can be converted by the compiler to the proper type before the function is called.
- These conversions occur as specified by C++'s promotion rules.
- The promotion rules indicate how to convert between types without losing data.
- An `int` can be converted to a `double` without changing its value.
- However, a `double` converted to an `int` truncates the fractional part of the `double` value.
- Keep in mind that `double` variables can hold numbers of much greater magnitude than `int` variables, so the loss of data may be considerable.

5.6 Function Prototypes and Argument Coercion (cont.)

- Values may also be modified when converting large integer types to small integer types (e.g., `long` to `short`), signed to unsigned or unsigned to signed.
- Unsigned integers range from 0 to approximately twice the positive range of the corresponding signed type.
- The promotion rules apply to expressions containing values of two or more data types; such expressions are also referred to as mixed-type expressions.
- The type of each value in a mixed-type expression is promoted to the "highest" type in the expression (actually a temporary version of each value is created and used for the expression—the original values remain unchanged).
- Promotion also occurs when the type of a function argument does not match the parameter type specified in the function definition or prototype.
- Figure 5.5 lists the fundamental data types in order from "highest type" to "lowest type."

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. 5.5 | Promotion hierarchy for fundamental data types.

5.6 Function Prototypes and Argument Coercion (cont.)

- Converting values to lower fundamental types can result in incorrect values.
- Therefore, a value can be converted to a lower fundamental type only by explicitly assigning the value to a variable of lower type (some compilers will issue a warning in this case) or by using a cast operator (see Section).
- Function argument values are converted to the parameter types in a function prototype as if they were being assigned directly to variables of those types.
- If a `square` function that uses an integer parameter is called with a floating-point argument, the argument is converted to `int` (a lower type), and `square` could return an incorrect value.
- For example, `square(4.5)` returns `16`, not `20.25`.

Common Programming Error 5.10

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 5.11

It's 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's also an error if the number of arguments in the call matches, but the arguments cannot be implicitly converted to the expected types.

5.7 C++ Standard Library Header Files

- The C++ Standard Library is divided into many portions, each with its own header file.
- The header files contain the function prototypes for the related functions that form each portion of the library.
- The header files also contain definitions of various class types and functions, as well as constants needed by those functions.
- A header file “instructs” the compiler on how to interface with library and user-written components.
- Figure 5.6 lists some common C++ Standard Library header files, most of which are discussed later in the book.

Standard Library header file	Explanation
`<iostream>`	Contains function prototypes for the standard input and standard output functions, introduced in Chapter 2, and is covered in more detail in Chapter 15, Stream Input/Output.
`<iomanip>`	Contains function prototypes for stream manipulators that format streams of data. This header file is first used in Section 3.9 and is discussed in more detail in Chapter 15, Stream Input/Output.
`<cmath>`	Contains function prototypes for math library functions (discussed in Section 5.3).
`<cstdlib>`	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 5.8; Chapter 11, Operator Overloading; Chapter 16, Exception Handling; Chapter 22, Bits, Characters, C Strings and `structs`; and Appendix F, C Legacy Code Topics.
`<ctime>`	Contains function prototypes and types for manipulating the time and date. This header file is used in Section 5.9.

Fig. 5.6 | C++ Standard Library header files. (Part 1 of 4.)

Standard Library header file	Explanation
`<vector>`, `<list>`, `<deque>`, `<queue>`, `<stack>`, `<map>`, `<set>`, `<bitset>`	These header files contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The `<vector>` header is first introduced in Chapter 6, Arrays and Vectors. We discuss all these header files in Chapter 21, Standard Template Library (STL).
`<cctype>`	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. These topics are discussed in Chapter 22, Bits, Characters, C Strings and `structs`.
`<cstring>`	Contains function prototypes for C-style string-processing functions. This header file is used in Chapter 11, Operator Overloading.
`<typeinfo>`	Contains classes for runtime type identification (determining data types at execution time). This header file is discussed in Section 13.8.
`<exception>`, `<stdexcept>`	These header files contain classes that are used for exception handling (discussed in Chapter 16, Exception Handling).

Fig. 5.6 | C++ Standard Library header files. (Part 2 of 4.)

Standard Library header file	Explanation
`<memory>`	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.
`<fstream>`	Contains function prototypes for functions that perform input from files on disk and output to files on disk (discussed in Chapter 17, Random-Access Files).
`<string>`	Contains the definition of class `string` from the C++ Standard Library (discussed in Chapter 18, Class `string` and String Stream Processing).
`<sstream>`	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 18, Class `string` and String Stream Processing).
`<functional>`	Contains classes and functions used by C++ Standard Library algorithms. This header file is used in Chapter 21.
`<iterator>`	Contains classes for accessing data in the C++ Standard Library containers. This header file is used in Chapter 21.
`<algorithm>`	Contains functions for manipulating data in C++ Standard Library containers. This header file is used in Chapter 21.

Fig. 5.6 | C++ Standard Library header files. (Part 3 of 4.)

Standard Library header file	Explanation
`<cassert>`	Contains macros for adding diagnostics that aid program debugging. This header file is used in Appendix E, Preprocessor.
`<cfloat>`	Contains the floating-point size limits of the system.
`<climits>`	Contains the integral size limits of the system.
`<cstdio>`	Contains function prototypes for the C-style standard input/output library functions.
`<locale>`	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.).
`<limits>`	Contains classes for defining the numerical data type limits on each computer platform.
`<utility>`	Contains classes and functions that are used by many C++ Standard Library header files.

Fig. 5.6 | C++ Standard Library header files. (Part 4 of 4.)

5.8 Case Study: Random Number Generation

- In this and the next section, we develop a game-playing program that includes multiple functions.
- The program uses many of the control statements and concepts discussed to this point.
- The element of chance can be introduced into computer applications by using the C++ Standard Library function `rand`.
- Consider the following statement:

```
i = rand();
```

- Function `rand` generates an unsigned integer between 0 and `RAND_MAX` (a constant defined in the `<cstdlib>` header file).
- The value of `RAND_MAX` must be at least 32767—the maximum positive value for a two-byte (16-bit) integer.
- For GNU C++, the value of `RAND_MAX` is 2147483647; for Visual Studio, the value of `RAND_MAX` is 32767.

5.8 Case Study: Random Number Generation (cont.)

- If `rand` truly produces integers at random, every number between 0 and `RAND_MAX` has an equal *chance* (or probability) of being chosen each time `rand` is called.
- The range of values produced directly by `rand` often is different than what a specific application requires.
- For example, a program that simulates coin tossing might require only 0 for "heads" and 1 for "tails." A program that simulates rolling a six-sided die would require random integers in the range 1 to 6.
- A program that randomly predicts the next type of spaceship (out of four possibilities) that will fly across the horizon in a video game might require random integers in the range 1 through 4.

5.8 Case Study: Random Number Generation (cont.)

- To demonstrate `rand`, Fig. 5.7 simulates 20 rolls of a six-sided die and displays the value of each roll.
- The function prototype for the `rand` function is in `<cstdlib>`.
- To produce integers in the range 0 to 5, we use the modulus operator (%) with `rand` as follows:

```
rand() % 6
```

- This is called scaling.
- The number 6 is called the scaling factor.
- We then shift the range of numbers produced by adding 1 to our previous result.
- Figure 5.7 confirms that the results are in the range 1–6.

```
1  // Fig. 5.7: fig05_07.cpp
2  // Shifted and scaled random integers.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdlib> // contains function prototype for rand
6  using namespace std;
7
8  int main()
9  {
10    // loop 20 times
11    for ( int counter = 1; counter <= 20; counter++ )
12    {
13       // pick random number from 1 to 6 and output it
14       cout << setw( 10 ) << ( 1 + rand() % 6 );
15
16       // if counter is divisible by 5, start a new line of output
17       if ( counter % 5 == 0 )
18          cout << endl;
19    } // end for
20 } // end main
```

Fig. 5.7 | Shifted, scaled integers produced by `1 + rand() % 6`. (Part 1 of 2.)

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

Fig. 5.7 | Shifted, scaled integers produced by `1 + rand() % 6`. (Part 2 of 2.)

5.8 Case Study: Random Number Generation (cont.)

- To show that the numbers produced by `rand` occur with approximately equal likelihood, Fig. 5.8 simulates 6,000,000 rolls of a die.
  - The values 1–6 should appear approximately 1,000,000 times each.
  - Confirmed by the program's output.
- As the output shows, we can simulate the rolling of a six-sided die by scaling and shifting the values produced by `rand`.
- The program should never get to the `default` case (lines 45–46) in the `switch` structure, because the `switch`'s controlling expression (`face`) always has values in the range 1–6; however, we provide the `default` case as a matter of good practice.
- In Chapter 6, we show how to replace the entire `switch` structure in Fig. 5.8 elegantly with a single-line statement.

Error-Prevention Tip 5.2

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

```
1  // Fig. 5.8: fig05_08.cpp
2  // Roll a six-sided die 6,000,000 times.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdlib> // contains function prototype for rand
6  using namespace std;
7
8  int main()
9  {
10    int frequency1 = 0; // count of 1s rolled
11    int frequency2 = 0; // count of 2s rolled
12    int frequency3 = 0; // count of 3s rolled
13    int frequency4 = 0; // count of 4s rolled
14    int frequency5 = 0; // count of 5s rolled
15    int frequency6 = 0; // count of 6s rolled
16
17    int face; // stores most recently rolled value
18
19    // summarize results of 6,000,000 rolls of a die
20    for ( int roll = 1; roll <= 6000000; roll++ )
21    {
22       face = 1 + rand() % 6; // random number from 1 to 6
23
```

Fig. 5.8 | Rolling a six-sided die 6,000,000 times. (Part 1 of 3.)

```
24         // determine roll value 1-6 and increment appropriate counter
25         switch ( face )
26         {
27            case 1:
28               ++frequency1; // increment the 1s counter
29               break;
30            case 2:
31               ++frequency2; // increment the 2s counter
32               break;
33            case 3:
34               ++frequency3; // increment the 3s counter
35               break;
36            case 4:
37               ++frequency4; // increment the 4s counter
38               break;
39            case 5:
40               ++frequency5; // increment the 5s counter
41               break;
42            case 6:
43               ++frequency6; // increment the 6s counter
44               break;
45            default: // invalid value
46               cout << "Program should never get here!";
47         } // end switch
48      } // end for
```

Fig. 5.8 | Rolling a six-sided die 6,000,000 times. (Part 2 of 3.)

```
49
50      cout << "Face" << setw( 13 ) << "Frequency" << endl; // output headers
51      cout << "   1" << setw( 13 ) << frequency1
52         << "\n   2" << setw( 13 ) << frequency2
53         << "\n   3" << setw( 13 ) << frequency3
54         << "\n   4" << setw( 13 ) << frequency4
55         << "\n   5" << setw( 13 ) << frequency5
56         << "\n   6" << setw( 13 ) << frequency6 << endl;
57   } // end main
```

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

Fig. 5.8 | Rolling a six-sided die 6,000,000 times. (Part 3 of 3.)

5.8 Case Study: Random Number Generation (cont.)

- Repeatability is an important characteristic of `rand`.
  - When debugging, repeatability is essential for proving that corrections to the program work properly.
- Function `rand` actually generates pseudorandom numbers.
- Repeatedly calling `rand` produces a sequence of numbers that appears to be random.
- However, the sequence repeats itself each time the program executes.
- Once a program has been thoroughly debugged, it can be conditioned to produce a different sequence of random numbers for each execution.

5.8 Case Study: Random Number Generation (cont.)

- This is called randomizing and is accomplished with the C++ Standard Library function `srand`.
- Function `srand` takes an `unsigned` integer argument and seeds the `rand` function to produce a different sequence of random numbers for each execution.

5.8 Case Study: Random Number Generation (cont.)

- Figure 5.9 demonstrates function `srand`.
- Data type `unsigned`, which is short for `unsigned int`.
- An `int` is stored in at least two bytes of memory (typically four bytes on 32-bit systems and as much as eight bytes on 64-bit systems) and can have positive and negative values.
- An `unsigned int` is also stored in at least two bytes of memory.
- A two-byte `unsigned int` can have only nonnegative values in the range 0–65535.
- A four-byte `unsigned int` can have only nonnegative values in the range 0–4294967295.
- `srand` takes an `unsigned int` value as an argument.
- The function prototype for `srand` is in header file `<cstdlib>`.

```
1  // Fig. 5.9: fig05_09.cpp
2  // Randomizing die-rolling program.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdlib> // contains prototypes for functions srand and rand
6  using namespace std;
7
8  int main()
9  {
10    unsigned seed; // stores the seed entered by the user
11
12    cout << "Enter seed: ";
13    cin >> seed;
14    srand( seed ); // seed random number generator
15
16    // loop 10 times
17    for ( int counter = 1; counter <= 10; counter++ )
18    {
19       // pick random number from 1 to 6 and output it
20       cout << setw( 10 ) << ( 1 + rand() % 6 );
21
```

Fig. 5.9 | Randomizing the die-rolling program. (Part 1 of 2.)

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

```
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
```

```
Enter seed: 67
        6       1       4       6       2
        1       6       1       6       4
```

Fig. 5.9 | Randomizing the die-rolling program. (Part 2 of 2.)

5.8 Case Study: Random Number Generation (cont.)

- To randomize without having to enter a seed each time, we may use a statement like

```
srand( time( 0 ) );
```

- This causes the computer to read its clock to obtain the value for the seed.
- Function `time` (with the argument `0` as written in the preceding statement) typically returns the current time as the number of seconds since January 1, 1970, at midnight Greenwich Mean Time (GMT).
- This value is converted to an `unsigned` integer and used as the seed to the random number generator.
- The function prototype for `time` is in `<ctime>`.

5.8 Case Study: Random Number Generation (cont.)

- Rolling a six-sided die with the statement

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

- always assigns an integer (at random) to variable `face` in the range $1 \leq$ `face` ≤ 6.
- The width of this range (i.e., the number of consecutive integers in the range) is 6 and the starting number in the range is 1.
- The width of the range is the number used to scale `rand` with the modulus operator (i.e., 6), and the starting number of the range is the number (i.e., 1) that's added to the expression `rand % 6`.
- We can generalize this result as

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

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

5.9 Case Study: Game of Chance; Introducing enum

- One of the most popular games of chance is a dice game known as “craps,” which is played in casinos and back alleys worldwide.
- The rules of the game are straightforward:
  - **A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5 and 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first roll, the player wins. If the sum is 2, 3 or 12 on the first roll (called “craps”), the player loses (i.e., the “house” wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first roll, then that sum becomes the player’s “point.” To win, you must continue rolling the dice until you “make your point.” The player loses by rolling a 7 before making the point.**

5.9 Case Study: Game of Chance; Introducing enum (cont.)

- The program in Fig. 5.10 simulates the game.
- In the rules, notice that the player must roll two dice on the first roll and on all subsequent rolls.
- We define function `rollDice` (lines 63–75) to roll the dice and compute and print their sum.
- The function is defined once, but called from lines 21 and 45.
- The function takes no arguments and returns the sum of the two dice, so empty parentheses and the return type `int` are indicated in the function prototype (line 8) and function header (line 63).

```
1  // Fig. 5.10: fig05_10.cpp
2  // Craps simulation.
3  #include <iostream>
4  #include <cstdlib> // contains prototypes for functions srand and rand
5  #include <ctime> // contains prototype for function time
6  using namespace std;
7
8  int rollDice(); // rolls dice, calculates and displays sum
9
10 int main()
11 {
12    // enumeration with constants that represent the game status
13    enum Status { CONTINUE, WON, LOST }; // all caps in constants
14
15    int myPoint; // point if no win or loss on first roll
16    Status gameStatus; // can contain CONTINUE, WON or LOST
17
18    // randomize random number generator using current time
19    srand( time( 0 ) );
20
21    int sumOfDice = rollDice(); // first roll of the dice
22
```

Fig. 5.10 | Craps simulation. (Part 1 of 6.)

```
23      // determine game status and point (if needed) based on first roll
24      switch ( sumOfDice )
25      {
26         case 7: // win with 7 on first roll
27         case 11: // win with 11 on first roll
28            gameStatus = WON;
29            break;
30         case 2: // lose with 2 on first roll
31         case 3: // lose with 3 on first roll
32         case 12: // lose with 12 on first roll
33            gameStatus = LOST;
34            break;
35         default: // did not win or lose, so remember point
36            gameStatus = CONTINUE; // game is not over
37            myPoint = sumOfDice; // remember the point
38            cout << "Point is " << myPoint << endl;
39            break; // optional at end of switch
40      } // end switch
41
```

Fig. 5.10 | Craps simulation. (Part 2 of 6.)

```
42      // while game is not complete
43      while ( gameStatus == CONTINUE ) // not WON or LOST
44      {
45         sumOfDice = rollDice(); // roll dice again
46
47         // determine game status
48         if ( sumOfDice == myPoint ) // win by making point
49            gameStatus = WON;
50         else
51            if ( sumOfDice == 7 ) // lose by rolling 7 before point
52               gameStatus = LOST;
53      } // end while
54
55      // display won or lost message
56      if ( gameStatus == WON )
57         cout << "Player wins" << endl;
58      else
59         cout << "Player loses" << endl;
60   } // end main
61
```

Fig. 5.10 | Craps simulation. (Part 3 of 6.)

```
62  // roll dice, calculate sum and display results
63  int rollDice()
64  {
65     // pick random die values
66     int die1 = 1 + rand() % 6; // first die roll
67     int die2 = 1 + rand() % 6; // second die roll
68
69     int sum = die1 + die2; // compute sum of die values
70
71     // display results of this roll
72     cout << "Player rolled " << die1 << " + " << die2
73        << " = " << sum << endl;
74     return sum; // end function rollDice
75  } // end function rollDice
```

Fig. 5.10 | Craps simulation. (Part 4 of 6.)

```
Player rolled 2 + 5 = 7
Player wins
```

```
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 6 + 6 = 12
Player loses
```

Fig. 5.10 | Craps simulation. (Part 5 of 6.)

```
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
```

Fig. 5.10 | Craps simulation. (Part 6 of 6.)

5.9 Case Study: Game of Chance; Introducing enum (cont.)

- Variable `gameStatus` is declared to be of new type `Status`.
- Line 13 declares a user-defined type called an enumeration.
- An enumeration, introduced by the keyword `enum` and followed by a type name (in this case, `Status`), is a set of integer constants represented by identifiers.
- The values of these enumeration constants start at `0`, unless specified otherwise, and increment by `1`.
- In the preceding enumeration, the constant `CONTINUE` has the value 0, `WON` has the value 1 and `LOST` has the value 2.
- The identifiers in an `enum` must be unique, but separate enumeration constants can have the same integer value.
- Variables of user-defined type `Status` can be assigned only one of the three values declared in the enumeration.

Good Programming Practice 5.3

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

Good Programming Practice 5.4

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

5.9 Case Study: Game of Chance; Introducing enum (cont.)

- Another popular enumeration is

```
enum Months {
   JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG,
   SEP, OCT, NOV, DEC };
```

  which creates user-defined type `Months` with enumeration constants representing the months of the year.
- The first value in the preceding enumeration is explicitly set to 1, so the remaining values increment from 1, resulting in the values 1 through 12.
- Any enumeration constant can be assigned an integer value in the enumeration definition, and subsequent enumeration constants each have a value 1 higher than the preceding constant in the list until the next explicit setting.

Good Programming Practice 5.5

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

Common Programming Error 5.12

Assigning the integer equivalent of an enumeration constant (rather than the enumeration constant, itself) to a variable of the enumeration type is a compilation error.

Common Programming Error 5.13

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

5.10 Storage Classes

- The programs you've seen so far use identifiers for variable names.
- The attributes of variables include name, type, size and value.
- This chapter also uses identifiers as names for user-defined functions.
- Actually, each identifier in a program has other attributes, including storage class, scope and linkage.
- C++ provides five storage-class specifiers: `auto`, `register`, `extern`, `mutable` and `static`.
- This section discusses storage-class specifiers `auto`, `register`, `extern` and `static`; `mutable` (discussed in Chapter 24, Other Topics) is used exclusively with classes.

5.10 Storage Classes (cont.)

- An identifier's storage class determines the period during which that identifier exists in memory.
- Some exist briefly, some are repeatedly created and destroyed and others exist for the entire execution of a program.
- First we discuss the storage classes static and automatic.
- An identifier's scope is where the identifier can be referenced in a program.
- Some identifiers can be referenced throughout a program; others can be referenced from only limited portions of a program.
- Section 5.11 discusses the scope of identifiers.

5.10 Storage Classes (cont.)

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

5.10 Storage Classes (cont.)

- The storage-class specifiers can be split into two storage classes: automatic storage class and static storage class.
- Keywords `auto` and `register` are used to declare variables of the automatic storage class.
- Such variables are created when program execution enters the block in which they're defined, they exist while the block is active and they're destroyed when the program exits the block.

5.10 Storage Classes (cont.)

- Only local variables of a function can be of automatic storage class.
- A function's local variables and parameters normally are of automatic storage class.
- The storage class specifier `auto` explicitly declares variables of automatic storage class.
- For example, the following declaration indicates that `double` variable `x` is a local variable of automatic storage class—it exists only in the nearest enclosing pair of curly braces within the body of the function in which the definition appears:

```
auto double x;
```

- Local variables are of automatic storage class by default, so keyword `auto` rarely is used.
- For the remainder of the text, we refer to variables of automatic storage class simply as automatic variables.

Performance Tip 5.1

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

Software Engineering Observation 5.7

Automatic storage is an example of the principle of least privilege. 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're not needed?

5.10 Storage Classes (cont.)

- Data in the machine-language version of a program is normally loaded into registers for calculations and other processing.
- The compiler might ignore `register` declarations.
- For example, there might not be a sufficient number of registers available for the compiler to use.
- The following definition *suggests* that the integer variable `counter` be placed in one of the computer's registers; regardless of whether the compiler does this, `counter` is initialized to 1:

```
register int counter = 1;
```

- The `register` keyword can be used only with local variables and function parameters.

Performance Tip 5.2

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 kept in hardware registers, the overhead of repeatedly loading the variables from memory into the registers and storing the results back into memory is eliminated.*

Performance Tip 5.3

Often, `register` *is unnecessary. Optimizing compilers can recognize frequently used variables and may place them in registers without needing a* `register` *declaration.*

5.10 Storage Classes (cont.)

- Keywords `extern` and `static` declare identifiers for variables of the static storage class and for functions.
- Static-storage-class variables exist from the point at which the program begins execution and last for the duration of the program.
- A static-storage-class variable's storage is allocated when the program begins execution.
- Such a variable is initialized once when its declaration is encountered.
- For 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.
- Storage class and scope (where a name can be used) are separate issues, as we'll see in Section 5.11.

5.10 Storage Classes (cont.)

- There are two types of identifiers with static storage class—external identifiers (such as global variables and global function names) and local variables declared with the storage-class specifier `static`.
- Global variables are created by placing variable declarations outside any class or function definition.
- Global variables retain their values throughout the execution of the program.
- Global variables and global functions can be referenced by any function that follows their declarations or definitions in the source file.

Software Engineering Observation 5.8

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 5.9

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

5.10 Storage Classes (cont.)

- Local variables declared `static` are still known only in the function in which they're declared, but, unlike automatic variables, `static` local variables retain their values when the function returns to its caller.
- The next time the function is called, the `static` local variables contain the values they had when the function last completed execution.
- The following statement declares local variable `count` to be `static` and to be initialized to 1:

```
static int count = 1;
```

- All numeric variables of the static storage class are initialized to zero if they're not explicitly initialized by you, but it's nevertheless a good practice to explicitly initialize all variables.
- Storage-class specifiers `extern` and `static` have special meaning when they're applied explicitly to external identifiers such as global variables and global function names.

5.11 Scope Rules

- The portion of the program where an identifier can be used is known as its scope.
- For example, when we declare a local variable in a block, it can be referenced only in that block and in blocks nested within that block.
- This section discusses four scopes for an identifier—function scope, global namespace scope, local scope and function-prototype scope.
- Later we'll see two other scopes—class scope (Chapter 9) and namespace scope (Chapter 24).

5.11 Scope Rules (cont.)

- An identifier declared outside any function or class has global namespace scope.
- Such an identifier is “known” in all functions from the point at which it’s declared until the end of the file.
- Global variables, function definitions and function prototypes placed outside a function all have global namespace scope.

5.11 Scope Rules (cont.)

- Labels (identifiers followed by a colon such as `start:`) are the only identifiers with function scope.
- Labels can be used anywhere in the function in which they appear, but cannot be referenced outside the function body.
- Labels are used in `goto` statements (Appendix F).
- Labels are implementation details that functions hide from one another.

5.11 Scope Rules (cont.)

- Identifiers declared inside a block have local scope.
- Local scope begins at the identifier's declaration and ends at the terminating right brace (}) of the block in which the identifier is declared.
- Local variables have local scope, as do function parameters, which are also local variables of the function.
- Any block can contain variable declarations.
- When blocks are nested and an identifier in an outer block has the same name as an identifier in an inner block, the identifier in the outer block is "hidden" until the inner block terminates.
- The inner block sees the value of its own local identifier and not that of the identically named identifier in the enclosing block.

5.11 Scope Rules (cont.)

- Local variables declared `static` still have local scope, even though they exist from the time the program begins execution.
- Storage duration does not affect the scope of an identifier.

5.11 Scope Rules (cont.)

- The only identifiers with function prototype scope are those used in the parameter list of a function prototype.
- As mentioned previously, function prototypes do not require names in the parameter list—only types are required.
- Names appearing in the parameter list of a function prototype are ignored by the compiler.
- Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity.
- In a single prototype, a particular identifier can be used only once.

Common Programming Error 5.14

Accidentally using the same name for an identifier in an inner block that's used for an identifier in an outer block, when in fact you want the identifier in the outer block to be active for the duration of the inner block, is typically a logic error.

Good Programming Practice 5.6

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

5.11 Scope Rules (cont.)

- The program of Fig. 5.11 demonstrates scoping issues with global variables, automatic local variables and `static` local variables.

```
1  // Fig. 5.11: fig05_11.cpp
2  // A scoping example.
3  #include <iostream>
4  using namespace std;
5
6  void useLocal(); // function prototype
7  void useStaticLocal(); // function prototype
8  void useGlobal(); // function prototype
9
10 int x = 1; // global variable
11
12 int main()
13 {
14    cout << "global x in main is " << x << endl;
15
16    int x = 5; // local variable to main
17
18    cout << "local x in main's outer scope is " << x << endl;
19
20    { // start new scope
21       int x = 7; // hides both x in outer scope and global x
22
23       cout << "local x in main's inner scope is " << x << endl;
24    } // end new scope
```

Fig. 5.11 | Scoping example. (Part 1 of 4.)

```
25
26      cout << "local x in main's outer scope is " << x << endl;
27
28      useLocal(); // useLocal has local x
29      useStaticLocal(); // useStaticLocal has static local x
30      useGlobal(); // useGlobal uses global x
31      useLocal(); // useLocal reinitializes its local x
32      useStaticLocal(); // static local x retains its prior value
33      useGlobal(); // global x also retains its prior value
34
35      cout << "\nlocal x in main is " << x << endl;
36  } // end main
37
38  // useLocal reinitializes local variable x during each call
39  void useLocal()
40  {
41      int x = 25; // initialized each time useLocal is called
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
47
```

Fig. 5.11 | Scoping example. (Part 2 of 4.)

```
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()
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     x++;
58     cout << "local static x is " << x << " on exiting useStaticLocal"
59        << endl;
60  } // end function useStaticLocal
61
62  // useGlobal modifies global variable x during each call
63  void useGlobal()
64  {
65     cout << "\nglobal x is " << x << " on entering useGlobal" << endl;
66     x *= 10;
67     cout << "global x is " << x << " on exiting useGlobal" << endl;
68  } // end function useGlobal
```

Fig. 5.11 | Scoping example. (Part 3 of 4.)

```
global x in main is 1
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
```

Fig. 5.11 | Scoping example. (Part 4 of 4.)

5.12 Function Call Stack and Activation Records

- To understand how C++ performs function calls, we first need to consider a data structure (i.e., collection of related data items) known as a stack.
- Think of a stack as analogous to a pile of dishes.
- When a dish is placed on the pile, it's normally placed at the top (referred to as pushing the dish onto the stack).
- Similarly, when a dish is removed from the pile, it's normally removed from the top (referred to as popping the dish off the stack).
- Stacks are known as last-in, first-out (LIFO) data structures—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

5.12 Function Call Stack and Activation Records (cont.)

- One of the most important mechanisms for computer science students to understand is the function call stack (sometimes referred to as the program execution stack).
- This data structure—working "behind the scenes"—supports the function call/return mechanism.
- It also supports the creation, maintenance and destruction of each called function's automatic variables.
- We explained the last-in, first-out (LIFO) behavior of stacks with our dish-stacking example.
- As we'll see in Figs. 5.13–5.15, this LIFO behavior is exactly what a function does when returning to the function that called it.

5.12 Function Call Stack and Activation Records (cont.)

- As each function is called, it may, in turn, call other functions, which may, in turn, call other functions—all before any of the functions return.
- Each function eventually must return control to the function that called it.
- So, somehow, we must keep track of the return addresses that each function needs to return control to the function that called it.

5.12 Function Call Stack and Activation Records (cont.)

- The function call stack is the perfect data structure for handling this information.
- Each time a function calls another function, an entry is pushed onto the stack.
- This entry, called a stack frame or an activation record, contains the return address that the called function needs in order to return to the calling function.
- It also contains some additional information we'll soon discuss.

5.12 Function Call Stack and Activation Records (cont.)

- If the called function returns, instead of calling another function before returning, the stack frame for the function call is popped, and control transfers to the return address in the popped stack frame.
- The beauty of the call stack is that each called function always finds the information it needs to return to its caller at the top of the call stack.
- If one function makes a call to another, a stack frame for the new function call is simply pushed onto the call stack.
- Thus, the return address required by the newly called function to return to its caller is now located at the top of the stack.

5.12 Function Call Stack and Activation Records (cont.)

- The stack frames have another important responsibility.
- Most functions have automatic variables—parameters and any local variables the function declares.
- Automatic variables need to exist while a function is executing.
- They need to remain active if the function makes calls to other functions.
- But when a called function returns to its caller, the called function's automatic variables need to "go away." The called function's stack frame is a perfect place to reserve the memory for the called function's automatic variables.
- That stack frame exists as long as the called function is active.
- When that function returns—and no longer needs its local automatic variables—its stack frame is popped from the stack, and those local automatic variables are no longer known to the program.

5.12 Function Call Stack and Activation Records (cont.)

- The amount of memory in a computer is finite, so only a certain amount of memory can be used to store activation records on the function call stack.
- If more function calls occur than can have their activation records stored on the function call stack, an error known as stack overflow occurs.

5.12 Function Call Stack and Activation Records (cont.)

- Consider how the call stack supports the operation of a `square` function called by `main` (lines 9–14 of Fig. 5.12).

```
1  // Fig. 5.12: fig05_12.cpp
2  // square function used to demonstrate the function
3  // call stack and activation records.
4  #include <iostream>
5  using namespace std;
6
7  int square( int ); // prototype for function square
8
9  int main()
10 {
11    int a = 10; // value to square (local automatic variable in main)
12
13    cout << a << " squared: " << square( a ) << endl; // display a squared
14 } // end main
15
16 // returns the square of an integer
17 int square( int x ) // x is a local variable
18 {
19    return x * x; // calculate square and return result
20 } // end function square
```

```
10 squared: 100
```

Fig. 5.12 | Demonstrating the function call stack and activation records.

5.12 Function Call Stack and Activation Records (cont.)

- First the operating system calls `main`—this pushes an activation record onto the stack (shown in Fig. 5.13).
- The activation record tells `main` how to return to the operating system (i.e., transfer to return address `R1`) and contains the space for `main`'s automatic variable (i.e., `a`, which is initialized to `10`).
- Function `main`—before returning to the operating system—now calls function `square` in line 13 of Fig. 5.12.
- This causes a stack frame for `square` (lines 17–20) to be pushed onto the function call stack (Fig. 5.14).
- This stack frame contains the return address that `square` needs to return to `main` (i.e., `R2`) and the memory for `square`'s automatic variable (i.e., `x`).

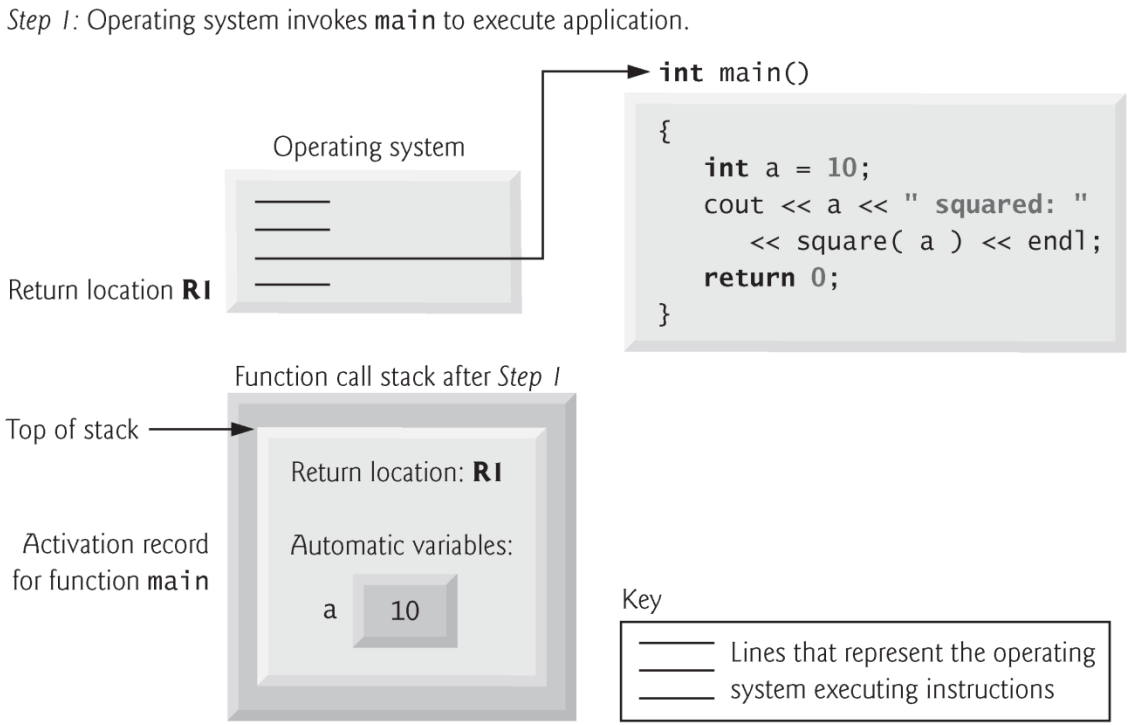


Fig. 5.13 | Function call stack after the operating system invokes `main` to execute the program.

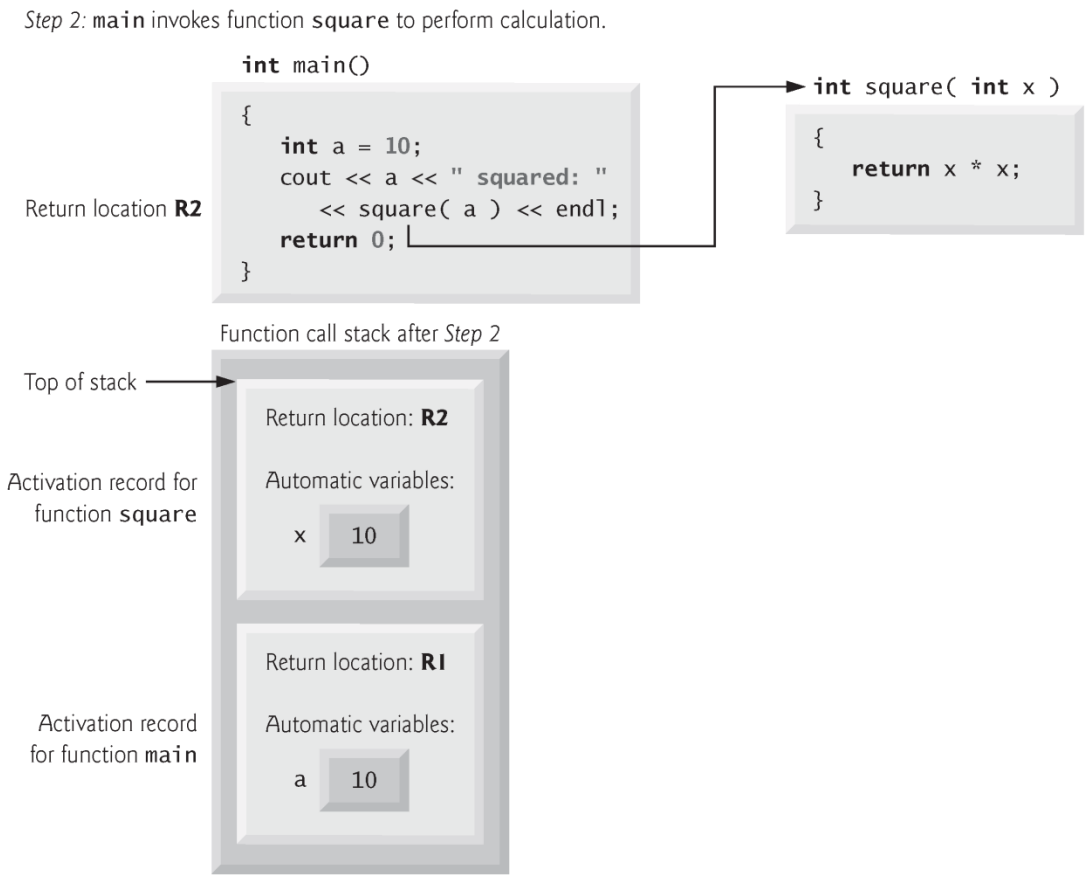


Fig. 5.14 | Function call stack after `main` invokes `square` to perform the calculation.

5.12 Function Call Stack and Activation Records (cont.)

- After `square` calculates the square of its argument, it needs to return to `main`—and no longer needs the memory for its automatic variable `x`.
- So the stack is popped—giving `square` the return location in `main` (i.e., `R2`) and losing `square`'s automatic variable.
- Figure 5.15 shows the function call stack after `square`'s activation record has been popped.

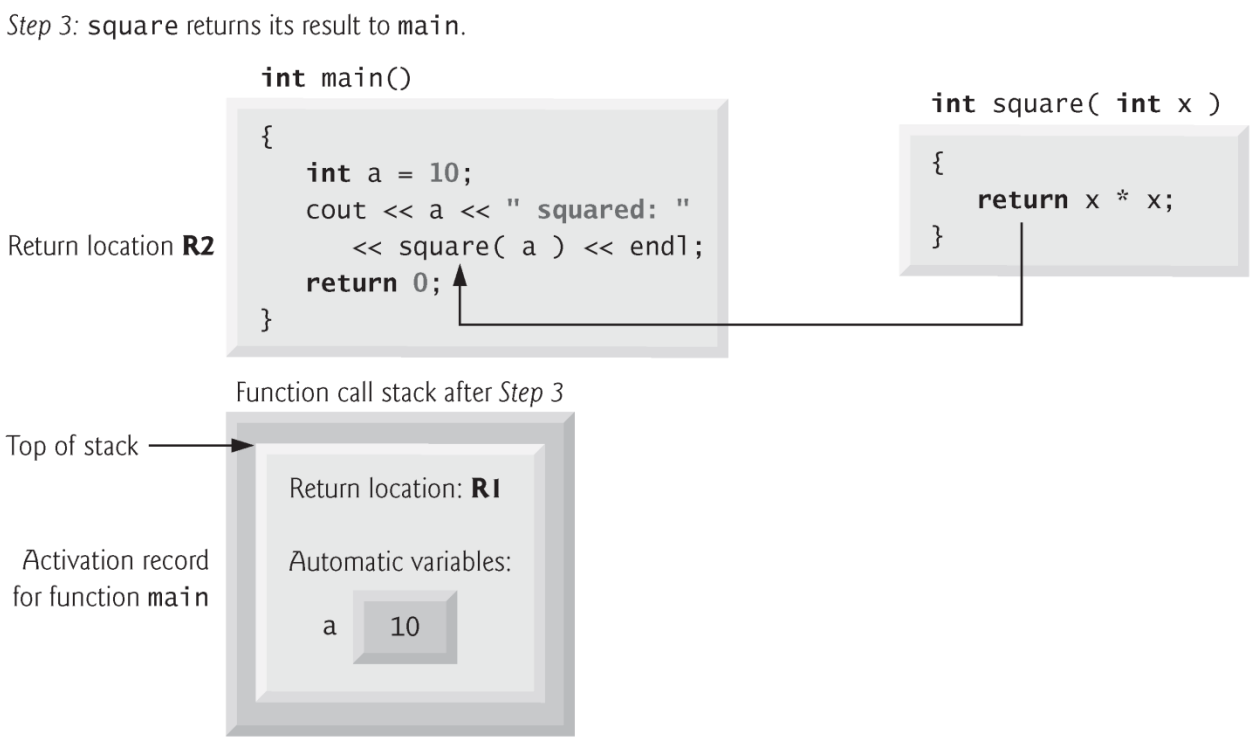


Fig. 5.15 | Function call stack after function `square` returns to `main`.

5.12 Function Call Stack and Activation Records (cont.)

- `main` now displays the result of calling `square` (line 13).
- Reaching the closing right brace of `main` causes its activation record to be popped from the stack and gives `main` the address it needs to return to the operating system (i.e., `R1` in Fig. 5.13) and causes the memory for `main`'s automatic variable (i.e., `a`) to become unavailable.
- You've now seen how valuable the stack data structure is in implementing a key mechanism that supports program execution.

5.13 Functions with Empty Parameter Lists

- In C++, an empty parameter list is specified by writing either `void` or nothing at all in parentheses.
- The prototype

```
void print();
```

  specifies that function `print` does not take arguments and does not return a value.
- Figure 5.16 shows both ways to declare and use functions with empty parameter lists.

```
1  // Fig. 5.16: fig05_16.cpp
2  // Functions that take no arguments.
3  #include <iostream>
4  using namespace std;
5
6  void function1(); // function that takes no arguments
7  void function2( void ); // function that takes no arguments
8
9  int main()
10 {
11    function1(); // call function1 with no arguments
12    function2(); // call function2 with no arguments
13 } // end main
14
15 // function1 uses an empty parameter list to specify that
16 // the function receives no arguments
17 void function1()
18 {
19    cout << "function1 takes no arguments" << endl;
20 } // end function1
21
```

Fig. 5.16 | Functions that take no arguments. (Part 1 of 2.)

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

```
function1 takes no arguments
function2 also takes no arguments
```

Fig. 5.16 | Functions that take no arguments. (Part 2 of 2.)

Common Programming Error 5.15

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

5.14 Inline Functions

- Function calls involve execution-time overhead.
- C++ provides inline functions to help reduce function call overhead—especially for small functions.
- Placing the 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.
- The trade-off is that multiple copies of the function code are inserted in the program (often making the program larger) rather than there being a single copy of the function to which control is passed each time the function is called.
- The compiler can ignore the `inline` qualifier and typically does so for all but the smallest functions.

Software Engineering Observation 5.10

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

Good Programming Practice 5.7

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

Performance Tip 5.4

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

5.14 Inline Functions (cont.)

- Figure 5.17 uses `inline` function `cube` (lines 9–12) to calculate the volume of a cube.
- Keyword `const` in function `cube`'s parameter list (line 9) tells the compiler that the function does not modify variable `side`.
- This ensures that `side`'s value is not changed by the function during the calculation.
- Notice that the complete definition of function `cube` appears before it's used in the program.
- This is required so that the compiler knows how to expand a `cube` function call into its inlined code.
- For this reason, reusable inline functions are typically placed in header files, so that their definitions can be included in each source file that uses them.

Software Engineering Observation 5.11

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. 5.17: fig05_17.cpp
2  // Using an inline function to calculate the volume of a cube.
3  #include <iostream>
4  using namespace std;
5
6  // Definition of inline function cube. Definition of function appears
7  // before function is called, so a function prototype is not required.
8  // First line of function definition acts as the prototype.
9  inline double cube( const double side )
10 {
11    return side * side * side; // calculate cube
12 } // end function cube
13
14 int main()
15 {
16    double sideValue; // stores value entered by user
17    cout << "Enter the side length of your cube: ";
18    cin >> sideValue; // read value from user
19
20    // calculate cube of sideValue and display result
21    cout << "Volume of cube with side "
22       << sideValue << " is " << cube( sideValue ) << endl;
23 } // end main
```

Fig. 5.17 | `inline` function that calculates the volume of a cube. (Part 1 of 2.)

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

Fig. 5.17 | `inline` function that calculates the volume of a cube. (Part 2 of 2.)

5.15 References and Reference Parameters

- Two ways to pass arguments to functions in many programming languages are pass-by-value and pass-by-reference.
- When an argument is passed by value, a *copy* of the argument's value is made and passed (on the function call stack) to the called function.
- Changes to the copy do not affect the original variable's value in the caller.
- This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems.
- Each argument in this chapter has been passed by value.

Performance Tip 5.5

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.

5.15 References and Reference Parameters (cont.)

- reference parameters—the first of the two means C++ provides for performing pass-by-reference.
- With pass-by-reference, the caller gives the called function the ability to access the caller's data directly, and to modify that data.
- A reference parameter is an alias for its corresponding argument in a function call.
- To indicate that a function parameter is passed by reference, simply follow the parameter's type in the function prototype by an ampersand (&); use the same convention when listing the parameter's type in the function header.
- Then, mentioning the variable by its parameter name in the body of the called function actually refers to the original variable in the calling function, and the original variable can be modified directly by the called function.
- As always, the function prototype and header must agree.

Performance Tip 5.6

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 5.12

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

5.15 References and Reference Parameters (cont.)

- Figure 5.18 compares pass-by-value and pass-by-reference with reference parameters.
- The "styles" of the arguments in the calls to function `squareByValue` and function `squareByReference` are identical—both variables are simply mentioned by name in the function calls.
- Without checking the function prototypes or function definitions, it isn't possible to tell from the calls alone whether either function can modify its arguments.
- Because function prototypes are mandatory, the compiler has no trouble resolving the ambiguity.

Common Programming Error 5.16

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

Performance Tip 5.7

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.

```
1  // Fig. 5.18: fig05_18.cpp
2  // Comparing pass-by-value and pass-by-reference with references.
3  #include <iostream>
4  using namespace std;
5
6  int squareByValue( int ); // function prototype (value pass)
7  void squareByReference( int & ); // function prototype (reference pass)
8
9  int main()
10 {
11    int x = 2; // value to square using squareByValue
12    int z = 4; // value to square using squareByReference
13
14    // demonstrate squareByValue
15    cout << "x = " << x << " before squareByValue\n";
16    cout << "Value returned by squareByValue: "
17       << squareByValue( x ) << endl;
18    cout << "x = " << x << " after squareByValue\n" << endl;
19
20    // demonstrate squareByReference
21    cout << "z = " << z << " before squareByReference" << endl;
22    squareByReference( z );
23    cout << "z = " << z << " after squareByReference" << endl;
24 } // end main
```

Fig. 5.18 | Passing arguments by value and by reference. (Part 1 of 2.)

```
25
26 // squareByValue multiplies number by itself, stores the
27 // result in number and returns the new value of number
28 int squareByValue( int number )
29 {
30    return number *= number; // caller's argument not modified
31 } // end function squareByValue
32
33 // squareByReference multiplies numberRef by itself and stores the result
34 // in the variable to which numberRef refers in function main
35 void squareByReference( int &numberRef )
36 {
37    numberRef *= numberRef; // caller's argument modified
38 } // end function squareByReference
```

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

z = 4 before squareByReference
z = 16 after squareByReference
```

Fig. 5.18 | Passing arguments by value and by reference. (Part 2 of 2.)

5.15 References and Reference Parameters (cont.)

- To specify a reference to a constant, place the `const` qualifier before the type specifier in the parameter declaration.
- Note the placement of `&` in function `squareByReference`'s parameter list (line 35, Fig. 5.18).
  - Some C++ programmers prefer to write the equivalent form `int& numberRef`.

5.15 References and Reference Parameters (cont.)

- References can also be used as aliases for other variables within a function (although they typically are used with functions as shown in Fig. 5.18).
- For example, the code

```
int count = 1; // declare integer variable count
int &cRef = count; // create cRef as an alias for count
cRef++; // increment count (using its alias cRef)
```

  increments variable `count` by using its alias `cRef`.
- Reference variables must be initialized in their declarations (see Fig. 5.19 and Fig. 5.20) and cannot be reassigned as aliases to other variables.
- Once a reference is declared as an alias for another variable, all operations performed on the alias are actually performed on the original variable.
- Unless it's a reference to a constant, a reference argument must be an *lvalue* (e.g., a variable name), not a constant or expression that returns an *rvalue* (e.g., the result of a calculation).

```
1  // Fig. 5.19: fig05_19.cpp
2  // Initializing and using a reference.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int x = 3;
9     int &y = x; // y refers to (is an alias for) x
10
11    cout << "x = " << x << endl << "y = " << y << endl;
12    y = 7; // actually modifies x
13    cout << "x = " << x << endl << "y = " << y << endl;
14 } // end main
```

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

Fig. 5.19 | Initializing and using a reference.

```
1  // Fig. 5.20: fig05_20.cpp
2  // References must be initialized.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int x = 3;
9     int &y; // Error: y must be initialized
10
11    cout << "x = " << x << endl << "y = " << y << endl;
12    y = 7;
13    cout << "x = " << x << endl << "y = " << y << endl;
14 } // end main
```

Fig. 5.20 | Uninitialized reference causes a compilation error. (Part 1 of 2.)

Microsoft Visual C++ compiler error message:

```
C:\cpphtp7_examples\ch05\Fig05_20\fig05_20.cpp(9) : error C2530: 'y' :
   references must be initialized
```

GNU C++ compiler error message:

```
fig05_20.cpp:9: error: 'y' declared as a reference but not initialized
```

Fig. 5.20 | Uninitialized reference causes a compilation error. (Part 2 of 2.)

5.15 References and Reference Parameters (cont.)

- Functions can return references, but this can be dangerous.
- When returning a reference to a variable declared in the called function, the variable should be declared `static` in that function.
- Otherwise, the reference refers to an automatic variable that's discarded when the function terminates; such a variable is said to be "undefined," and the program's behavior is unpredictable.
- References to undefined variables are called dangling references
- The C++ standard does not specify the error messages that compilers use to indicate particular errors.
- For this reason, Fig. 5.20 shows the error messages produced by the Microsoft Visual C++ compiler and GNU C++ compiler when a reference is not initialized.

Common Programming Error 5.17

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

5.16 Default Arguments

- It isn't uncommon for a program to invoke a function repeatedly with the same argument value for a particular parameter.
- In such cases, you can specify that such a parameter has a default argument, i.e., a default value to be passed to that parameter.
- When a program omits an argument for a parameter with a default argument in a function call, the compiler rewrites the function call and inserts the default value of that argument.
- Default arguments must be the rightmost (trailing) arguments in a function's parameter list.

5.16 Default Arguments (cont.)

- When calling a function with two or more default arguments, if an omitted argument is not the rightmost argument in the argument list, then all arguments to the right of that argument also must be omitted.
- Default arguments must be specified with the first occurrence of the function name—typically, in the function prototype.
- If the function prototype is omitted because the function definition also serves as the prototype, then the default arguments should be specified in the function header.
- Default values can be any expression, including constants, global variables or function calls.
- Default arguments also can be used with `inline` functions.
- Figure 5.21 demonstrates using default arguments to calculate a box's volume.

```
1  // Fig. 5.21: fig05_21.cpp
2  // Using default arguments.
3  #include <iostream>
4  using namespace std;
5
6  // function prototype that specifies default arguments
7  int boxVolume( int length = 1, int width = 1, int height = 1 );
8
9  int main()
10 {
11    // no arguments--use default values for all dimensions
12    cout << "The default box volume is: " << boxVolume();
13
14    // specify length; default width and height
15    cout << "\n\nThe volume of a box with length 10,\n"
16       << "width 1 and height 1 is: " << boxVolume( 10 );
17
18    // specify length and width; default height
19    cout << "\n\nThe volume of a box with length 10,\n"
20       << "width 5 and height 1 is: " << boxVolume( 10, 5 );
21
```

Fig. 5.21 | Default arguments to a function. (Part 1 of 2.)

```
22      // specify all arguments
23      cout << "\n\nThe volume of a box with length 10,\n"
24         << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
25         << endl;
26   } // end main
27
28   // function boxVolume calculates the volume of a box
29   int boxVolume( int length, int width, int height )
30   {
31      return length * width * height;
32   } // 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
```

Fig. 5.21 | Default arguments to a function. (Part 2 of 2.)

Good Programming Practice 5.8

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

Software Engineering Observation 5.13

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

Common Programming Error 5.18

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

5.17 Unary Scope Resolution Operator

- It's possible to declare local and global variables of the same name.
- C++ provides the unary scope resolution operator (`::`) to access a global variable when a local variable of the same name is in scope.
- The unary scope resolution operator cannot be used to access a local variable of the same name in an outer block.
- A global variable can be accessed directly without the unary scope resolution operator if the name of the global variable is not the same as that of a local variable in scope.
- Figure 5.22 shows the unary scope resolution operator with local and global variables of the same name (lines 6 and 10).
- Using the unary scope resolution operator (`::`) with a given variable name is optional when the only variable with that name is a global variable.

```
1  // Fig. 5.22: fig05_22.cpp
2  // Using the unary scope resolution operator.
3  #include <iostream>
4  using namespace std;
5
6  int number = 7; // global variable named number
7
8  int main()
9  {
10    double number = 10.5; // local variable named number
11
12    // display values of local and global variables
13    cout << "Local double value of number = " << number
14       << "\nGlobal int value of number = " << ::number << endl;
15 } // end main
```

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

Fig. 5.22 | Unary scope resolution operator.

Good Programming Practice 5.9

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're intending to access a global variable rather than a nonglobal variable.

Software Engineering Observation 5.14

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 5.3

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

Error-Prevention Tip 5.4

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.

5.18 Function Overloading

- C++ enables several functions of the same name to be defined, as long as they have different signatures.
- This is called function overloading.
- The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call.
- Function overloading is used to create several functions of the same name that perform similar tasks, but on different data types.
- For example, many functions in the math library are overloaded for different numeric types—the C++ standard requires `float`, `double` and `long double` overloaded versions of the math library functions discussed in Section 5.3.

Good Programming Practice 5.10

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

5.18 Function Overloading (cont.)

- Figure 5.23 uses overloaded `square` functions to calculate the square of an `int` (lines 7–11) and the square of a `double` (lines 14–18).
- Line 22 invokes the `int` version of function `square` by passing the literal value `7`.
- C++ treats whole number literal values as type `int`.
- Similarly, line 24 invokes the `double` version of function `square` by passing the literal value `7.5`, which C++ treats as a `double` value.
- In each case the compiler chooses the proper function to call, based on the type of the argument.
- The last two lines of the output window confirm that the proper function was called in each case.

```
1  // Fig. 5.23: fig05_23.cpp
2  // Overloaded functions.
3  #include <iostream>
4  using namespace std;
5
6  // function square for int values
7  int square( int x )
8  {
9     cout << "square of integer " << x << " is ";
10    return x * x;
11 } // end function square with int argument
12
13 // function square for double values
14 double square( double y )
15 {
16    cout << "square of double " << y << " is ";
17    return y * y;
18 } // end function square with double argument
19
```

Fig. 5.23 | Overloaded `square` functions. (Part 1 of 2.)

```
20  int main()
21  {
22     cout << square( 7 ); // calls int version
23     cout << endl;
24     cout << square( 7.5 ); // calls double version
25     cout << endl;
26  } // end main
```

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

Fig. 5.23 | Overloaded `square` functions. (Part 2 of 2.)

5.18 Function Overloading (cont.)

- Overloaded functions are distinguished by their signatures.
- A signature is a combination of a function's name and its parameter types (in order).
- The compiler encodes each function identifier with the number and types of its parameters (sometimes referred to as name mangling or name decoration) to enable type-safe linkage.
- Type-safe linkage ensures that the proper overloaded function is called and that the types of the arguments conform to the types of the parameters.
- Figure 5.24 was compiled with GNU C++.
- Function-name mangling is compiler specific.
- Also, function `main` is not mangled, because it cannot be overloaded.

Common Programming Error 5.19

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

```
1  // Fig. 5.24: fig05_24.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
22
```

Fig. 5.24 | Name mangling to enable type-safe linkage. (Part 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  } // end main
```

```
__Z6squarei
__Z6squared
__Z8nothing1ifcRi
__Z8nothing2ciRfRd
_main
```

Fig. 5.24 | Name mangling to enable type-safe linkage. (Part 2 of 2.)

5.18 Function Overloading (cont.)

- The compiler uses only the parameter lists to distinguish between overloaded functions.
- Such functions need not have the same number of parameters.
- Use caution when overloading functions with default parameters, because this may cause ambiguity.

Common Programming Error 5.20

A function with default arguments omitted might be called identically to another overloaded function; this is a compilation error. For example, having a program that contains 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.

5.19 Function Templates

- Overloaded functions are normally used to perform similar operations that involve different program logic on different data types.
- If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently by using function templates.
- You write a single function template definition.
- Given the argument types provided in calls to this function, C++ automatically generates separate function template specializations to handle each type of call appropriately.
- Thus, defining a single function template essentially defines a whole family of overloaded functions.

5.19 Function Templates (cont.)

- Figure 5.25 contains the definition of a function template (lines 3–17) for a `maximum` function that determines the largest of three values.
- Function template definitions begin with `template` (line 3) followed by a template parameter list enclosed in angle brackets (< and >).
- Every parameter in the template parameter list (often referred to as a formal type parameter) is preceded by keyword `typename` or keyword `class` (which are synonyms in this context).
- The formal type parameters are placeholders for fundamental types or user-defined types.
- These placeholders are used to specify the types of the function's parameters (line 4), to specify the function's return type (line 4) and to declare variables within the body of the function definition (line 6).
- A function template is defined like any other function, but uses the formal type parameters as placeholders for actual data types.

```
1  // Fig. 5.25: maximum.h
2  // Definition of function template maximum.
3  template < class T >  // or template< typename T >
4  T maximum( T value1, T value2, T value3 )
5  {
6     T maximumValue = value1; // assume value1 is maximum
7
8     // determine whether value2 is greater than maximumValue
9     if ( value2 > maximumValue )
10       maximumValue = value2;
11
12    // determine whether value3 is greater than maximumValue
13    if ( value3 > maximumValue )
14       maximumValue = value3;
15
16    return maximumValue;
17 } // end function template maximum
```

Fig. 5.25 | Function template `maximum` header file.

5.19 Function Templates (cont.)

- The function template in Fig. 5.25 declares a single formal type parameter `T` (line 3) as a placeholder for the type of the data to be tested by function `maximum`.
- The name of a type parameter must be unique in the template parameter list for a particular template definition.
- When the compiler detects a `maximum` invocation in the program source code, the type of the data passed to `maximum` is substituted for `T` throughout the template definition, and C++ creates a complete function for determining the maximum of three values of the specified data type—all three must have the same type, since we use only one type parameter in this example.
- Then the newly created function is compiled.
- Thus, templates are a means of code generation.
- Figure 5.26 uses the `maximum` function template (lines 17, 27 and 37) to determine the largest of three `int` values, three `double` values and three `char` values, respectively.

```
1  // Fig. 5.26: fig05_26.cpp
2  // Function template maximum test program.
3  #include <iostream>
4  #include "maximum.h" // include definition of function template maximum
5  using namespace std;
6
7  int main()
8  {
9     // demonstrate maximum with int values
10    int int1, int2, int3;
11
12    cout << "Input three integer values: ";
13    cin >> int1 >> int2 >> int3;
14
15    // invoke int version of maximum
16    cout << "The maximum integer value is: "
17       << maximum( int1, int2, int3 );
18
19    // demonstrate maximum with double values
20    double double1, double2, double3;
21
22    cout << "\n\nInput three double values: ";
23    cin >> double1 >> double2 >> double3;
24
```

Fig. 5.26 | Demonstrating function template `maximum`. (Part 1 of 2.)

```
25      // invoke double version of maximum
26      cout << "The maximum double value is: "
27         << maximum( double1, double2, double3 );
28
29      // demonstrate maximum with char values
30      char char1, char2, char3;
31
32      cout << "\n\nInput three characters: ";
33      cin >> char1 >> char2 >> char3;
34
35      // invoke char version of maximum
36      cout << "The maximum character value is: "
37         << maximum( char1, char2, char3 ) << endl;
38   } // end main
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

Fig. 5.26 | Demonstrating function template `maximum`. (Part 2 of 2.)

5.20 Recursion

- The programs we've discussed are generally structured as functions that call one another in a disciplined, hierarchical manner.
- For some problems, it's useful to have functions call themselves.
- A recursive function is a function that calls itself, either directly, or indirectly (through another function).
- Recursion is an important topic discussed at length in upper-level computer science courses.
- This section and the next present simple examples of recursion.
- This book contains an extensive treatment of recursion.

5.20 Recursion (cont.)

- Recursive problem-solving approaches have a number of elements in common.
- A recursive function is called to solve a problem.
- The function actually knows how to solve only the simplest case(s), or so-called base case(s).
- If the function is called with a base case, the function simply returns a result.
- If the function is called with a more complex problem, it typically divides the problem into two conceptual pieces—a piece that the function knows how to do and a piece that it does not know how to do.
- To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or smaller version.
- This new problem looks like the original, so the function calls a copy of itself to work on the smaller problem—this is referred to as a recursive call and is also called the recursion step.

5.20 Recursion (cont.)

- The recursion step often includes the keyword `return`, because its result will be combined with the portion of the problem the function knew how to solve to form the result passed back to the original caller, possibly `main`.
- The recursion step executes while the original call to the function is still "open," i.e., it has not yet finished executing.
- The recursion step can result in many more such recursive calls, as the function keeps dividing each new subproblem with which the function is called into two conceptual pieces.
- In order for the recursion to eventually terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller and smaller problems must eventually converge on the base case.
- At that point, the function recognizes the base case and returns a result to the previous copy of the function, and a sequence of returns ensues up the line until the original call eventually returns the final result to `main`.

5.20 Recursion (cont.)

- The factorial of a nonnegative integer *n*, written *n!* (and pronounced "n factorial"), is the product

  $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

  with 1! equal to 1, and 0! defined to be 1.
- For example, 5! is the product $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is equal to 120.
- The factorial of an integer, `number`, greater than or equal to 0, can be calculated iteratively (nonrecursively) by using a `for` statement as follows:

```
factorial = 1;
for ( int counter = number; counter >= 1; counter-- )
   factorial *= counter;
```

5.20 Recursion (cont.)

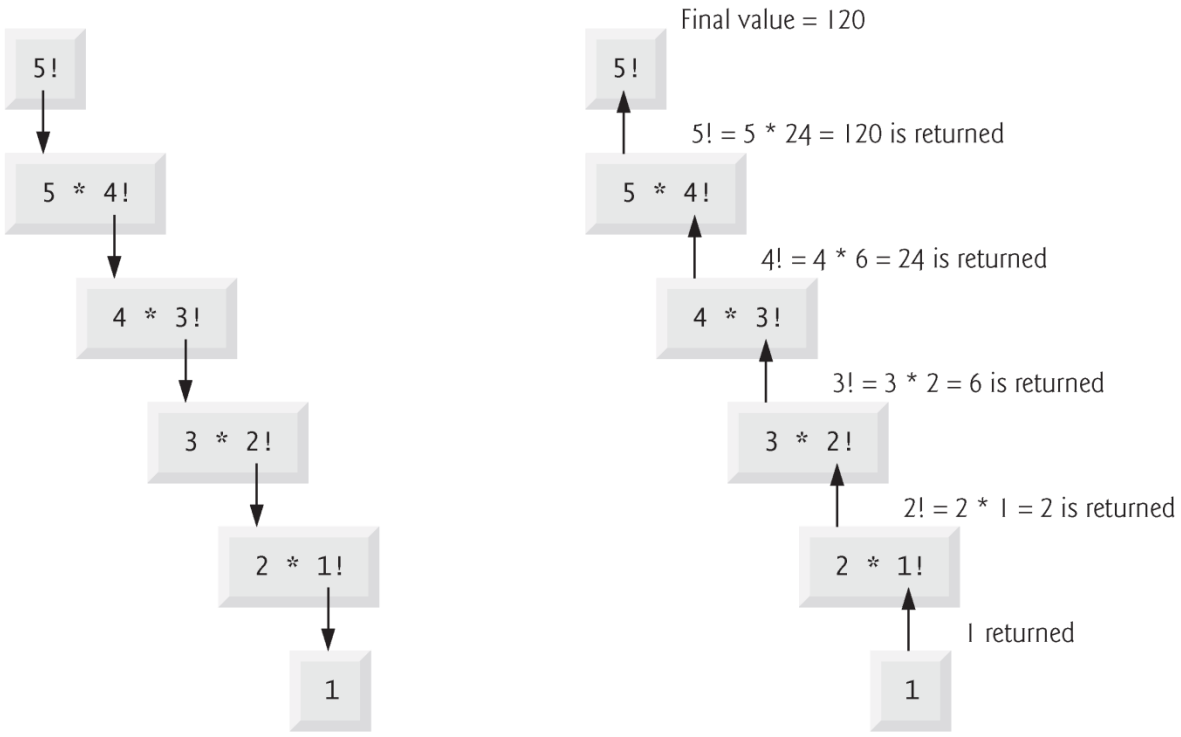
- A recursive definition of the factorial function is arrived at by observing the following algebraic relationship:

  $n! = n \cdot (n - 1)!$

- 5! is clearly equal to 5 * 4! as is shown by the following:

  $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$
  $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$
  $5! = 5 \cdot (4!)$

- The evaluation of 5! would proceed as shown in Fig. 5.27.
  - Figure 5.27(a) shows how the succession of recursive calls proceeds until 1! is evaluated to be 1, which terminates the recursion.
  - Figure 5.27(b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.



(a) Procession of recursive calls.

(b) Values returned from each recursive call.

Fig. 5.27 | Recursive evaluation of 5!.

5.20 Recursion (cont.)

- Figure 5.28 uses recursion to calculate and print the factorials of the integers 0–10.
- The recursive function `factorial` (lines 18–24) first determines whether the terminating condition `number <= 1` (line 20) is true.
- If `number` is less than or equal to 1, the `factorial` function returns 1 (line 21), no further recursion is necessary and the function terminates.
- If `number` is greater than 1, line 23 expresses the problem as the product of `number` and a recursive call to `factorial` evaluating the factorial of `number - 1`, which is a slightly simpler problem than the original calculation `factorial(number)`.

```
1  // Fig. 5.28: fig05_28.cpp
2  // Demonstrating the recursive function factorial.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  unsigned long factorial( unsigned long ); // function prototype
8
9  int main()
10 {
11    // calculate the factorials of 0 through 10
12    for ( int counter = 0; counter <= 10; counter++ )
13       cout << setw( 2 ) << counter << "! = " << factorial( counter )
14          << endl;
15 } // end main
16
17 // recursive definition of function factorial
18 unsigned long factorial( unsigned long number )
19 {
20    if ( number <= 1 ) // test for base case
21       return 1; // base cases: 0! = 1 and 1! = 1
22    else // recursion step
23       return number * factorial( number - 1 );
24 } // end function factorial
```

Fig. 5.28 | Demonstrating the recursive function `factorial`. (Part 1 of 2.)

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

Fig. 5.28 | Demonstrating the recursive function `factorial`. (Part 2 of 2.)

5.20 Recursion (cont.)

- Function `factorial` has been declared to receive a parameter of type `unsigned long` and return a result of type `unsigned long`.
- This is shorthand notation for `unsigned long int`.
- The C++ standard requires that a variable of type `unsigned long int` be at least as big as an `int`.
- Typically, an `unsigned long int` is stored in at least four bytes (32 bits); such a variable can hold a value in the range 0 to at least 4294967295.
  - (The data type `long int` is also stored in at least four bytes and can hold a value at least in the range –2147483648 to 2147483647.)

5.20 Recursion (cont.)

- As can be seen in Fig. 5.28, factorial values become large quickly.
- We chose the data type `unsigned long` so that the program can calculate factorials greater than 7! on computers with small (such as two-byte) integers.
- Unfortunately, the function `factorial` produces large values so quickly that even `unsigned long` does not help us compute many factorial values before even the size of an `unsigned long` variable is exceeded.
- Variables of type `double` could be used to calculate factorials of larger numbers.

Common Programming Error 5.21

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.

5.21 Example Using Recursion: Fibonacci Series

- The Fibonacci series

  0, 1, 1, 2, 3, 5, 8, 13, 21, …

  begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.
- The series occurs in nature and, in particular, describes a form of spiral.
- The ratio of successive Fibonacci numbers converges on a constant value of 1.618….
  - This number, too, frequently occurs in nature and has been called the golden ratio or the golden mean.
  - Humans tend to find the golden mean aesthetically pleasing.
  - Architects often design windows, rooms and buildings whose length and width are in the ratio of the golden mean.
  - Postcards are often designed with a golden mean length/width ratio.

5.21 Example Using Recursion: Fibonacci Series (cont.)

- The Fibonacci series can be defined recursively as follows:

  fibonacci(0) = 0
  fibonacci(1) = 1
  fibonacci(*n*) = fibonacci(*n* – 1) + fibonacci(*n* – 2)

- The program of Fig. 5.29 calculates the *n*th Fibonacci number recursively by using function `fibonacci`.
- Fibonacci numbers tend to become large quickly.
  - We chose the data type `unsigned long` for the parameter type and the return type in function `fibonacci`.
- Figure 5.29 shows the execution of the program, which displays the Fibonacci values for several numbers.

```
1  // Fig. 5.29: fig05_29.cpp
2  // Testing the recursive fibonacci function.
3  #include <iostream>
4  using namespace std;
5
6  unsigned long fibonacci( unsigned long ); // function prototype
7
8  int main()
9  {
10    // calculate the fibonacci values of 0 through 10
11    for ( int counter = 0; counter <= 10; counter++ )
12       cout << "fibonacci( " << counter << " ) = "
13          << fibonacci( counter ) << endl;
14
15    // display higher fibonacci values
16    cout << "fibonacci( 20 ) = " << fibonacci( 20 ) << endl;
17    cout << "fibonacci( 30 ) = " << fibonacci( 30 ) << endl;
18    cout << "fibonacci( 35 ) = " << fibonacci( 35 ) << endl;
19 } // end main
20
```

Fig. 5.29 | Demonstrating function `fibonacci`. (Part 1 of 2.)

```
21 // recursive function fibonacci
22 unsigned long fibonacci( unsigned long number )
23 {
24    if ( ( number == 0 ) || ( number == 1 ) ) // base cases
25       return number;
26    else // recursion step
27       return fibonacci( number - 1 ) + fibonacci( number - 2 );
28 } // 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
```

Fig. 5.29 | Demonstrating function `fibonacci`. (Part 2 of 2.)

5.21 Example Using Recursion: Fibonacci Series (cont.)

- Interestingly, if `number` is greater than 1, the recursion step (line 27) generates *two* recursive calls, each for a slightly smaller problem than the original call to `fibonacci`.
- Figure 5.30 shows how function `fibonacci` would evaluate `fibonacci(3)`.
- This figure raises some interesting issues about the order in which C++ compilers evaluate the operands of operators.
- This is a separate issue from the order in which operators are applied to their operands, namely, the order dictated by the rules of operator precedence and associativity.
- Figure 5.30 shows that evaluating `fibonacci(3)` causes two recursive calls, namely, `fibonacci(2)` and `fibonacci(1)`.

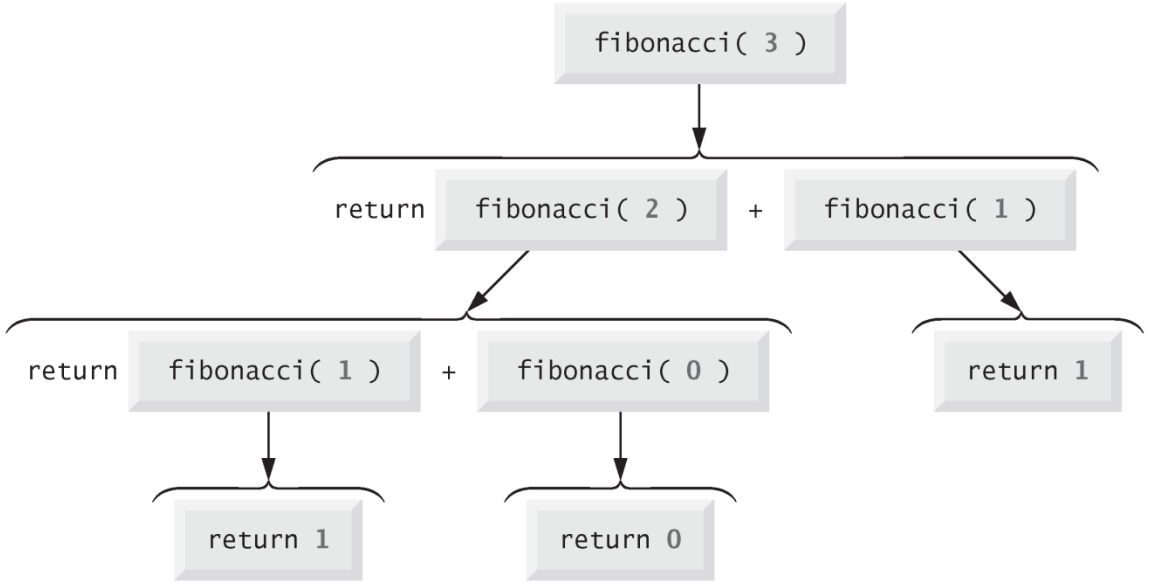


Fig. 5.30 | Set of recursive calls to function `fibonacci`.

5.21 Example Using Recursion: Fibonacci Series (cont.)

- C++ does not specify the order in which the operands of most operators (including +) are to be evaluated.
- Therefore, you must make no assumption about the order in which these calls execute.
- The calls could in fact execute `fibonacci(2)` first, then `fibonacci(1)`, or they could execute in the reverse order: `fibonacci(1)`, then `fibonacci(2)`.
- In this program and in most others, it turns out that the final result would be the same.
- However, in some programs the evaluation of an operand can have side effects (changes to data values) that could affect the final result of the expression.

5.21 Example Using Recursion: Fibonacci Series (cont.)

- C++ specifies the order of evaluation of the operands of only four operators—`&&`, `||`, comma (`,`) and `?:`.
- The first three are binary operators whose two operands are guaranteed to be evaluated left to right.
- The last operator is C++'s only ternary operator.
- Its leftmost operand is always evaluated first; if it evaluates to nonzero (true), the middle operand evaluates next and the last operand is ignored; if the leftmost operand evaluates to zero (false), the third operand evaluates next and the middle operand is ignored.

Common Programming Error 5.22

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 5.3

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

5.21 Example Using Recursion: Fibonacci Series (cont.)

- A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers.
- 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.
- Calculating only the 20th Fibonacci number would require on the order of 2^{20} or about a million calls, calculating the 30th Fibonacci number would require on the order of 2^{30} or about a billion calls, and so on.
- Computer scientists refer to this as exponential complexity.

Performance Tip 5.8

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

5.22 Recursion vs. Iteration

- In the two previous sections, we studied two functions that easily can be implemented recursively or iteratively.
- This section compares the two approaches and discusses why you might choose one approach over the other in a particular situation.

5.22 Recursion vs. Iteration (cont.)

- Both iteration and recursion are based on a control statement: Iteration uses a repetition structure; recursion uses a selection structure.
- Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated function calls.

5.22 Recursion vs. Iteration (cont.)

- Iteration and recursion both involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.
- Iteration with counter-controlled repetition and recursion both gradually approach termination: Iteration modifies a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion produces simpler versions of the original problem until the base case is reached.

5.22 Recursion vs. Iteration (cont.)

- Both iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem during each recursive call in a manner that converges on the base case.

5.22 Recursion vs. Iteration (cont.)

- To illustrate the differences between iteration and recursion, let's examine an iterative solution to the factorial problem (Fig. 5.31).
- A repetition statement is used (lines 23–24 of Fig. 5.31) rather than the selection statement of the recursive solution (lines 20–23 of Fig. 5.28).
- Both solutions use a termination test.
- In the recursive solution, line 20 tests for the base case.
- In the iterative solution, line 23 tests the loop-continuation condition—if the test fails, the loop terminates.
- Finally, instead of producing simpler versions of the original problem, the iterative solution uses a counter that's modified until the loop-continuation condition becomes false.

```
1  // Fig. 5.31: fig05_31.cpp
2  // Testing the iterative factorial function.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  unsigned long factorial( unsigned long ); // function prototype
8
9  int main()
10 {
11    // calculate the factorials of 0 through 10
12    for ( int counter = 0; counter <= 10; counter++ )
13       cout << setw( 2 ) << counter << "! = " << factorial( counter )
14          << endl;
15 } // end main
16
```

Fig. 5.31 | Iterative factorial solution. (Part 1 of 2.)

```
17  // iterative function factorial
18  unsigned long factorial( unsigned long number )
19  {
20     unsigned long result = 1;
21
22     // iterative factorial calculation
23     for ( unsigned long i = number; i >= 1; i-- )
24        result *= i;
25
26     return result;
27  } // end function factorial
```

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

Fig. 5.31 | Iterative factorial solution. (Part 2 of 2.)

5.22 Recursion vs. Iteration (cont.)

- Recursion has many negatives.
- It repeatedly invokes the mechanism, and consequently the overhead, of function calls.
- This 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; this can consume considerable memory.
- Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted.

Software Engineering Observation 5.15

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

Performance Tip 5.9

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

Common Programming Error 5.23

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

5.22 Recursion vs. Iteration (cont.)

- Figure 5.32 summarizes the recursion examples and exercises in the text.

Location in text	Recursion examples and exercises
Chapter 5	
Section 5.20, Fig. 5.28	Factorial function
Section 5.21, Fig. 5.29	Fibonacci function
Exercise 5.36	Raising an integer to an integer power
Exercise 5.38	Towers of Hanoi
Exercise 5.40	Visualizing recursion
Exercise 5.41	Greatest common divisor
Exercise 5.45, Exercise 5.46	Mystery "What does this program do?" exercise
Chapter 6	
Exercise 6.18	Mystery "What does this program do?" exercise
Exercise 6.21	Mystery "What does this program do?" exercise
Exercise 6.31	Selection sort
Exercise 6.32	Determine whether a string is a palindrome
Exercise 6.33	Linear search
Exercise 6.34	Eight Queens
Exercise 6.35	Print an array

Fig. 5.32 | Summary of recursion examples and exercises in the text. (Part 1 of 3.)

Location in text	Recursion examples and exercises
Exercise 6.36	Print a string backward
Exercise 6.37	Minimum value in an array
Chapter 7	
Exercise 7.15	Quicksort
Exercise 7.16	Maze traversal
Exercise 7.17	Generating mazes randomly
Chapter 19	
Section 19.3.3, Figs. 19.5–19.7	Mergesort
Exercise 19.8	Linear search
Exercise 19.9	Binary search
Exercise 19.10	Quicksort
Chapter 20	
Section 20.7, Figs. 20.20–20.22	Binary tree insert
Section 20.7, Figs. 20.20–20.22	Preorder traversal of a binary tree
Section 20.7, Figs. 20.20–20.22	Inorder traversal of a binary tree

Fig. 5.32 | Summary of recursion examples and exercises in the text. (Part 2 of 3.)

Location in text	Recursion examples and exercises
Section 20.7, Figs. 20.20–20.22	Postorder traversal of a binary tree
Exercise 20.20	Print a linked list backward
Exercise 20.21	Search a linked list
Exercise 20.22	Binary tree delete
Exercise 20.23	Binary tree search
Exercise 20.24	Level order traversal of a binary tree
Exercise 20.25	Printing tree

Fig. 5.32 | Summary of recursion examples and exercises in the text. (Part 3 of 3.)