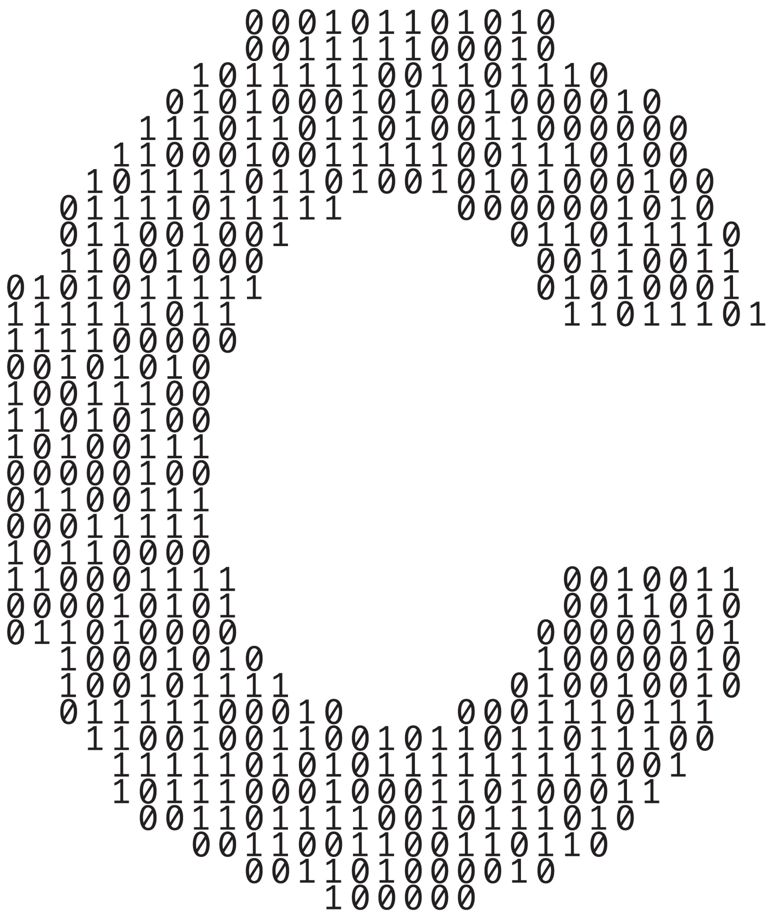
Functions

5



Objectives

In this chapter, you'll:

- Construct programs modularly from small pieces called functions.
- Use common math functions from the C standard library.
- Create new functions.
- Understand how function prototypes help the compiler ensure that you use functions correctly.
- Use the mechanisms that pass information between functions.
- See some commonly used C standard library headers.
- Learn how the function call and return mechanism is supported by the function-call stack and stack frames.
- Build a rock, paper, scissors game using simulation techniques and random-number generation.
- Understand how an identifier's storage class affects its storage duration, scope and linkage.
- Write and use recursive functions, i.e., functions that call themselves.
- Continue our presentation of Secure C programming with a look at secure random-number generation.

Outline

5.1 Introduction

Most computer programs that solve real-world problems are much larger than those presented in the first few chapters. Experience has shown that the best way to develop and maintain a program is to construct it from smaller pieces, each of which is more manageable than the original program. This technique is called divide and conquer. We'll describe some key C features for designing, implementing, operating and maintaining large programs.

5.2 Modularizing Programs in C

In C, you use functions to modularize programs by combining the new functions you write with prepackaged C standard library functions. 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. Prepackaged functions make your job easier because they provide many of the capabilities you need.

The C standard includes the C language and its standard library—standard C compilers implement both.[1] The functions `printf`, `scanf` and `pow` that we've used in previous chapters are from the standard library.

SE

Avoid Reinventing the Wheel

Familiarize yourself with the rich collection of C standard library functions to help reduce program-development time. When possible, use standard functions instead of writing new ones. The C standard library functions are written by experts, well tested

1. Some C standard library portions are designated as optional and are not available in all standard C compilers.

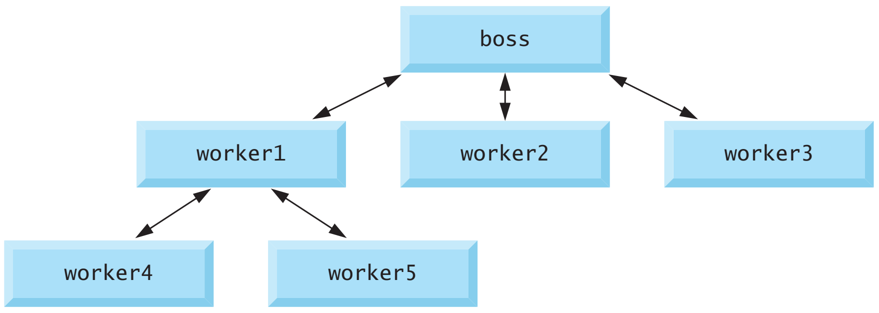
and efficient. Also, using the functions in the C standard library helps make programs more portable. PERF

Defining Functions

You can define functions to perform specific tasks that may be used at many points in a program. The statements defining the function are written once and are hidden from other functions. As we'll see, such hiding is crucial to good software engineering.

Calling and Returning from 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.[2] 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 it's done. For example, a function that displays data on the screen calls the worker function `printf` to perform that task. Function `printf` displays the data and reports back—or **returns**—to the caller when it completes its task. The boss function does not know how the worker function performs its designated task. The worker may call other worker functions, and the boss will be unaware of this. The following diagram shows a boss function hierarchically communicating with several worker functions:



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.

✓ Self Check

1 *(Fill-In)* Programs are typically written by combining new functions you write with prepackaged functions available in the ________.
Answer: C standard library.

2 *(Fill-In)* Functions are invoked by a function ________, which specifies the function name and provides information (as arguments) that the function needs to perform its designated task.
Answer: call.

2. In Chapter 7, Pointers, we'll discuss function pointers. You'll see that you also can call a function through a function pointer, and that you can actually pass functions to other functions.

5.3 Math Library Functions

C's math library functions (header `math.h`) allow you to perform common mathematical calculations. We use many of these functions in this section. To calculate and print the square root of `900.0` you might write

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

When this statement executes, it calls the math library function `sqrt` to calculate the square root of `900.0`, then prints the result as `30.00`. The `sqrt` function takes an argument of type `double` and returns a result of type `double`. In fact, 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. You may store a function call's result in a variable for later use as in

```
double result = sqrt(900.0);
```

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

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

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

The following table summarizes several C math library functions. In the table, the variables `x` and `y` are of type `double`. The C11 standard added complex-number capabilities via the `complex.h` header.

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`[5] `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`

Function	Description	Example
`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*Ẋn radians)	`cos(0.0)` is `1.0`
`tan(x)`	trigonometric tangent of *x* (*x*Ẋn radians)	`tan(0.0)` is `0.0`

✓ Self Check

1 *(Multiple Choice)* Which of the following statements a), b) or c) is *false*?

a) You call a function by writing its name followed by its argument, or a comma-separated list of arguments, in parentheses.

b) The following statement calculates and stores the square root of `900.0`:

```
double result = sqrt(900.0);
```

c) To use the math library functions, you must include the `math.h` header.

d) All of the above statements are *true*.

Answer: d.

2 *(True/False)* Function arguments may be constants, variables or expressions. If `c = 16.0`, `d = 4.0` and `f = 5.0`, then the following statement calculates and prints the square root of `100.00`:

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

Answer: *False*. Actually, it calculates the square root of `36.0` and prints it as `6.00`.

5.4 Functions

Functions allow you to modularize a program. 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.

Functionalizing Programs

There are several motivations for "functionalizing" a program. The **divide-and-conquer** approach makes program development more manageable. Another motivation is building new programs by using existing functions. Such **software reusability** is a key concept in object-oriented programming languages derived from C, such as C++, Java, C# (pronounced "C sharp"), Objective-C and Swift.

With good function naming and definition, you can create programs from standardized functions that accomplish specific tasks, rather than custom 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 it to be executed from other program locations by calling that function.

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

SE software reuse. If you cannot choose a concise name to describe what the function does, it may be performing too many diverse tasks. It's usually best to break such a function into smaller functions—a process called *decomposition*.

Self Check

1 *(Fill-In)* There are several motivations for "functionalizing" a program. The divide-and-conquer approach makes program development more manageable. Another motivation is ________—using existing functions as building blocks to create new programs.
Answer: software reusability.

2 *(True/False)* Each function should perform a rich collection of related tasks, and the function name should describe those tasks.
Answer: *False*. Actually, each function should be limited to performing a single, well-defined task, and the function name should describe that task. This facilitates abstraction and promotes software reuse.

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. Now we consider how to write custom functions.

5.5.1 square Function

Consider a program that uses a function `square` to calculate and print the squares of the integers from 1 to 10 (Fig. 5.1).

```c
// fig05_01.c
// Creating and using a function.
#include <stdio.h>

int square(int number); // function prototype

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

   puts("");
}

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

Fig. 5.1 | Creating and using a function. (Part 1 of 2.)

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

Fig. 5.1 | Creating and using a function. (Part 2 of 2.)

Calling Function square

Function `square` is **invoked** or **called** in `main` within the `printf` statement (line 10):

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

Function `square` receives a copy of the argument `x`'s value in the parameter `number` (line 17). Then `square` calculates `number * number` and passes the result back to line 10 in `main` where `square` was invoked. Line 10 passes `square`'s result to function `printf`, which displays the result on the screen. This process repeats 10 times—once for each iteration of the `for` statement.

square Function Definition

Function `square`'s definition (lines 17–19) shows that it expects an `int` parameter `number`. The keyword `int` preceding the function name (line 17) indicates that `square` returns an integer result. The **`return` statement** in `square` passes the result of `number * number` back to the calling function.

SE Choosing meaningful function names and meaningful parameter names makes programs more readable and helps avoid excessive comments. Programs should be written as collections of small functions. This makes programs easier to write, debug, maintain, modify and reuse.

SE 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's return type, name and parameter list should fit on one line if possible.

Local Variables

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 **parameters** that enable communicating between functions via arguments in function calls. A function's parameters are also local variables of that function.

square Function Prototype

Line 5

```
int square(int number); // 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. Forgetting the semicolon at the end of a function prototype is a syntax error. ERR

The compiler compares `square`'s call (line 10) to its prototype to ensure that:

- the number of arguments is correct,

- the arguments are of the correct types,
- the argument types are in the correct order, and
- the return type is consistent with the context in which the function is called.

The function prototype, first line of the function definition and function calls should all agree in the number, type and order of arguments and parameters. The function prototype and function header must have the same return type, which affects where the function can be called. For example, a function with the return type `void` cannot be used in an assignment statement to store a value or in a call to `printf` to display a value. Function prototypes are discussed in detail in Section 5.6.

Format of a Function Definition

The format of a function definition is

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

The *function-name* is any valid identifier. The ***return-value-type*** is the 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**.

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 parameters, *parameter-list* should contain the keyword `void`. Each parameter must include its type; otherwise, a compilation error occurs. ERR

Placing a semicolon after the *parameter-list*'s right parenthesis in a function definition is an error, as is redefining a parameter as a local variable in a function. ERR 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.

Function Body

The *statements* within braces form the **function body**, which also is a block. Local variables can be declared in any block, and blocks can be nested. Functions cannot be nested—defining a function inside another function is a syntax error. ERR

Returning Control from a Function

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 *expression*'s value to the caller.

main's Return Type

The main function's int return value indicates whether the program executed correctly. In earlier versions of C, we'd explicitly place

```
return 0; // 0 indicates successful program termination
```

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

5.5.2 maximum Function

Let's consider a custom maximum function that returns the largest of three integers (Fig. 5.2). Next, they're passed to maximum (line 17), which determines the largest integer. This value is returned to main by the return statement in maximum (line 32). The printf statement in line 17 then prints the value returned by maximum.

```
 1  // fig05_02.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  int main(void) {
 8     int number1 = 0; // first integer entered by the user
 9     int number2 = 0; // second integer entered by the user
10     int number3 = 0; // third integer entered by the user
11
12     printf("%s", "Enter three integers: ");
13     scanf("%d%d%d", &number1, &number2, &number3);
14
15     // number1, number2 and number3 are arguments
16     // to the maximum function call
17     printf("Maximum is: %d\n", maximum(number1, number2, number3));
18  }
19
20  // Function maximum definition
21  int maximum(int x, int y, int z) {
22     int max = x; // assume x is largest
23
24     if (y > max) { // if y is larger than max,
25        max = y; // assign y to max
26     }
27
28     if (z > max) { // if z is larger than max,
29        max = z; // assign z to max
30     }
31
32     return max; // max is largest value
33  }
```

Fig. 5.2 | Finding the maximum of three integers. (Part 1 of 2.)

```
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. 5.2 | Finding the maximum of three integers. (Part 2 of 2.)

The function initially assumes that its first argument (stored in the parameter `x`) is the largest and assigns it to `max` (line 22). Next, the `if` statement at lines 24–26 determines whether `y` is greater than `max` and, if so, assigns `y` to `max`. Then, the `if` statement at lines 28–30 determines whether `z` is greater than `max` and, if so, assigns `z` to `max`. Finally, line 32 returns `max` to the caller.

✓ Self Check

1 *(Multiple Choice)* The following line of code is a ________.

```
int square(int y);
```

a) Function definition.
b) Function statement.
c) Function prototype.
d) None of the above.

Answer: c.

2 *(Multiple Choice)* Consider the maximum function in Fig. 5.2. Which of the following statements is *false*?

a) The code determines the largest of three integer values.
b) The statement `return max;` sends the result back to the calling function.
c) The code in line 21—`int maximum(int x, int y, int z)`—is commonly called a function header.
d) If `int max = x;` (line 22) were accidentally replaced by `int max = y;` the function would still return the same result.

Answer: d) is *false*. The function would then incorrectly return the larger of the values contained in only parameters `y` and `z`.

5.6 Function Prototypes: A Deeper Look

An important C feature is the function prototype, which was borrowed from C++. The compiler uses function prototypes to validate function calls. Pre-standard 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.

You should include function prototypes for all functions to take advantage of C's type-checking capabilities. Use `#include` preprocessor directives to obtain function prototypes from standard library headers, third-party library headers and headers for functions developed by you or your team members. SE

The function prototype for `maximum` in Fig. 5.2 (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 an `int` result. Notice that the function prototype (omitting the semicolon) is the same as the `maximum` definition's first line. We include parameter names in function prototypes for documentation purposes. The compiler ignores these names, so the following prototype also is valid: SE

```
int maximum(int, int, int);
```

Compilation Errors

A function call that does not match the function prototype is a compilation error. It also is an error if the function prototype and the function definition disagree. For example, in Fig. 5.2, if the function prototype had been written ERR

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

the compiler would generate an error, because the function prototype's `void` return type 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 **argument coercion**, i.e., implicitly converting arguments to the appropriate type. For example, calling the math library function `sqrt` with an integer argument still works even though the function prototype in `<math.h>` specifies a `double` parameter. The following statement correctly evaluates `sqrt(4)` and prints `2.000`:

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

The function prototype causes the compiler to convert a copy of the `int` value `4` to the `double` value `4.0` before passing it to `sqrt`. In general, argument values that do not correspond precisely to the function prototype's parameter types are converted to the proper type before the function is called. Such 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. ERR

In our `sqrt` example, an `int` is automatically converted to a `double` without changing its value—`double` can represent a much wider range of values than `int`. However, a `double` converted to an `int` truncates the `double`'s fractional part, thus changing the original value. Converting large integer types to small integer types (e.g., `long` to `short`) can also change values.

Mixed-Type Expressions

The usual arithmetic conversion rules are handled by the compiler. They apply to **mixed-type expressions**—that is, expressions containing values of multiple data types. In such expressions, the compiler makes temporary copies of values that need to be converted, then converts the *copies* to the "highest" type in the expression—this is known as **promotion**. For mixed-type expressions containing at least one floating-point value:

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

If the mixed-type expression contains only integer types, then the usual arithmetic conversions specify a set of integer promotion rules.

Section 6.3.1 of the C standard document specifies the complete details of arithmetic operands and the usual arithmetic conversion rules. The following table lists the floating-point and integer data types with each type's `printf` and `scanf` conversion specifications. In most cases, the integer types lower in the following table are converted to higher types:

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`

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 are converted to the parameter types specified in a function prototype as if the arguments were being assigned to variables of those types. So, if we pass a `double` to our `square` function in Fig. 5.1, the `double` is converted to `int` (a lower type), and `square` usually returns an incorrect value. For example, `square(4.5)` returns `16`, not `20.25`.

ERR 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.

Function Prototype Notes

If there's no function prototype for a function, the compiler forms one from 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. ERR

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

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

✓ Self Check

1 *(Fill-In)* In a mixed-type expression, the compiler makes a temporary copy of each value that needs to be converted, then converts the copies to the "highest" type in the expression—this is known as ________.
Answer: promotion.

2 *(Multiple Choice)* Consider the following function prototype for a `maximum` function:

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

Which of the following statements is false?

a) It states that `maximum` takes three arguments of type `int` and returns a result of type `int`.
b) Parameter names are required in function prototypes.
c) A function's prototype is often the same as the function's header, except that the header does not end in a semicolon.
d) Forgetting the semicolon at the end of a function prototype is a syntax error.

Answer: b) is *false*. Parameter names in function prototypes are for documentation purposes. The compiler ignores these names, so `int maximum(int, int, int);` is equivalent to the preceding prototype.

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. You usually place a dish at the top—referred to as pushing the dish onto the stack. Similarly, you typically remove a dish 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.

Function-Call Stack

An important mechanism for computing 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. As you'll see in this section, the function-call stack also supports creating, maintaining and destroying each called function's local variables.

Stack Frames

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 its caller. 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. When a called function returns, the stack frame for the function call is popped, and control transfers to the return address specified in the popped stack frame.

Each called function always finds at the *top* of the call stack the information it needs to return to its caller. If a called function calls another function, a stack frame for the new function call is 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 local variables, which must 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 local variables need to "go away." The called function's stack frame is a perfect place to reserve the memory for local variables. That stack frame exists only as long as the called function is active. When that function returns—and no longer needs its local variables—its stack frame is popped from the stack. Those local variables are no longer known to the program.

Stack Overflow

Of course, the amount of memory in a computer is finite, so only limited 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**[3] 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 8–12 of Fig. 5.3).

3. This is how the website `stackoverflow.com` got its name—a popular website for getting answers to your programming questions.

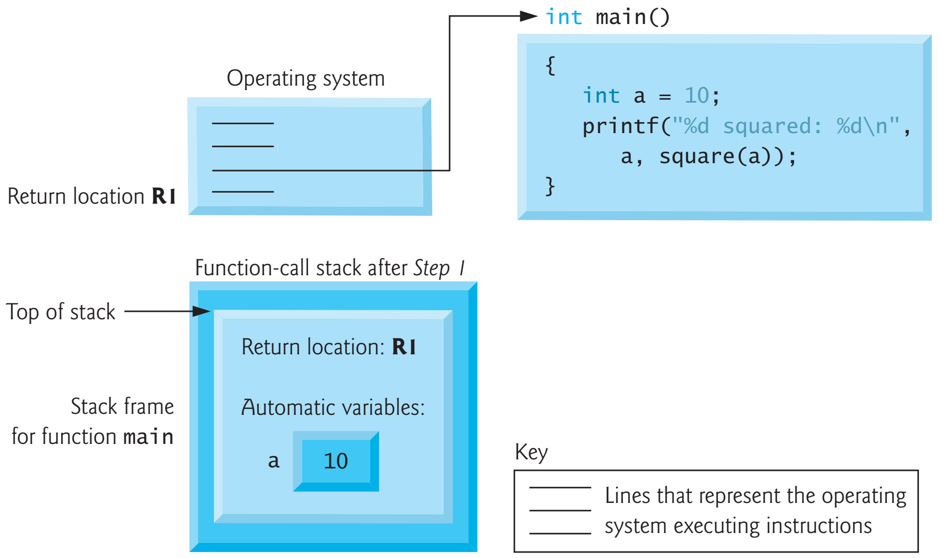
```
 1  // fig05_03.c
 2  // Demonstrating the function-call stack
 3  // and stack frames using a function square.
 4  #include <stdio.h>
 5
 6  int square(int x); // prototype for function square
 7
 8  int main() {
 9     int a = 10; // value to square (local variable in main)
10
11     printf("%d squared: %d\n", a, square(a)); // display a squared
12  }
13
14  // returns the square of an integer
15  int square(int x) { // x is a local variable
16     return x * x; // calculate square and return result
17  }
```

```
10 squared: 100
```

Fig. 5.3 | Demonstrating the function-call stack and stack frames using a function `square`.

Step 1: Operating System Invokes `main` to Execute Application

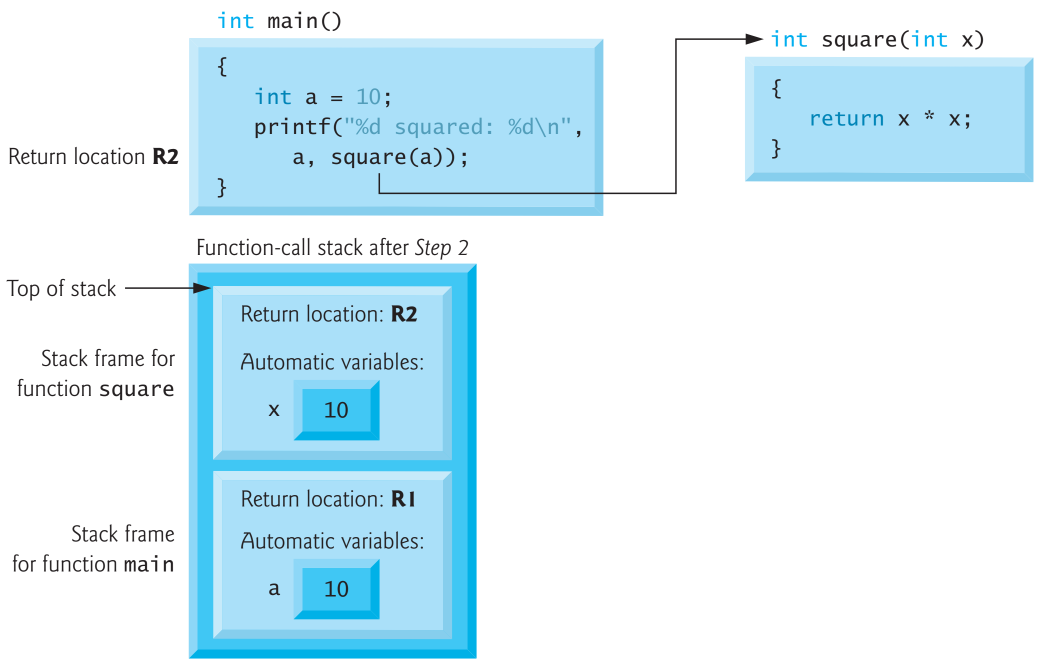
First, the operating system calls `main`—this pushes a stack frame onto the stack (as shown in the following diagram). The stack frame tells `main` how to return to the operating system (that is, transfer to return address `R1`) and contains the space for `main`'s local variable `a`, which is initialized to `10`.


Operating system
Return location R1
Function-call stack after Step 1
Top of stack
Stack frame for function main
Return location: R1
Automatic variables:
a
10
Key
Lines that represent the operating system executing instructions

```
int main()
{
   int a = 10;
   printf("%d squared: %d\n",
      a, square(a));
}
```

Step 2: `main` Invokes Function `square` to Perform Calculation

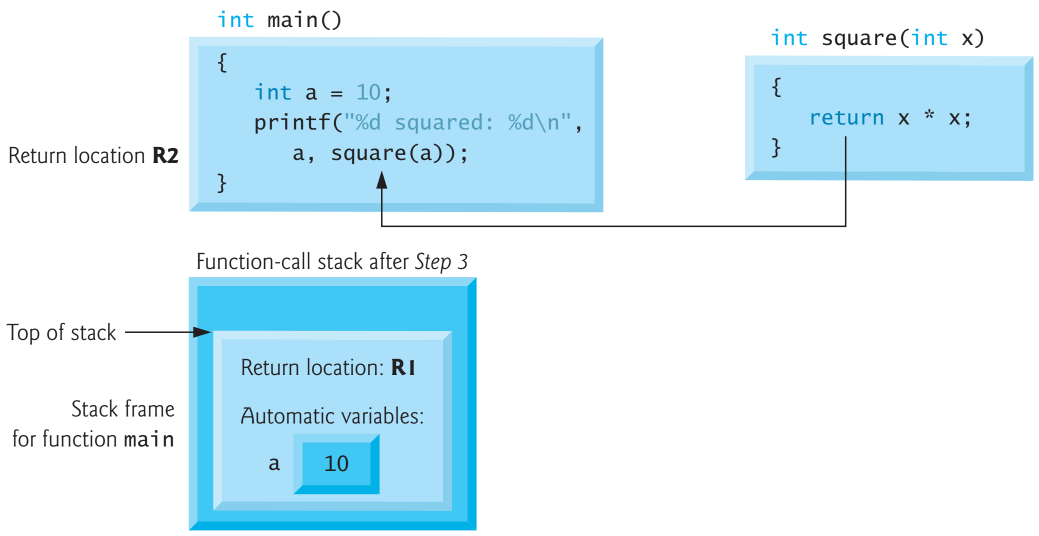
Function `main`—before returning to the operating system—now calls function `square` in line 11 of Fig. 5.3. This causes a stack frame for `square` (lines 15–17) to be pushed onto the function-call stack, as shown in the following diagram:



This stack frame contains the return address that `square` needs to return to `main` (i.e., `R2`) and the memory for `square`'s local variable (i.e., `x`).

Step 3: `square` Returns Its Result to `main`

After `square` calculates the square of its argument, it needs to return to `main`—and no longer needs the memory for its local variable `x`. So the stack is popped—giving `square` the return location in `main` (i.e., `R2`) and losing `square`'s local variable. The following diagram shows the function-call stack *after* `square`'s stack frame has been popped:



Function `main` now displays the result of calling `square` (line 11 in Fig. 5.3). Reaching `main`'s closing right brace pops its stack frame from the stack. This gives `main` the address it needs to return to the operating system (i.e., `R1` in the preceding diagram). At this point, the memory for `main`'s local variable (i.e., `a`) is unavailable.

Flaw in Our Discussion

There is a flaw in the preceding discussion and diagrams. We showed `main` calling out to `square` and `square` returning to `main`, but, of course, `printf` is a function too. As you study the code in Fig. 5.3, you might be inclined to say that `main` calls `printf`, then `printf` calls `square`. However, `printf`'s argument values must be known in full before `printf` can be called. So execution proceeds as follows:

1. The operating system calls `main`, so `main`'s stack frame is pushed onto the stack.
2. `main` calls `square`, so `square`'s stack frame is pushed onto the stack.
3. `square` calculates and returns to `main` a value for use in `printf`'s argument list, so `square`'s stack frame is popped from the stack.
4. `main` calls `printf`, so `printf`'s stack frame is pushed onto the stack.
5. `printf` displays its arguments, then returns to `main`, so `printf`'s stack frame is popped from the stack.
6. `main` terminates, so `main`'s stack frame is popped from the stack.

Data Structures

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 and trees in Chapter 12.

✓ Self Check

1 *(Fill-In)* Each time a function calls another function, an entry is pushed onto the stack. This entry, called a ________, contains the return address that the called function needs in order to return to the calling function.
Answer: stack frame.

2 *(True/False)* A called function's stack frame is a perfect place to reserve the memory for local variables. That stack frame exists only as long as the called function is active. When the function returns—and no longer needs its local variables—its stack frame is popped from the stack. At that point, those local variables are no longer known to the program.
Answer: *True.*

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. The following table alphabetically lists several standard library headers that may be included in programs. The C standard includes additional headers. The term "macros"—used several times in this table—is discussed in detail in Chapter 14.

Header	Explanation
Headers we use or discuss in this book:	
`<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.
`<float.h>`	Contains the floating-point size limits of the system.
`<limits.h>`	Contains the integral size limits of the system.
`<math.h>`	Contains function prototypes for math library functions.
`<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.
`<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.
Other headers:	
`<errno.h>`	Defines macros that are useful for reporting error conditions.
`<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 locale notion enables the computer system to handle different conventions for expressing data such as dates, times, currency amounts and large numbers throughout the world.
`<setjmp.h>`	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.
`<stddef.h>`	Contains common type definitions used by C.

You can create custom headers. A programmer-defined header can be included by using the `#include` preprocessor directive. For example, if the prototype for our square function was located 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 14.2 presents additional information on including headers, such as why programmer-defined headers are enclosed in quotes (`""`) rather than angle brackets (`<>`).

✓ Self Check

1 *(Fill-In)* The _______ header contains function prototypes for string-processing functions.
Answer: `<string.h>`.

2 *(Fill-In)* The ________ header contains information for adding diagnostics that aid program debugging.
Answer: `<assert.h>`.

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 an argument is passed by value, a copy of the argument's value is made and passed to the 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 accidental side effects (variable modifications) that can 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. In **Chapter 7, Pointers**, we'll show how to achieve pass-by-reference. 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.

✓ Self Check

1 *(True/False)* When an argument is passed by value, a copy of the argument's value is made and passed to the function. Changes to the copy also are applied to the original variable's value in the caller.
Answer: *False*. With pass-by-value, changes to the copy do not affect an original variable's value in the caller.

2 *(True/False)* Pass-by-reference should be used only with *trusted* called functions that need to modify the original variable.
Answer: *True*.

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 custom functions. The program uses functions and several 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`[4] from the `<stdlib.h>` header.

4. C standard library function `rand` is known to be "predictable," which can create security breach opportunities. Each of our preferred platforms offers a non-standard secure random-number generator. We'll mention these in Section 5.17, Secure C Programming—Secure Random-Number Generation.

Obtaining a Random Integer Value

Consider the following statement:

```
int value = rand();
```

The `rand` function generates an integer between 0 and `RAND_MAX` (a symbolic constant defined in the `<stdlib.h>` header). The C standard states that `RAND_MAX`'s value must be at least 32,767, 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 32,767, and on GNU `gcc` and Xcode Clang with a maximum `RAND_MAX` value of 2,147,483,647. 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 (Fig. 5.4) to simulate 10 rolls of a six-sided die and print each roll's value.

```
 1  // fig05_04.c
 2  // Shifted, scaled random integers produced by 1 + rand() % 6.
 3  #include <stdio.h>
 4  #include <stdlib.h>
 5
 6  int main(void) {
 7
 8     for (int i = 1; i <= 10; ++i) {
 9        printf("%d  ", 1 + (rand() % 6)); // display random die value
10     }
11
12     puts("");
13  }
```

```
6  6  5  5  6  5  1  1  5  3
```

Fig. 5.4 | Shifted, scaled random integers produced by `1 + rand() % 6`.

The `rand` function's prototype is in `<stdlib.h>`. In line 9, 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 confirms that the results are in the range 1 to 6—the order in which these random values are chosen might vary by compiler.

Rolling a Six-Sided Die 60,000,000 Times

To show that these numbers occur approximately with *equal likelihood*, let's simulate 60,000,000 rolls of a die with the program of Fig. 5.5. Each integer from 1 to 6 should appear approximately 10,000,000 times.

```c
// fig05_05.c
// Rolling a six-sided die 60,000,000 times.
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int frequency1 = 0; // rolled 1 counter
   int frequency2 = 0; // rolled 2 counter
   int frequency3 = 0; // rolled 3 counter
   int frequency4 = 0; // rolled 4 counter
   int frequency5 = 0; // rolled 5 counter
   int frequency6 = 0; // rolled 6 counter

   // loop 60000000 times and summarize results
   for (int roll = 1; roll <= 60000000; ++roll) {
      int face = 1 + rand() % 6; // random number from 1 to 6

      // determine face value and increment appropriate counter
      switch (face) {
         case 1: // rolled 1
            ++frequency1;
            break;
         case 2: // rolled 2
            ++frequency2;
            break;
         case 3: // rolled 3
            ++frequency3;
            break;
         case 4: // rolled 4
            ++frequency4;
            break;
         case 5: // rolled 5
            ++frequency5;
            break;
         case 6: // rolled 6
            ++frequency6;
            break; // optional
      }
   }

   // display results in tabular format
   printf("%s%13s\n", "Face", "Frequency");
   printf("   1%13d\n", frequency1);
   printf("   2%13d\n", frequency2);
   printf("   3%13d\n", frequency3);
   printf("   4%13d\n", frequency4);
```

Fig. 5.5 | Rolling a six-sided die 60,000,000 times. (Part 1 of 2.)

```
47      printf("    5%13d\n", frequency5);
48      printf("    6%13d\n", frequency6);
49   }
```

```
Face    Frequency
   1      9999294
   2     10002929
   3      9995360
   4     10000409
   5     10005206
   6      9996802
```

Fig. 5.5 | Rolling a six-sided die 60,000,000 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 specification to print the character strings `"Face"` and `"Frequency"` as column headers (line 42). After we study arrays in Chapter 6, we'll show how to replace this 20-line `switch` statement elegantly with a single-line statement.

Randomizing the Random-Number Generator

Executing the program of Fig. 5.4 again produces

```
6   6   5   5   6   5   1   1   5   3
```

This is the exact sequence of values we showed in Fig. 5.4. 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 `int` argument and **seeds** function `rand` to produce a different sequence of random numbers for each program execution.

We demonstrate function `srand` in Fig. 5.6. The function prototype for `srand` is found in `<stdlib.h>`.

```
// fig05_06.c
// Randomizing the die-rolling program.
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   printf("%s", "Enter seed: ");
```

Fig. 5.6 | Randomizing the die-rolling program. (Part 1 of 2.)

```
    int seed = 0; // number used to seed the random-number generator
    scanf("%d", &seed);

    srand(seed); // seed the random-number generator

    for (int i = 1; i <= 10; ++i) {
       printf("%d  ", 1 + (rand() % 6)); // display random die value
    }

    puts("");
}
```

```
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. 5.6 | Randomizing the die-rolling program. (Part 2 of 2.)

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. The first and last outputs use the same seed value, so they show the same results.

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 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.

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:

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

This statement always assigns an integer value (at random) to the variable `face` in the range 1 ≤ `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:

```
int 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, you'll choose integers at random from sets of values other than ranges of consecutive integers.

✓ Self Check

1 *(Fill-In)* ________ is an important characteristic of function `rand`. When we're debugging a program, this characteristic is essential for proving that corrections to a program work properly.
Answer: Repeatability.

2 *(True/False)* 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.
Answer: *True.*

3 *(Fill-In)* 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 ________.
Answer: `srand`.

5.11 Game Simulation Case Study: Rock, Paper, Scissors

In this section, we simulate a popular hand game; rock, paper, scissors. The game is played between two people. The rules of the game are straightforward:

> *Each player simultaneously forms one of three shapes with an outstretched hand. The shapes are "rock", "paper", and "scissors". A simultaneous, zero-sum game, it has three possible outcomes: a draw, a win, or a loss. "rock" breaks (wins) "scissors", "scissors" cuts (wins) "paper", and "paper" covers (wins) "rock". If both players choose the same shape, the game is a tie.*

Figure 5.7 simulates the game and shows several sample executions.

```c
// fig05_07.c
// Simulating the game of rock, paper, scissors
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h> // contains prototype for function time

enum GameStatus {CONTINUE, GAME_WON, GAME_LOST}; // constants represent game
// status
enum RoundStatus {DRAW, WON, LOST}; // constants represent round status
enum Shape {ROCK, PAPER, SCISSORS}; // constants representing possible shapes

enum Shape computerPlayRandomShape(void); // playRandomShape function
// prototype;
enum RoundStatus getRoundStatus(enum Shape, enum Shape);
enum Shape convertIntToShape(int);
char* getStringFromShape(enum Shape);

int main(void) {
   srand(time(NULL)); // randomize based on current time
   enum GameStatus gameStatus = CONTINUE; // may be CONTINUE, GAME_WON, or
   // GAME_LOST
   int bestOfN = 3; // best of 3 game
   int winThreshold = (bestOfN / 2) + 1;

   int playerWinCount = 0;
   int computerWinCount = 0;

   int roundNum = 0;

   // while game not complete
   while (CONTINUE == gameStatus) { // should keep playing
      printf("==================\nRound %d\n==================\n", roundNum++);
      printf("Enter 0 (ROCK), or 1 (PAPER), or 2 (SCISSORS): ");

      int playerShapeInt = -1;
      scanf(" %d", &playerShapeInt);

      enum Shape playerShape = convertIntToShape(playerShapeInt); // player
      // shape
      enum Shape computerShape = computerPlayRandomShape(); // computer plays
      // random shape

      enum RoundStatus roundStatus = getRoundStatus(playerShape, computerShape);
      // get winner for current round
      switch (roundStatus) {
         case DRAW:
            printf("It's a DRAW!\n");
            break;
```

Fig. 5.7 | Simulating the game of rock, paper, scissors. (Part 1 of 4.)

```
         case WON:
            printf("Player WON!\n");
            if (++playerWinCount == winThreshold) {
               // Increment win count, if win count reaches threshold,
               // player wins the game
               gameStatus = GAME_WON;
            }
            break;
         case LOST:
            printf("Player LOST!\n");
            if (++computerWinCount == winThreshold) {
               // Increment win count, if win count reaches threshold,
               // computer wins the game
               gameStatus = GAME_LOST;
            }
            break;
      }
   }

   switch (gameStatus) {
      case GAME_WON:
         printf("**Best of %d game is WON by player!**\n", bestOfN);
         break;
      case GAME_LOST:
         printf("**Best of %d game is WON by computer!**\n", bestOfN);
         break;
      case CONTINUE:
      default:
         printf("**Error! Should not reach here!**");
   }
}

// play a random shape and display result
enum Shape computerPlayRandomShape(void) {
   enum Shape randomShape = convertIntToShape(rand() % 3);
   printf("Computer played %s\n", getStringFromShape(randomShape));
   return randomShape;
}

enum RoundStatus getRoundStatus(enum Shape playerShape, enum Shape
computerShape) {
   if (playerShape == computerShape) {
      return DRAW;
   } else if ((playerShape == ROCK && computerShape == SCISSORS)
      || (playerShape == SCISSORS && computerShape == PAPER)
      || (playerShape == PAPER && computerShape == ROCK)) {
      return WON;
   } else {
      return LOST;
   }
}
```

Fig. 5.7 | Simulating the game of rock, paper, scissors. (Part 2 of 4.)

```
enum Shape convertIntToShape(int shapeValue) {
   return (enum Shape) shapeValue;
}

char * getStringFromShape(enum Shape shape) {
   switch (shape) {
      case ROCK:
         return "ROCK";
      case PAPER:
         return "PAPER";
      case SCISSORS:
         return "SCISSORS";
   }
   return "";
}
```

Player wins after a few rounds:

```
==================
Round 0
==================
Enter 0 (ROCK), or 1 (PAPER), or 2 (SCISSORS): 0
Computer played SCISSORS
player WON!
==================
Round 1
==================
Enter 0 (ROCK), or 1 (PAPER), or 2 (SCISSORS): 1
Computer played PAPER
It's a DRAW!
==================
Round 2
==================
Enter 0 (ROCK), or 1 (PAPER), or 2 (SCISSORS): 0
Computer played ROCK
It's a DRAW!
==================
Round 3
==================
Enter 0 (ROCK), or 1 (PAPER), or 2 (SCISSORS): 2
Computer played ROCK
player LOST!
==================
Round 4
==================
Enter 0 (ROCK), or 1 (PAPER), or 2 (SCISSORS): 2
Computer played PAPER
player WON!
**Best of 3 game is WON by player!**
```

Fig. 5.7 | Simulating the game of rock, paper, scissors. (Part 3 of 4.)

Player wins after winning 2 straight rounds:

```
==================
Round 0
==================
Enter 0 (ROCK), or 1 (PAPER), or 2 (SCISSORS): 0
Computer played SCISSORS
player WON!
==================
Round 1
==================
Enter 0 (ROCK), or 1 (PAPER), or 2 (SCISSORS): 1
Computer played ROCK
player WON!
**Best of 3 game is WON by player!**
```

Player loses after losing 2 straight rounds:

```
==================
Round 0
==================
Enter 0 (ROCK), or 1 (PAPER), or 2 (SCISSORS): 0
Computer played PAPER
player LOST!
==================
Round 1
==================
Enter 0 (ROCK), or 1 (PAPER), or 2 (SCISSORS): 2
Computer played ROCK
player LOST!
**Best of 3 game is WON by computer!**
```

Fig. 5.7 | Simulating the game of rock, paper, scissors. (Part 4 of 4.)

In Fig. 5.7, we have implemented a best of 3 game, that is, whoever is first to win a total of 2 rounds, wins. We define a few prototypes (lines 12–15). The function `computerPlayRandomShape(void)` does not take any arguments, and randomly picks and returns a shape for the computer to play. `getRoundStatus(Shape, Shape)` accepts two shapes (player's and computer's, respectively) and returns whether the player wins, loses, or it's a draw. `convertIntToShape(int)` is a helper method to convert integer value of the shape into its enumeration constants. Enumerations will be described in detail in the next section. `getStringFromShape(Shape)` simply returns the string representation of the shape (for display purposes).

Enumerations

The game is reasonably involved. The player may win or lose at various rounds. Variable `gameStatus`, defined to be of a new type—`enum GameStatus`—stores the current game status. Line 8 creates a new type called an **enumeration**. An enumeration, introduced by the keyword **enum**, is a set of integer constants represented by identifiers. **Enumeration constants** help make programs more readable and easier to maintain. Values in an `enum` start with 0 and are incremented by 1. In line 8, the constant `CONTINUE` has the value 0, `GAME_WON` has the value 1 and `GAME_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. Use only uppercase letters in `enum` constant names to make them stand out in a program and to indicate they are not variables.

When the game is won, `gameStatus` is set to `GAME_WON`. When the game is lost, `gameStatus` is set to `GAME_LOST`. Otherwise, `gameStatus` is set to `CONTINUE`, and the game continues.

The concept above is also used for the other two `enum`s, representing the `Shape (ROCK, PAPER, SCISSORS)` and the `RoundStatus (DRAW, WON, LOST)`.

Round Logic

In each round, we prompt the player to enter 0, 1, or 2 which represents playing "rock," "paper," and "scissors," respectively (lines 31–34). Once the player enters a shape, we randomly pick a shape for the computer to play (line 37) and obtain the winner of the round (line 39).

If the player wins the round, we increment the player winning counter (line 46). If the player loses the round, we increment the computer winning counter (line 53). If it's a draw, do nothing, proceed to the next round.

Game Ends

The game ends when a player or computer wins 2 rounds (in a best of 3 game). This number can be easily adjusted to be best of 5 or any other number by modifying line 20.

Control Architecture

Note the program's control architecture. We've used multiple functions—`main`, `computerPlayRandomShape`, `getRoundStatus`, `convertIntToShape`, `getStringFromShape`—and the `switch`, `while` and nested `if...else` statements.

Related Exercises

This case study is supported by the following exercises:

- Exercise 5.47 (Game Modification).
- Exercise 6.20 (Game Statistics).

✓ Self Check

1 *(Fill-In)* An enumeration, introduced by the keyword ________, is a set of integer constants represented by identifiers.
Answer: `enum`.

2 *(Fill-In)* In the following statement

```
enum Shape {ROCK, PAPER, SCISSORS};
```

The values of `ROCK`, `PAPER`, and `SCISSORS` are ________, ________, and ________.
Answer: `0`, `1` and `2`.

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`,[5] `extern` and **`static`**.[6] A **storage class** determines an identifier's storage duration, scope and linkage. **Storage duration** is the period during which an identifier exists in memory. Some exist briefly, some are repeatedly created and destroyed, and others exist for the entire program execution. **Scope** determines where a program can reference an identifier. Some can be referenced throughout a program, others from only portions of a program. For a multiple-source-file program, an identifier's **linkage** determines 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, and Section 5.13 discusses scope. Chapter 15 discusses identifier linkage and programming with multiple source files.

Local Variables and Automatic Storage Duration

The storage-class specifiers are split between **automatic storage duration** and **static storage duration**. The **`auto`** keyword declares that a variable has automatic storage duration. Such variables are created when program control enters the block in which they're defined. They exist while the block is active, and they're destroyed when program control exits the block.

Only variables can have automatic storage duration. A function's local variables—those declared in the parameter list or function body—have automatic storage duration by default, so the `auto` keyword is rarely used. Automatic storage duration is a means of conserving memory because local variables exist only when they're needed. We'll refer to variables with automatic storage duration simply as local variables.

PERF

Static Storage Class

Keywords `extern` and `static` declare identifiers for variables and functions with *static storage duration*. Identifiers of static storage duration exist from the time at which the program begins execution until it 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 these names exist from the start of program execution, they are not always

5. Keyword `register` is archaic and should not be used.
6. C11 added the storage-class specifier `_Thread_local`, which is beyond this book's scope.

accessible. 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 have storage class `extern` by default. Global variables are created by placing variable declarations outside any function definition. They retain their values throughout program execution. Global variables and functions can be referenced by any function that follows their declarations 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.

ERR 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, you should avoid global variables except in situations PERF with unique performance requirements (as discussed in Chapter 15). Variables used only in a particular function should be defined as local variables in that function.

Local `static` variables are still known only in the function in which they're defined and retain their value when the function returns. 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. Chapter 15 discusses the explicit use of `extern` and `static` with external identifiers and multiple-source-file programs.

✓ Self Check

1 *(Fill-In)* Each identifier in a program has attributes, including storage class, storage duration, ________ and ________.
Answer: scope, linkage.

2 *(Multiple Choice)* Which of the following statements a), b) or c) is *false*?
   a) An identifier's storage duration is the period during which the identifier exists in memory.
   b) An identifier's scope is where the identifier can be referenced in a program.
   c) Keyword `auto` declares variables of automatic storage duration. Such variables are created when program control enters the block in which they're defined. They exist while the block is active, and they're destroyed when program control exits the block.
   d) All of the above statements are *true*.
Answer: d.

5.13 Scope Rules

The scope of an identifier is the portion of the program in which the identifier can be referenced. For example, a local variable in a block 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.

Function Scope

Labels are identifiers followed by a colon such as `start:`. Labels are the *only* identifiers with function scope. Labels can be used anywhere in the function in which they appear, but they cannot be referenced outside the function body. Labels are used in `switch` statements (as `case` labels) and in `goto` statements (see Chapter 15). Labels are hidden in the function in which they're defined. This 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.

File Scope

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.

Block 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 outer block's identifier has the same name as an inner block's identifier, the outer block's identifier is hidden until the inner block terminates. While executing in the inner block, the inner block sees its local identifier's value, *not* the value of the enclosing block's identically named identifier. For this reason, you generally should avoid variable names that hide names in outer scopes. 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.

Function-Prototype Scope

The only identifiers with function-prototype scope are those used in the parameter list of a function prototype. As mentioned previously, function prototypes do not require names in the parameter list—only types are required. If a name is used in the parameter list of a function prototype, the compiler ignores it. Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity.

Scoping Example

Figure 5.8 demonstrates scoping issues with global variables, 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 12). 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 17). 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  // fig05_08.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  int main(void) {
12     int x = 5; // local variable to main
13
14     printf("local x in outer scope of main is %d\n", x);
15
16     { // start new scope
17        int x = 7; // local variable to new scope
18
19        printf("local x in inner scope of main is %d\n", x);
20     } // end new scope
21
22     printf("local x in outer scope of main is %d\n", x);
23
24     useLocal(); // useLocal has automatic local x
25     useStaticLocal(); // useStaticLocal has static local x
26     useGlobal(); // useGlobal uses global x
27     useLocal(); // useLocal reinitializes automatic local x
28     useStaticLocal(); // static local x retains its prior value
29     useGlobal(); // global x also retains its value
30
31     printf("\nlocal x in main is %d\n", x);
32  }
33
34  // useLocal reinitializes local variable x during each call
35  void useLocal(void) {
36     int x = 25; // initialized each time useLocal is called
37
38     printf("\nlocal x in useLocal is %d after entering useLocal\n", x);
39     ++x;
```

Fig. 5.8 | Scoping. (Part 1 of 2.)

```
40     printf("local x in useLocal is %d before exiting useLocal\n", x);
41  }
42
43  // useStaticLocal initializes static local variable x only the first time
44  // the function is called; value of x is saved between calls to this
45  // function
46  void useStaticLocal(void) {
47     static int x = 50; // initialized once
48
49     printf("\nlocal static x is %d on entering useStaticLocal\n", x);
50     ++x;
51     printf("local static x is %d on exiting useStaticLocal\n", x);
52  }
53
54  // function useGlobal modifies global variable x during each call
55  void useGlobal(void) {
56     printf("\nglobal x is %d on entering useGlobal\n", x);
57     x *= 10;
58     printf("global x is %d on exiting useGlobal\n", x);
59  }
```

```
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. 5.8 | Scoping. (Part 2 of 2.)

The program defines three functions that each take no arguments and return nothing. Function `useLocal` defines a local variable `x` and initializes it to 25 (line 36). When function `useLocal` is called, the variable is printed, incremented, and printed again before exiting the function. Each time this function is called, the local variable `x` is reinitialized to 25.

Function `useStaticLocal` defines a `static` variable `x` and initializes it to 50 in line 47 (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, the `static` local variable `x` will contain the previously incremented value 51.

Function `useGlobal` does not define any variables, so 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 31) to show that none of the function calls modified `x`'s value because the functions all referred to variables in other scopes.

✓ Self Check

1 *(Fill-In)* The ________ of an identifier is the portion of the program in which the identifier can be referenced. For example, a local variable in a block can be referenced only following its definition in that block or in blocks nested within that block.
Answer: scope.

2 *(True/False)* Any block may contain variable definitions. When blocks are nested and an outer block's identifier has the same name as an inner block's identifier, the inner block's identifier is hidden until the outer block terminates.
Answer: *False*. Actually, when blocks are nested and an outer block's identifier has the same name as an inner block's identifier, the outer block's identifier is hidden until the inner block terminates.

5.14 Recursion

For some types of problems, it's actually useful to have functions call themselves. A recursive function is one 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, we present simple recursion examples. We present an extensive treatment of recursion, which is spread throughout Chapters 5–8, 12 and 13. The table in Section 5.16 summarizes the book's recursion examples and exercises.

Base Cases and Recursive Calls

We consider recursion conceptually first, then examine several programs containing recursive functions. Recursive problem-solving approaches have several 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, it simply returns a result. When called with a more complex problem, the function typically divides the problem into two conceptual pieces:

- one that the function knows how to do, and
- one 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 work on the smaller problem—this is referred to as a **recursive call** or the **recursion step**. The recursion step also includes a `return` statement, 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 is paused, waiting for the result from the recursion step. The recursion step can result in many more such recursive calls, as the function keeps dividing each problem with which it's called 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 its caller. 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:

```
unsigned long long int factorial = 1;

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

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

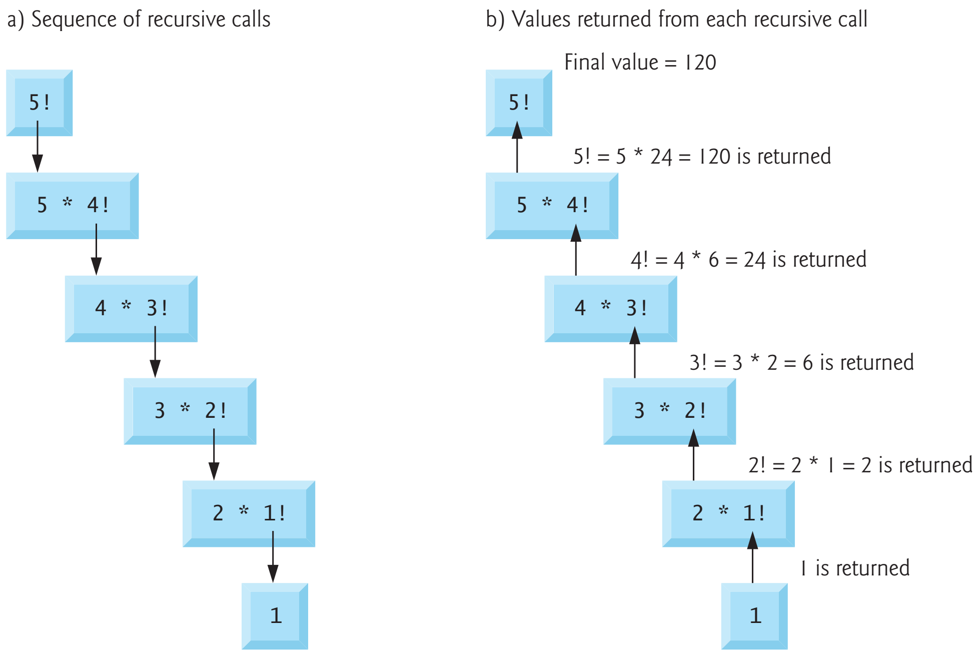
$$n! = n \cdot (n-1)!$$

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

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

Evaluating 5! Recursively

The evaluation of 5! would proceed as shown in the following diagram. Part (a) of the following diagram shows how the succession of recursive calls proceeds until 1! is evaluated to be 1 (i.e., the *base case*), terminating the recursion. Part (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
5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
b) Values returned from each recursive call
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 is returned
1

Implementing Recursive Factorial Calculations

Figure 5.9 uses recursion to calculate and print the factorials of the integers 0–21 (the choice of the type `unsigned long long int` will be explained momentarily).

```c
// fig05_09.c
// Recursive factorial function.
#include <stdio.h>

unsigned long long int factorial(int number);

int main(void) {
   // calculate factorial(i) and display result
   for (int i = 0; i <= 21; ++i) {
      printf("%d! = %llu\n", i, factorial(i));
   }
}

// recursive definition of function factorial
unsigned long long int factorial(int number) {
   if (number <= 1) { // base case
      return 1;
   }
   else { // recursive step
      return (number * factorial(number - 1));
   }
}
```

Fig. 5.9 | Recursive factorial function. (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
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 14197454024290336768
```

Fig. 5.9 | Recursive factorial function. (Part 2 of 2.)

Function `factorial`

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)`.

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, though infinite loops do not typically exhaust memory. ERR

Factorials Become Large Quickly

Function `factorial` (lines 15–22) receives an `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. 5.9, 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 specification `%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 many factorial values, because that type's maximum value is quickly exceeded.

Integer Types Have Limitations

Even when we use `unsigned long long int`, we still can't calculate factorials beyond 21! This points to a weakness in procedural programming languages like C—the language is not easily extended to handle the unique requirements of various applications. Object-oriented languages like C++ are extensible. Through a language feature called classes, programmers can create new data types, even ones that could hold arbitrarily large integers.

✓ Self Check

1 *(Fill-In)* Omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case will cause ________, eventually exhausting memory.
Answer: infinite recursion.

2 *(Fill-In)* The following code should iteratively calculate the factorial of an integer, `number`, but the code contains a bug:

```
unsigned long long int factorial = 1;

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

You can correct the bug by changing ________ to ________.
Answer: *, *=.

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:

$$\begin{aligned}\text{fibonacci}(0) &= 0\\ \text{fibonacci}(1) &= 1\\ \text{fibonacci}(n) &= \text{fibonacci}(n-1) + \text{fibonacci}(n-2)\end{aligned}$$

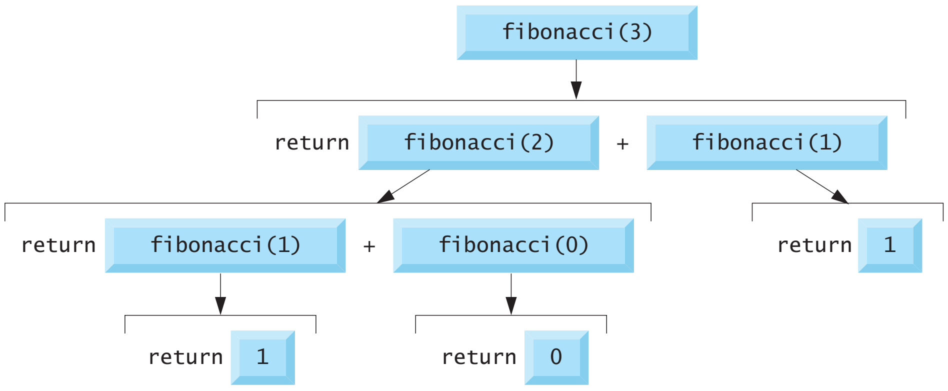
Figure 5.10 calculates the *n*th Fibonacci number recursively using function `fibonacci`. Fibonacci numbers tend to become large quickly. So, we've chosen the data type `unsigned long long int` for the return type in function `fibonacci`.

```
 1  // fig05_10.c
 2  // Recursive fibonacci function.
 3  #include <stdio.h>
 4
 5  unsigned long long int fibonacci(int n); // function prototype
 6
 7  int main(void) {
 8     // calculate and display fibonacci(number) for 0-10
 9     for (int number = 0; number <= 10; number++) {
10        printf("Fibonacci(%d) = %llu\n", number, fibonacci(number));
11     }
12
13     printf("Fibonacci(20) = %llu\n", fibonacci(20));
14     printf("Fibonacci(30) = %llu\n", fibonacci(30));
15     printf("Fibonacci(40) = %llu\n", fibonacci(40));
16  }
17
18  // Recursive definition of function fibonacci
19  unsigned long long int fibonacci(int n) {
20     if (0 == n || 1 == n) { // base case
21        return n;
22     }
23     else { // recursive step
24        return fibonacci(n - 1) + fibonacci(n - 2);
25     }
26  }
```

```
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(40) = 102334155
```

Fig. 5.10 | Recursive fibonacci function.

The `fibonacci` calls from `main` are *not* recursive (lines 10 and 13–15), but all subsequent calls to `fibonacci` are recursive (line 24). 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`. The following diagram shows how function `fibonacci` would evaluate `fibonacci(3)`:


fibonacci(3)
return fibonacci(2) + fibonacci(1)
return fibonacci(1) + fibonacci(0)
return 1
return 1
return 0

Order of Evaluation of Operands

This figure raises some interesting issues about the *order* in which C compilers will evaluate operators' operands. This is a different issue from the order in which operators are applied to their operands—that is, the order dictated by the rules of operator precedence and grouping. The preceding diagram 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, C does *not* specify the order in which the operands of most operators (including +) are to be evaluated. Therefore, you should make no assumption about the order in which these calls will execute. The calls could execute `fibonacci(2)` first and then `fibonacci(1)`, or the calls could execute in the reverse order, `fibonacci(1)` then `fibonacci(2)`. In this and most other programs, the final result would be the same. But in some programs, the evaluation of an operand may have *side effects* that could affect the final result of the expression.

Operators for which Operand Evaluation Order Is Specified

C specifies the operand evaluation order of only four operators—`&&`, `||`, the comma (`,`) operator and `?:`. The first three are binary operators whose operands are guaranteed to be evaluated *left-to-right*. [*Note:* The commas used to separate the arguments in a function call are *not* comma operators.] The last operator is C's only *ternary* operator. Its leftmost operand is always evaluated first. If the leftmost operand evaluates to nonzero (true), the middle operand is evaluated next, and the last operand is ignored. If the leftmost operand evaluates to zero (false), the third operand is evaluated next, and the middle operand is ignored.

Exponential Complexity

A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers. Each level of recursion in the `fibonacci` function has a doubling effect on the number of calls. The number of recursive calls that execute to calculate the *n*th Fibonacci number is "on the order of 2^n." This rapidly gets out of hand. Calculating only the 20th Fibonacci number would require on the order of 2^{20} or about a million calls, calculating the 30th Fibonacci number would require on the order of 2^{30} or about a billion calls, and so on. Computer scientists refer to this as exponential complexity. Problems of this nature can humble even the world's most powerful computers! Complexity issues in general, and exponential complexity in particular, are discussed in detail in the upper-level computer-science course generally called "Algorithms."

The example we showed in this section used an intuitively appealing solution to calculate Fibonacci numbers, but there are better approaches. Exercise 5.48 asks you to investigate recursion in more depth and propose alternate approaches to implementing the recursive Fibonacci algorithm.

✓ Self Check

1 *(True/False)* For optimization reasons, C specifies the order in which the operands of most operators (including `+`) are to be evaluated.
Answer: *False*. For optimization reasons, C does *not* specify the order in which the operands of most operators (including `+`) are to be evaluated. C specifies the order of evaluation of the operands of *only four* operators—`&&`, `||`, the comma (`,`) operator and `?:`.

2 *(Multiple Choice)* Consider the code in Fig. 5.10, which implements a recursive `fibonacci` function. Which of the following statements a), b) or c) is *false*?

a) All `fibonacci` calls in Fig. 5.10 are recursive calls.
b) 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.
c) If `n` is greater than 1, the recursion step generates *two* recursive calls, each a slightly simpler problem than the original call to `fibonacci`.
d) All of the above statements are *true*.

Answer: a) is *false*. Actually, the calls to `fibonacci` from `main` are not recursive calls, but all subsequent calls to `fibonacci` are recursive (line 24).

5.16 Recursion vs. Iteration

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

Common Features of Iteration and Recursion

- Both iteration and recursion are based on a *control statement*: Iteration uses an *iteration statement*; recursion uses a *selection statement*.
- Both iteration and recursion involve *repetition*: Iteration uses an *iteration statement*; recursion achieves repetition through *repeated function calls*.
- Iteration and recursion each have a *termination test*: Iteration terminates when the *loop-continuation condition fails*; recursion when a *base case is recognized*.
- Counter-controlled iteration and recursion both *gradually approach termination*: Iteration keeps modifying a counter until the counter assumes a value

that makes the *loop-continuation condition fail*; recursion keeps producing simpler versions of the original problem until the *base case is reached*.

- Both iteration and recursion can occur *infinitely*: An *infinite loop* occurs with iteration if the loop-continuation test *never* becomes false; *infinite recursion* occurs if the recursion step does *not* reduce the problem each time in a manner that *converges on the base case*. Infinite iteration and recursion typically occur as a result of errors in a program's logic.

Negatives of Recursion

Recursion has many negatives. It *repeatedly* invokes the mechanism—and consequently the *overhead*—of function calls. This can be expensive in both processor time and memory space. Each recursive call causes *another copy* of the function (actually only the function's variables) to be created; this can *consume considerable memory*. Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted. So why choose recursion?

Recursion Is Not Required

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

Recursion Examples and Exercises Throughout This Book

Most programming textbooks introduce recursion much later than we've done here. We feel that recursion is such a sufficiently rich and complex topic that it's better to introduce it earlier and spread the examples over the remainder of the text. The following table summarizes by chapter the recursion examples and exercises in the text.

Recursion examples and exercises

Chapter 5
Factorial function
Fibonacci function
Greatest common divisor
Multiply two integers
Raising an integer to an integer power
Towers of Hanoi
Recursive `main`
Visualizing recursion

Chapter 6
Sum the elements of an array
Print an array
Print an array backward
Print a string backward
Check whether a string is a palindrome
Minimum value in an array
Linear search
Binary search
Eight Queens

Chapter 7
Maze traversal

Chapter 8
Printing a string input at the keyboard backward

Chapter 12
Search a linked list
Print a linked list backward
Binary tree insert
Preorder traversal of a binary tree
Inorder traversal of a binary tree
Postorder traversal of a binary tree
Printing trees

Chapter 13
Selection sort
Quicksort

Closing Observations

Let's close this discussion with some observations that we make repeatedly throughout the book. Good software engineering is important, and high performance is important. So, we've included extensive software-engineering and performance tips throughout the book. Unfortunately, these goals are often at odds with one another. Good software engineering is key to making more manageable the task of developing the larger and more complex software systems we need. High performance is key to realizing the systems of the future that will place ever greater computing demands on hardware. Where do functions fit in here?

Software Engineering

SE

Dividing a large program into functions promotes good software engineering. But it has a price. A heavily functionalized program—compared to a monolithic (i.e., one-piece) program without functions—makes potentially large numbers of function calls. These consume execution time on a computer's processor(s). Although monolithic programs may perform better, they're more difficult to program, test, debug, maintain and evolve.

Performance

PERF

Today's hardware architectures are tuned to make function calls efficient. C compilers help optimize your code, and today's hardware processors and multicore architecture are incredibly fast. For the vast majority of applications and software systems you'll build, concentrating on good software engineering will be more important than high-performance programming. Nevertheless, in many applications and systems, such as game programming, real-time systems, operating systems and embedded systems, performance is crucial, so we include performance tips throughout the book.

✓ Self Check

1 *(True/False)* Dividing a large program into functions promotes good software engineering. But a heavily functionalized program—compared to a monolithic (i.e., one-piece) program without functions—makes a potentially large number of function calls. These consume execution time on a computer's processor(s). Although monolithic programs may perform better, they're more difficult to program, test, debug, maintain and evolve.
Answer: *True.*

2 *(Multiple Choice)* Which of the following statements is *false*?

a) Recursion repeatedly invokes the mechanism, and consequently the overhead, of function calls. This can be expensive in both processor time and memory space.

b) Each recursive call causes another copy of the function's statements and variables to be created; this can consume considerable memory.

c) Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted.
d) A recursive approach is chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug.

Answer: b) is *false*. Actually, each recursive call causes another copy of *only the function's variables* to be created.

SEC 5.17 Secure C Programming—Secure Random-Number Generation

In Section 5.10, we introduced the `rand` function for generating pseudorandom numbers. This function is sufficient for textbook examples but is not meant for use in industrial-strength applications. According to the C standard document's description of function `rand`, "There are no guarantees as to the quality of the random sequence produced and some implementations are known to produce sequences with *distressingly* non-random low-order bits." The CERT guideline MSC30-C indicates that implementation-specific random-number generation functions must be used to ensure that the random numbers produced are *not predictable*—this is extremely important, for example, in cryptography and other security applications.

The guideline presents several platform-specific random-number generators that are considered to be secure. For more information, see guideline MSC30-C at `https://wiki.sei.cmu.edu/`. If you're building industrial-strength applications that require random numbers, you should investigate for your platform the recommended function(s) to use. For example:

- Microsoft Windows provides the `BCryptGenRandom` function, which is part of Microsoft's "Cryptography API: Next Generation:"

  `https://docs.microsoft.com/en-us/windows/win32/seccng/cng-portal`

- POSIX-based systems (such as Linux) provide a `random` function, which you can learn more about by executing the following command in a Terminal or shell:

  ```
  man random
  ```

- MacOS's `stdlib.h` header provides the `arc4random` function, which you can learn more about by executing the following command in a macOS Terminal:

  ```
  man arc4random
  ```

✓ Self Check

1 *(True/False)* The CERT guideline MSC30-C indicates that implementation-specific random-number generation functions must be used to ensure that the random numbers produced are predictable—this is extremely important, for example, in cryptography and other security applications.

Answer: *False*. Actually, the CERT guideline MSC30-C indicates that implementation-specific random-number generation functions must be used to ensure that the random numbers produced are not predictable.

Summary

Section 5.1 Introduction

- The best way to develop and maintain a large program is to **divide** (p. 232) it into several smaller pieces, each more manageable than the original program.

Section 5.2 Modularizing Programs in C

- A **function** (p. 232) is invoked by a **function call** (p. 233), which specifies the function name and provides information (as arguments) that the function needs to perform its task.

Section 5.3 Math Library Functions

- A function is invoked by writing its name followed by a left parenthesis, the argument (or a comma-separated list of arguments) and a right parenthesis.
- Each argument may be a constant, a variable or an expression.

Section 5.4 Functions

- There are several motivations for "functionalizing" a program. The divide-and-conquer approach makes program development more manageable. Another is building new programs by using existing functions. Such software reusability is a key concept in object-oriented programming languages derived from C, such as C++, Java, C# (pronounced "C sharp"), Objective-C and Swift.
- With good function naming and definition, you can create programs from standardized functions that accomplish specific tasks, rather than custom 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 it to be executed from other program locations by calling that function.

Section 5.5 Function Definitions

- The arguments passed to a function should match in number, type and order with the **parameters** (p. 237) in the function definition.
- When a program encounters a function call, control transfers from the point of invocation to the called function, the statements of that function execute, then control returns to the caller.
- A **called function** can **return control** to the caller in one of three ways. If the function does not return a value, control is returned when the function-ending right brace is reached, or by executing the statement

  ```
  return;
  ```

  If the function does return a value, the statement

  `return` *expression*;

  returns the value of *expression*.
- A **local variable** (p. 237) is known only in a function definition. Other functions are not allowed to know the names of a function's local variables, nor is any function allowed to know the implementation details of any other function.

- A **function prototype** (p. 237) declares the function's name, its return type and the number, types and order of the parameters the function expects to receive.
- The general **format for a function definition** is

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

  If a function does not return a value, the *return-value-type* is declared as `void`. The *function-name* is any valid identifier. The *parameter-list* (p. 238) is a comma-separated list containing the definitions of the variables that will be passed to the function. If a function does not receive any values, *parameter-list* is declared as `void`.

Section 5.6 Function Prototypes: A Deeper Look

- Function prototypes enable the compiler to verify that functions are called correctly.
- The compiler ignores variable names mentioned in the function prototype.
- The C standard's **usual arithmetic-conversion rules** (p. 241) arguments in a **mixed-type expression** (p. 242) are converted to the same type.

Section 5.7 Function-Call Stack and Stack Frames

- **Stacks** (p. 243) are known as **last-in, first-out** (**LIFO**; p. 243) data structures—the last item pushed (inserted) onto the stack is the first item popped (removed) from the stack.
- A called function must know how to return to its caller, so the return address of the calling function is pushed onto the **program execution stack** (p. 243) when the function is called. If a series of function calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out order so that the last function to execute will be the first to return to its caller.
- The program execution stack contains the **memory for the local variables** used in each function invocation during a program's execution. This data is known as the **stack frame** (p. 243) of the function call. When a function call is made, the stack frame for that function call is pushed onto the program execution stack. When the function returns to its caller, the stack frame is popped off the stack and those local variables are no longer known to the program.
- If there are more function calls than can have their stack frames stored on the program execution stack, an error known as a **stack overflow** occurs.

Section 5.8 Headers

- Each standard library has a corresponding **header** (p. 247) containing the function prototypes for that library's functions.
- You can create and include your own headers.

Section 5.9 Passing Arguments by Value and by Reference

- When an argument is **passed by value** (p. 249), a copy is made and passed to the called function. Changes to the copy do not affect the original variable's value in the caller.
- When an argument is **passed by reference** (p. 249), the caller allows the called function to modify the original variable's value.
- All calls in C are pass-by-value by default.

Section 5.10 Random-Number Generation

- **Function `rand`** generates an integer between 0 and `RAND_MAX` which is defined by the C standard to be at least 32767.

- Values produced by rand can be **scaled** and **shifted** to produce values in a specific range (p. 250).
- To **randomize** a program, use the C standard library function srand.
- The **srand function** seeds (p. 252) the random-number generator. An srand call is ordinarily inserted in a program only after it has been thoroughly debugged. This ensures repeatability, which is essential to proving that corrections to a random-number generation program work properly.
- The function prototypes for rand and srand are contained in <stdlib.h>.
- To randomize without the need for entering a seed each time, we use srand(time(NULL)).
- The general equation for scaling and shifting a random number is

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

  where a is the shifting value (i.e., the first number in the desired range of consecutive integers) and b is the scaling factor (i.e., the width of the desired range of consecutive integers).

Section 5.11 Example: A Game of Chance; Introducing enum

- An **enumeration** (p. 260), introduced by the keyword **enum**, is a set of integer constants. Values in an enum start with 0 and are incremented by 1. You also can assign an integer to each identifier in an enum. The identifiers in an enumeration must be unique, but the values may be duplicated.

Section 5.12 Storage Classes

- Each identifier in a program has the attributes **storage class**, **storage duration**, **scope** and **linkage** (p. 260).
- C provides four **storage classes** indicated by the **storage class specifiers**: **auto**, **register**, **extern** and **static** (p. 260).
- An identifier's **storage duration** is when that identifier exists in memory.
- An identifier's **linkage** (p. 260) determines for a multiple-source-file program whether an identifier is known only in the current source file or in any source file with proper declarations.
- A function's local variables have **automatic storage duration** (p. 260)—they're created when program control enters the block in which they're defined, exist while the block is active and are destroyed when program control exits the block.
- Keywords **extern** and **static** declare identifiers for variables and functions of static storage duration. **Static storage duration** (p. 260) variables are allocated and initialized once, before the program begins execution.
- There are two 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** are created by placing variable definitions outside any function definition. Global variables retain their values throughout the program execution.
- **Local static variables** retain their value between calls to the function in which they're defined.
- All numeric variables of static storage duration are initialized to zero by default.

Section 5.13 Scope Rules

- An identifier's **scope** (p. 262) is where the identifier can be referenced in a program.

- The purpose of **information hiding** is to give functions access only to the information they need to complete their tasks. This is a means of implementing the **principle of least privilege**.
- An identifier can have **function scope**, **file scope**, **block scope** or **function-prototype scope** (p. 262).
- **Labels** 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.
- An identifier declared outside any function has file scope. Such an identifier is "known" in all functions from the point at which it's declared until the end of the file.
- Identifiers defined inside a block have block scope. Block scope ends at the terminating right brace (`}`) of the block.
- Local variables have block scope, as do function parameters, which are local variables.
- 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.
- The only identifiers with function-prototype scope are those used in the parameter list of a function prototype.

Section 5.14 Recursion

- A **recursive function** (p. 265) is a function that calls itself either directly or indirectly.
- If a recursive function is called with a **base case** (p. 265), the function simply returns a result. If it's called with a more complex problem, it divides the problem into two conceptual pieces: a piece that the function knows how to do and a slightly smaller version of the original problem. Because this new problem looks like the original, the function launches a recursive call to work on the smaller problem.
- For recursion to terminate, each time the recursive function calls itself with a slightly simpler version of the original problem, the sequence of smaller and smaller problems must converge on the **base case**. When the function recognizes the base case, the result is returned to the previous function call, and a sequence of returns ensues all the way up the line until the original call of the function eventually returns the final result.
- Standard C does not specify the order in which the operands of most operators (including `+`) are to be evaluated. Of C's many operators, the standard specifies the order of evaluation of the operands of only the operators `&&`, `||`, the comma (`,`) operator and `?:`. The first three of these are binary operators whose two operands are evaluated left-to-right. The last operator is C's only ternary operator. Its leftmost operand is evaluated first; if it evaluates to nonzero, the middle operand is evaluated next and the last operand is ignored; if the leftmost operand evaluates to zero, the third operand is evaluated next and the middle operand is ignored.

Section 5.16 Recursion vs. Iteration

- Both iteration and recursion are based on a control structure: Iteration uses an iteration statement; recursion uses a selection statement.
- Both iteration and recursion involve repetition: Iteration uses an iteration statement; recursion achieves repetition through repeated function calls.
- Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.

- Iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on the base case.
- Recursion repeatedly invokes the mechanism, and consequently the overhead, of function calls. This can be expensive in both processor time and memory space.

Self-Review Exercises

5.1 Answer each of the following:

a) ________ are used to modularize programs.
b) A function is invoked with a(n) ________.
c) A variable known only within the function in which it's defined is called a(n) ________.
d) The ________ statement is used to pass an expression's value back to a calling function.
e) Keyword ________ is used in a function header to indicate that a function does not return a value or to indicate that a function contains no parameters.
f) The ________ of an identifier is the portion of the program in which the identifier can be used.
g) The three ways to return control from a called function to a caller are ________, ________ and ________.
h) A(n) ________ allows the compiler to check the number, types, and order of the arguments passed to a function.
i) The ________ function is used to produce random numbers.
j) The ________ function is used to set the random number seed to randomize a program.
k) The storage-class specifiers are ________, ________, ________ and ________.
l) Variables declared in a block or in the parameter list of a function have storage class ________, unless specified otherwise.
m) A non-`static` variable defined outside any block or function is a(n) ________ variable.
n) For a local variable in a function to retain its value between calls to the function, it must be declared with the ________ storage-class specifier.
o) The four identifier scopes are ________, ________, ________ and ________.
p) A function that calls itself either directly or indirectly is a(n) ________ function.
q) A recursive function typically has two components: one that provides a means for the recursion to terminate by testing for a(n) ________ case, and one that expresses the problem as a recursive call for a slightly simpler problem than the original call.

5.2 Consider the following program:

```c
#include <stdio.h>
int cube(int y);

int main(void) {
   for (int x = 1; x <= 10; ++x) {
      printf("%d\n", cube(x));
   }
}

int cube(int y) {
   return y * y * y;
}
```

State the scope (function scope, file scope, block scope or function-prototype scope) of each of the following elements:

a) The variable `x` in `main`.
b) The variable `y` in `cube`.
c) The function `cube`.
d) The function `main`.
e) The function prototype for `cube`.
f) The identifier `y` in the function prototype for `cube`.

5.3 Write a program that tests whether the examples of the math library function calls shown in the table of Section 5.3 actually produce the indicated results.

5.4 Give the function header for each of the following functions:

a) Function `hypotenuse` that takes two `double` arguments, `side1` and `side2`, and returns a `double` result.
b) Function `smallest` that takes three integers, `x`, `y`, `z`, and returns an integer.
c) Function `instructions` that does not receive any arguments and does not return a value.
d) Function `intToFloat` that takes an integer argument, `number`, and returns a `float`.

5.5 Give the function prototype for each of the following:

a) The function described in Exercise 5.4(a).
b) The function described in Exercise 5.4(b).
c) The function described in Exercise 5.4(c).
d) The function described in Exercise 5.4(d).

5.6 Write a declaration for floating-point variable `lastValue` that's to retain its value between calls to the function in which it's defined.

5.7 Find the error in each of the following program segments and explain how the error can be corrected (see also Exercise 5.46):

a)
```
int g(void) {
   printf("%s", "Inside function g\n");
   int h(void) {
      printf("%s", "Inside function h\n");
   }
}
```

b)
```
int sum(int x, int y) {
    int result = x + y;
}
```
c)
```
void f(float a); {
    float a;
    printf("%f", a);
}
```
d)
```
int sum(int n) {
    if (0 == n) {
       return 0;
    }
    else {
       n + sum(n - 1);
    }
}
```
e)
```
void product(void) {
    printf("%s", "Enter three integers: ")
    int a;
    int b;
    int c;
    scanf("%d%d%d", &a, &b, &c);
    int result = a * b * c;
    printf("Result is %d", result);
    return result;
}
```

Answers to Self-Review Exercises

5.1 a) functions. b) function call. c) local variable. d) `return`. e) `void`. f) scope. g) `return;` or `return expression;` or encountering the closing right brace of a function. h) function prototype. i) `rand`. j) `srand`. k) `auto`, `register`, `extern`, `static`. l) `auto`. m) external, global. n) `static`. o) function scope, file scope, block scope, function-prototype scope. p) recursive. q) base.

5.2 a) Block scope. b) Block scope. c) File scope. d) File scope. e) File scope. f) Function-prototype scope.

5.3 See below. [*Note:* On most Linux systems, you must use the `-lm` option when compiling this program.]

```
// ex05_03.c
// Testing the math library functions
#include <stdio.h>
#include <math.h>

```

```
int main(void) {
   // calculates and outputs the square root
   printf("sqrt(%.1f) = %.1f\n", 900.0, sqrt(900.0));
   printf("sqrt(%.1f) = %.1f\n", 9.0, sqrt(9.0));

   // calculates and outputs the cube root
   printf("cbrt(%.1f) = %.1f\n", 27.0, cbrt(27.0));
   printf("cbrt(%.1f) = %.1f\n", -8.0, cbrt(-8.0));

   // calculates and outputs the exponential function e to the x
   printf("exp(%.1f) = %f\n", 1.0, exp(1.0));
   printf("exp(%.1f) = %f\n", 2.0, exp(2.0));

   // calculates and outputs the logarithm (base e)
   printf("log(%f) = %.1f\n", 2.718282, log(2.718282));
   printf("log(%f) = %.1f\n", 7.389056, log(7.389056));

   // calculates and outputs the logarithm (base 10)
   printf("log10(%.1f) = %.1f\n", 1.0, log10(1.0));
   printf("log10(%.1f) = %.1f\n", 10.0, log10(10.0));
   printf("log10(%.1f) = %.1f\n", 100.0, log10(100.0));

   // calculates and outputs the absolute value
   printf("fabs(%.1f) = %.1f\n", 13.5, fabs(13.5));
   printf("fabs(%.1f) = %.1f\n", 0.0, fabs(0.0));
   printf("fabs(%.1f) = %.1f\n", -13.5, fabs(-13.5));

   // calculates and outputs ceil(x)
   printf("ceil(%.1f) = %.1f\n", 9.2, ceil(9.2));
   printf("ceil(%.1f) = %.1f\n", -9.8, ceil(-9.8));

   // calculates and outputs floor(x)
   printf("floor(%.1f) = %.1f\n", 9.2, floor(9.2));
   printf("floor(%.1f) = %.1f\n", -9.8, floor(-9.8));

   // calculates and outputs pow(x, y)
   printf("pow(%.1f, %.1f) = %.1f\n", 2.0, 7.0, pow(2.0, 7.0));
   printf("pow(%.1f, %.1f) = %.1f\n", 9.0, 0.5, pow(9.0, 0.5));

   // calculates and outputs fmod(x, y)
   printf("fmod(%.3f, %.3f) = %.3f\n", 13.657, 2.333,
      fmod(13.657, 2.333));

   // calculates and outputs sin(x)
   printf("sin(%.1f) = %.1f\n", 0.0, sin(0.0));

   // calculates and outputs cos(x)
   printf("cos(%.1f) = %.1f\n", 0.0, cos(0.0));

   // calculates and outputs tan(x)
   printf("tan(%.1f) = %.1f\n", 0.0, tan(0.0));
}
```

```
sqrt(900.0) = 30.0
sqrt(9.0) = 3.0
cbrt(27.0) = 3.0
cbrt(-8.0) = -2.0
exp(1.0) = 2.718282
exp(2.0) = 7.389056
log(2.718282) = 1.0
log(7.389056) = 2.0
log10(1.0) = 0.0
log10(10.0) = 1.0
log10(100.0) = 2.0
fabs(13.5) = 13.5
fabs(0.0) = 0.0
fabs(-13.5) = 13.5
ceil(9.2) = 10.0
ceil(-9.8) = -9.0
floor(9.2) = 9.0
floor(-9.8) = -10.0
pow(2.0, 7.0) = 128.0
pow(9.0, 0.5) = 3.0
fmod(13.657, 2.333) = 1.992
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0
```

5.4 See the answers below:

a) `double hypotenuse(double side1, double side2)`

b) `int smallest(int x, int y, int z)`

c) `void instructions(void)`

d) `float intToFloat(int number)`

5.5 See the answers below:

a) `double hypotenuse(double side1, double side2);`

b) `int smallest(int x, int y, int z);`

c) `void instructions(void);`

d) `float intToFloat(int number);`

5.6 `static float lastValue;`

5.7 See the answers below:

a) Error: Function `h` is defined in function `g`.
Correction: Move the definition of `h` out of the definition of `g`.

b) Error: The function body is supposed to return an integer, but does not.
Correction: Replace the statement in the function body with:

```
return x + y;
```

c) Error: Semicolon after the right parenthesis that encloses the parameter list, and redefining the parameter `a` in the function definition.
Correction: Delete the semicolon after the right parenthesis of the parameter list, and delete the declaration `float a;` in the function body.

d) Error: The result of `n + sum(n - 1)` is not returned; `sum` returns an improper result.
Correction: Rewrite the statement in the `else` clause as

```
return n + sum(n - 1);
```

e) Error: The function returns a value when it's not supposed to.
Correction: Eliminate the `return` statement.

Exercises

5.8 Show the value of `x` after each of the following statements is performed:

a) `x = fabs(10.85);`
b) `x = floor(10.85);`
c) `x = fabs(-0.678);`
d) `x = ceil(9.234);`
e) `x = fabs(0.0);`
f) `x = ceil(-34.87);`
g) `x = ceil(-fabs(-12 - floor(-9.5)));`

5.9 *(Car Rental Services)* A car rental service charges a minimum fee of $25.00 to rent a car for 8 hours and charges an additional $5 per hour after 8 hours. The maximum charge per day is $50, exclusive of service tax. The company charges an additional $0.50 per hour as service tax. Assume that no car is rented for more than 72 hours at a time. If a car is rented for more than 24 hours, then rent is calculated on a per day basis. Write a program that calculates and prints the rental charges for each of three customers who rented cars from this agency yesterday. You should enter the hours for which the car has been rented for each customer. Your program should print the results in a tabular format and should calculate and print the total of yesterday's receipts. The program should use the function `calculateCharges` to determine the charges for each customer. Your outputs should appear in the following format:

```
Car       Hours        Charge
1          12           56.00
2          34          117.00
3          48          124.00
TOTAL      94          297.00
```

5.10 *(Rounding Numbers)* An application of function `floor` is rounding a value to the nearest integer. The statement

```
y = floor(x + .5);
```

rounds `x` to the nearest integer and assigns the result to `y`. Write a program that reads several numbers and rounds each of these numbers to the nearest integer. For each number processed, print both the original number and the rounded number.

5.11 *(Rounding Numbers)* Function `floor` may be used to round a number to a specific decimal place. The statement

```
y = floor(x * 10 + .5) / 10;
```

rounds `x` to the tenths position (the first position to the right of the decimal point). The statement

```
y = floor(x * 100 + .5) / 100;
```

rounds `x` to the hundredths position (the second position to the right of the decimal point). Write a program that defines four functions to round a number `x` in various ways:

a) `roundToInteger(number)`
b) `roundToTenths(number)`
c) `roundToHundreths(number)`
d) `roundToThousandths(number)`

For each value the program inputs, display the original value, the number rounded to the nearest integer, the number rounded to the nearest tenth, the number rounded to the nearest hundredth, and the number rounded to the nearest thousandth.

5.12 Answer each of the following questions:

a) What is the difference between passing arguments by arguments and passing arguments by reference?
b) What values does the `rand` function generate?
c) How do you randomize a program? How do you scale or shift the values produced by the `rand` function?
d) What is a recursive function? What is a base case?

5.13 Write statements that assign random integers to the variable *n* in the following ranges:

a) $1 \le n \le 6$
b) $100 \le n \le 1000$
c) $0 \le n \le 19$
d) $1000 \le n \le 2222$
e) $-1 \le n \le 1$
f) $-3 \le n \le 11$

5.14 For each of the following sets of integers, write a single statement that will print a number at random from the set:

a) 3, 6, 9, 12, 15, 18, 21, 24, 27, 30.
b) 3, 5, 7, 9, 11, 13, 15, 17, 19.
c) 3, 8, 13, 18, 23, 28, 33.

5.15 *(Hypotenuse Calculations)* Define a function called `hypotenuse` that calculates a right triangle's hypotenuse, based on the values of the other two sides. The function should take two `double` arguments and return the hypotenuse as a `double`. Test your program with the side values specified in the following table:

Side 1	Side 2
3.0	4.0
5.0	12.0
8.0	15.0

5.16 *(Sides of a Triangle)* Write a function that reads three nonzero double values as the sides of a triangle, and calculates and returns the area of the triangle as a double variable. It should also check whether the numbers represent the sides of a triangle before calculating the area. Use this function in a program that inputs a series of sets of numbers.

5.17 *(Sides of a Right Triangle)* Write a function that reads three nonzero integers and determines whether they're the sides of a right-angled triangle. The function should take three integer arguments and return `1` (true) if the arguments comprise a right-angled triangle, and `0` (false) otherwise. Use this function in a program that inputs a series of sets of integers.

5.18 *(Even or Odd)* Write a program that inputs a series of integers and passes them one at a time to function `isEven`, which uses the remainder operator to determine whether an integer is even. The function should take an integer argument and return `1` if the integer is even and `0` otherwise.

5.19 *(Rectangle of Asterisks)* Write a function that displays a solid rectangle of asterisks whose sides are specified in the integer parameters `side1` and `side2`. For example, if the sides are 4 and 5, the function displays the following

```
*****
*****
*****
```

5.20 *(Displaying a Rectangle of Any Character)* Modify the function created in Exercise 5.19 to form the rectangle out of whatever character is contained in character parameter `fillCharacter`. Thus, if the `sides` are `5` and `4`, and `fillCharacter` is "`@`", then the function should print the following:

```
@@@@
@@@@
@@@@
@@@@
@@@@
```

5.21 *(Project: Drawing Shapes with Characters)* Use techniques similar to those developed in Exercises 5.19 and 5.20 to produce a program that graphs a wide range of shapes.

5.22 *(Separating Digits)* Write program segments to accomplish each of the following:

a) Calculate the `int` part of the quotient when `int a` is divided by `int b`.
b) Calculate the `int` remainder when `int a` is divided by `int b`.
c) Use the program pieces developed in a) and b) to write a function that inputs an integer between `1` and `32767` and prints it as a series of digits, with two spaces between each digit. For example, `4562` should be printed as:

```
4  5  6  2
```

5.23 *(Time in Seconds)* Write a function that takes the time as three integer arguments (for hours, minutes and seconds) and returns the number of seconds since the last time the clock "struck 12." Use this function to calculate the amount of time in seconds between two times, both of which are within one 12-hour cycle of the clock.

5.24 *(Currency Conversion)* Implement the following double functions:

a) Function `toYen` takes an amount in US dollars and returns the Yen equivalent.
b) Function `toEuro` takes an amount in US dollars and returns the Euro equivalent.
c) Use these functions to write a program that prints charts showing the Yen and Euro equivalents of a range of dollar amounts. Print the outputs in a neat tabular format. Use an exchange rate of 1 USD = 118.87 Yen, and 1 USD = 0.92 Euro.

5.25 *(Find the Maximum)* Write a function that returns the largest of four floating-point numbers.

5.26 *(Perfect Numbers)* An integer number is said to be a *perfect number* if its factors, including 1 (but not the number itself), sum to the number. For example, 6 is a perfect number because 6 = 1 + 2 + 3. Write a function `isPerfect` that determines whether parameter `number` is a perfect number. Use this function in a program that determines and prints all the perfect numbers between 1 and 1000. Print the factors of each perfect number to confirm that the number is indeed perfect. Challenge the power of your computer by testing numbers much larger than 1000.

5.27 *(Roots of a Quadratic Equation)* A quadratic equation is any equation of the form $ax^2 + bx + c = 0$ where a, b, and c are the coefficients of x. The roots of a quadratic equation can be calculated by the formula $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. If the expression, $b^2 - 4ac$, which is also called the discriminant, is positive then the equation has real roots. If the discriminant is negative, the equation has imaginary (or complex) roots. Write a function that accepts the coefficients of an equation as parameters, checks if the roots are real, and calculates the roots of the equation. Write a program to test this function.

5.28 *(Sum of Digits)* Write a function that takes an integer and returns the sum of its digits. For example, given the number 7631, the function should return 17 (Explanation: 7 + 6 + 3 + 1 = 17).

5.29 *(Lowest Common Multiple)* The *lowest common multiple (LCM)* of two integers is the smallest positive integer that is a multiple of both numbers. Write a function `lcm` that returns the lowest common multiple of two numbers.

5.30 *(Quality Points for Student's Grades)* Write a function `toQualityPoints` that inputs a student's average and returns 4 if it's 90–100, 3 if it's 80–89, 2 if it's 70–79, 1 if it's 60–69, and 0 if the average is lower than 60.

5.31 *(Coin Tossing)* Write a program that simulates coin tossing. For each toss, display `Heads` or `Tails`. Let the program toss the coin 100 times, and count the number of heads and tails. Display the results. The program should call a function `flip` that takes no arguments and returns `0` for tails and `1` for heads. If the program realistically simulates the coin tossing, then each side of the coin should appear approximately half the time for a total of approximately 50 heads and 50 tails.

5.32 *(Guess the Number)* Write a C program that plays the game of "guess the number" as follows: Your program chooses the number to be guessed by selecting an integer at random in the range 1 to 1000. The program then types:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
```

The player types a first guess. The program responds with one of the following:

```
1. Excellent! You guessed the number!
   Would you like to play again (y or n)?
2. Too low. Try again.
3. Too high. Try again.
```

If the guess is incorrect, your program should loop until the player guesses the number. Your program should keep telling the player `Too high` or `Too low` to help the player "zero in" on the correct answer.

5.33 *(Guess the Number Modification)* Modify your Exercise 5.32 solution to count the number of guesses the player makes. If the number is 10 or fewer, print `"Either you know the secret or you got lucky!"` If the player guesses the number in 10 tries, then print `"Aha! You know the secret!"` If the player makes more than 10 guesses, then print `"You should be able to do better!"` Why should it take no more than 10 guesses? Well, with each "good guess" the player should be able to eliminate half of the numbers. Now show why any number 1 to 1000 can be guessed in 10 or fewer tries.

5.34 *(Recursive Exponentiation)* Write a recursive function `power(base, exponent)` that when invoked returns

$$\texttt{base}^{\texttt{exponent}}$$

For example, `power(3, 4) = 3 * 3 * 3 * 3`. Assume that `exponent` is an integer greater than or equal to 1. *Hint:* The recursion step would use the relationship

$$\texttt{base}^{\texttt{exponent}} = \texttt{base} * \texttt{base}^{\texttt{exponent-1}}$$

and the terminating condition occurs when `exponent` is equal to `1` because

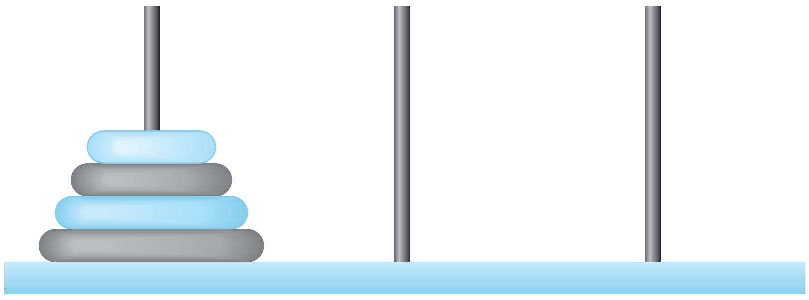
$$\texttt{base}^{1} = \texttt{base}$$

5.35 *(Fibonacci)* The Fibonacci series

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

begins with the terms 0 and 1 and has the property that each succeeding term is the sum of the two preceding terms. First, write a *nonrecursive* function `fibonacci(n)` that calculates the `n`th Fibonacci number. Use `int` for the function's parameter and `unsigned long long int` for its return type. Then, determine the largest Fibonacci number that can be printed on your system.

5.36 *(Towers of Hanoi)* Every budding computer scientist must grapple with certain classic problems, and the Towers of Hanoi (shown in the following diagram) is one of the most famous of these:



Legend has it that in a temple in the Far East, priests are attempting to move a stack of disks from one peg to another. The initial stack had 64 disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from this peg to a second peg under the constraints that exactly one disk is moved at a time and at no time may a larger disk be placed above a smaller disk. A third peg is available for temporarily holding the disks. Supposedly the world will end when the priests complete their task, so there's little incentive for us to facilitate their efforts.

Let's assume that the priests are attempting to move the disks from peg 1 to peg 3. We wish to develop an algorithm that will print the precise sequence of disk-to-disk peg transfers.

If we were to approach this problem with conventional methods, we'd rapidly find ourselves hopelessly knotted up in managing the disks. Instead, if we attack the problem with recursion in mind, it immediately becomes tractable. Moving n disks can be viewed in terms of moving only $n - 1$ disks (and hence the recursion) as follows:

a) Move $n - 1$ disks from peg 1 to peg 2, using peg 3 as a temporary holding area.
b) Move the last disk (the largest) from peg 1 to peg 3.
c) Move the $n - 1$ disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

The process ends when the last task involves moving $n = 1$ disk, i.e., the base case. This is accomplished by trivially moving the disk without the need for a temporary holding area.

Write a program to solve the Towers of Hanoi problem. Use a recursive function with four parameters:

a) The number of disks to be moved.
b) The peg on which these disks are initially threaded.
c) The peg to which this stack of disks is to be moved.
d) The peg to be used as a temporary holding area.

Your program should print the precise instructions it will take to move the disks from the starting peg to the destination peg. For example, to move a stack of three disks from peg 1 to peg 3, your program should print the following series of moves:

$1 \rightarrow 3$ (This means move one disk from peg 1 to peg 3.)
$1 \rightarrow 2$
$3 \rightarrow 2$
$1 \rightarrow 3$
$2 \rightarrow 1$
$2 \rightarrow 3$
$1 \rightarrow 3$

5.37 *(Towers of Hanoi: Iterative Solution)* Any program that can be implemented recursively can be implemented iteratively, although sometimes with considerably more difficulty and considerably less clarity. Try writing an iterative version of the Towers of Hanoi. If you succeed, compare your iterative version with the recursive version you developed in Exercise 5.36. Investigate issues of performance, clarity and your ability to demonstrate the correctness of the programs.

5.38 *(Visualizing Recursion)* It's interesting to watch recursion "in action." Modify the factorial function of Fig. 5.9 to print its local variable and recursive call parameter. For each recursive call, display the outputs on a separate line and add a level of indentation. Do your utmost to make the outputs clear, interesting and meaningful. Your goal here is to design and implement an output format that helps a person understand recursion better. You may want to add such display capabilities to the many other recursion examples and exercises throughout the text.

5.39 *(Recursive Greatest Common Divisor)* The greatest common divisor of integers `x` and `y` is the largest integer that evenly divides both `x` and `y`. Write a recursive function `gcd` that returns the greatest common divisor of `x` and `y`. The greatest common divisor of `x` and `y` is defined recursively as follows: If `y` is equal to `0`, then `gcd(x, y)` is `x`; otherwise `gcd(x, y)` is `gcd(y, x % y)`, where `%` is the remainder operator.

5.40 *(Recursive `main`)* Can `main` be called recursively? Write a program containing a function `main`. Include `static` local variable `count` initialized to 1. Postincrement and print the value of `count` each time `main` is called. Run your program. What happens?

5.41 *(Recursive Prime)* Write a recursive function `isPrime` that determines whether the given input is a prime number. Use this function in a program.

5.42 What does the following program do? What happens if you exchange lines 7 and 8?

```
 1  #include <stdio.h>
 2
 3  int main(void) {
 4     int c = '\0'; // variable to hold character input by user
 5
 6     if ((c = getchar()) != EOF) {
 7        main();
 8        printf("%c", c);
 9     }
10  }
```

5.43 What does the following program do?

```c
#include <stdio.h>

int mystery(int a, int b); // function prototype

int main(void) {
   printf("%s", "Enter two positive integers: ");
   int x = 0; // first integer
   int y = 0; // second integer
   scanf("%d%d", &x, &y);

   printf("The result is %d\n", mystery(x, y));
}

// Parameter b must be a positive integer
// to prevent infinite recursion
int mystery(int a, int b) {
   // base case
   if (1 == b) {
      return a;
   }
   else { // recursive step
      return a + mystery(a, b - 1);
   }
}
```

5.44 After you determine what the program of Exercise 5.43 does, modify it to function properly after removing the restriction that the second argument must be positive.

5.45 *(Testing Math Library Functions)* Write a program that tests the math library functions shown in Section 5.3's table. Exercise each of these functions by having your program print out tables of return values for a diversity of argument values.

5.46 Find the error in each of the following program segments and explain how to correct it:

a)
```c
double cube(float); // function prototype
cube(float number) { // function definition
   return number * number * number;
}
```

b)
```c
int randomNumber = srand();
```

c)
```c
double y = 123.45678;
int x;
x = y;
printf("%f\n", (double) x);
```

d)
```c
double square(double number) {
   double number;
   return number * number;
}
```

e)
```
int sum(int n) {
   if (0 == n) {
      return 0;
   }
   else {
      return n + sum(n);
   }
}
```

5.47 *(Rock, Paper, Scissors Game Modification)* Modify the program of Fig. 5.7 to have a method with custom logic to programmatically choose the player's Shape to play instead of entering from user input. We can apply any strategies to choose the Shape to increase our chance of winning. One possible strategy: we can count the number of each rock, paper, and scissors that has been played by the computer, and the least played shape has a higher chance to be played by the computer in the next round.

5.48 *(Research Project: Improving the Recursive Fibonacci Implementation)* In Section 5.15, the recursive algorithm we used to calculate Fibonacci numbers was intuitively appealing. However, recall that the algorithm resulted in the exponential explosion of recursive function calls. Research the recursive Fibonacci implementation online. Study the various approaches, including the iterative version in Exercise 5.35 and versions that use only so-called "tail recursion." Discuss the relative merits of each.

Computer-Assisted Instruction

Computers create exciting possibilities for improving the educational experience of all students worldwide, as suggested by the next five exercises. [*Note:* Check out initiatives such as the One Laptop Per Child Project (`www.laptop.org`).]

5.49 *(Computer-Assisted Instruction)* The use of computers in education is referred to as *computer-assisted instruction* (*CAI*). Write a program that will help an elementary-school student learn multiplication. Use the `rand` function to produce two positive one-digit integers. The program should then prompt the user with a question, such as

```
How much is 6 times 7?
```

The student then inputs the answer. Next, the program checks the student's answer. If it's correct, display the message `"Very good!"` and ask another multiplication question. If the answer is wrong, display the message `"No. Please try again."` and let the student try the same question repeatedly until the student finally gets it right. A separate function should be used to generate each new question. This function should be called once when the application begins execution and each time the user answers the question correctly.

5.50 *(Computer-Assisted Instruction: Reducing Student Fatigue)* One problem in CAI environments is student fatigue. This can be reduced by varying the computer's responses to hold the student's attention. Modify the program of Exercise 5.49 so that various comments are displayed for each answer as follows:

Possible responses to a correct answer:

```
Very good!
Excellent!
Nice work!
Keep up the good work!
```

Possible responses to an incorrect answer:

```
No. Please try again.
Wrong. Try once more.
Don't give up!
No. Keep trying.
```

Use random-number generation to choose a number from 1 to 4 that will be used to select one of the four appropriate responses to each correct or incorrect answer. Use a `switch` statement to issue the responses.

5.51 *(Computer-Assisted Instruction: Monitoring Student Performance)* More sophisticated computer-assisted instruction systems monitor the student's performance over a period of time. The decision to begin a new topic is often based on the student's success with previous topics. Modify the program of Exercise 5.50 to count the number of correct and incorrect responses typed by the student. After the student types 10 answers, your program should calculate the percentage that are correct. If the percentage is lower than 75%, display "`Please ask your teacher for extra help.`", then reset the program so another student can try it. If the percentage is 75% or higher, display "`Congratulations, you are ready to go to the next level!`", then reset the program so another student can try it.

5.52 *(Computer-Assisted Instruction: Difficulty Levels)* Exercises 5.49–5.51 developed a computer-assisted instruction program to help teach an elementary-school student multiplication. Modify the program to allow the user to enter a difficulty level. At a difficulty level of 1, the program should use only single-digit numbers in the problems; at a difficulty level of 2, numbers as large as two digits, and so on.

5.53 *(Computer-Assisted Instruction: Varying the Types of Problems)* Modify the program of Exercise 5.52 to allow the user to pick a type of arithmetic problem to study. An option of `1` means addition problems only, `2` means subtraction problems only, `3` means multiplication problems only and `4` means a random mixture of all these types.

Random-Number Simulation Case Study: The Tortoise and the Hare

5.54 *(The Tortoise and the Hare Race)* In this problem, you'll recreate one of the truly great moments in history—the classic race of the tortoise and the hare. You'll use random-number generation to develop a simulation of this memorable event.

Our contenders begin the race at "square 1" of 70 squares. Each square represents a possible position along the racecourse. The finish line is at square 70. The first contender to reach or pass square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up a slippery mountainside, so occasionally, the contenders lose ground.

There's a clock that ticks once per second. With each tick, adjust the animals' positions according to the following rules:

Animal	Move type	Percentage of the time	Actual move
Tortoise	Fast plod	50%	3 squares forward
	Slip	20%	6 squares backward
	Slow plod	30%	1 square forward
Hare	Sleep	20%	No move at all
	Big hop	20%	9 squares forward
	Big slip	10%	12 squares backward
	Small hop	30%	1 square forward
	Small slip	20%	2 squares backward

Use variables to keep track of the animals' positions (i.e., position numbers are 1–70). Start each animal at position 1 (i.e., the "starting gate"). If an animal slips left before square 1, move the animal back to square 1. If an animal moves past square 70, move the animal back to square 80.

Generate the percentages in the preceding table by producing a random integer, x, in the range 1 ≤ x ≤ 10. For the tortoise, perform a "fast plod" when 1 ≤ x ≤ 5, a "slip" when 6 ≤ x ≤ 7, or a "slow plod" when 8 ≤ x ≤ 10. Use a similar technique to move the hare.

Begin the race by printing

```
ON YOUR MARK, GET SET
BANG              !!!!
AND THEY'RE OFF   !!!!
```

Then, for each tick (i.e., each iteration of a loop), print a 70-position line showing the letter `T` in the tortoise's position and the letter `H` in the hare's position. Occasionally, the contenders will land on the same square. In this case, the tortoise bites the hare, and your program should print `"OUCH!!!"` beginning at that position. All print positions other than the `T`, the `H`, or the `OUCH!!!` (in case of a tie) should be blank.

After printing each line, test whether either animal has reached or passed square 70. If so, then print the winner and terminate the simulation. If the tortoise wins, print `"TORTOISE WINS!!! YAY!!!"` If the hare wins, print `"Hare wins. Yuch."` If both animals win on the same tick of the clock, you may want to favor the turtle (the "underdog"), or you may want to print `"It's a tie"`. If neither animal wins, perform the loop again to simulate the next tick of the clock. When you're ready to run your program, assemble a group of fans to watch the race. You'll be amazed at how involved your audience gets!

This page is intentionally left blank