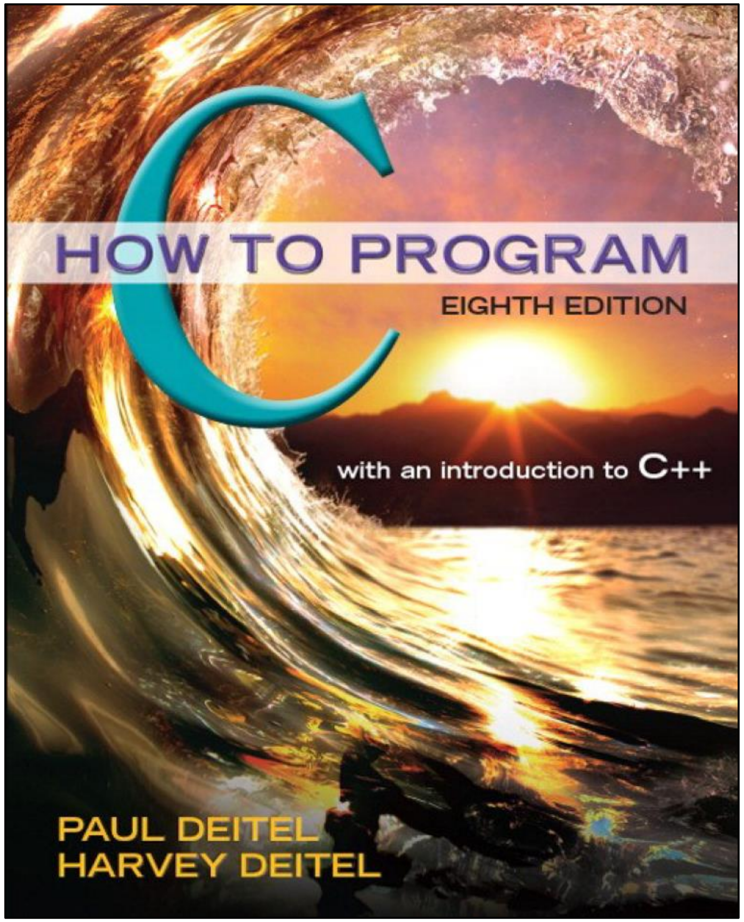
C How to Program

Eighth Edition



Chapter 5

C Functions

Learning Objectives

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

Outline (1 of 2)

Outline (2 of 2)

5.1 Introduction

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

5.2 Modularizing Programs in C (1 of 4)

- **Functions** are used to modularize programs
- C programs are typically written by combining new functions you write with **prepackaged** functions available in the **C standard library.**
- 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.

5.2 Modularizing Programs in C (2 of 4)

- The functions `printf`, `scanf` and `pow` that we've used in previous chapters are standard library functions.
- You can write your own functions to define tasks that may be used at many points in a program.
- These are sometimes referred to as **programmer-defined functions.**
- The statements defining the function are written only once, and the statements are hidden from other functions.
- Functions are **invoked** by a **function call,** which specifies the function name and provides information (as **arguments**) that the called function needs to perform its designated task.

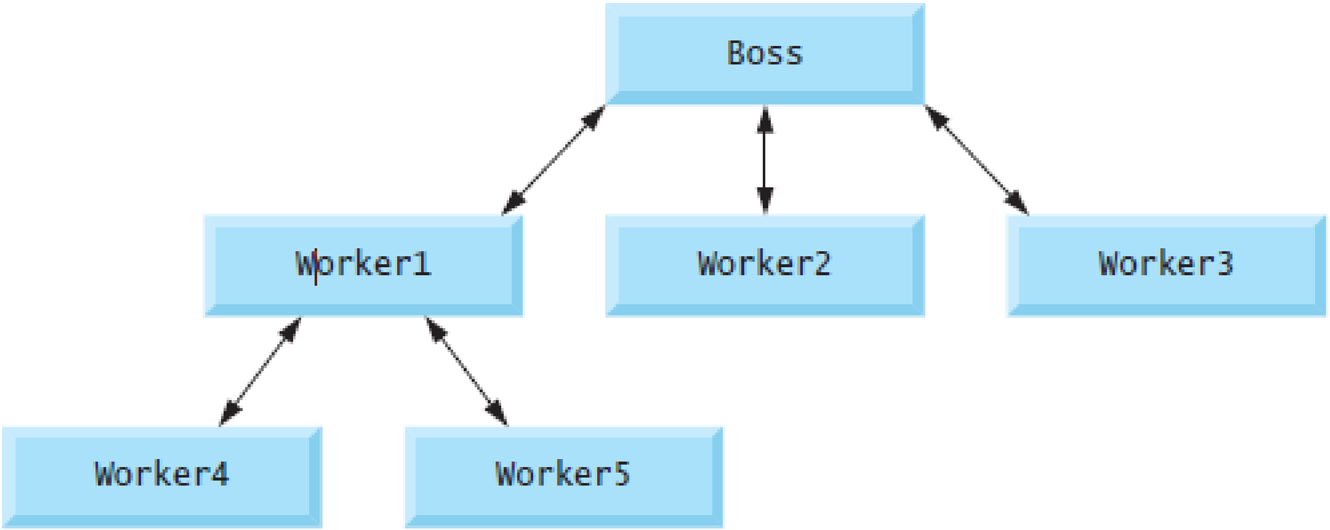
5.2 Modularizing Programs in C (3 of 4)

- A common analogy for this is the hierarchical form of management.
- A boss (the **calling function** or **caller**) asks a worker (the **called function**) to perform a task and report back when the task is done (Figure 5.1 (see slide 13)).
- For example, a function needing to display information on the screen calls the worker function `printf` to perform that task, then `printf` displays the information and reports back—or **returns**—to the calling function when its task is completed.
- The boss function does not know how the worker function performs its designated tasks.

5.2 Modularizing Programs in C (4 of 4)

- The worker may call other worker functions, and the boss will be unaware of this.
- We'll soon see how this "hiding" of implementation details promotes good software engineering.
- Figure 5.1 shows a boss function communicating with several worker functions in a hierarchical manner.
- Note that `worker1` acts as a boss function to `worker4` and `worker5`.
- Relationships among functions may differ from the hierarchical structure shown in this figure.

Figure 5.1 Hierarchical Boss-Function/Worker-Function Relationship



5.3 Math Library Functions (1 of 3)

- Math library functions allow you to perform certain common mathematical calculations.
- Functions are normally used in a program by writing the name of the function followed by a left parenthesis followed by the **argument** (or a comma-separated list of arguments) of the function followed by a right parenthesis.
- For example, a programmer desiring to calculate and print the square root of `900.0` you might write

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

- When this statement executes, the math library function `sqrt` is called to calculate the square root of the number contained in the parentheses (`900.0`).

5.3 Math Library Functions (2 of 3)

- The number `900.0` is the argument of the `sqrt` function.
- The preceding statement would print `30.00`.
- The `sqrt` function takes an argument of type double and returns a result of type `double`.
- All functions in the math library that return floating-point values return the data type `double`.
- Note that `double` values, like `float` values, can be output using the `%f` conversion specification.

5.3 Math Library Functions (3 of 3)

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

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

- calculates and prints the square root of `13.0` + `3.0` * `4.0` = `25.0`, namely `5.00`.
- In the figure, the variables `x` and `y` are of type `double`.

Figure 5.2 Commonly Used Math Library Functions (1 of 2)

Function	Description	Example
sqrt(x)	square root of *x*	sqrt(**900.0**) is 30.0 sqrt(**9.0**) is 3.0
cbrt(x)	cube root of *x* (C99 and C11 only)	cbrt(**27.0**) is 3.0 cbrt(**−8.0**) is **−**2.0
exp(x)	exponential function e^x	exp(**1.0**) is 2.718282 exp(**2.0**) is 7.389056
log(x)	natural logarithm of *x* (base *e*)	log(**2.718282**) is 1.0 log(**7.389056**) is 2.0
log10(x)	logarithm of *x* (base 10)	log10(**1.0**) is 0.0 log10(**10.0**) is 1.0 log10(**100.0**) is 2.0
fabs(x)	absolute value of *x* as a floating-point number	fabs(**13.5**) is 13.5 fabs(**0.0**) is 0.0 fabs(**−13.5**) is 13.5
ceil(x)	rounds *x* to the smallest integer not less than *x*	ceil(**9.2**) is 10.0 ceil(**−9.8**) is **−**9.0

Figure 5.2 Commonly Used Math Library Functions (2 of 2)

Function	Description	Example
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* $\left(x^y\right)$	pow(**2**, **7**) is 128.0 pow(**9**, **.5**) is 3.0
fmod(x, y)	remainder of $\frac{x}{y}$ as a floating-point number	fmod(**13.657**, **2.333**) is 1.992
sin(x)	trigonometric sine of x (x in radians)	sin(**0.0**) is 0.0
cos(x)	trigonometric cosine of x (x in radians)	cos(**0.0**) is 1.0
tan(x)	trigonometric tangent of x (x in radians)	tan(**0.0**) is 0.0

5.4 Functions (1 of 2)

- Functions allow you to modularize a program.
- All variables defined in function definitions are **local variables**—they can be accessed **only** in the function in which they're defined.
- Most functions have a list of **parameters** that provide the means for communicating information between functions.
- A function's parameters are also local variables of that function.

5.4 Functions (2 of 2)

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

5.5 Function Definitions (1 of 10)

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

Figure 5.3 Creating and Using a Programmer-Defined Function

```
// Fig. 5.3: fig05_03.c
// Creating and using a programmer-defined function.
#include <stdio.h>

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

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

5.5 Function Definitions (2 of 10)

- Function square is **invoked** or **called** in `main` within the `printf` statement

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

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

5.5 Function Definitions (3 of 10)

- The definition of function `square` shows that `square` expects an integer parameter `y`.
- The keyword `int` preceding the function name indicates that `square` **returns** an integer result.
- The **`return` statement** in `square` passes the value of the expression `y * y` (that is, the result of the calculation) back to the calling function.
- `int square(int y); // function prototype`
  - The `int` in parentheses informs the compiler that `square` expects to **receive** an integer value from the caller.

5.5 Function Definitions (4 of 10)

- The `int` to the **left** of the function name `square` informs the compiler that `square` returns an integer result to the caller.
- The compiler refers to the function prototype to check that any calls to `square` contain the **correct return type**, the **correct number of arguments**, the **correct argument types**, and that the **arguments are in the correct order**.
- The format of a function definition is

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

5.5 Function Definitions (5 of 10)

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

5.5 Function Definitions (6 of 10)

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

5.5 Function Definitions (7 of 10)

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

5.5 Function Definitions (8 of 10)

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

```
return;
```

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

`return` *expression;*

- returns the value of **expression** to the caller**.**

5.5 Function Definitions (9 of 10)

`main`'s Return Type

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

```
return 0;
```

- at the end of `main`—`0` indicates that a program ran successfully.
- The C standard indicates that main implicitly returns 0 if you to omit the preceding statement—as we've done throughout this book.

5.5 Function Definitions (10 of 10)

- You can explicitly return non-zero 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 environment.

5.5 Function Definitions (11 of 10)

Function `maximum`

- Our second example uses a programmer-defined function `maximum` to determine and return the largest of three integers (Figure 5.4).
- Next, they're passed to `maximum,` which determines the largest integer.
- This value is returned to main by the `return` statement in `maximum.`

Figure 5.4 Finding the Maximum of Three Integers (1 of 3)

```c
// Fig. 5.4: fig05_04.c
// Finding the maximum of three integers.
#include <stdio.h>

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

int main(void)
{
   int number1; // first integer entered by the user
   int number2; // second integer entered by the user
   int number3; // third integer entered by the user

   printf("%s", "Enter three integers: ");
   scanf("%d%d%d", &number1, &number2, &number3);

   // number1, number2 and number3 are arguments
   // to the maximum function call
   printf("Maximum is: %d\n", maximum(number1, number2, number3));
}

```

Figure 5.4 Finding the Maximum of Three Integers (2 of 3)

```
// Function maximum definition
// x, y and z are parameters
int maximum(int x, int y, int z)
{
   int max = x; // assume x is largest

   if (y > max) { // if y is larger than max,
      max = y; // assign y to max
   }

   if (z > max) { // if z is larger than max,
      max = z; // assign z to max
   }

   return max; // max is largest value
}
```

Figure 5.4 Finding the Maximum of Three Integers (3 of 3)

```
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
```

5.6 Function Prototypes: A Deeper Look (3 of 12)

- The function prototype for `maximum` in Figure 5.4 (see slides 49-51) is

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

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

5.6 Function Prototypes: A Deeper Look (4 of 12)

Compilation Errors

- A function call that does not match the function prototype is a compilation error.
- An error is also generated if the function prototype and the function definition disagree.
- For example, in Figure 5.4 (see slides 30-31), if the function prototype had been written

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

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

5.6 Function Prototypes: A Deeper Look (5 of 12)

Argument Coercion and "Usual Arithmetic Conversion Rules"

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

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

correctly evaluates `sqrt(4)` and prints the value `2.000`.

5.6 Function Prototypes: A Deeper Look (7 of 12)

- In our `sqrt` example above, an `int` is automatically converted to a `double` without changing its value.
- However, a `double` converted to an `int` **truncates** the fractional part of the `double` value, thus changing the original value.
- Converting large integer types to small integer types (e.g., `long` to `short`) may also result in changed values.

5.6 Function Prototypes: A Deeper Look (8 of 12)

- The usual arithmetic conversion rules automatically apply to expressions containing values of two or more data types (also referred to as **mixed-type expressions**) and are handled for you by the compiler.
- In a mixed-type expression, the compiler makes a temporary copy of the value that needs to be converted then converts the copy to the “highest” type in the expression—the original value remains unchanged.

5.6 Function Prototypes: A Deeper Look (9 of 12)

- The usual arithmetic conversion rules for a mixed-type expression containing at least one floating-point value are:
  - If one of the values is a long double, the other is converted to a long double.
  - If one of the values is a double, the other is converted to a double.
  - If one of the values is a float, the other is converted to a float.

5.6 Function Prototypes: A Deeper Look (10 of 12)

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

Figure 5.5 Arithmetic Data Types and Their Conversion Specifications

Data type	`printf` conversion specification	`scanf` conversion specification
Floating-point types		
long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
Integer types		
unsigned long long int	%llu	%llu
long long int	%llu	%llu
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

5.6 Function Prototypes: A Deeper Look (11 of 12)

- 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.
- Function argument values are converted to the parameter types in a function prototype as if they were being assigned directly to variables of those types.
- If our `square` function that uses an integer parameter (Figure 5.3 (see slide 27)) is called with a floating-point argument, the argument is converted to `int` (a lower type), and `square` usually returns an incorrect value.
- For example, `square(4.5)` returns `16`, not `20.25`.

5.6 Function Prototypes: A Deeper Look (12 of 12)

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

5.7 Function Call Stack and Stack Frames (1 of 12)

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

5.7 Function Call Stack and Stack Frames (2 of 12)

- An important mechanism for computer science students to understand is the **function call stack** (sometimes referred to as the **program execution stack**).
- This data structure—working “behind the scenes”—supports the function call/return mechanism.
- It also supports the creation, maintenance and destruction of each called function’s local variables.

5.7 Function Call Stack and Stack Frames (3 of 12)

- As each function is called, it may call other functions, which may call other functions—all before any function returns.
- Each function eventually must return control to the function that called it.
- So, we must keep track of the return addresses that each function needs to return control to the function that called it.
- The function call stack is the perfect data structure for handling this information.

5.7 Function Call Stack and Stack Frames (9 of 12)

Function Call Stack in Action

- Now let's consider how the call stack supports the operation of a square function called by `main`
- First the operating system calls `main`—this pushes a stack frame onto the stack.
- The stack frame tells `main` how to return to the operating system (i.e., transfer to return address `R1`) and contains the space for `main`'s automatic variable (i.e., a, which is initialized to `10`).

Figure 5.6 Demonstrating the Function Call Stack and Stack Frames Using a Function Square

```c
// Fig. 5.6: fig05_06.c
// Demonstrating the function call stack
// and stack frames using a function square.
#include <stdio.h>

int square(int); // prototype for function square

int main()
{
   int a = 10; // value to square (local automatic variable in main)

   printf("%d squared: %d\n", a, square(a)); // display a squared
}

// returns the square of an integer
int square(int x) // x is a local variable
{
   return x * x; // calculate square and return result
}
```

```
10 squared: 100
```

5.7 Function Call Stack and Stack Frames (10 of 12)

- Function `main`—before returning to the operating system—now calls function `square`
- This causes a stack frame for `square` to be pushed onto the function call stack (Figure 5.8 (see slides 50-51)).
- This stack frame contains the return address that `square` needs to return to `main` (i.e., `R2`) and the memory for `square`'s automatic variable (i.e., `x`).

Figure 5.7 Function Call Stack After the Operating System Invokes `Main` to Execute the Program

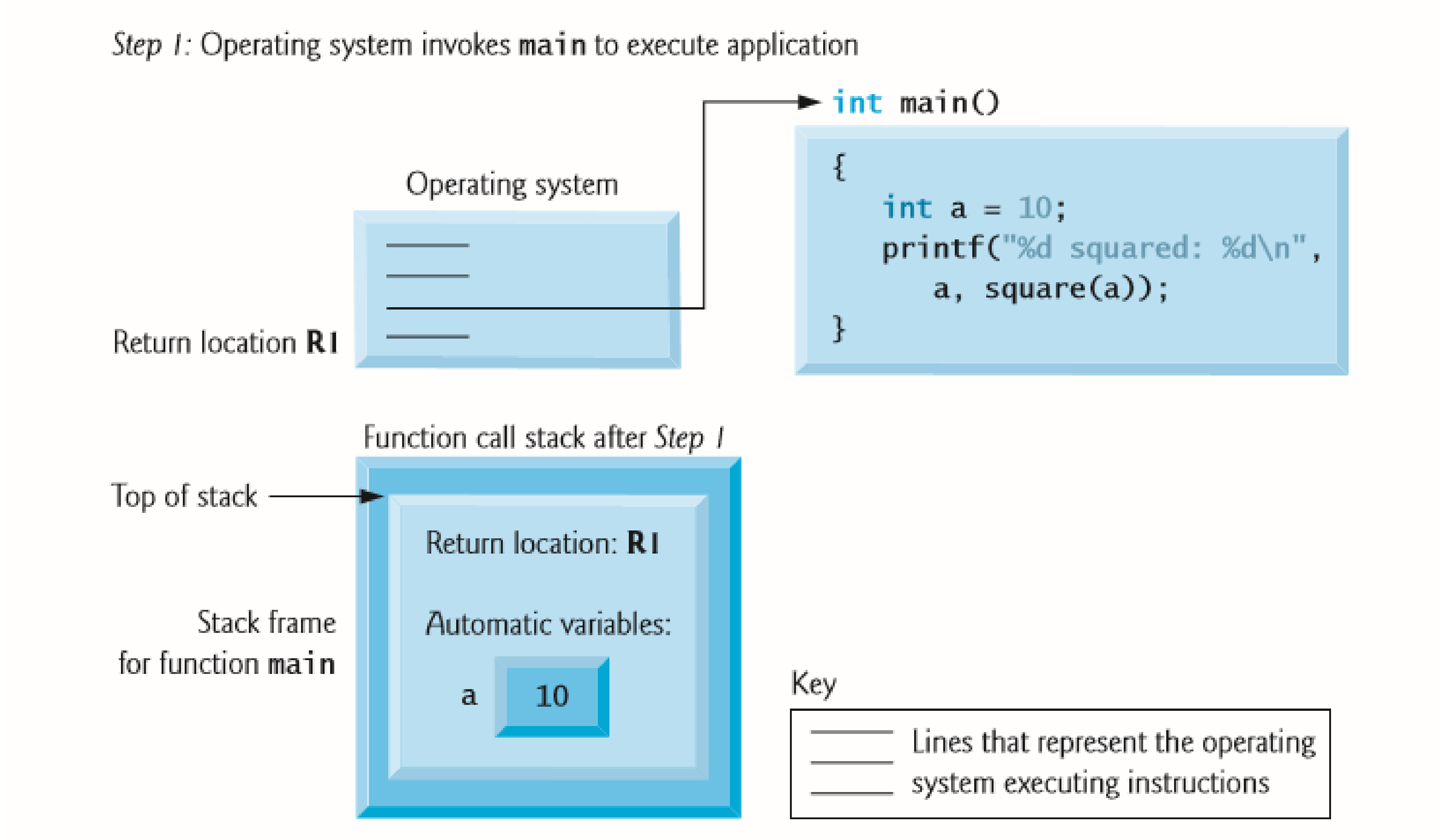


Figure 5.8 Function Call Stack After `Main` Invokes `Square` to Perform the Calculation (1 of 2)

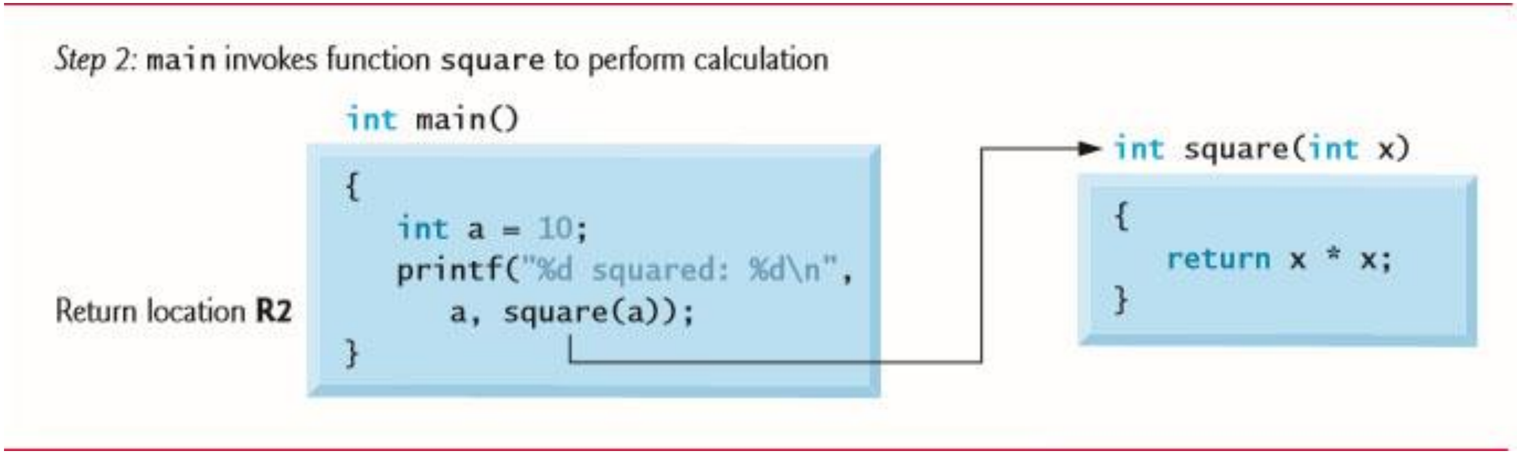
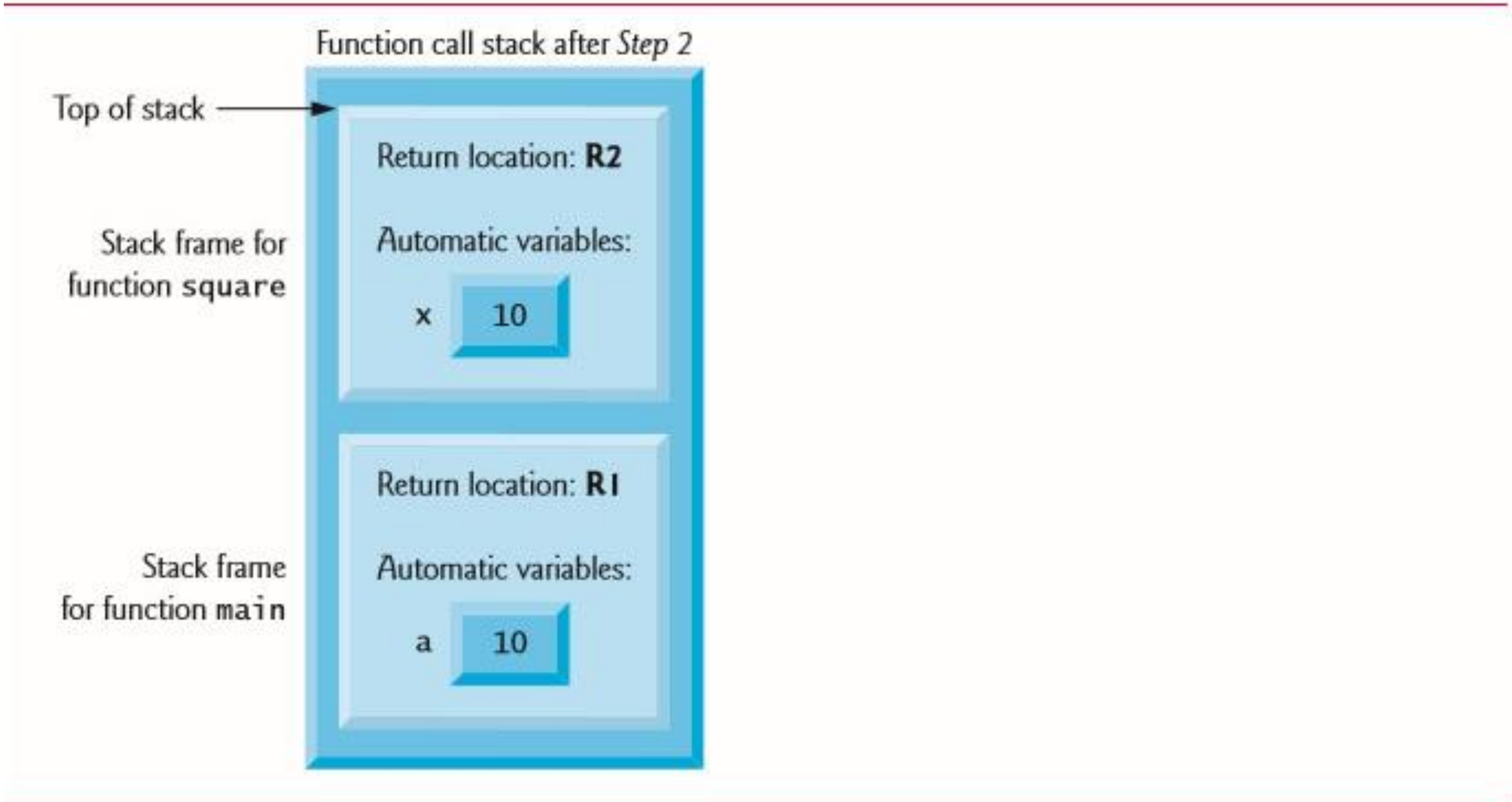


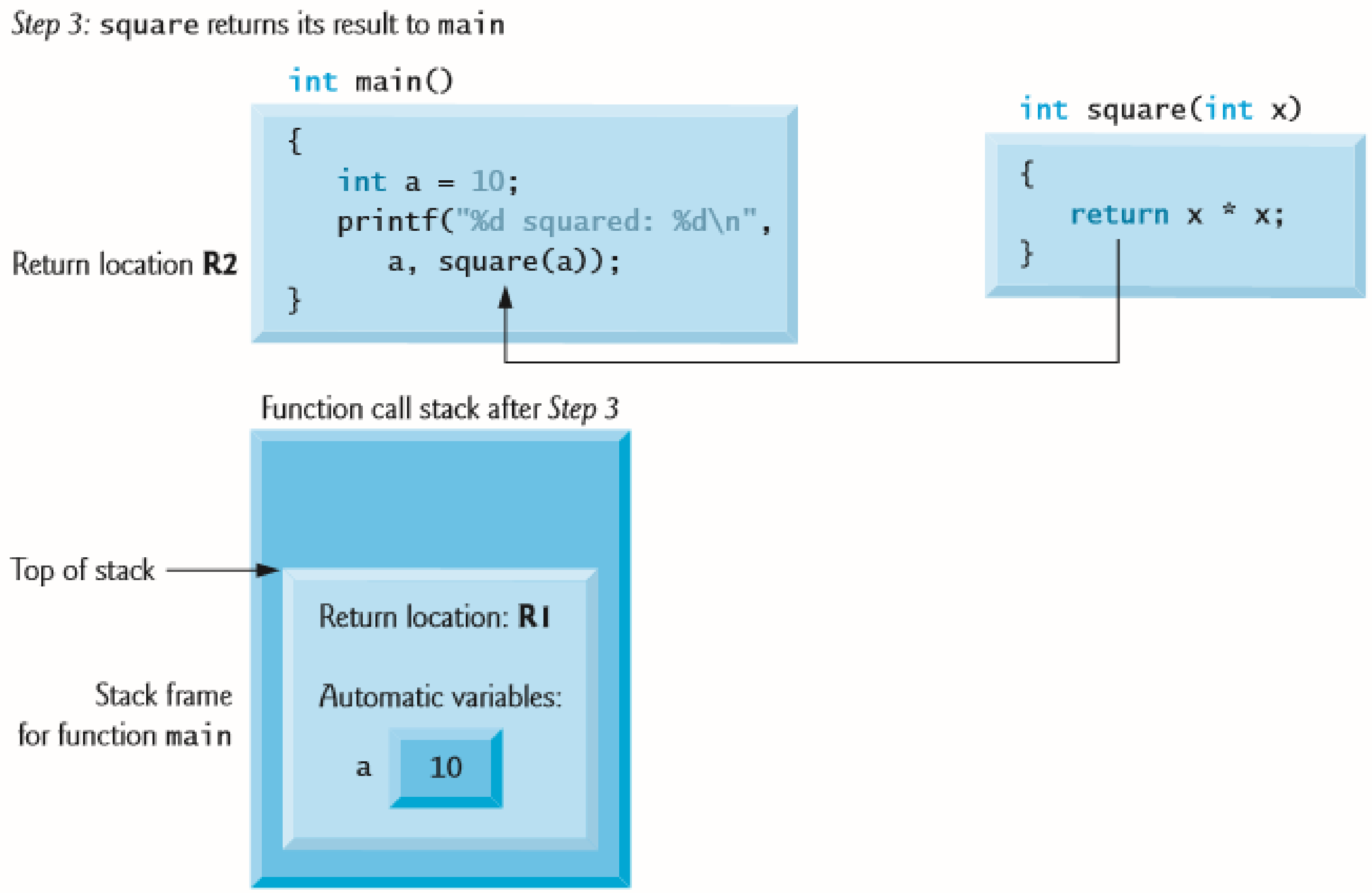
Figure 5.8 Function Call Stack After `Main` Invokes `Square` to Perform the Calculation (2 of 2)



5.7 Function Call Stack and Stack Frames (11 of 12)

- After `square` calculates the square of its argument, it needs to return to `main`—and no longer needs the memory for its automatic variable `x`.
- So the stack is popped—giving `square` the return location in `main` (i.e., `R2`) and losing `square’s` automatic variable.
- Figure 5.9 shows the function call stack after square’s stack frame has been popped.

Figure 5.9 Function Call Stack After Function `square` Returns to `main`



5.8 Headers (1 of 2)

- Each standard library has a corresponding **header** containing the function prototypes for all the functions in that library and definitions of various data types and constants needed by those functions.
- Figure 5.6 lists alphabetically some of the standard library headers that may be included in programs.
- The term “macros” that’s used several times in Figure 5.6 is discussed in detail in Chapter 13.
- You can create custom headers.
- Programmer-defined headers should also use the `.h` filename extension.

5.8 Headers (2 of 2)

- 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"
```

Figure 5.10 Some of the Standard Library Headers (1 of 2)

Header	Explanation
< assert.h >	Contains information for adding diagnostics that aid program debugging.
< ctype.h >	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
< errno.h >	Defines macros that are useful for reporting error conditions.
< float.h >	Contains the floating-point size limits of the system.
< limits.h >	Contains the integral size limits of the system.
< locale.h >	Contains function prototypes and other information that enables a program to be modified for the current locale on which it's running. The notion of locale enables the computer system to handle different conventions for expressing data such as dates, times, currency amounts and large numbers throughout the world.
< math.h >	Contains function prototypes for math library functions.

Figure 5.10 Some of the Standard Library Headers (2 of 2)

Header	Explanation
< setjmp.h >	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.
< signal.h >	Contains function prototypes and macros to handle various conditions that may arise during program execution.
< stdarg.h >	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
< stddef.h >	Contains common type definitions used by C for performing calculations.
< stdio.h >	Contains function prototypes for the standard input/output library functions, and information used by them.
< stdlib.h >	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers and other utility functions.
< string.h >	Contains function prototypes for string-processing functions.
< time.h >	Contains function prototypes and types for manipulating the time and date.

5.9 Passing Arguments by Value and by Reference (1 of 2)

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

5.9 Passing Arguments by Value and by Reference (2 of 2)

- This prevents the accidental **side effects** (variable modifications) that so greatly hinder the development of correct and reliable software systems.
- Pass-by-reference should be used only with trusted called functions that need to modify the original variable.
- In C, all arguments are passed by value.
- In Chapter 6, we'll see that array arguments are automatically passed by reference for performance reasons.

5.10 Random Number Generation (1 of 13)

- We now take a brief and, hopefully, entertaining diversion into **simulation** and **game playing**.
- The element of chance can be introduced into computer applications by using the C standard library function `rand` from the `<stdlib.h>` header.
- Consider the following statement:

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

- The rand function generates an integer between 0 and `RAND_MAX` (a symbolic constant defined in the `<stdlib.h>` header).

5.10 Random Number Generation (2 of 13)

- Standard C states that the value of `RAND_MAX` must be at least 32767, which is the maximum value for a two-byte (i.e., 16-bit) integer.
- The programs in this section were tested on Microsoft Visual C++ with a maximum `RAND_MAX` value of 32767 and on GNU `gcc` and Xcode LLVM with a `RAND_MAX` value of 2147483647.
- If `rand` truly produces integers at random, every number between 0 and `RAND_MAX` has an equal chance (or probability) of being chosen each time `rand` is called.
- The range of values produced directly by `rand` is often different from what's needed in a specific application.

5.10 Random Number Generation (3 of 13)

- For example, a program that simulates coin tossing might require only 0 for “heads” and 1 for “tails.”
- A dice-rolling program that simulates a six-sided die would require random integers from 1 to 6.

Rolling a Six-Sided Die

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

```
rand() % 6
```

- to produce integers in the range 0 to 5.

5.10 Random Number Generation (4 of 13)

- This is called **scaling.**
- The number 6 is called the **scaling factor.**
- We then **shift** the range of numbers produced by adding 1 to our previous result.
- The output of Figure 5.7 (see slide 82) confirms that the results are in the range 1 to 6—the actual random values chosen might vary by compiler.

Figure 5.11 Shifted, Scaled Random Integers Produced By `1 + rand() % 6`

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

int main(void)
{
   // loop 20 times
   for (unsigned int i = 1; i <= 20; ++i) {

      // pick random number from 1 to 6 and output it
      printf("%10d", 1 + (rand() % 6));

      // if counter is divisible by 5, begin new line of output
      if (i % 5 == 0) {
         puts("");
      }
   }
}
```

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

5.10 Random Number Generation (5 of 13)

- Note the use of the `%s` conversion specifier to print the character strings “`Face`” and “`Frequency`” as column headers.
- After we study arrays in Chapter 6, we’ll show how to replace this `switch` statement elegantly with a single-line statement.

Figure 5.12 Rolling a Six-Sided Die 60,000,000 Times (1 of 3)

```c
// Fig. 5.12: fig05_12.c
// Rolling a six-sided die 60,000,000 times.
#include <stdio.h>
#include <stdlib.h>

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

   // loop 60000000 times and summarize results
   for (unsigned 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;
```

Figure 5.12 Rolling a Six-Sided Die 60,000,000 Times (2 of 3)

```

            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
         }
      }

```

Figure 5.12 Rolling a Six-Sided Die 60,000,000 Times (3 of 3)

```
    // display results in tabular format
    printf("%s%13s\n", "Face", "Frequency");
    printf("   1%13u\n", frequency1);
    printf("   2%13u\n", frequency2);
    printf("   3%13u\n", frequency3);
    printf("   4%13u\n", frequency4);
    printf("   5%13u\n", frequency5);
    printf("   6%13u\n", frequency6);
}
```

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

5.10 Random Number Generation (6 of 13)

Randomizing the Random Number Generator

- Executing the program of Figure 5.11 again produces exactly the same sequence of values.
- How can these be **random numbers?** Ironically, this repeatability is an important characteristic of function `rand`.

5.10 Random Number Generation (7 of 13)

- 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.

5.10 Random Number Generation (8 of 13)

- This is called **randomizing** and is accomplished with the standard library function `srand`.
- Function `srand` takes an `unsigned` integer argument and **seeds** function rand to produce a different sequence of random numbers for each execution of the program.
- We demonstrate function `srand` in Figure 5.13.

5.10 Random Number Generation (9 of 13)

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

Figure 5.13 Randomizing the Die-Rolling Program (1 of 3)

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

int main(void)
{
   unsigned int seed; // number used to seed the random number generator

   printf("%s", "Enter seed: ");
   scanf("%u", &seed); // note %u for unsigned int

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

```

Figure 5.13 Randomizing the Die-Rolling Program (2 of 3)

```
    // loop 10 times
    for (unsigned int i = 1; i <= 10; ++i) {

       // pick a random number from 1 to 6 and output it
       printf("%10d", 1 + (rand() % 6));

       // if counter is divisible by 5, begin a new line of output
       if (i % 5 == 0) {
          puts("");
       }
    }
}
```

Figure 5.13 Randomizing the Die-Rolling Program (3 of 3)

```
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
```

5.10 Random Number Generation (10 of 13)

- Let's run the program several times and observe the results.
- Notice that a different sequence of random numbers is obtained each time the program is run, provided that a different seed is supplied.
- To randomize without entering a seed each time, use a statement like

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

- This causes the computer to read its clock to obtain the value for the seed automatically.
- Function `time` returns the number of seconds that have passed since midnight on January 1, 1970.

5.10 Random Number Generation (11 of 13)

- This value is converted to an unsigned integer and used as the seed to the random number generator.
- The function prototype for `time` is in `<time.h>`.

5.10 Random Number Generation (12 of 13)

Generalized Scaling and Shifting of Random Numbers

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

```
0 ≤ rand() ≤ RAND_MAX
```

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

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

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

5.10 Random Number Generation (14 of 14)

- Referring to the preceding statement, we see that the width of the range is determined by the number used to **scale** `rand` with the **remainder operator** (i.e., 6), and the **starting number** of the range is equal to the number (i.e., 1) that's added to `rand % 6`.
- We can generalize this result as follows

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

- where `a` is the **shifting value** (which is equal to the **first** number in the desired range of consecutive integers) and `b` is the **scaling factor** (which is equal to the **width** of the desired range of consecutive integers).

5.11 Example: A Game of Chance; Introducing Enum (1 of 9)

- One of the most popular games of chance is a dice game known as “craps.” The rules of the game are simple.
  - A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5, and 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3, or 12 on the first throw (called “craps”), the player loses (i.e., the “house” wins). If the sum is 4, 5, 6, 8, 9, or 10 on the first throw, then that sum becomes the player’s “point.” To win, you must continue rolling the dice until you “make your point.” The player loses by rolling a 7 before making the point.
- Figure 5.14 simulates the game of craps and Figure 5.15 shows several sample executions.

Figure 5.14 Simulating the Game of Craps (1 of 4)

```c
// Fig. 5.14: fig05_14.c
// Simulating the game of craps.
#include <stdio.h>
#include <stdlib.h>
#include <time.h> // contains prototype for function time

// enumeration constants represent game status
enum Status { CONTINUE, WON, LOST };

int rollDice(void); // function prototype

int main(void)
{
   // randomize random number generator using current time
   srand(time(NULL));

   int myPoint; // player must make this point to win
   enum Status gameStatus; // can contain CONTINUE, WON, or LOST
   int sum = rollDice(); // first roll of the dice

```

Figure 5.14 Simulating the Game of Craps (2 of 4)

```
    // determine game status based on sum of dice
    switch(sum) {

       // win on first roll
       case 7: // 7 is a winner
       case 11: // 11 is a winner
          gameStatus = WON;
          break;

       // lose on first roll
       case 2: // 2 is a loser
       case 3: // 3 is a loser
       case 12: // 12 is a loser
          gameStatus = LOST;
          break;

       // remember point
       default:
          gameStatus = CONTINUE; // player should keep rolling
          myPoint = sum; // remember the point
          printf("Point is %d\n", myPoint);
          break; // optional
    }

```

Figure 5.14 Simulating the Game of Craps (3 of 4)

```c
   // while game not complete
   while (CONTINUE == gameStatus) { // player should keep rolling
      sum = rollDice(); // roll dice again

      // determine game status
      if (sum == myPoint) { // win by making point
         gameStatus = WON;
      }
      else {
         if (7 == sum) { // lose by rolling 7
            gameStatus = LOST;
         }
      }
   }

   // display won or lost message
   if (WON == gameStatus) { // did player win?
      puts("Player wins");
   }
   else { // player lost
      puts("Player loses");
   }
}

```

Figure 5.14 Simulating the Game of Craps (4 of 4)

```
// roll dice, calculate sum and display results
int rollDice(void)
{
   int die1 = 1 + (rand() % 6); // pick random die1 value
   int die2 = 1 + (rand() % 6); // pick random die2 value

   // display results of this roll
   printf("Player rolled %d + %d = %d\n", die1, die2, die1 + die2);
   return die1 + die2; // return sum of dice
}
```

Figure 5.15 Sample Runs for the Game of Craps (1 of 2)

Player wins on the first roll

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

Player wins on a subsequent roll

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

Figure 5.15 Sample Runs for the Game of Craps (2 of 2)

Player loses on the first roll

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

Player loses on a subsequent roll

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

5.11 Example: A Game of Chance; Introducing Enum (2 of 9)

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

5.11 Example: A Game of Chance; Introducing Enum (3 of 9)

- The player may win or lose on the first roll, or may win or lose on any subsequent roll.
- Variable `gameStatus,` defined to be of a new type—`enum Status`—stores the current status.
- An **enumeration,** introduced by the keyword **enum,** is a set of integer constants represented by identifiers.
- **Enumeration constants** help make programs easier to read
- Values in an `enum` start with `0` and are incremented by `1`.

5.11 Example: A Game of Chance; Introducing Enum (4 of 9)

Enumerations

- The constant `CONTINUE` has the value `0`, `WON` has the value `1` and `LOST` has the value `2`.
- It’s also possible to assign an integer value to each identifier in an `enum` (see Chapter 10).
- The identifiers in an enumeration must be unique, but the values may be duplicated.

5.11 Example: A Game of Chance; Introducing Enum (5 of 9)

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

5.11 Example: A Game of Chance; Introducing Enum (6 of 9)

Game Ends on First Roll

- After the first roll, if the game is over, the while statement is skipped because `gameStatus` is not `CONTINUE`.
- The program proceeds to the `if…else` statement, which prints `"Player wins"` if `gameStatus` is `WON` and `"Player loses"` otherwise.

5.11 Example: A Game of Chance; Introducing Enum (7 of 9)

Game Ends on a Subsequent Roll

- After the first roll, if the game is not over, then `sum` is saved in `myPoint.`
- Execution proceeds with the while statement because `gameStatus` is `CONTINUE.`
- Each time through the while, `rollDice` is called to produce a new `sum.`
- If sum matches `myPoint, gameStatus` is set to `WON` to indicate that the player won, the `while`-test fails, the `if…else` statement prints `"Player wins"` and execution terminates.

5.11 Example: A Game of Chance; Introducing Enum (8 of 9)

- If `sum` is equal `to 7, gameStatus` is set to `LOST` to indicate that the player lost, the `while`-test fails, the `if`…`else` statement prints `"Player loses"` and execution terminates.

5.11 Example: A Game of Chance; Introducing Enum (9 of 9)

Control Architecture

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

5.12 Storage Classes (1 of 10)

- 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`, `extern` and **static.**
- An identifier's **storage class** determines its storage duration, scope and linkage.
- An identifier's **storage duration** is the period during which the identifier exists **in memory**.

5.12 Storage Classes (2 of 10)

- Some exist briefly, some are repeatedly created and destroyed, and others exist for the program's entire execution.
- An identifier's **scope** is where the identifier can be referenced in a program.
- Some can be referenced throughout a program, others from only portions of a program.
- An identifier's **linkage** determines for a multiple-source-file program whether the identifier is known only in the current source file or in any source file with proper declarations.
- This section discusses storage classes and storage duration.

5.12 Storage Classes (3 of 10)

- Section 5.13 discusses scope.
- Chapter 14 discusses identifier linkage and programming with multiple source files.
- The storage-class specifiers can be split into **automatic storage duration** and **static storage duration.**
- Keyword `auto` is used to declare variables of automatic storage duration.
- Variables with automatic storage duration are created when the block in which they're defined is entered; they exist while the block is active, and they're destroyed when the block is exited.

5.12 Storage Classes (4 of 10)

Local Variables

- Only variables can have automatic storage duration.
- A function's local variables (those declared in the parameter list or function body) normally have automatic storage duration.
- Keyword `auto` explicitly declares variables of automatic storage duration.

5.12 Storage Classes (5 of 10)

- Local variables have automatic storage duration by **default**, so keyword `auto` is rarely used.
- For the remainder of the text, we'll refer to variables with automatic storage duration simply as **automatic variables.**

5.12 Storage Classes (6 of 10)

Static Storage Class

- Keywords `extern` and `static` are used in the declarations of identifiers for variables and functions of static storage duration.
- Identifiers of static storage duration exist from the time at which the program begins execution until the program terminates.
- For static variables, storage is allocated and initialized **only once, before** the program begins execution.
- For functions, the name of the function exists when the program begins execution.

5.12 Storage Classes (7 of 10)

- However, even though the variables and the function names exist from the start of program execution, this does not mean that these identifiers can be accessed throughout the program.
- Storage duration and scope (where a name can be used) are separate issues, as we'll see in Section 5.13.
- There are several types of identifiers with static storage duration: **external identifiers** (such as global variables and function names) and local variables declared with the storage-class specifier `static`.
- Global variables and function names are of storage class `extern` by default.

5.12 Storage Classes (8 of 10)

- Global variables are created by placing variable declarations **outside** any function definition, and they retain their values throughout the execution of the program.
- Global variables and functions can be referenced by any function that follows their declarations 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.

5.12 Storage Classes (9 of 10)

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

5.12 Storage Classes (10 of 10)

- All numeric variables of static storage duration are initialized to zero by default if you do not explicitly initialize them.
- Keywords `extern` and `static` have special meaning when explicitly applied to external identifiers.
- In Chapter 14 we discuss the explicit use of `extern` and `static` with external identifiers and multiple-source-file programs.

5.13 Scope Rules (1 of 9)

- The **scope of an identifier** is the portion of the program in which the identifier can be referenced.
- For example, when we define a local variable in a block, it can be referenced only following its definition in that block or in blocks nested within that block.
- The four identifier scopes are **function scope, file scope, block scope,** and **function-prototype scope.**
- Labels (identifiers followed by a colon such as `start:`) are the only identifiers with **function scope.**
- Labels can be used anywhere in the function in which they appear, but cannot be referenced outside the function body.

5.13 Scope Rules (2 of 9)

- Labels are used in `switch` statements (as `case` labels) and in `goto` statements (see Chapter 14).
- Labels are hidden in the function in which they're defined.
- This hiding—more formally called **information hiding**—is a means of implementing the **principle of least privilege**—a fundamental principle of good software engineering.
- 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.

5.13 Scope Rules (3 of 9)

- Global variables, function definitions, and function prototypes placed outside a function all have file scope.
- Identifiers defined inside a block have **block scope.**
- Block scope ends at the terminating right brace (}) of the block.
- Local variables defined at the beginning of a function have block scope as do function parameters, which are considered local variables by the function.
- Any block may contain variable definitions.

5.13 Scope Rules (4 of 9)

- When blocks are nested, and an identifier in an outer block has the same name as an identifier in an inner block, the identifier in the outer block is “hidden” until the inner block terminates.
- This means that while executing in the inner block, the inner block sees the value of its own local identifier and not the value of the identically named identifier in the enclosing block.
- Local variables declared `static` still have block scope, even though they exist from before program startup.

5.13 Scope Rules (5 of 9)

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

5.13 Scope Rules (6 of 9)

- Figure 5.16 demonstrates scoping issues with global variables, automatic local variables, and `static` local variables.
- A global variable `x` is defined and initialized to `1`.
- 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
- 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

5.13 Scope Rules (7 of 9)

- 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.
- The program defines three functions that each take no arguments and return nothing.
- Function `useLocal` defines an automatic variable `x` and initializes it to 25
- When `useLocal` is called, the variable is printed, incremented, and printed again before exiting the function.

5.13 Scope Rules (8 of 9)

- Each time this function is called, automatic variable `x` is reinitialized to 25.
- Function `useStaticLocal` defines a static variable `x` and initializes it to 50
- Local variables declared as `static` retain their values even when they're out of scope.
- When `useStaticLocal` is called, `x` is printed, incremented, and printed again before exiting the function.
- In the next call to this function, `static` local variable `x` will contain the value 51.
- Function `useGlobal` does not define any variables.

5.13 Scope Rules (9 of 9)

- Therefore, when it refers to variable `x`, the global `x` 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 to show that none of the function calls modified the value of `x` because the functions all referred to variables in other scopes.

Figure 5.16 Scoping (1 of 4)

```c
// Fig. 5.16: fig05_16.c
// Scoping.
#include <stdio.h>

void useLocal(void); // function prototype
void useStaticLocal(void); // function prototype
void useGlobal(void); // function prototype

int x = 1; // global variable

int main(void)
{
   int x = 5; // local variable to main

   printf("local x in outer scope of main is %d\n", x);

   { // start new scope
      int x = 7; // local variable to new scope

      printf("local x in inner scope of main is %d\n", x);
   } // end new scope

   printf("local x in outer scope of main is %d\n", x);

```

Figure 5.16 Scoping (2 of 4)

```c

    useLocal(); // useLocal has automatic local x
    useStaticLocal(); // useStaticLocal has static local x
    useGlobal(); // useGlobal uses global x
    useLocal(); // useLocal reinitializes automatic local x
    useStaticLocal(); // static local x retains its prior value
    useGlobal(); // global x also retains its value

    printf("\nlocal x in main is %d\n", x);
}

// useLocal reinitializes local variable x during each call
void useLocal(void)
{
    int x = 25; // initialized each time useLocal is called

    printf("\nlocal x in useLocal is %d after entering useLocal\n", x);
    ++x;
    printf("local x in useLocal is %d before exiting useLocal\n", x);
}

```

Figure 5.16 Scoping (3 of 4)

```
// useStaticLocal initializes static local variable x only the first time
// the function is called; value of x is saved between calls to this
// function
void useStaticLocal(void)
{
    // initialized once
    static int x = 50;

    printf("\nlocal static x is %d on entering useStaticLocal\n", x);
    ++x;
    printf("local static x is %d on exiting useStaticLocal\n", x);
}

// function useGlobal modifies global variable x during each call
void useGlobal(void)
{
    printf("\nglobal x is %d on entering useGlobal\n", x);
    x *= 10;
    printf("global x is %d on exiting useGlobal\n", x);
}
```

Figure 5.16 Scoping (4 of 4)

```
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
```

5.14 Recursion (1 of 12)

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

5.14 Recursion (2 of 12)

- This book contains an extensive treatment of recursion, which is spread throughout Chapters 5–8, 12 and Appendix F.
- Figure 5.17, in Section 5.16, summarizes the 31 recursion examples and exercises in the book.
- We consider recursion conceptually first, then examine several programs containing recursive functions.
- Recursive problem-solving approaches have a number of elements in common.
- A recursive function is called to solve a problem.
- The function actually knows how to solve only the simplest case(s), or so-called **base case(s).**

5.14 Recursion (3 of 12)

- If the function is called with a base case, the function simply returns a result.
- If the function is called with a more complex problem, the function divides the problem into two conceptual pieces: a piece that the function knows how to do and a piece that it does not know how to do.
- To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or smaller version.

5.14 Recursion (4 of 12)

- Because this new problem looks like the original problem, the function launches (calls) a fresh copy of itself to go to work on the smaller problem—this is referred to as a **recursive call** or the **recursion step.**
- The recursion step also includes the keyword `return`, because its result will be combined with the portion of the problem the function knew how to solve to form a result that will be passed back to the original caller.
- The recursion step executes while the original call to the function is paused, waiting for the result from the recursion step.

5.14 Recursion (5 of 12)

- The recursion step can result in many more such recursive calls, as the function keeps dividing each problem it's called with into two conceptual pieces.
- For the recursion to terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller problems must eventually converge on the base case.
- When the function recognizes the base case, returns a result to the previous copy of the function, and a sequence of returns ensues all the way up the line until the original call of the function eventually returns the final result to `main`.

5.14 Recursion (6 of 12)

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 (non recursively) using a `for` statement as follows:

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

5.14 Recursion (7 of 12)

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

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

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

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

- The evaluation of $5!$ would proceed as shown in Figure 5.17.

5.14 Recursion (8 of 12)

- Figure 5.17(a) shows how the succession of recursive calls proceeds until $1!$ is evaluated to be 1 (i.e., the **base** case), which terminates the recursion.
- Figure 5.17(b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.
- Figure 5.18 uses recursion to calculate and print the factorials of the integers 0–10 (the choice of the type `unsigned long long int` will be explained momentarily).

5.14 Recursion (9 of 12)

- 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)`.

5.14 Recursion (10 of 12)

- Function `factorial` has been declared to receive a parameter of type `long` and return a result of type `long`.
- This is shorthand notation for `long int`.
- The C standard specifies that a variable of type `long int` is stored in at least 4 bytes, and thus may hold a value as large as +2147483647.
- As can be seen in Figure 5.14, factorial values become large quickly.
- We've chosen the data type `long` so the program can calculate factorials greater than 7! on computers with small (such as 2-byte) integers.

Figure 5.17 Recursive Evaluation of 5!

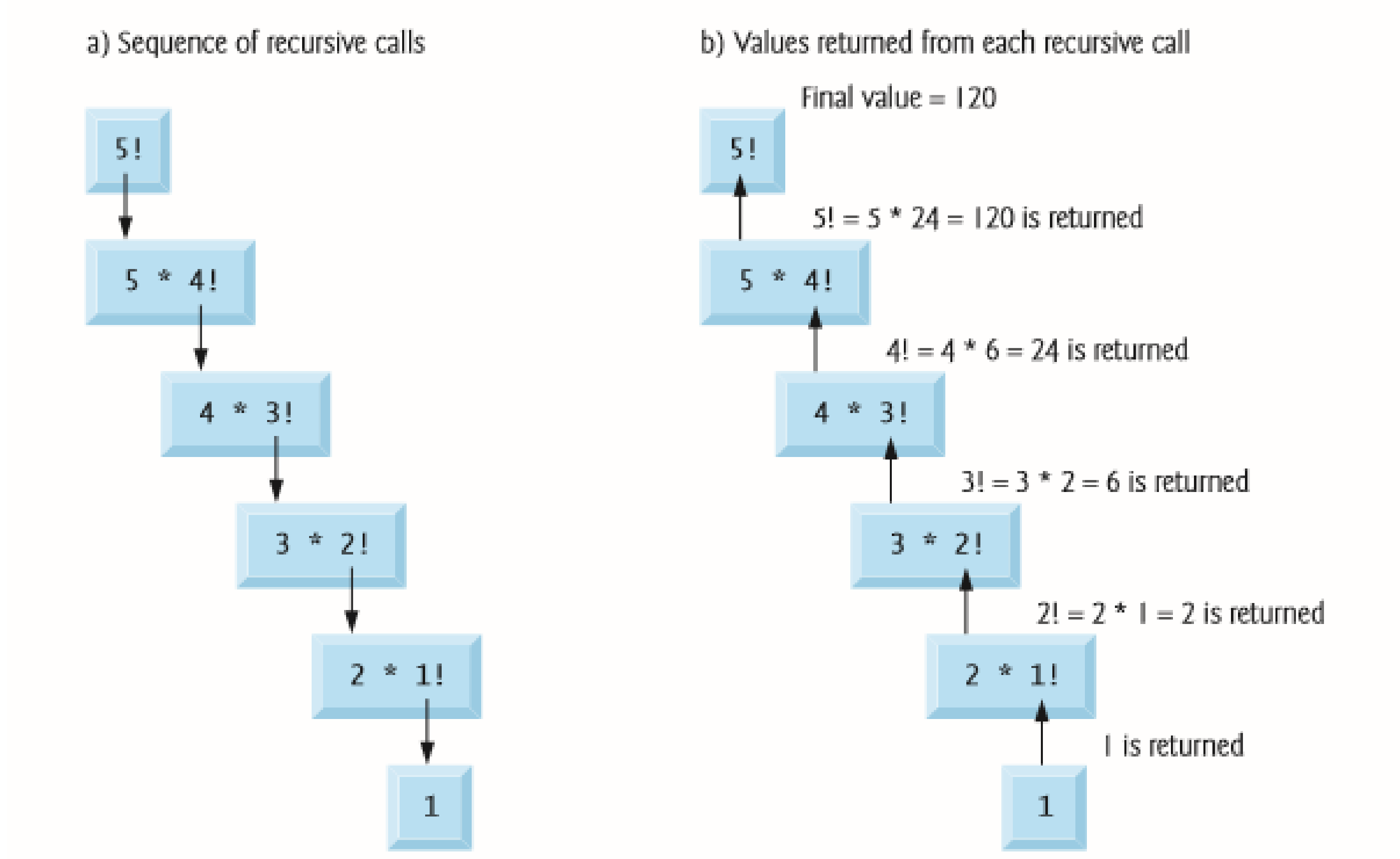


Figure 5.18 Recursive Factorial Function (1 of 3)

```c
// Fig. 5.18: fig05_18.c
// Recursive factorial function.
#include <stdio.h>

unsigned long long int factorial(unsigned int number);

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

```

Figure 5.18 Recursive Factorial Function (2 of 3)

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

Figure 5.18 Recursive Factorial Function (3 of 3)

```
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
```

5.14 Recursion (11 of 12)

- The recursive `factorial` function first tests whether a **terminating condition** is true, i.e., whether `number` is less than or equal to `1`.
- The conversion specifier `%llu` is used to print `unsigned long long int` values.
- Unfortunately, the `factorial` function produces large values so quickly that even `unsigned long long int` does not help us print very many factorial values before the maximum value of an `unsigned long long int` variable is exceeded.
- Even when we use `unsigned long long int`, we still can’t calculate factorials beyond `21`!

5.14 Recursion (12 of 12)

- This points to a weakness in C (and most other procedural programming languages)—namely that the language is not easily **extended** to handle the unique requirements of various applications.
- As we'll see later in the book, C++ is an **extensible** language that, through "classes," allows us to create new data types, including ones that could hold arbitrarily large integers if we wish.

Common Programming Error 5.9

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

Common Programming Error 5.10

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

5.15 Example Using Recursion: Fibonacci Series (1 of 11)

- 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….

5.15 Example Using Recursion: Fibonacci Series (2 of 11)

- 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.

5.15 Example Using Recursion: Fibonacci Series (3 of 11)

- The Fibonacci series may be defined recursively as follows:

$$\text{fibonacci}(0) = 0$$
$$\text{fibonacci}(1) = 1$$
$$\text{fibonacci}(n) = \textit{fibonacci}(n - 1) + \textit{fibonacci}(n - 2)$$

- Figure 5.19 calculates the n^{th} Fibonacci number recursively using function `fibonacci`.
- Notice that Fibonacci numbers tend to become large quickly.
- Therefore, we've chosen the data type `unsigned int` for the parameter type and the data type `unsigned long long int` for the return type in function `fibonacci`.
- In Figure 5.19, each pair of output lines shows a separate run of the program.

Figure 5.19 Recursive Fibonacci Function (1 of 3)

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

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

int main(void)
{
   unsigned int number; // number input by user

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

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

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

```

Figure 5.19 Recursive Fibonacci Function (2 of 3)

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

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

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

Figure 5.19 Recursive Fibonacci Function (3 of 3)

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

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

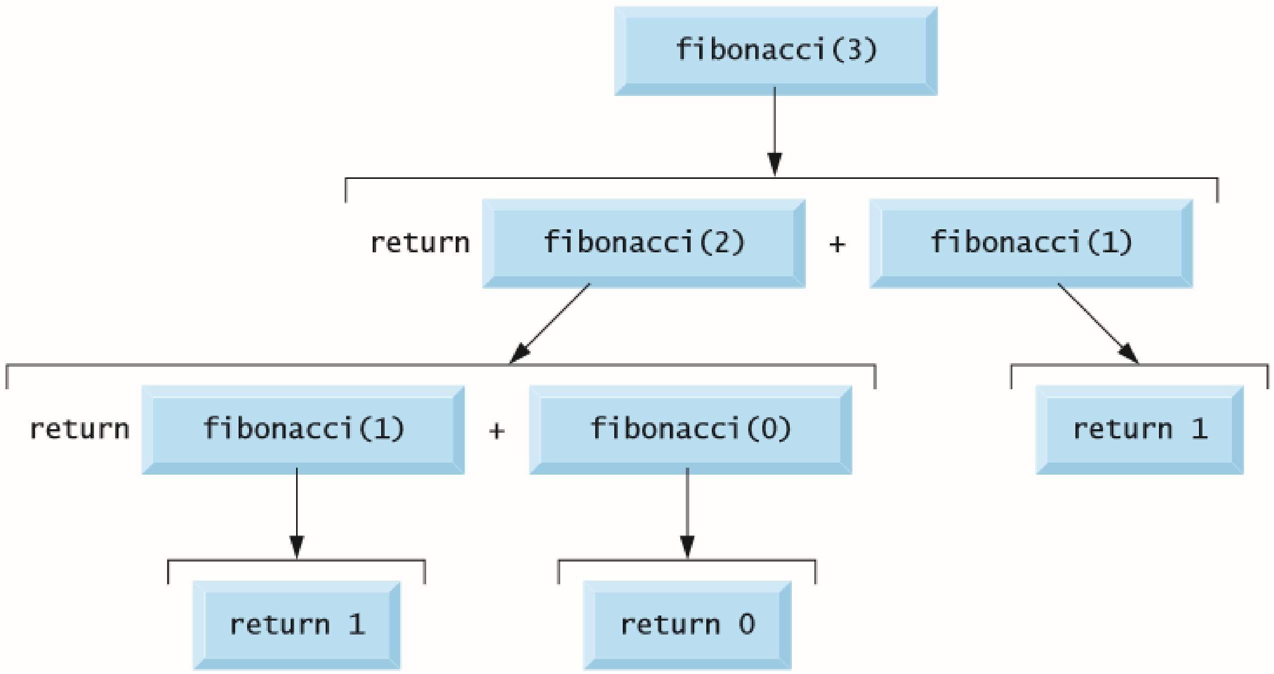
```
Enter an integer: 10
Fibonacci(10) = 55
```

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

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

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

Figure 5.20 Set of Recursive Calls for Fibonacci(3)



5.15 Example Using Recursion: Fibonacci Series (5 of 11)

Order of Evaluation of Operands

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

5.15 Example Using Recursion: Fibonacci Series (6 of 11)

- 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 in fact execute `fibonacci(2)` first and then `fibonacci(1)`, or the calls could execute in the reverse order, `fibonacci(1)` then `fibonacci(2)`.
- In this program and in most other programs, the final result would be the same.

5.15 Example Using Recursion: Fibonacci Series (7 of 11)

- But in some programs the evaluation of an operand may have side effects that could affect the final result of the expression.
- C specifies the order of evaluation of the operands of only four operators—namely &&, ||, the comma (,) operator and ?:.
- The first three of these are binary operators whose operands are guaranteed to be evaluated left to right.

5.15 Example Using Recursion: Fibonacci Series (8 of 11)

- [**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, 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.

5.15 Example Using Recursion: Fibonacci Series (9 of 11)

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 will be executed 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.

5.15 Example Using Recursion: Fibonacci Series (10 of 11)

- Computer scientists refer to this as exponential complexity.
- Problems of this nature 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 curriculum course generally called “Algorithms.”

5.15 Example Using Recursion: Fibonacci Series (11 of 11)

- 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.

5.16 Recursion vs. Iteration (1 of 6)

- Both iteration and recursion are based on a control statement: Iteration uses a repetition statement; recursion uses a **selection statement**.
- Both iteration and recursion involve repetition: Iteration explicitly uses a repetition 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 when a **base case is recognized**.

5.16 Recursion vs. Iteration (2 of 6)

- Iteration with counter-controlled repetition and recursion each **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.

5.16 Recursion vs. Iteration (3 of 6)

- 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.

5.16 Recursion vs. Iteration (4 of 6)

- 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?

Software Engineering Observation 5.12

Any problem that can be solved recursively can also be solved iteratively (non recursively). A recursive approach is normally 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.

Figure 5.21 Recursion Examples and Exercises in the Text (1 of 2)

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

Figure 5.21 Recursion Examples and Exercises in the Text (2 of 2)

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

Appendix D

Selection sort

Quick sort

Appendix E

Fibonacci function.

5.16 Recursion vs. Iteration (6 of 6)

- Good software engineering is important.
- High performance is important.
- 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?

Performance Tip 5.2

Dividing a large program into functions promotes good software engineering. But it has a price. A heavily functionalized program—as compared to a monolithic (i.e., one-piece) program without functions—makes potentially large numbers of function calls, and 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.

Copyright