EE1003
Introduction to Computer I

Chapter 5
Functions and an Introduction to Recursion

Andy, Yu-Guang Chen
Assistant Professor, Department of EE
National Central University
andyygchen@ee.ncu.edu.tw

Learning 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.

Outline

Outline

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.)

- 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 you write are referred to as user-defined functions or programmer-defined functions.
- 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.
    - Programs can be created from standardized functions that accomplish specific tasks.
  - 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.

5.2 Program Components in C++ (cont.)

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
- 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).

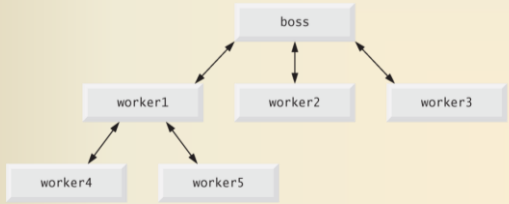


Fig. 5.1 | Hierarchical boss function/worker function relationship.

5.3 Math Library Functions

- 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.

5.3 Math Library Functions

Function	Description	Example
ceil(x)	rounds *x* to the smallest integer not less than *x*	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
cos(x)	trigonometric cosine of *x* (*x* in radians)	cos(0.0) is 1.0
exp(x)	exponential function e^x	exp(1.0) is 2.718282 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.)

5.3 Math Library Functions

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

- 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 returned result to the caller.
  - The type `void` indicates that a function does not return a value.
- All variables defined in a function 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.)

```
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 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.

```
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
```

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.

```
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
```


5.4 Function Definitions (cont.)

- Line 6 is a function prototype.

```
6  int square( int ); // 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 the number, types, and order of input/output arguments.
- 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.

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.
- 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.

5.4 Function Definitions (cont.)

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.*

5.4 Function Definitions (cont.)

Common Programming Error 5.3

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

```
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
```

5.4 Function Definitions (cont.)

Common Programming Error 5.4
Defining a function parameter again as a local variable in the function is a compilation error.

Common Programming Error 5.5
Defining a function inside another function is a syntax error.

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

```
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.)

5.5 Functions with Multiple Parameters

```
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.)

5.5 Functions with Multiple Parameters

```
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.)

5.5 Functions with Multiple Parameters

- 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.

5.5 Functions with Multiple Parameters

- The function 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 `number1`, the parameter `y` is initialized with the value of the argument `number2` and the parameter `z` is initialized with the value of the argument `number3`.
- 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.)

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.

5.6 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.
- 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.
- Functions in the same scope must have unique signatures.

5.6 Function Prototypes and Argument Coercion

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.*

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.)

- An important feature of function prototypes is argument coercion—i.e., forcing arguments to the appropriate types specified by the parameter declarations.
- The argument values that do not correspond precisely to the parameter types can be converted by the compiler to the proper type before the function is called.
  - These conversions follow C++'s promotion rules.
- 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.

5.6 Function Prototypes and Argument Coercion (cont.)

- Values may also be modified when converting to different types.
  - Especially for the conversion from "large" type to "small" type.
- The promotion rules apply to expressions containing values of two or more data types (referred to as mixed-type expressions).
  - Promotion also occurs when the type of a function argument does not match the specified parameter type of the function.
- The type of each value in a mixed-type expression is promoted to the "highest" type in the expression.
  - A temporary version of each value is created and used for the expression—the original values remain unchanged.
- Figure 5.5 lists the fundamental data types in order from "highest type" to "lowest type."

5.6 Function Prototypes and Argument Coercion (cont.)

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.
- 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.

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.

5.7 C++ Standard Library Header Files

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.)

5.7 C++ Standard Library Header Files

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.)

5.7 C++ Standard Library Header Files

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.)

5.7 C++ Standard Library Header Files

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

- The element of chance can be introduced into computer applications by using the C++ Standard Library function **rand**.
  - The function prototype for the `rand` function is in `<cstdlib>`.
  - For example: `i = rand();`
- Function `rand` generates an unsigned integer between 0 and `RAND_MAX` (a constant defined in the `<cstdlib>` header file).
  - For GNU C++, the value of `RAND_MAX` is 2147483647; for Visual Studio, the value of `RAND_MAX` is 32767.
- To produce integers in the range 0 to 5, we use the modulus operator (%) with `rand` as follows → `rand() % 6`
  - The number 6 is called the scaling factor.
- Shifting the range of numbers produces the integers from 1 to 6.

```
1 + rand() % 6
```


5.8 Case Study: Random Number Generation

```
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
```

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

5.8 Case Study: Random Number Generation (cont.)

```
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.)

5.8 Case Study: Random Number Generation (cont.)

```
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.)

5.8 Case Study: Random Number Generation (cont.)

```
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.)

- Fig. 5.8 simulates 6,000,000 rolls of a die to show the probability of the produced numbers by `rand`.
  - The values 1–6 should appear approximately 1,000,000 times each.
  - Confirmed by the program's output.
- The program should never get to the `default` case (lines 45–46) in the `switch` structure, because the 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.

5.8 Case Study: Random Number Generation (cont.)

- Function `rand` actually generates pseudorandom numbers.
- The numbers in the sequence appear to be random, but the sequence repeats itself each time the program executes.
- It can be conditioned to produce a different sequence of random numbers for each execution.
- 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.
  - The function prototype for `srand` is in header file `<cstdlib>`.

5.8 Case Study: Random Number Generation (cont.)

```
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.)

5.8 Case Study: Random Number Generation (cont.)

```
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) 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>`.

5.9 Case Study: Game of Chance; Introducing enum

- One of the popular dice games, known as "**craps**," is played in casinos and back alleys worldwide.
- A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5 and 6 spots.
- If the sum of the two upward faces 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).

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

```
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.)

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

```
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.)

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

```
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.)

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

```
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.)

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

```
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.)

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

- Variable `gameStatus` is declared to be of new type `Status`.
- 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 this program, 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.
  - Improve the readability of programs.

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

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.

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

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 5.13

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

5.10 Storage Classes

- An identifier's storage class determines the period during which that identifier exists in memory.
- Automatic storage
  - Object created and destroyed within its block
  - **auto**: default storage for local variables (rarely explicitly used)
    - `auto double x, y;`
- Static storage
  - Variables exist for entire program execution
  - **static**: local variables defined in functions
    - Keep value after function ends
    - Only known in their own function
  - **extern**: default for global variables and functions
    - Known in any function

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
  - Local variables: declared with the storage-class specifier `static`
  - Both variables can retain their values throughout the execution
- 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 functions can be referenced by any function after their declarations or definitions in the code.

5.10 Storage Classes (cont.)

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.

5.11 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

- Global namespace scope
  - Identifier declared outside any function or class → known in all functions from the point at which it is declared
  - Used for global variables, function definitions, function prototypes
- Function scope
  - Only for labels (identifiers followed by a colon, such as `start:`)
  - Can only be referenced inside the function in which they appear

5.11 Scope Rules

- Local scope
  - Identifier declared inside a block
    - Local scope begins at definition, ends at right brace (})
  - Used for local variables, static variables, and function parameters
  - Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block
  - The inner block sees the value of its own local identifier and not that of the identically named identifier in the enclosing block.
  - 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

- Function prototype scope
  - 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.

5.11 Scope Rules

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

```
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.)

5.11 Scope Rules

```
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.)

5.11 Scope Rules

```
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.)

5.11 Scope Rules

```
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

- 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

- 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

- 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

- 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

- 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

- 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

```
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

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

5.12 Function Call Stack and Activation Records

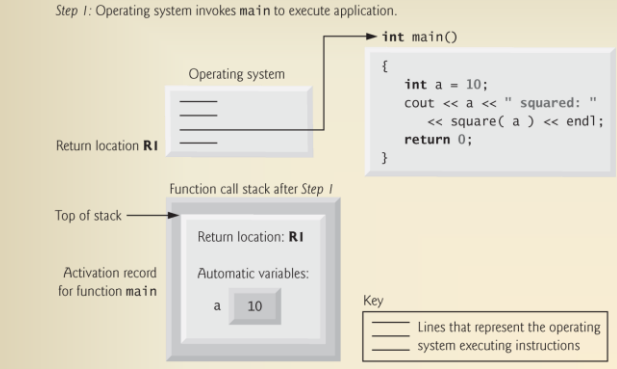


Fig. 5.13 | Function call stack after the operating system invokes `main` to execute the program.

5.12 Function Call Stack and Activation Records

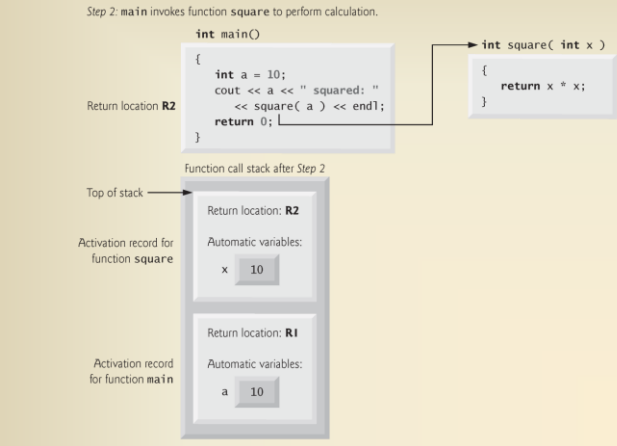


Fig. 5.14 | Function call stack after `main` invokes `square` to perform the calculation.

5.12 Function Call Stack and Activation Records

- 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.

5.12 Function Call Stack and Activation Records

Step 3: `square` returns its result to `main`.

```
int main()
{
   int a = 10;
   cout << a << " squared: "
      << square( a ) << endl;
   return 0;
}
```

Return location **R2**

```
int square( int x )
{
   return x * x;
}
```

Function call stack after *Step 3*

Top of stack →

Return location: **R1**

Automatic variables:

a 10

Activation record for function `main`

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

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.

5.13 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
```

5.13 Functions with Empty Parameter Lists

```
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
```

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.

5.15 References and Reference Parameters

- There are two ways to pass arguments to functions.
- Pass-by-value: (the default way)
  - A copy of the argument's value is passed to the called function
  - Changes to the copy do not affect the original variable's value in the caller
  - Prevents accidental side effects of functions → more reliable
- Pass-by-reference:
  - Gives called function the ability to access and modify the caller's argument data directly
  - More convenient, but hard to debug …
    - Who changed the data ?? When were the data changed??

5.15 References and Reference Parameters

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.

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.)

- Reference parameters—the first of the two means C++ provides for performing pass-by-reference.
- A reference parameter is an alias for its corresponding argument in a function call.
- To declare a reference parameter, simply place an ampersand (&) after the parameter type in the function prototype and function header
- Example: `int &count` in a function header
  - Pronounced as "count is a reference to an int"
- Parameter name in the body of the called function actually refers to the original variable in the calling function
  - The original variable can be modified directly by the called function.

5.15 References and Reference Parameters (cont.)

```
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.)

5.15 References and Reference Parameters (cont.)

```
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.)

- References can also be used as aliases for other variables within a function.
- 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 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.

5.15 References and Reference Parameters (cont.)

```
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.

5.15 References and Reference Parameters (cont.)

```
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
```

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.)

5.16 Default Arguments

- If a function is frequently used with the same argument value for a particular parameter, you can specify that such a parameter has a default value to be passed to that parameter.
- When an argument is omitted 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.
- Default arguments must be specified with the first occurrence of the function name—typically, in the function prototype.
- Default values can be any expression, including constants, global variables or function calls.

5.16 Default Arguments

```
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.)

5.16 Default Arguments

```
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.)

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.

5.17 Unary Scope Resolution Operator

```
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.

5.17 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.

5.17 Unary Scope Resolution Operator

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.

5.18 Function Overloading

Good Programming Practice 5.10
Overloading functions that perform closely related tasks can make programs more readable and understandable.

Common Programming Error 5.19
Creating overloaded functions with identical parameter lists and different return types is a compilation error.

5.18 Function Overloading

```
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

- 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.

5.18 Function Overloading

- 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.

5.18 Function Overloading

```
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.)

5.18 Function Overloading

```
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

- 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

```
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

- 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.

5.19 Function Templates

- 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.

5.19 Function Templates

```
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.)

5.19 Function Templates

```
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

- Divide and conquer is often adopted to solve complex problems.
  - **Divide**: Recursively break down a problem into two or more sub-problems of the same (or related) type
  - **Conquer**: Until these become simple enough to be solved directly
  - **Combine**: The solutions to the sub-problems are then combined to give a solution to the original problem
- Correctness: proved by mathematical induction
- The first domino falls.
- If a domino falls, so will the next domino.
- **All dominoes will fall!**

5.20 Recursion (cont.)

- A recursive function is a function that calls itself, either directly, or indirectly (through another function).
- The function only knows how to solve 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
  - For complex problem, the function divides a problem into
    - What it can do (base case) → return the result
    - What it cannot do → resemble the original problem, but be a slightly simpler or smaller version
    - The function calls a new copy of itself (recursion step) to solve the smaller problem
  - Eventually base case gets solved
    - Gets plugged in, works its way up and solves whole problem

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 can be calculated iteratively (nonrecursively) by using a `for` statement.
- A recursive definition of the factorial function can be observed from the following algebraic relationship:

  $n! = n \cdot (n-1)!$
- 5! is clearly equal to 5 * 4! as is shown by the following:

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

5.20 Recursion (cont.)

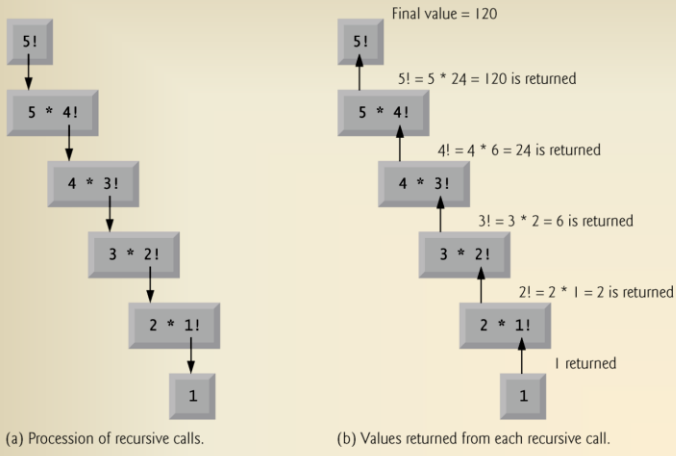


Fig. 5.27 | Recursive evaluation of 5!.

5.20 Recursion (cont.)

```
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.)

5.20 Recursion (cont.)

```
 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.)

5.21 Example Using Recursion: Fibonacci Series

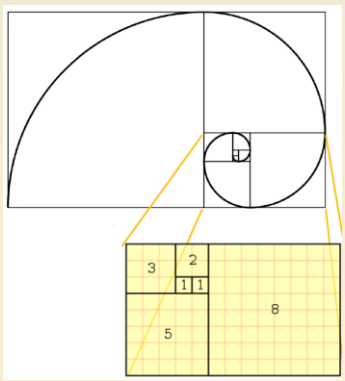
- The Fibonacci series can be defined recursively as follows:

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

- The series occurs in nature and describes a form of spiral.

  Ex: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

- Fibonacci numbers tend to become large quickly.
  - Chose the data type `unsigned long` for the parameter type and the return type in function `fibonacci`.



5.21 Example Using Recursion: Fibonacci Series

```
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.)

5.21 Example Using Recursion: Fibonacci Series

```
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

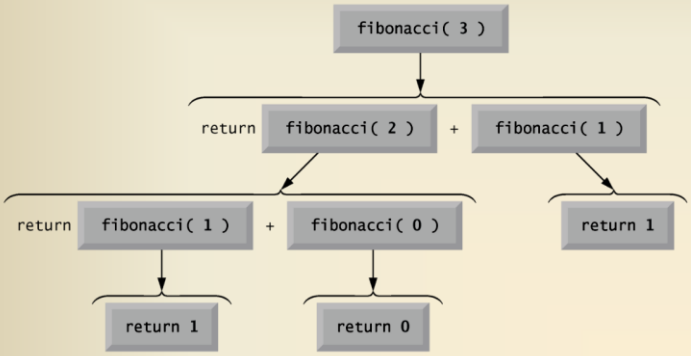


Fig. 5.30 | Set of recursive calls to function `fibonacci`.

5.22 Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance
  - Choice between performance (iteration) and good software engineering (recursion)
  - Avoid using recursion in performance situations
    - Recursive calls take time and consume additional memory

5.22 Recursion vs. Iteration

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.

5.22 Recursion vs. Iteration

```
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.)

5.22 Recursion vs. Iteration

```
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.)

Summary

- Math library
- Function and Parameters
- Function Prototypes and Argument Coercion
- Random number and enum
- Variable Scope
- Overloading and Templates
- Recursion