Chapter 6, Functions and an Introduction to Recursion

C++ How to Program, 7/e

6.1 Introduction

- Experience has shown that the best way to develop and maintain a large program is to construct it from small, simple pieces, or components.
  - This technique is called divide and conquer.
- Emphasize how to declare and use functions to facilitate the design, implementation, operation and maintenance of large programs.

6.2 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.
- 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.

6.2 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 6.1).
  - A boss (similar to the calling function) asks a worker (similar to the called function) to perform a task and report back (i.e., return) the results after completing the task.
  - The boss function does not know how the worker function performs its designated tasks.
  - The worker may also call other worker functions, unbeknownst to the boss.
- This hiding of implementation details promotes good software engineering.

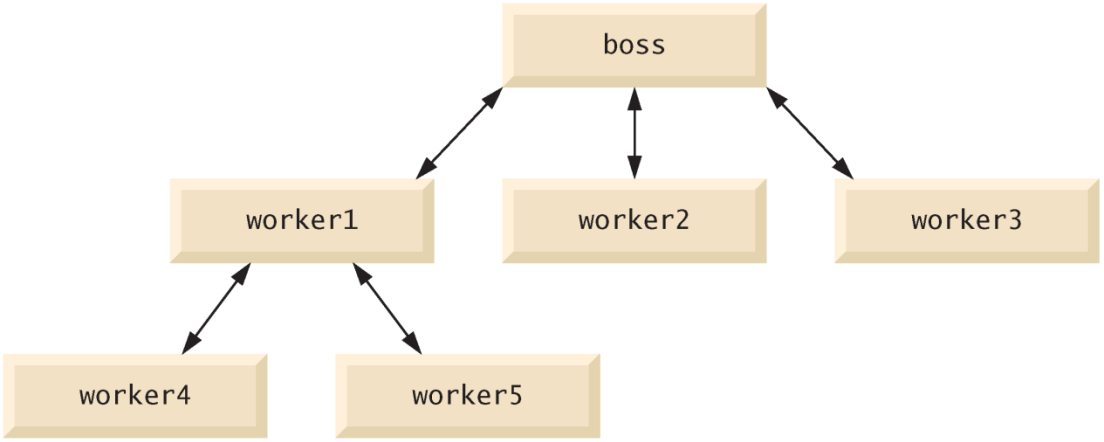


Fig. 6.1 | Hierarchical boss function/worker function relationship.

6.3 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—therefore, each is called simply by specifying the name of the function followed by parentheses containing the function's arguments.*
- Some math library functions are summarized in Fig. 6.2.
  - 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. 6.2 | 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. 6.2 | Math library functions. (Part 2 of 2.)

6.4 Function Definitions with Multiple Parameters

- Functions often require more than one piece of information to perform their tasks.
- Such functions have multiple parameters.
- There must be one argument in the function call for each parameter (also called a formal parameter) in the function definition.
- Multiple parameters are specified in both the function prototype and the function header as a comma-separated list.

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

int maximum(int, int, int);

int main()
{
    int maximumGrade = 0;
    cout << "Enter 3 grades : " << endl;
    int first, second, third;
    cin >> first >> second >> third;
    maximumGrade = maximum(first, second, third);
    cout << "maximum number is " << maximumGrade << endl;
    return 0;
}
```

```
int maximum(int a, int b, int c)
{
    int result = a;
    if (result < b)
        result = b;
    else if (result < c)
        result = c;
    return result;
}
```


6.5 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.

6.5 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.
  - Signature does not specify the function's return type.
- Functions in the same scope must have unique signatures.

6.5 Function Prototypes and Argument Coercion (cont.)

- An important feature of function prototypes is argument coercion
  - forcing arguments to the appropriate types specified by the parameter declarations.
  - These conversions occur as specified by C++'s promotion rules.
  - The promotion rules indicate how to convert between types without losing data.
- The promotion rules apply to expressions containing values of two or more data types
  - 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.

6.5 Function Prototypes and Argument Coercion (cont.)

- Promotion also occurs when the type of a function argument does not match the parameter type specified in the function definition or prototype.
- Figure 6.6 lists the fundamental data types in order from “highest type” to “lowest type.”
- 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 or by using a cast operator.

Data types
`long double`
`double`
`float`
`unsigned long int` (synonymous with `unsigned long`)
`long int` (synonymous with `long`)
`unsigned int` (synonymous with `unsigned`)
`int`
`unsigned short int` (synonymous with `unsigned short`)
`short int` (synonymous with `short`)
`unsigned char`
`char`
`bool`

Fig. 6.6 | Promotion hierarchy for fundamental data types.

6.7 Case Study: Random Number Generation

- The element of chance can be in-troduced into computer applications by using the C++ Standard Library function `rand`.
- The function `rand` generates an unsigned integer between 0 and `RAND_MAX` (a symbolic 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.
- 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.*

6.7 Case Study: Random Number Generation (cont.)

- 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`:
  - `rand() % 6`
  - This is called scaling.
  - The number 6 is called the scaling factor. Six values are produced.
- We can shift the range of numbers produced by adding a value.

```
1  // Fig. 6.8: fig06_08.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. 6.8 | 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. 6.8 | Shifted, scaled integers produced by `1 + rand() % 6`. (Part 2 of 2.)

6.7 Case Study: Random Number Generation (cont.)

- To show that the numbers produced by `rand` occur with approximately equal likelihood, Fig. 6.9 simu-lates 6,000,000 rolls of a die.
- Each integer in the range 1 to 6 should ap-pear approximately 1,000,000 times.

```
1  // Fig. 6.9: fig06_09.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. 6.9 | 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. 6.9 | 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. 6.9 | Rolling a six-sided die 6,000,000 times. (Part 3 of 3.)

6.7 Case Study: Random Number Generation (cont.)

- Executing the program of Fig. 6.8 again produces exactly the same sequence of values.
  - This repeatability is an important characteristic of function `rand`.
  - Essential for proving that program works and for debugging.
- Function `rand` actually generates pseudorandom numbers.
  - Repeatedly calling `rand` produces a sequence of numbers that appears to be random.
  - The sequence repeats itself each time the program executes.

6.7 Case Study: Random Number Generation (cont.)

- Once a program has been debugged, 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.

```
1  // Fig. 6.10: fig06_10.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. 6.10 | 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. 6.10 | Randomizing the die-rolling program. (Part 2 of 2.)

6.7 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 re-turns the current time as the number of seconds since January 1, 1970, at midnight Greenwich Mean Time (GMT).
- The function prototype for `time` is in `<ctime>`.

6.7 Case Study: Random Number Generation (cont.)

- To produce random numbers in a specific range use:
  - `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 de-sired range of consecutive integers.*

6.10 Scope Rules

- The portion of the program where an identifier can be used is known as its scope.

```
1  // Fig. 6.12: fig06_12.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. 6.12 | 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. 6.12 | 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. 6.12 | 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. 6.12 | Scoping example. (Part 4 of 4.)

6.16 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.
- 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. 6.23: fig06_23.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. 6.23 | Unary scope resolution operator.

6.12 Functions with Empty Parameter Lists

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

```
1  // Fig. 6.17: fig06_17.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. 6.17 | 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. 6.17 | Functions that take no arguments. (Part 2 of 2.)

6.14 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.
- To specify a reference to a constant, place the `const` qualifier before the type specifier in the parameter declaration.

6.14 References and Reference Parameters (cont.)

- 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 pa-rameter's type in the function prototype by an ampersand (&); use the same convention when listing the pa-rameter's type in the function header.

```
1  // Fig. 6.19: fig06_19.cpp
2  // Comparing pass-by-value and pass-by-referentces.
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. 6.19 | 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. 6.19 | Passing arguments by value and by reference. (Part 2 of 2.)

6.15 Default Arguments

- It isn't uncommon for a program to invoke a function repeatedly with the same argument value for a particular parameter.
- 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 parame-ter list.

```
1  // Fig. 6.22: fig06_22.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. 6.22 | 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. 6.22 | Default arguments to a function. (Part 2 of 2.)

6.17 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 sev-eral functions of the same name that perform similar tasks, but on different data types.

```
1  // Fig. 6.24: fig06_24.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. 6.24 | 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. 6.24 | Overloaded `square` functions. (Part 2 of 2.)

6.19 Recursion

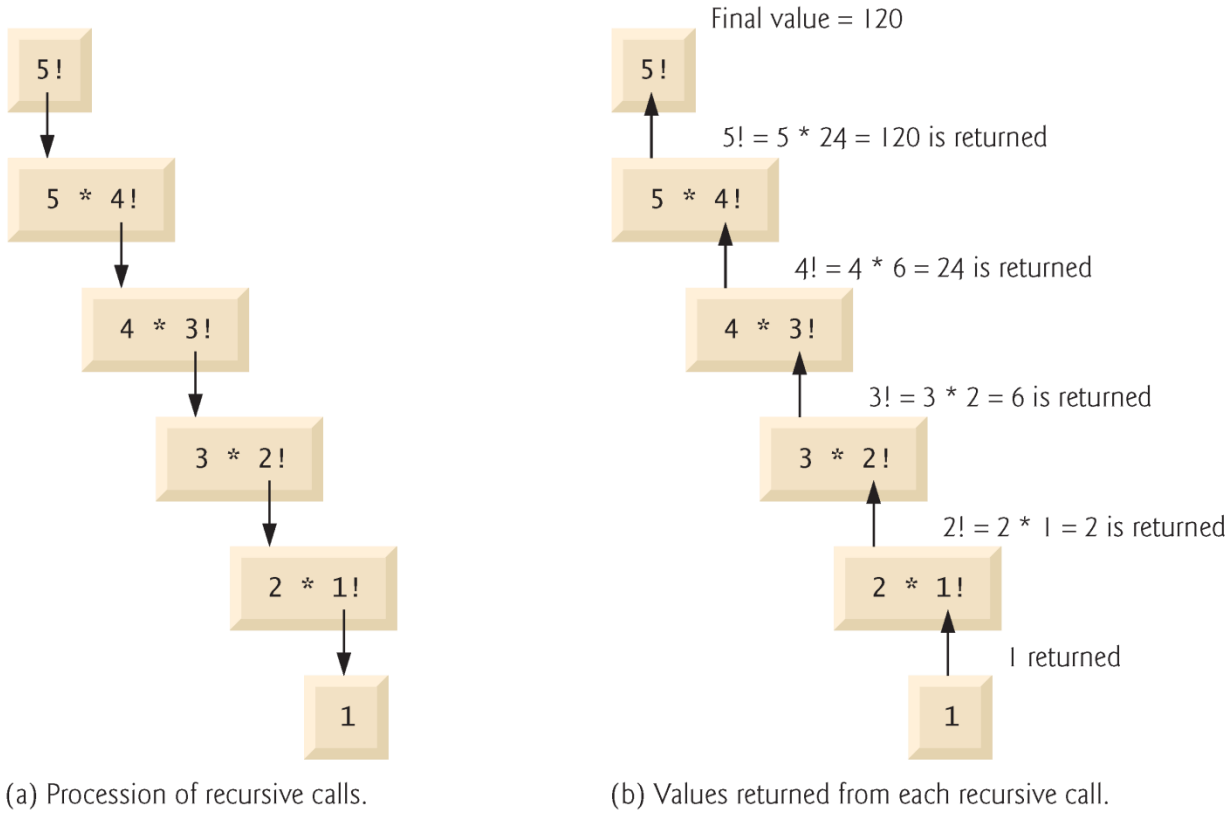
- A recursive function is a function that calls itself, either directly, or indirectly (through another function).
- Recursive problem-solving approaches have a number of elements in common.
  - A recursive function is called to solve a problem.
  - The function actu-ally knows how to solve only the simplest case(s), or so-called base case(s).
  - If the func-tion 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.
  - 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.

6.19 Recursion (cont.)

- The recursion step often includes the key-word `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.

6.19 Recursion (cont.)

- The factorial of a nonnegative integer *n*, *written n! (and pronounced "n factorial"), is the product*
  - *n · (n – 1) · (n – 2) · … · 1*
- with 1! equal to 1, and 0! defined to be 1.
- The factorial of an integer, `number`, greater than or equal to 0, can be calculated iteratively (nonrecursively) by using a loop.
- A recursive definition of the factorial function is arrived at by observing the follow-ing algebraic relationship:
  - *n! = n · (n – 1)!*



(a) Procession of recursive calls.

(b) Values returned from each recursive call.

Fig. 6.28 | Recursive evaluation of 5!.

```
1  // Fig. 6.29: fig06_29.cpp
2  // 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. 6.29 | 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. 6.29 | Demonstrating the recursive function `factorial`. (Part 2 of 2.)

6.20 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.

6.20 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. 6.30 calculates the *nth Fibonacci number recursively by using function* `fibonacci`.

```
1  // Fig. 6.30: fig06_30.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. 6.30 | 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. 6.30 | Demonstrating function `fibonacci`. (Part 2 of 2.)

6.20 Example Using Recursion: Fibonacci Series (cont.)

- Figure 6.31 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.
- Most programmers simply assume that operands are evaluated left to right.
- C++ does not specify the order in which the operands of most operators (including +) are to be evaluated.
- Therefore, you must make no as-sumption about the order in which these calls execute.

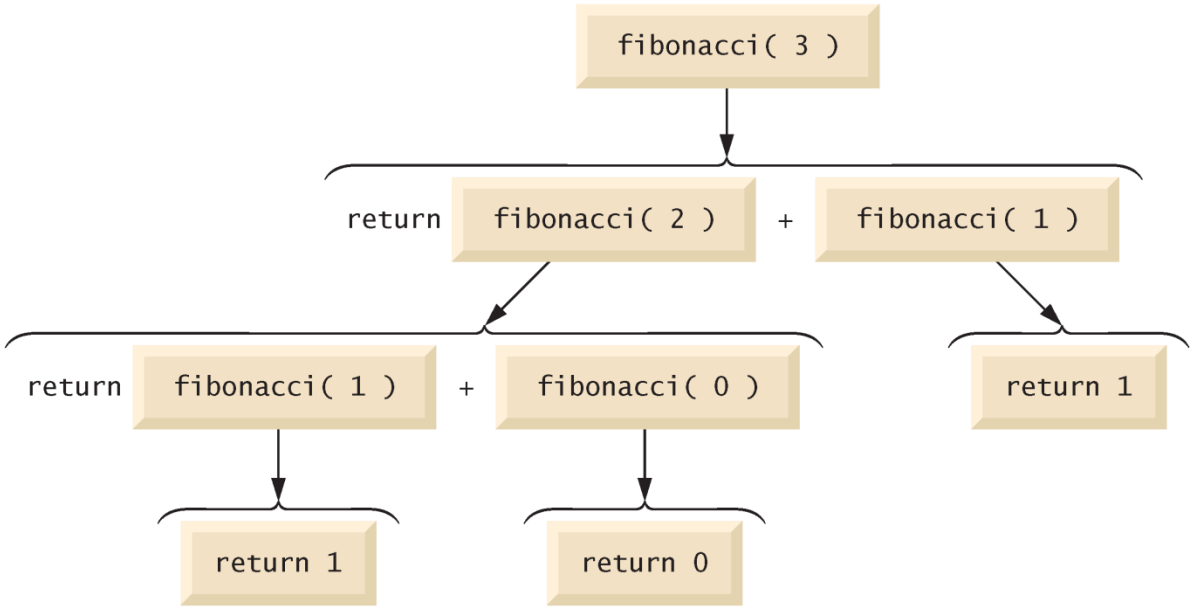


Fig. 6.31 | Set of recursive calls to function `fibonacci`.

6.21 Recursion vs. Iteration

- Both iteration and recursion are based on a control statement: Iteration uses a repeti-tion 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.
- Iteration and recursion both involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.

6.21 Recursion vs. Iteration (cont.)

- 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.
- Both it-eration 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.

```
1  // Fig. 6.32: fig06_32.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. 6.32 | 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. 6.32 | Iterative factorial solution. (Part 2 of 2.)