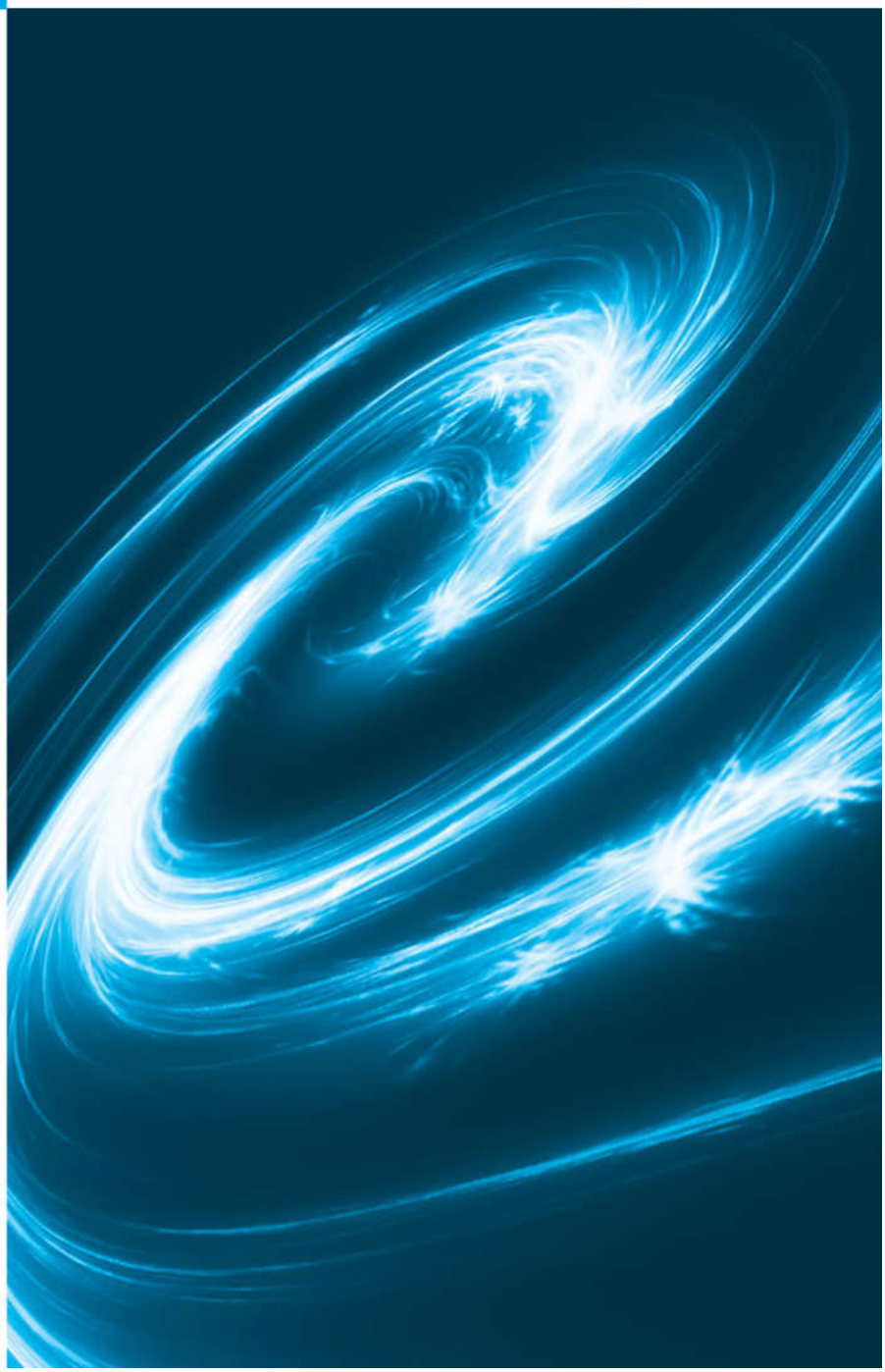
5 C Functions



Form ever follows function.
—Louis Henri Sullivan

O! call back yesterday, bid time return.
—William Shakespeare

Answer me in one word.
—William Shakespeare

There is a point at which methods devour themselves.
—Frantz Fanon

Objectives

In this chapter, you'll:

- Construct programs modularly from small pieces called functions.
- Use common math functions in the C standard library.
- Create new functions.
- Use the mechanisms that pass information between functions.
- Learn how the function call/return mechanism is supported by the function call stack and stack frames.
- Use simulation techniques based on random number generation.
- Write and use functions that call themselves.

5.1 Introduction

Most computer programs that solve real-world problems are much larger than the programs presented in the first few chapters. Experience has shown that the best way to develop and maintain a large program is to construct it from smaller pieces or modules, each of which is more manageable than the original program. This technique is called divide and conquer. This chapter describes some key features of the C language that facilitate the design, implementation, operation and maintenance of large programs.

5.2 Program Modules in C

Modules in C are called functions. C programs are typically written by combining new functions you write with *prepackaged* functions available in the C standard library. We discuss both kinds of functions in this chapter. The C standard library provides a rich collection of functions for performing common *mathematical calculations*, *string manipulations*, *character manipulations*, *input/output*, and many other useful operations. This makes your job easier, because these functions provide many of the capabilities you need.

Good Programming Practice 5.1

Familiarize yourself with the rich collection of functions in the C standard library.

Software Engineering Observation 5.1

Avoid reinventing the wheel. When possible, use C standard library functions instead of writing new functions. This can reduce program development time.

Portability Tip 5.1

Using the functions in the C standard library helps make programs more portable.

The C language and the standard library are *both* specified by the C standard, and they're both provided with standard C systems (with the exception that some of the

libraries are designated as optional). The functions `printf`, `scanf` and `pow` that we've used in previous chapters are standard library functions.

You can write functions to define specific tasks that may be used at many points in a program. These are sometimes referred to as **programmer-defined functions**. The actual statements defining the function are written only once, and the statements are hidden from other functions.

Functions are **invoked** by a **function call**, which specifies the function name and provides information (as arguments) that the function needs to perform its designated task. A common analogy for this is the hierarchical form of management. A boss (the **calling function** or **caller**) asks a worker (the **called function**) to perform a task and report back when the task is done (Fig. 5.1). For example, a function needing to display information on the screen calls the worker function `printf` to perform that task, then `printf` displays the information and reports back—or **returns**—to the calling function when its task is completed. The boss function does *not* know how the worker function performs its designated tasks. The worker may call other worker functions, and the boss will be unaware of this. We'll soon see how this "hiding" of implementation details promotes good software engineering. Figure 5.1 shows a boss function communicating with several worker functions in a hierarchical manner. Note that `Worker1` acts as a boss function to `worker4` and `worker5`. Relationships among functions may differ from the hierarchical structure shown in this figure.

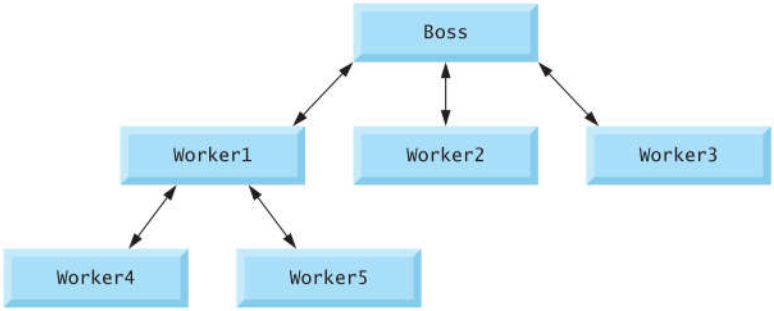


Fig. 5.1 | Hierarchical boss-function/worker-function relationship.

5.3 Math Library Functions

Math library functions allow you to perform certain common mathematical calculations. We use some of them here to introduce the concept of functions. Later in the book, we'll discuss many of the other functions in the C standard library.

Functions are normally used in a program by writing the name of the function followed by a left parenthesis followed by the **argument** (or a comma-separated list of arguments) of the function followed by a right parenthesis. For example, to calculate and print the square root of `900.0` you might write

```
printf( "%.2f", sqrt( 900.0 ) );
```

When this statement executes, the math library function `sqrt` is *called* to calculate the square root of the number contained in the parentheses (`900.0`). The number `900.0` is the *argument* of the `sqrt` function. The preceding statement would print `30.00`. The `sqrt` function takes

an argument of type `double` and returns a result of type `double`. All functions in the math library that return floating-point values return the data type `double`. Note that `double` values, like `float` values, can be output using the `%f` conversion specification.

Error-Prevention Tip 5.1

Include the math header by using the preprocessor directive `#include <math.h>` *when using functions in the math library.*

Function arguments may be constants, variables, or expressions. If `c1 = 13.0`, `d = 3.0` and `f = 4.0`, then the statement

```
printf( "%.2f", sqrt( c1 + d * f ) );
```

calculates and prints the square root of `13.0 + 3.0 * 4.0 = 25.0`, namely `5.00`.

Figure 5.2 summarizes a small sample of the C math library functions. In the figure, the variables `x` and `y` are of type `double`. The C11 standard adds a wide range of floating-point and complex-number capabilities.

Function	Description	Example
`sqrt( x )`	square root of x	`sqrt( 900.0 )` is `30.0` `sqrt( 9.0 )` is `3.0`
`cbrt( x )`	cube root of x (C99 and C11 only)	`cbrt( 27.0 )` is `3.0` `cbrt( -8.0 )` is `-2.0`
`exp( x )`	exponential function e^x	`exp( 1.0 )` is `2.718282` `exp( 2.0 )` is `7.389056`
`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( 1.0 )` is `0.0`
		`log10( 10.0 )` is `1.0`
		`log10( 100.0 )` is `2.0`

`fabs( x )`	absolute value of x as a floating-point number	`fabs( 13.5 )` is `13.5`
		`fabs( 0.0 )` is `0.0`
		`fabs( -13.5 )` is `13.5`

`ceil( x )`	rounds x to the smallest integer not less than x	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`
`floor( x )`	rounds x to the largest integer not greater than x	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`pow( x, y )`	x raised to power y (x^y)	`pow( 2, 7 )` is `128.0` `pow( 9, .5 )` is `3.0`
`fmod( x, y )`	remainder of x/y as a floating-point number	`fmod( 13.657, 2.333 )` is `1.992`
`sin( x )`	trigonometric sine of x (x in radians)	`sin( 0.0 )` is `0.0`
`cos( x )`	trigonometric cosine of x (x in radians)	`cos( 0.0 )` is `1.0`
`tan( x )`	trigonometric tangent of x (x in radians)	`tan( 0.0 )` is `0.0`

Fig. 5.2 | Commonly used math library functions.

5.4 Functions

Functions allow you to modularize a program. All variables defined in function definitions are local variables—they can be accessed *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.

Software Engineering Observation 5.2

In programs containing many functions, `main` is often implemented as a group of calls to functions that perform the bulk of the program's work.

There are several motivations for "functionalizing" a program. The *divide-and-conquer* approach makes program development more manageable. Another motivation is software reusability—using existing functions as *building blocks* to create new programs. Software reusability is a major factor in the *object-oriented programming* movement that you'll learn more about when you study languages derived from C, such as C++, Java and C# (pronounced "C sharp"). With good function naming and definition, programs can be created from standardized functions that accomplish specific tasks, rather than being built by using customized code. This is known as abstraction. We use abstraction each time we use standard library functions like `printf`, `scanf` and `pow`. A third motivation is to avoid repeating code in a program. Packaging code as a function allows the code to be executed from other locations in a program simply by calling the function.

Software Engineering Observation 5.3

Each function should be limited to performing a single, well-defined task, and the function name should express that task. This facilitates abstraction and promotes software reusability.

Software Engineering Observation 5.4

If you cannot choose a concise name that expresses what the function does, it's possible that your function is attempting to perform too many diverse tasks. It's usually best to break such a function into several smaller functions—this is sometimes called decomposition.

5.5 Function Definitions

Each program we've presented has consisted of a function called `main` that called standard library functions to accomplish its tasks. We now consider how to write *custom* functions. Consider a program that uses a function `square` to calculate and print the squares of the integers from 1 to 10 (Fig. 5.3).

```
1  // Fig. 5.3: fig05_03.c
2  // Creating and using a programmer-defined function.
3  #include <stdio.h>
4
5  int square( int y ); // function prototype
6
```

Fig. 5.3 | Creating and using a programmer-defined function. (Part 1 of 2.)

```
// function main begins program execution
int main( void )
{
   int x; // counter

   // loop 10 times and calculate and output square of x each time
   for ( x = 1; x <= 10; ++x ) {
      printf( "%d  ", square( x ) ); // function call
   } // end for

   puts( "" );
} // end main

// square function definition returns the square of its parameter
int square( int y ) // y is a copy of the argument to the function
{
   return y * y; // returns the square of y as an int
} // end function square
```

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

Fig. 5.3 | Creating and using a programmer-defined function. (Part 2 of 2.)

Function `square` is **invoked** or **called** in `main` within the `printf` statement (line 14)

```
   printf( "%d  ", square( x ) ); // function call
```

Function `square` receives a *copy* of the value of `x` in the parameter `y` (line 21). Then `square` calculates `y * y`. The result is passed back returned to function `printf` in `main` where `square` was invoked (line 14), and `printf` displays the result. This process is repeated 10 times using the `for` statement.

The definition of function `square` (lines 21–24) shows that `square` expects an integer parameter `y`. The keyword `int` preceding the function name (line 21) indicates that `square` *returns* an integer result. The **`return` statement** in `square` passes the value of the expression `y * y` (that is, the result of the calculation) back to the calling function.

Line 5

```
   int square( int y ); // function prototype
```

is a **function prototype**. The `int` in parentheses informs the compiler that `square` expects to *receive* an integer value from the caller. The `int` to the *left* of the function name `square` informs the compiler that `square` *returns* an integer result to the caller. The compiler refers to the function prototype to check that any calls to `square` (line 14) contain the *correct return type*, the *correct number of arguments* and the *correct argument types*, and that the *arguments are in the correct order*. Function prototypes are discussed in detail in Section 5.6.

The format of a function definition is

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

The *function-name* is any valid identifier. The *return-value-type* is the data type of the result returned to the caller. The *return-value-type* `void` indicates that a function does *not* return a value. Together, the *return-value-type*, *function-name* and *parameter-list* are sometimes referred to as the function header.

Error-Prevention Tip 5.2

Check that your functions that are supposed to return values do so. Check that your functions that are not supposed to return values do not.

The *parameter-list* is a comma-separated list that specifies the parameters received by the function when it's called. If a function does *not* receive any values, *parameter-list* is `void`. A type *must* be listed *explicitly* for each parameter.

Common Programming Error 5.1

Specifying function parameters of the same type as `double x, y` *instead of* `double x, double y` *results in a compilation error.*

Common Programming Error 5.2

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

Common Programming Error 5.3

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

Good Programming Practice 5.2

Although it's not incorrect to do so, do not use the same names for a function's arguments and the corresponding parameters in the function definition. This helps avoid ambiguity.

The *definitions* and *statements* within braces form the function body, which is also referred to as a block. Variables can be declared in any block, and blocks can be nested.

Common Programming Error 5.4

Defining a function inside another function is a syntax error.

Good Programming Practice 5.3

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

Software Engineering Observation 5.5

Small functions promote software reusability.

Software Engineering Observation 5.6

Programs should be written as collections of small functions. This makes programs easier to write, debug, maintain and modify.

Software Engineering Observation 5.7

A function requiring a large number of parameters may be performing too many tasks. Consider dividing the function into smaller functions that perform the separate tasks. The function header should fit on one line if possible.

Software Engineering Observation 5.8

The function prototype, function header and function calls should all agree in the number, type, and order of arguments and parameters, and in the type of return value.

There are three ways to return control from a called function to the point at which a function was invoked. If the function does *not* return a result, control is returned simply when the function-ending right brace is reached, or by executing the statement

```
return;
```

If the function *does* return a result, the statement

```
return expression;
```

returns the value of *expression* to the caller.

`main`'s Return Type

Notice that `main` has an `int` return type. The return value of `main` is used to indicate whether the program executed correctly. In earlier versions of C, we'd explicitly place

```
return 0;
```

at the end of `main`—`0` indicates that a program ran successfully. The C standard indicates that `main` implicitly returns `0` if you to omit the preceding statement—as we've done throughout this book. You can explicitly return non-zero values from `main` to indicate that a problem occured during your program's execution. For information on how to report a program failure, see the documentation for your particular operating-system environment.

Function `maximum`

Our second example uses a programmer-defined function `maximum` to determine and return the largest of three integers (Fig. 5.4). The integers are input with `scanf` (line 15). Next, they're passed to `maximum` (line 19), which determines the largest integer. This value is returned to main by the `return` statement in `maximum` (line 36). The value returned is then printed in the `printf` statement (line 19).

```
1  // Fig. 5.4: fig05_04.c
2  // Finding the maximum of three integers.
3  #include <stdio.h>
4
5  int maximum( int x, int y, int z ); // function prototype
6
7  // function main begins program execution
8  int main( void )
9  {
```

Fig. 5.4 | Finding the maximum of three integers. (Part 1 of 2.)

```
10     int number1; // first integer entered by the user
11     int number2; // second integer entered by the user
12     int number3; // third integer entered by the user
13
14     printf( "%s", "Enter three integers: " );
15     scanf( "%d%d%d", &number1, &number2, &number3 );
16
17     // number1, number2 and number3 are arguments
18     // to the maximum function call
19     printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );
20  } // end main
21
22  // Function maximum definition
23  // x, y and z are parameters
24  int maximum( int x, int y, int z )
25  {
26     int max = x; // assume x is largest
27
28     if ( y > max ) { // if y is larger than max,
29        max = y; // assign y to max
30     } // end if
31
32     if ( z > max ) { // if z is larger than max,
33        max = z; // assign z to max
34     } // end if
35
36     return max; // max is largest value
37  } // end function maximum
```

```
Enter three integers: 22 85 17
Maximum is: 85
```

```
Enter three integers: 47 32 14
Maximum is: 47
```

```
Enter three integers: 35 8 79
Maximum is: 79
```

Fig. 5.4 | Finding the maximum of three integers. (Part 2 of 2.)

5.6 Function Prototypes: A Deeper Look

An important feature of C is the function prototype. This feature was borrowed from C++. The compiler uses function prototypes to validate function calls. Early versions of C did *not* perform this kind of checking, so it was possible to call functions improperly without the compiler detecting the errors. Such calls could result in fatal execution-time errors or nonfatal errors that caused subtle, difficult-to-detect problems. Function prototypes correct this deficiency.

Good Programming Practice 5.4

Include function prototypes for all functions to take advantage of C's type-checking capabilities. Use `#include` *preprocessor directives to obtain function prototypes for the standard library functions from the headers for the appropriate libraries, or to obtain headers containing function prototypes for functions developed by you and/or your group members.*

The function prototype for `maximum` in Fig. 5.4 (line 5) is

```
int maximum( int x, int y, int z ); // function prototype
```

It states that `maximum` takes three arguments of type `int` and returns a result of type `int`. Notice that the function prototype is the same as the first line of `maximum`'s definition.

Good Programming Practice 5.5

Parameter names are sometimes included in function prototypes (our preference) for documentation purposes. The compiler ignores *these names.*

Common Programming Error 5.5

Forgetting the semicolon at the end of a function prototype is a syntax error.

Compilation Errors

A function call that does not match the function prototype is a *compilation* error. An error is also generated if the function prototype and the function definition disagree. For example, in Fig. 5.4, if the function prototype had been written

```
void maximum( int x, int y, int z );
```

the compiler would generate an error because the `void` return type in the function prototype would differ from the `int` return type in the function header.

Argument Coercion and "Usual Arithmetic Conversion Rules"

Another important feature of function prototypes is the **coercion of arguments**, i.e., the forcing of arguments to the appropriate type. For example, the math library function `sqrt` can be called with an integer argument even though the function prototype in `<math.h>` specifies a `double` parameter, and the function will still work correctly. The statement

```
printf( "%.3f\n", sqrt( 4 ) );
```

correctly evaluates `sqrt(4)` and prints the value `2.000`. The function prototype causes the compiler to convert a *copy* of the integer value `4` to the `double` value `4.0` before the *copy* is passed to `sqrt`. In general, *argument values that do not correspond precisely to the parameter types in the function prototype are converted to the proper type before the function is called.* These conversions can lead to incorrect results if C's **usual arithmetic conversion rules** are not followed. These rules specify how values can be converted to other types without losing data. In our `sqrt` example above, an `int` is automatically converted to a `double` without changing its value (because `double` can represent a much larger range of values than `int`). However, a `double` converted to an `int` *truncates* the fractional part of the `double` value, thus changing the original value. Converting large integer types to small integer types (e.g., `long` to `short`) may also result in changed values.

The usual arithmetic conversion rules automatically apply to expressions containing values of two data types (also referred to as **mixed-type expressions**), and are handled for

you by the compiler. In a mixed-type expression, the compiler makes a temporary copy of the value that needs to be converted then converts the copy to the "highest" type in the expression—the original value remains unchanged. The usual arithmetic conversion rules for a mixed-type expression containing at least one floating-point value are:

- If one of the values is a `long double`, the other is converted to a `long double`.
- If one of the values is a `double`, the other is converted to a `double`.
- If one of the values is a `float`, the other is converted to a `float`.

If the mixed-type expression contains only integer types, then the usual arithmetic conversions specify a set of integer promotion rules. In *most* cases, the integer types lower in Fig. 5.5 are converted to types higher in the figure. Section 6.3.1 of the C standard document specifies the complete details of arithmetic operands and the usual arithmetic conversion rules. Figure 5.5 lists the floating-point and integer data types with each type's `printf` and `scanf` conversion specifications.

Data type	`printf` conversion specification	`scanf` conversion specification
Floating-point types		
`long double`	`%Lf`	`%Lf`
`double`	`%f`	`%lf`
`float`	`%f`	`%f`
Integer types		
`unsigned long long int`	`%llu`	`%llu`
`long long int`	`%lld`	`%lld`
`unsigned long int`	`%lu`	`%lu`
`long int`	`%ld`	`%ld`
`unsigned int`	`%u`	`%u`
`int`	`%d`	`%d`
`unsigned short`	`%hu`	`%hu`
`short`	`%hd`	`%hd`
`char`	`%c`	`%c`

Fig. 5.5 | Arithmetic data types and their conversion specifications.

Converting values to *lower* types in Fig. 5.5 can result in incorrect values, so the compiler typically issues warnings for such cases. A value can be converted to a lower type *only* by explicitly assigning the value to a variable of lower type or by using a *cast* operator. Arguments in a function call are converted to the parameter types specified in a function prototype as if the arguments were being assigned directly to variables of those types. If our `square` function that uses an `int` parameter (Fig. 5.3) is called with a floating-point argument, the argument is converted to `int` (a lower type), and `square` usually returns an incorrect value. For example, `square(4.5)` returns `16`, not `20.25`.

Common Programming Error 5.6

Converting from a higher data type in the promotion hierarchy to a lower type can change the data value. Many compilers issue warnings in such cases.

If there's no function prototype for a function, the compiler forms its own function prototype using the first occurrence of the function—either the function definition or a call to the function. This typically leads to warnings or errors, depending on the compiler.

Error-Prevention Tip 5.3

Always include function prototypes for the functions you define or use in your program to help prevent compilation errors and warnings.

Software Engineering Observation 5.9

A function prototype placed outside any function definition applies to all calls to the function appearing after *the function prototype in the file. A function prototype placed in a function applies only to calls made in that function.*

5.7 Function Call Stack and Stack Frames

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.

An important mechanism 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. 5.7–5.9, this LIFO behavior is *exactly* what a function does when returning to the function that called it.

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

Each called function *always* finds the information it needs to return to its caller at the *top* of the call stack. And, if a function makes a call to another function, 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.

The stack frames have another important responsibility. Most functions have *automatic variables*—parameters and some or all of their local variables. 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 automatic variables. That stack frame exists only 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.

Of course, the amount of memory in a computer is finite, so only a certain amount of memory can be used to store stack frames on the function call stack. If more function calls occur than can have their stack frames stored on the function call stack, a *fatal* error known as stack overflow occurs.

Function Call Stack in Action

Now let's consider how the call stack supports the operation of a `square` function called by `main` (lines 8–13 of Fig. 5.6). First the operating system calls `main`—this pushes a stack frame onto the stack (shown in Fig. 5.7). The stack frame 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 12 of Fig. 5.6. This causes a stack frame for `square` (lines 16–19) to be pushed onto the function call stack (Fig. 5.8). 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`).

```
1  // Fig. 5.6: fig05_06.c
2  // Demonstrating the function call stack
3  // and stack frames using a function square.
4  #include <stdio.h>
5
6  int square( int ); // prototype for function square
7
8  int main()
9  {
10    int a = 10; // value to square (local automatic variable in main)
11
12    printf( "%d squared: %d\n", a, square( a ) ); // display a squared
13 } // end main
14
15 // returns the square of an integer
16 int square( int x ) // x is a local variable
17 {
18    return x * x; // calculate square and return result
19 } // end function square
```

```
10 squared: 100
```

Fig. 5.6 | Demonstrating the function call stack and stack frames using a function square.

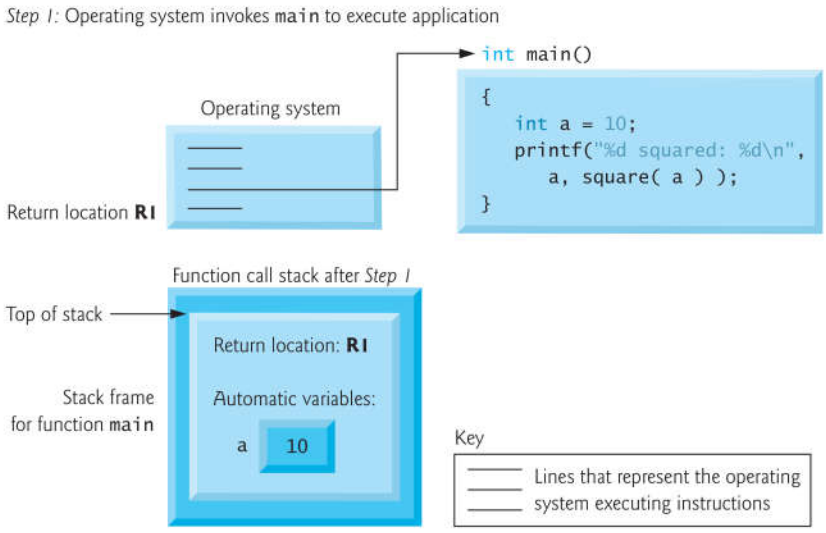


Fig. 5.7 | Function call stack after the operating system invokes `main` to execute the program.

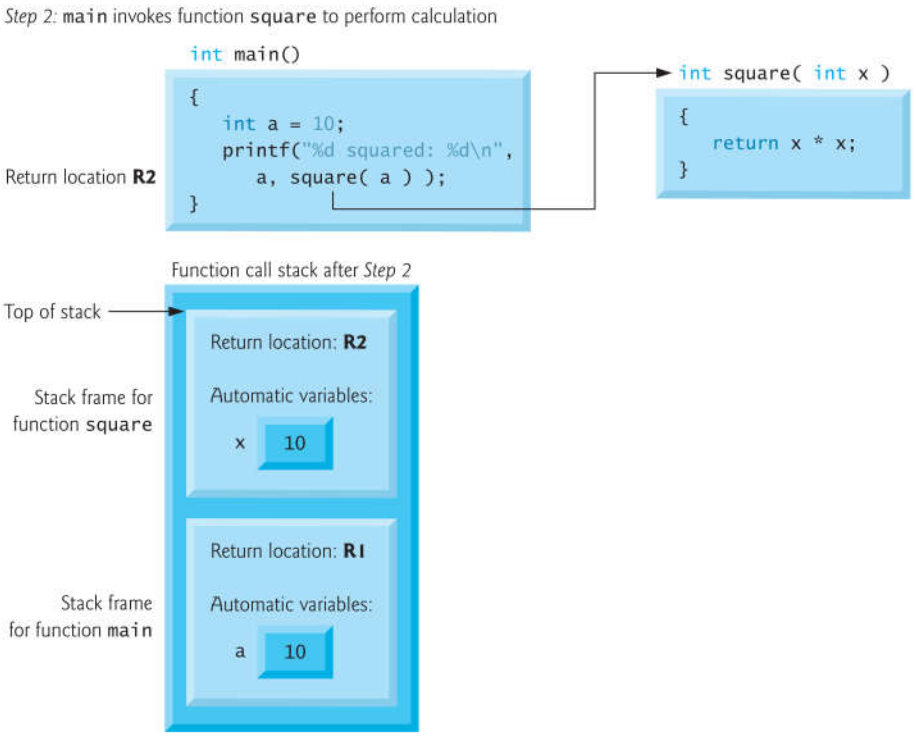


Fig. 5.8 | Function call stack after `main` invokes `square` to perform the calculation.

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 5.9 shows the function call stack *after* `square`'s stack frame has been popped.

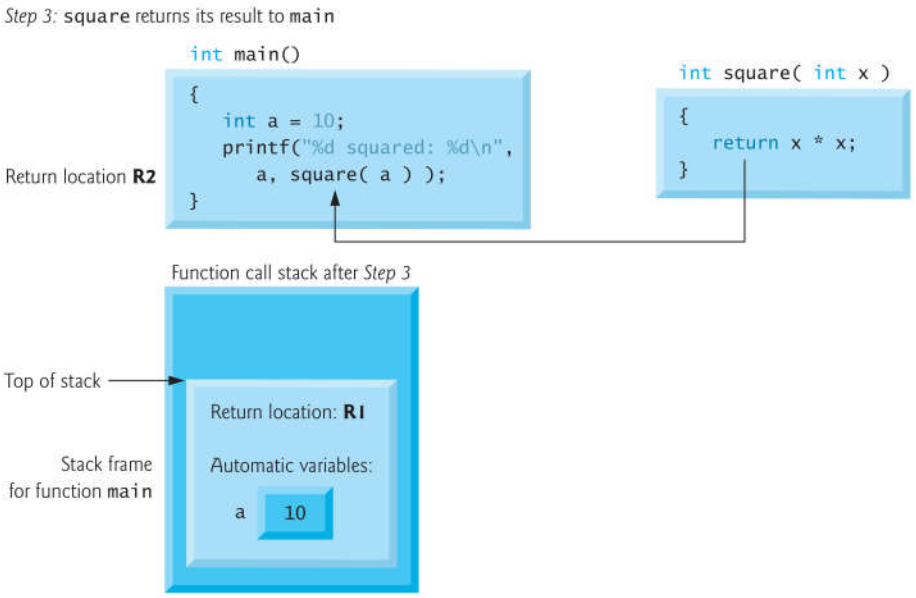


Fig. 5.9 | Function call stack after function `square` returns to `main`.

Function `main` now displays the result of calling `square` (line 12). Reaching the closing right brace of `main` causes its stack frame to be popped from the stack, gives `main` the address it needs to return to the operating system (i.e., `R1` in Fig. 5.7) and causes the memory for `main`'s automatic variable (i.e., `a`) to become unavailable.

You've now seen how valuable the stack data structure is in implementing a key mechanism that supports program execution. Data structures have many important applications in computer science. We discuss stacks, queues, lists, trees and other data structures in Chapter 12.

5.8 Headers

Each standard library has a corresponding **header** containing the function prototypes for all the functions in that library and definitions of various *data types* and constants needed by those functions. Figure 5.10 lists alphabetically some of the standard library headers that may be included in programs. The C standard includes additional headers. The term "macros" that's used several times in Fig. 5.10 is discussed in detail in Chapter 13.

You can create custom headers. Programmer-defined headers should also use the `.h` filename extension. A programmer-defined header can be included by using the `#include` preprocessor directive. For example, if the prototype for our square function was located

Header	Explanation
`<assert.h>`	Contains information for adding diagnostics that aid program debugging.
`<ctype.h>`	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
`<errno.h>`	Defines macros that are useful for reporting error conditions.
`<float.h>`	Contains the floating-point size limits of the system.
`<limits.h>`	Contains the integral size limits of the system.
`<locale.h>`	Contains function prototypes and other information that enables a program to be modified for the current locale on which it's running. The notion of locale enables the computer system to handle different conventions for expressing data such as dates, times, currency amounts and large numbers throughout the world.
`<math.h>`	Contains function prototypes for math library functions.
`<setjmp.h>`	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.
`<signal.h>`	Contains function prototypes and macros to handle various conditions that may arise during program execution.
`<stdarg.h>`	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
`<stddef.h>`	Contains common type definitions used by C for performing calculations.
`<stdio.h>`	Contains function prototypes for the standard input/output library functions, and information used by them.
`<stdlib.h>`	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers, and other utility functions.
`<string.h>`	Contains function prototypes for string-processing functions.
`<time.h>`	Contains function prototypes and types for manipulating the time and date.

Fig. 5.10 | Some of the standard library headers.

in the header `square.h`, we'd include that header in our program by using the following directive at the top of the program:

```
#include "square.h"
```

Section 13.2 presents additional information on including headers.

5.9 Passing Arguments By Value and By Reference

In many programming languages, there are two ways to pass arguments—pass-by-value and pass-by-reference. When arguments are *passed by value*, a *copy* of the argument's value is made and passed to the called function. Changes to the copy do *not* affect an original variable's value in the caller. When an argument is *passed by reference*, the caller allows the called function to *modify* the original variable's value.

Pass-by-value should be used whenever the called function does not need to modify the value of the caller's original variable. This prevents the accidental side effects (variable

modifications) that so greatly hinder the development of correct and reliable software systems. Pass-by-reference should be used only with *trusted* called functions that need to modify the original variable.

In C, all arguments are passed by value. As we'll see in Chapter 7, it's possible to simulate pass-by-reference by using the *address operator* and the *indirection operator*. In Chapter 6, we'll see that array arguments are automatically passed by reference for performance reasons. We'll see in Chapter 7 that this is *not* a contradiction. For now, we concentrate on pass-by-value.

5.10 Random Number Generation

We now take a brief and, hopefully, entertaining diversion into *simulation* and *game playing*. In this and the next section, we'll develop a nicely structured game-playing program that includes multiple functions. The program uses most of the control statements we've studied. The *element of chance* can be introduced into computer applications by using the C standard library function `rand` from the `<stdlib.h>` header.

Consider the following statement:

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

The `rand` function generates an integer between 0 and `RAND_MAX` (a symbolic constant defined in the `<stdlib.h>` header). Standard C states that the value of `RAND_MAX` must be at least 32767, which is the maximum value for a two-byte (i.e., 16-bit) integer. The programs in this section were tested on Microsoft Visual C++ with a maximum `RAND_MAX` value of 32767 and on GNU `gcc` with a maximum `RAND_MAX` value of 2147483647. If `rand` truly produces integers *at random*, every number between 0 and `RAND_MAX` has an equal chance (or probability) of being chosen each time `rand` is called.

The range of values produced directly by `rand` is often different from what's needed in a specific application. For example, a program that simulates coin tossing might require only 0 for "heads" and 1 for "tails." A dice-rolling program that simulates a six-sided die would require random integers from 1 to 6.

Rolling a Six-Sided Die

To demonstrate `rand`, let's develop a program to simulate 20 rolls of a six-sided die and print the value of each roll. The function prototype for function `rand` is in `<stdlib.h>`. We use the remainder operator (`%`) in conjunction with `rand` as follows

```
rand() % 6
```

to produce integers in the range 0 to 5. This is called **scaling**. The number 6 is called the **scaling factor**. We then **shift** the range of numbers produced by adding 1 to our previous result. The output of Fig. 5.11 confirms that the results are in the range 1 to 6—the actual random values chosen might vary by compiler.

```
1  // Fig. 5.11: fig05_11.c
2  // Shifted, scaled random integers produced by 1 + rand() % 6.
3  #include <stdio.h>
4  #include <stdlib.h>
```

Fig. 5.11 | Shifted, scaled random integers produced by `1 + rand() % 6`. (Part 1 of 2.)

```
5
6  // function main begins program execution
7  int main( void )
8  {
9     unsigned int i; // counter
10
11    // loop 20 times
12    for ( i = 1; i <= 20; ++i ) {
13
14       // pick random number from 1 to 6 and output it
15       printf( "%10d", 1 + ( rand() % 6 ) );
16
17       // if counter is divisible by 5, begin new line of output
18       if ( i % 5 == 0 ) {
19          puts( "" );
20       } // end if
21    } // end for
22 } // end main
```

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

Fig. 5.11 | Shifted, scaled random integers produced by `1 + rand() % 6`. (Part 2 of 2.)

Rolling a Six-Sided Die 6,000,000 Times

To show that these numbers occur approximately with *equal likelihood*, let's simulate 6,000,000 rolls of a die with the program of Fig. 5.12. Each integer from 1 to 6 should appear approximately 1,000,000 times.

```
1  // Fig. 5.12: fig05_12.c
2  // Rolling a six-sided die 6,000,000 times.
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  // function main begins program execution
7  int main( void )
8  {
9     unsigned int frequency1 = 0; // rolled 1 counter
10    unsigned int frequency2 = 0; // rolled 2 counter
11    unsigned int frequency3 = 0; // rolled 3 counter
12    unsigned int frequency4 = 0; // rolled 4 counter
13    unsigned int frequency5 = 0; // rolled 5 counter
14    unsigned int frequency6 = 0; // rolled 6 counter
15
16    unsigned int roll; // roll counter, value 1 to 6000000
17    int face; // represents one roll of the die, value 1 to 6
```

Fig. 5.12 | Rolling a six-sided die 6,000,000 times. (Part 1 of 2.)

```
18
19      // loop 6000000 times and summarize results
20      for ( roll = 1; roll <= 6000000; ++roll ) {
21         face = 1 + rand() % 6; // random number from 1 to 6
22
23         // determine face value and increment appropriate counter
24         switch ( face ) {
25
26            case 1: // rolled 1
27               ++frequency1;
28               break;
29
30            case 2: // rolled 2
31               ++frequency2;
32               break;
33
34            case 3: // rolled 3
35               ++frequency3;
36               break;
37
38            case 4: // rolled 4
39               ++frequency4;
40               break;
41
42            case 5: // rolled 5
43               ++frequency5;
44               break;
45
46            case 6: // rolled 6
47               ++frequency6;
48               break; // optional
49         } // end switch
50      } // end for
51
52      // display results in tabular format
53      printf( "%s%13s\n", "Face", "Frequency" );
54      printf( "   1%13u\n", frequency1 );
55      printf( "   2%13u\n", frequency2 );
56      printf( "   3%13u\n", frequency3 );
57      printf( "   4%13u\n", frequency4 );
58      printf( "   5%13u\n", frequency5 );
59      printf( "   6%13u\n", frequency6 );
60   } // end main
```

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

Fig. 5.12 | Rolling a six-sided die 6,000,000 times. (Part 2 of 2.)

As the program output shows, by scaling and shifting we've used the `rand` function to realistically simulate the rolling of a six-sided die. Note the use of the `%s` conversion specifier to print the character strings `"Face"` and `"Frequency"` as column headers (line 53). After we study arrays in Chapter 6, we'll show how to replace this 26-line `switch` statement elegantly with a single-line statement.

Randomizing the Random Number Generator

Executing the program of Fig. 5.11 again produces

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

Notice that *exactly the same sequence of values* was printed. How can these be *random* numbers? Ironically, this *repeatability* is an important characteristic of function `rand`. When *debugging* a program, this repeatability is essential for proving that corrections to a program work properly.

Function `rand` actually generates **pseudorandom numbers**. Calling `rand` repeatedly produces a sequence of numbers that *appears* to be random. However, the sequence repeats itself each time the program is executed. Once a program has been thoroughly 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 standard library function **`srand`**. Function `srand` takes an `unsigned` integer argument and **seeds** function `rand` to produce a different sequence of random numbers for each execution of the program.

We demonstrate function `srand` in Fig. 5.13. Function `srand` takes an `unsigned int` value as an argument. The conversion specifier `%u` is used to read an `unsigned int` value with `scanf`. The function prototype for `srand` is found in `<stdlib.h>`.

Let's run the program several times and observe the results. Notice that a *different* sequence of random numbers is obtained each time the program is run, provided that a *different* seed is supplied.

To randomize *without* entering a seed each time, use a statement like

```
srand( time( NULL ) );
```

This causes the computer to read its clock to obtain the value for the seed automatically. Function `time` returns the number of seconds that have passed since midnight on January 1, 1970. 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 `<time.h>`. We'll say more about `NULL` in Chapter 7.

```
1  // Fig. 5.13: fig05_13.c
2  // Randomizing the die-rolling program.
3  #include <stdlib.h>
4  #include <stdio.h>
5
```

Fig. 5.13 | Randomizing the die-rolling program. (Part 1 of 2.)

```
6  // function main begins program execution
7  int main( void )
8  {
9     unsigned int i; // counter
10    unsigned int seed; // number used to seed the random number generator
11
12    printf( "%s", "Enter seed: " );
13    scanf( "%u", &seed ); // note %u for unsigned int
14
15    srand( seed );  // seed the random number generator
16
17    // loop 10 times
18    for ( i = 1; i <= 10; ++i ) {
19
20       // pick a random number from 1 to 6 and output it
21       printf( "%10d", 1 + ( rand() % 6 ) );
22
23       // if counter is divisible by 5, begin a new line of output
24       if ( i % 5 == 0 ) {
25          puts( "" );
26       } // end if
27    } // end for
28 } // end main
```

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

```
Enter seed: 867
         2         4         6         1         6
         1         1         3         6         2
```

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

Fig. 5.13 | Randomizing the die-rolling program. (Part 2 of 2.)

Generalized Scaling and Shifting of Random Numbers

The values produced directly by `rand` are always in the range:

```
0 ≤ rand() ≤ RAND_MAX
```

As you know, the following statement simulates rolling a six-sided die:

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

This statement always assigns an integer value (at random) to the variable `face` in the range $1 \leq$ `face` ≤ 6. The *width* of this range (i.e., the number of consecutive integers in the range) is 6 and the *starting number* in the range is 1. Referring to the preceding statement, we see that the width of the range is determined by the number used to *scale* `rand`

with the *remainder operator* (i.e., 6), and the *starting number* of the range is equal to the number (i.e., 1) that's added to `rand % 6`. We can generalize this result as follows:

```
n = a + rand() % b;
```

where `a` is the shifting value (which is equal to the *first* number in the desired range of consecutive integers) and `b` is the *scaling factor* (which is equal to the *width* of the desired range of consecutive integers). In the exercises, we'll see that it's possible to choose integers at random from sets of values other than ranges of consecutive integers.

Common Programming Error 5.7

Using `srand` in place of `rand` to generate random numbers.

5.11 Example: A Game of Chance

One of the most popular games of chance is a dice game known as "craps," which is played in casinos and back alleys throughout the world. The rules of the game are straightforward:

> *A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5, and 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3, or 12 on the first throw (called "craps"), the player loses (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9, or 10 on the first throw, 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.*

Figure 5.14 simulates the game of craps and Fig. 5.15 shows several sample executions.

```
1  // Fig. 5.14: fig05_14.c
2  // Simulating the game of craps.
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h> // contains prototype for function time
6
7  // enumeration constants represent game status
8  enum Status { CONTINUE, WON, LOST };
9
10 int rollDice( void ); // function prototype
11
12 // function main begins program execution
13 int main( void )
14 {
15    int sum; // sum of rolled dice
16    int myPoint; // player must make this point to win
17
18    enum Status gameStatus; // can contain CONTINUE, WON, or LOST
19
20    // randomize random number generator using current time
21    srand( time( NULL ) );
22
```

Fig. 5.14 | Simulating the game of craps. (Part 1 of 3.)

```
23      sum = rollDice(); // first roll of the dice
24
25      // determine game status based on sum of dice
26      switch( sum ) {
27
28         // win on first roll
29         case 7: // 7 is a winner
30         case 11: // 11 is a winner
31            gameStatus = WON; // game has been won
32            break;
33
34         // lose on first roll
35         case 2: // 2 is a loser
36         case 3: // 3 is a loser
37         case 12: // 12 is a loser
38            gameStatus = LOST; // game has been lost
39            break;
40
41         // remember point
42         default:
43            gameStatus = CONTINUE; // player should keep rolling
44            myPoint = sum; // remember the point
45            printf( "Point is %d\n", myPoint );
46            break; // optional
47      } // end switch
48
49      // while game not complete
50      while ( CONTINUE == gameStatus ) { // player should keep rolling
51         sum = rollDice(); // roll dice again
52
53         // determine game status
54         if ( sum == myPoint ) { // win by making point
55            gameStatus = WON; // game over, player won
56         } // end if
57         else {
58            if ( 7 == sum ) { // lose by rolling 7
59               gameStatus = LOST; // game over, player lost
60            } // end if
61         } // end else
62      } // end while
63
64      // display won or lost message
65      if ( WON == gameStatus ) { // did player win?
66         puts( "Player wins" );
67      } // end if
68      else { // player lost
69         puts( "Player loses" );
70      } // end else
71   } // end main
72
73   // roll dice, calculate sum and display results
74   int rollDice( void )
75   {
```

Fig. 5.14 | Simulating the game of craps. (Part 2 of 3.)

```
76      int die1; // first die
77      int die2; // second die
78      int workSum; // sum of dice
79
80      die1 = 1 + ( rand() % 6 ); // pick random die1 value
81      die2 = 1 + ( rand() % 6 ); // pick random die2 value
82      workSum = die1 + die2; // sum die1 and die2
83
84      // display results of this roll
85      printf( "Player rolled %d + %d = %d\n", die1, die2, workSum );
86      return workSum; // return sum of dice
87   } // end function rollRice
```

Fig. 5.14 | Simulating the game of craps. (Part 3 of 3.)

Player wins on the first roll

```
Player rolled 5 + 6 = 11
Player wins
```

Player wins on a subsequent roll

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

Player loses on the first roll

```
Player rolled 1 + 1 = 2
Player loses
```

Player loses on a subsequent roll

```
Player rolled 6 + 4 = 10
Point is 10
Player rolled 3 + 4 = 7
Player loses
```

Fig. 5.15 | Sample runs for the game of craps.

In the rules of the game, notice that the player must roll *two* dice on the first roll, and must do so later on all subsequent rolls. We define a function `rollDice` to roll the dice and compute and print their sum. Function `rollDice` is defined *once*, but it's called from *two* places in the program (lines 23 and 51). Interestingly, `rollDice` takes no arguments, so we've indicated `void` in the parameter list (line 74). Function `rollDice` does return the sum of the two dice, so a return type of `int` is indicated in its function header and in its function prototype.

Enumerations
The game is reasonably involved. The player may win or lose on the first roll, or may win or lose on any subsequent roll. Variable `gameStatus`, defined to be of a new type—`enum Status`—stores the current status. Line 8 creates a programmer-defined type called an **enumeration**. An enumeration, introduced by the keyword **`enum`**, is a set of integer constants represented by identifiers. **Enumeration constants** are sometimes called symbolic constants. Values in an `enum` start with `0` and are incremented by `1`. In line 8, the constant `CONTINUE` has the value `0`, `WON` has the value `1` and `LOST` has the value `2`. It's also possible to assign an integer value to each identifier in an `enum` (see Chapter 10). The *identifiers* in an enumeration must be *unique*, but the *values* may be *duplicated*.

Common Programming Error 5.8
Assigning a value to an enumeration constant after it has been defined is a syntax error.

Good Programming Practice 5.6
Use only uppercase letters in the names of enumeration constants to make these constants stand out in a program and to indicate that enumeration constants are not *variables.*

When the game is won, either on the first roll or on a subsequent roll, `gameStatus` is set to `WON`. When the game is lost, either on the first roll or on a subsequent roll, `gameStatus` is set to `LOST`. Otherwise `gameStatus` is set to `CONTINUE` and the game continues.

Game Ends on First Roll
After the first roll, if the game is over, the `while` statement (lines 50–62) is skipped because `gameStatus` is not `CONTINUE`. The program proceeds to the `if...else` statement at lines 65–70, which prints `"Player wins"` if `gameStatus` is `WON` and `"Player loses"` otherwise.

Game Ends on a Subsequent Roll
After the first roll, if the game is *not* over, then `sum` is saved in `myPoint`. Execution proceeds with the `while` statement because `gameStatus` is `CONTINUE`. Each time through the `while`, `rollDice` is called to produce a new `sum`. If `sum` matches `myPoint`, `gameStatus` is set to `WON` to indicate that the player won, the `while`-test fails, the `if...else` statement prints `"Player wins"` and execution terminates. If `sum` is equal to `7` (line 58), `gameStatus` is set to `LOST` to indicate that the player lost, the `while`-test fails, the `if...else` statement prints `"Player loses"` and execution terminates.

Control Architecture
Note the program's interesting control architecture. We've used two functions—`main` and `rollDice`—and the `switch`, `while`, nested `if...else` and nested `if` statements. In the exercises, we'll investigate various interesting characteristics of the game of craps.

5.12 Storage Classes

In Chapters 2–4, we used identifiers for variable names. The attributes of variables include *name*, *type*, *size* and *value*. In this chapter, we also use identifiers as names for user-defined functions. Actually, each identifier in a program has other attributes, including storage class, storage duration, scope and linkage.

C provides the **storage class specifiers** `auto`, `register`[1], `extern` and `static`.[2] An identifier's **storage class** determines its storage duration, scope and linkage. An identifier's **storage duration** is the period during which the identifier *exists in memory*. Some exist briefly, some are repeatedly created and destroyed, and others exist for the program's entire execution. An identifier's **scope** is *where* the identifier can be referenced in a program. Some can be referenced throughout a program, others from only portions of a program. An identifier's **linkage** determines for a multiple-source-file program whether the identifier is known *only* in the current source file or in *any* source file with proper declarations. This section discusses storage classes and storage duration. Section 5.13 discusses scope. Chapter 14 discusses identifier linkage and programming with multiple source files.

The storage-class specifiers can be split **automatic storage duration** and **static storage duration**. Keyword `auto` is used to declare variables of automatic storage duration. Variables with automatic storage duration are created when the block in which they're defined is entered; they exist while the block is active, and they're destroyed when the block is exited.

Local Variables

Only variables can have automatic storage duration. A function's local variables (those declared in the parameter list or function body) normally have automatic storage duration. Keyword `auto` explicitly declares variables of automatic storage duration. Local variables have automatic storage duration by *default*, so keyword `auto` is rarely used. For the remainder of the text, we'll refer to variables with automatic storage duration simply as **automatic variables**.

Performance Tip 5.1

Automatic storage is a means of conserving memory, because automatic variables exist only *when they're needed. They're created when a function is entered and destroyed when the function is exited.*

Static Storage Class

Keywords `extern` and `static` are used in the declarations of identifiers for variables and functions of *static storage duration*. Identifiers of static storage duration exist from the time at which the program begins execution until the program terminates. For `static` variables, storage is allocated and initialized *only once*, *before the program begins execution*. For functions, the name of the function exists when the program begins execution. However, even though the variables and the function names exist from the start of program execution, this does *not* mean that these identifiers can be accessed throughout the program. Storage duration and scope (*where* a name can be used) are separate issues, as we'll see in Section 5.13.

There are several types of identifiers with static storage duration: *external identifiers* (such as global variables and function names) and local variables declared with the storage-class specifier `static`. Global variables and function names are of storage class `extern` by default. Global variables are created by placing variable declarations *outside* any function definition, and they retain their values throughout the execution of the program. Global variables and functions can be referenced by any function that follows their declarations

1. Keyword `register` is archaic and should not be used.
2. The new C standard adds the storage class specifier `_Thread_local`, which is beyond this book's scope.

or definitions in the file. This is one reason for using function prototypes—when we include `stdio.h` in a program that calls `printf`, the function prototype is placed at the start of our file to make the name `printf` known to the rest of the file.

Software Engineering Observation 5.10

Defining 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. In general,* global variables should be avoided *except in certain situations with unique performance requirements (as discussed in Chapter 14).*

Software Engineering Observation 5.11

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

Local variables declared with the keyword `static` are still known *only* in the function in which they're defined, but unlike automatic variables, `static` local variables *retain* their value when the function is exited. The next time the function is called, the `static` local variable contains the value it had when the function last exited. The following statement declares local variable `count` to be `static` and initializes it to 1.

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

All numeric variables of static storage duration are initialized to zero by default if you do not explicitly initialize them.

Keywords `extern` and `static` have special meaning when explicitly applied to external identifiers. In Chapter 14 we discuss the explicit use of `extern` and `static` with external identifiers and multiple-source-file programs.

5.13 Scope Rules

The scope of an identifier is the portion of the program in which the identifier can be referenced. For example, when we define a local variable in a block, it can be referenced *only* following its definition in that block or in blocks nested within that block. The four identifier scopes are function scope, file scope, block scope, and function-prototype scope.

Labels (identifiers followed by a colon such as `start:`) are the *only* identifiers with function scope. Labels can be used *anywhere* in the function in which they appear, but cannot be referenced outside the function body. Labels are used in `switch` statements (as `case` labels) and in `goto` statements (see Chapter 14). Labels are implementation details that functions hide from one another. This hiding—more formally called information hiding—is a means of implementing the principle of least privilege—a fundamental principle of good software engineering. In the context of an application, the principle states that code should be granted *only* the amount of privilege and access that it needs to accomplish its designated task, but no more.

An identifier declared outside any function has file scope. Such an identifier is "known" (i.e., accessible) in all functions from the point at which the identifier is declared until the end of the file. Global variables, function definitions, and function prototypes placed outside a function all have file scope.

Identifiers defined inside a block have block scope. Block scope ends at the terminating right brace (`}`) of the block. Local variables defined at the beginning of a function

have block scope, as do function parameters, which are considered local variables by the function. *Any block may contain variable definitions.* When blocks are nested, and an identifier in an outer block has the *same* name as an identifier in an inner block, the identifier in the outer block is *hidden* until the inner block terminates. This means that while executing in the inner block, the inner block sees the value of its own local identifier and *not* the value of the identically named identifier in the enclosing block. Local variables declared `static` still have block scope, even though they exist from before program startup. Thus, storage duration does *not* affect the scope of an identifier.

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. If a name is used in the parameter list of a function prototype, the compiler *ignores* the name. Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity.

Common Programming Error 5.9

Accidentally using the same name for an identifier in an inner block as is 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.*

Error-Prevention Tip 5.4

Avoid variable names that hide names in outer scopes.

Figure 5.16 demonstrates scoping issues with global variables, automatic local variables and `static` local variables. A global variable `x` is defined and initialized to 1 (line 9). This global variable is hidden in any block (or function) in which a variable named `x` is defined. In `main`, a local variable `x` is defined and initialized to 5 (line 14). This variable is then printed to show that the global `x` is hidden in `main`. Next, a new block is defined in `main` with another local variable `x` initialized to 7 (line 19). This variable is printed to show that it hides `x` in the outer block of `main`. The variable `x` with value 7 is automatically destroyed when the block is exited, and the local variable `x` in the outer block of `main` is printed again to show that it's no longer hidden.

```
1  // Fig. 5.16: fig05_16.c
2  // Scoping.
3  #include <stdio.h>
4
5  void useLocal( void ); // function prototype
6  void useStaticLocal( void ); // function prototype
7  void useGlobal( void ); // function prototype
8
9  int x = 1; // global variable
10
11 // function main begins program execution
12 int main( void )
13 {
```

Fig. 5.16 | Scoping. (Part 1 of 3.)

```
14    int x = 5; // local variable to main
15
16    printf("local x in outer scope of main is %d\n", x );
17
18    { // start new scope
19       int x = 7; // local variable to new scope
20
21       printf( "local x in inner scope of main is %d\n", x );
22    } // end new scope
23
24    printf( "local x in outer scope of main is %d\n", x );
25
26    useLocal(); // useLocal has automatic local x
27    useStaticLocal(); // useStaticLocal has static local x
28    useGlobal(); // useGlobal uses global x
29    useLocal(); // useLocal reinitializes automatic local x
30    useStaticLocal(); // static local x retains its prior value
31    useGlobal(); // global x also retains its value
32
33    printf( "\nlocal x in main is %d\n", x );
34 } // end main
35
36 // useLocal reinitializes local variable x during each call
37 void useLocal( void )
38 {
39    int x = 25; // initialized each time useLocal is called
40
41    printf( "\nlocal x in useLocal is %d after entering useLocal\n", x );
42    ++x;
43    printf( "local x in useLocal is %d before exiting useLocal\n", x );
44 } // end function useLocal
45
46 // useStaticLocal initializes static local variable x only the first time
47 // the function is called; value of x is saved between calls to this
48 // function
49 void useStaticLocal( void )
50 {
51    // initialized once before program startup
52    static int x = 50;
53
54    printf( "\nlocal static x is %d on entering useStaticLocal\n", x );
55    ++x;
56    printf( "local static x is %d on exiting useStaticLocal\n", x );
57 } // end function useStaticLocal
58
59 // function useGlobal modifies global variable x during each call
60 void useGlobal( void )
61 {
62    printf( "\nglobal x is %d on entering useGlobal\n", x );
63    x *= 10;
64    printf( "global x is %d on exiting useGlobal\n", x );
65 } // end function useGlobal
```

Fig. 5.16 | Scoping. (Part 2 of 3.)

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before 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 in useLocal is 25 after entering useLocal
local x in useLocal is 26 before 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. 5.16 | Scoping. (Part 3 of 3.)

The program defines three functions that each take no arguments and return nothing. Function `useLocal` defines an automatic variable `x` and initializes it to 25 (line 39). When `useLocal` is called, the variable is printed, incremented, and printed again before exiting the function. Each time this function is called, automatic variable `x` is *reinitialized* to 25. Function `useStaticLocal` defines a `static` variable `x` and initializes it to 50 in line 52 (recall that the storage for `static` variables is allocated and initialized *only once*, *before the program begins execution*). Local variables declared as `static` *retain* their values even when they're out of scope. When `useStaticLocal` is called, `x` is printed, incremented, and printed again before exiting the function. In the next call to this function, `static` local variable `x` will contain the value 51. Function `useGlobal` does not define any variables. Therefore, when it refers to variable `x`, the global `x` (line 9) is used. When `useGlobal` is called, the global variable is printed, multiplied by 10, and printed again before exiting the function. The next time function `useGlobal` is called, the global variable still has its modified value, 10. Finally, the program prints the local variable `x` in `main` again (line 33) to show that none of the function calls modified the value of `x` because the functions all referred to variables in other scopes.

5.14 Recursion

The programs we've discussed are generally structured as functions that call one another in a disciplined, hierarchical manner. For some types of problems, it's useful to have functions call themselves. A recursive function is a function that *calls itself* either directly or indirectly through another function. Recursion is a complex topic discussed at length in upper-level computer science courses. In this section and the next, simple examples of re-

cursion are presented. This book contains an extensive treatment of recursion, which is spread throughout Chapters 5–8 and 12 and Appendix F. Figure 5.21, in Section 5.16, summarizes the recursion examples and exercises in the book.

We consider recursion conceptually first, then examine several programs containing recursive functions. Recursive problem-solving approaches have a number of elements in common. A recursive function is called to solve a problem. The function actually knows how to solve only the *simplest* case(s), or so-called **base case(s)**. If the function is called with a base case, the function simply returns a result. If the function is called with a more complex problem, the function divides the problem into two conceptual pieces: a piece that the function knows how to do and a piece that it does not know how to do. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or smaller version. Because this new problem looks like the original problem, the function launches (calls) a fresh copy of itself to go to work on the smaller problem—this is referred to as a **recursive call** or the **recursion step**. The recursion step also includes the keyword `return`, because its result will be combined with the portion of the problem the function knew how to solve to form a result that will be passed back to the original caller.

The recursion step executes while the original call to the function has not yet finished executing. The recursion step can result in many more such recursive calls, as the function keeps dividing each problem it's called with into two conceptual pieces. For the recursion to terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller problems must eventually *converge on the base case*. When the function recognizes the base case, it returns a result to the previous copy of the function, and a sequence of returns ensues all the way up the line until the original call of the function eventually returns the final result to `main`. All of this sounds quite exotic compared to the kind of problem solving we've been using with conventional function calls to this point. It can take a great deal of practice writing recursive programs before the process will appear natural. As an example of these concepts at work, let's write a recursive program to perform a popular mathematical calculation.

Recursively Calculating Factorials

The factorial of a nonnegative integer n, written $n!$ (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 * 4 * 3 * 2 * 1, which is equal to 120.

The factorial of an integer, `number`, greater than or equal to 0 can be calculated *iteratively* (nonrecursively) using a `for` statement as follows:

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

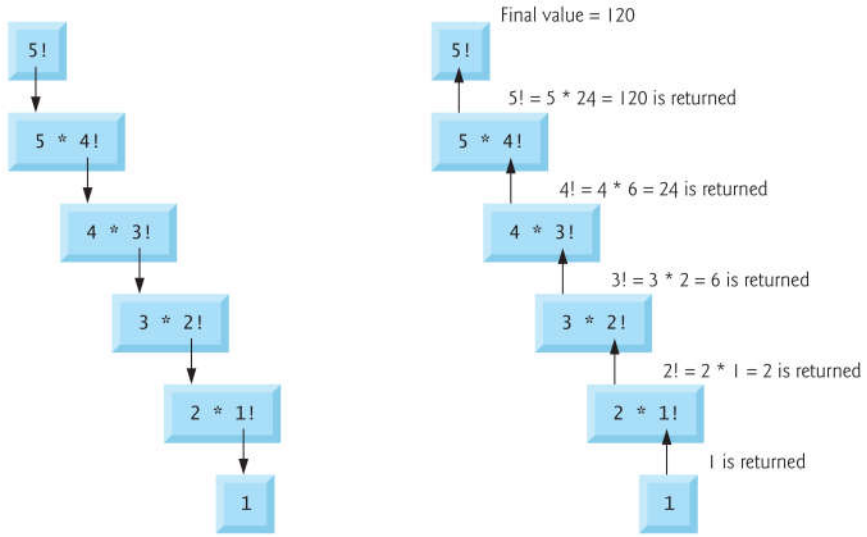
A *recursive* definition of the factorial function is arrived at by observing the following relationship:

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

For example, 5! is clearly equal to 5 * 4! as is shown by the following:

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

The evaluation of 5! would proceed as shown in Fig. 5.17. Figure 5.17(a) shows how the succession of recursive calls proceeds until 1! is evaluated to be 1 (i.e., the *base case*), which terminates the recursion. Figure 5.17(b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.



(a) Sequence of recursive calls (b) Values returned from each recursive call

Fig. 5.17 | Recursive evaluation of 5!.

Figure 5.18 uses recursion to calculate and print the factorials of the integers 0–10 (the choice of the type `unsigned long long int` will be explained momentarily).

```
1  // Fig. 5.18: fig05_18.c
2  // Recursive factorial function.
3  #include <stdio.h>
4
5  unsigned long long int factorial( unsigned int number );
6
7  // function main begins program execution
8  int main( void )
9  {
10    unsigned int i; // counter
11
```

Fig. 5.18 | Recursive factorial function. (Part 1 of 2.)

```
12      // during each iteration, calculate
13      // factorial( i ) and display result
14      for ( i = 0; i <= 21; ++i ) {
15         printf( "%u! = %llu\n", i, factorial( i ) );
16      } // end for
17   } // end main
18
19   // recursive definition of function factorial
20   unsigned long long int factorial( unsigned int number )
21   {
22      // base case
23      if ( number <= 1 ) {
24         return 1;
25      } // end if
26      else { // recursive step
27         return ( number * factorial( number - 1 ) );
28      } // end else
29   } // 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
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 14197454024290336768
```

Fig. 5.18 | Recursive factorial function. (Part 2 of 2.)

The recursive `factorial` function first tests whether a *terminating condition* is true, i.e., whether `number` is less than or equal to 1. If `number` is indeed less than or equal to 1, `factorial` returns 1, no further recursion is necessary, and the program terminates. If `number` is greater than 1, the statement

```
return number * factorial( number - 1 );
```

expresses the problem as the product of `number` and a recursive call to `factorial` evaluating the factorial of `number - 1`. The call `factorial( number - 1 )` is a slightly simpler problem than the original calculation `factorial( number )`.

Function `factorial` (lines 20–29) receives an `unsigned int` and returns a result of type `unsigned long long int`. The C standard specifies that a variable of type `unsigned long long int` can hold a value at least as large as 18,446,744,073,709,551,615. As can be seen in Fig. 5.18, factorial values become large quickly. We've chosen the data type `unsigned long long int` so the program can calculate larger factorial values. The conversion specifier `%llu` is used to print `unsigned long long int` values. Unfortunately, the `factorial` function produces large values so quickly that even `unsigned long long int` does not help us print very many factorial values before the maximum value of a `unsigned long long int` variable is exceeded.

Even when we use `unsigned long long int`, we still can't calculate factorials beyond 21! This points to a weakness in C (and most other procedural programming languages)—namely that the language is not easily *extended* to handle the unique requirements of various applications. As we'll see later in the book, C++ is an *extensible* language that, through "classes," allows us to create new data types, including ones that could hold arbitrarily large integers if we wish.

Common Programming Error 5.10

Forgetting to return a value from a recursive function when one is needed.

Common Programming Error 5.11

Either omitting the base case, or writing the recursion step incorrectly so that it does not converge on the base case, will cause infinite recursion, eventually exhausting memory. This is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution. Infinite recursion can also be caused by providing an unexpected input.

5.15 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 to a constant value of 1.618…. This number, too, repeatedly occurs in nature and has been called the *golden ratio* or the *golden mean*. Humans tend to find the golden mean aesthetically pleasing. Architects often design windows, rooms, and buildings whose length and width are in the ratio of the golden mean. Postcards are often designed with a golden mean length/width ratio.

The Fibonacci series may be defined recursively as follows:

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)
```

Figure 5.19 calculates the n^{th} Fibonacci number recursively using function `fibonacci`. Notice that Fibonacci numbers tend to become large quickly. Therefore, we've chosen the data type `unsigned int` for the parameter type and the data type `unsigned long long int`

for the return type in function `fibonacci`. In Fig. 5.19, each pair of output lines shows a separate run of the program.

```c
// Fig. 5.19: fig05_19.c
// Recursive fibonacci function
#include <stdio.h>

unsigned long long int fibonacci( unsigned int n ); // function prototype

// function main begins program execution
int main( void )
{
   unsigned long long int result; // fibonacci value
   unsigned int number; // number input by user

   // obtain integer from user
   printf( "%s", "Enter an integer: " );
   scanf( "%u", &number );

   // calculate fibonacci value for number input by user
   result = fibonacci( number );

   // display result
   printf( "Fibonacci( %u ) = %llu\n", number, result );
} // end main

// Recursive definition of function fibonacci
unsigned long long int fibonacci( unsigned int n )
{
   // base case
   if ( 0 == n || 1 == n ) {
      return n;
   } // end if
   else { // recursive step
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
   } // end else
} // end function fibonacci
```

```
Enter an integer: 0
Fibonacci( 0 ) = 0
```

```
Enter an integer: 1
Fibonacci( 1 ) = 1
```

```
Enter an integer: 2
Fibonacci( 2 ) = 1
```

```
Enter an integer: 3
Fibonacci( 3 ) = 2
```

Fig. 5.19 | Recursive fibonacci function. (Part 1 of 2.)

```
Enter an integer: 10
Fibonacci( 10 ) = 55
```

```
Enter an integer: 20
Fibonacci( 20 ) = 6765
```

```
Enter an integer: 30
Fibonacci( 30 ) = 832040
```

```
Enter an integer: 40
Fibonacci( 40 ) = 102334155
```

Fig. 5.19 | Recursive fibonacci function. (Part 2 of 2.)

The call to `fibonacci` from `main` is *not* a recursive call (line 18), but all subsequent calls to `fibonacci` are recursive (line 32). Each time `fibonacci` is invoked, it immediately tests for the *base case*—`n` is equal to 0 or 1. If this is true, `n` is returned. Interestingly, if `n` is greater than 1, the recursion step generates *two* recursive calls, each a slightly simpler problem than the original call to `fibonacci`. Figure 5.20 shows how function `fibonacci` would evaluate `fibonacci(3)`.

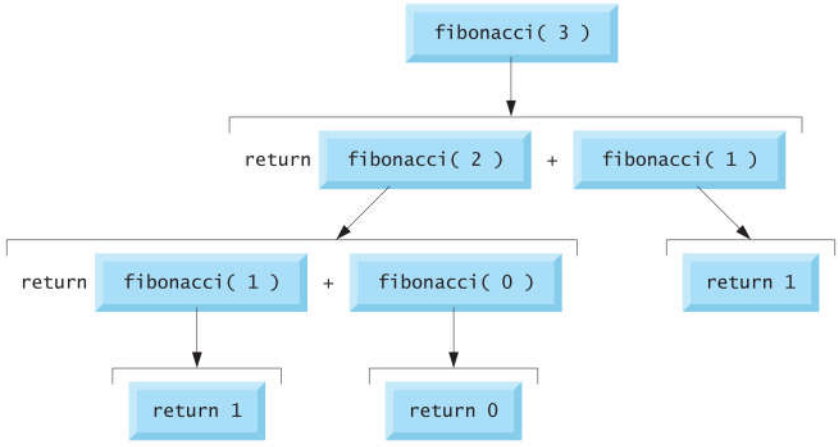


Fig. 5.20 | Set of recursive calls for `fibonacci( 3 )`.

Order of Evaluation of Operands

This figure raises some interesting issues about the *order* in which C compilers will evaluate the operands of operators. This is a different issue from the order in which operators are applied to their operands, namely the order dictated by the rules of operator precedence. Figure 5.20 shows that while evaluating `fibonacci(3)`, *two* recursive calls will be made, namely `fibonacci(2)` and `fibonacci(1)`. But in what order will these calls be made? You might simply assume the operands will be evaluated left to right. For optimization reasons,