02_05_C Functions

Objectives

In this chapter, you'll:

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

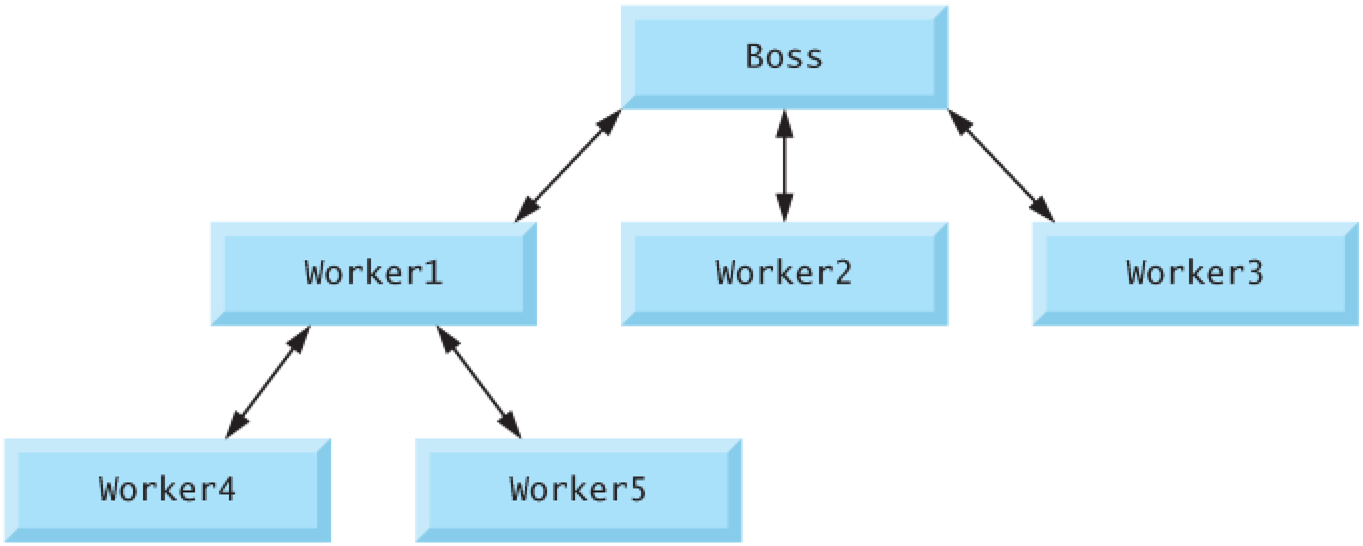

Fig. 5.1 | Hierarchical boss-function/worker-function relationship.

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`

Fig. 5.2 | Commonly used math library functions. (Part 1 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 (x^y)	`pow(2, 7)` is `128.0` `pow(9, .5)` is `3.0`
`fmod(x, y)`	remainder of x/y as a floating-point number	`fmod(13.657, 2.333)` is `1.992`
`sin(x)`	trigonometric sine of x (x in radians)	`sin(0.0)` is `0.0`
`cos(x)`	trigonometric cosine of x (x in radians)	`cos(0.0)` is `1.0`
`tan(x)`	trigonometric tangent of x (x in radians)	`tan(0.0)` is `0.0`

Fig. 5.2 | Commonly used math library functions. (Part 2 of 2.)

```c
// 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
```

Fig. 5.3 | Creating and using a programmer-defined function.

5.5 Function Definitions (Cont.)

Function `maximum`

- Our second example uses a programmer-defined function `maximum` to determine and return the largest of three integers (Fig. 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`.

```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));
}

```

Fig. 5.4 | Finding the maximum of three integers. (Part 1 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
}
```

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

Fig. 5.4 | Finding the maximum of three integers. (Part 3 of 3.)

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

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

```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
```

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

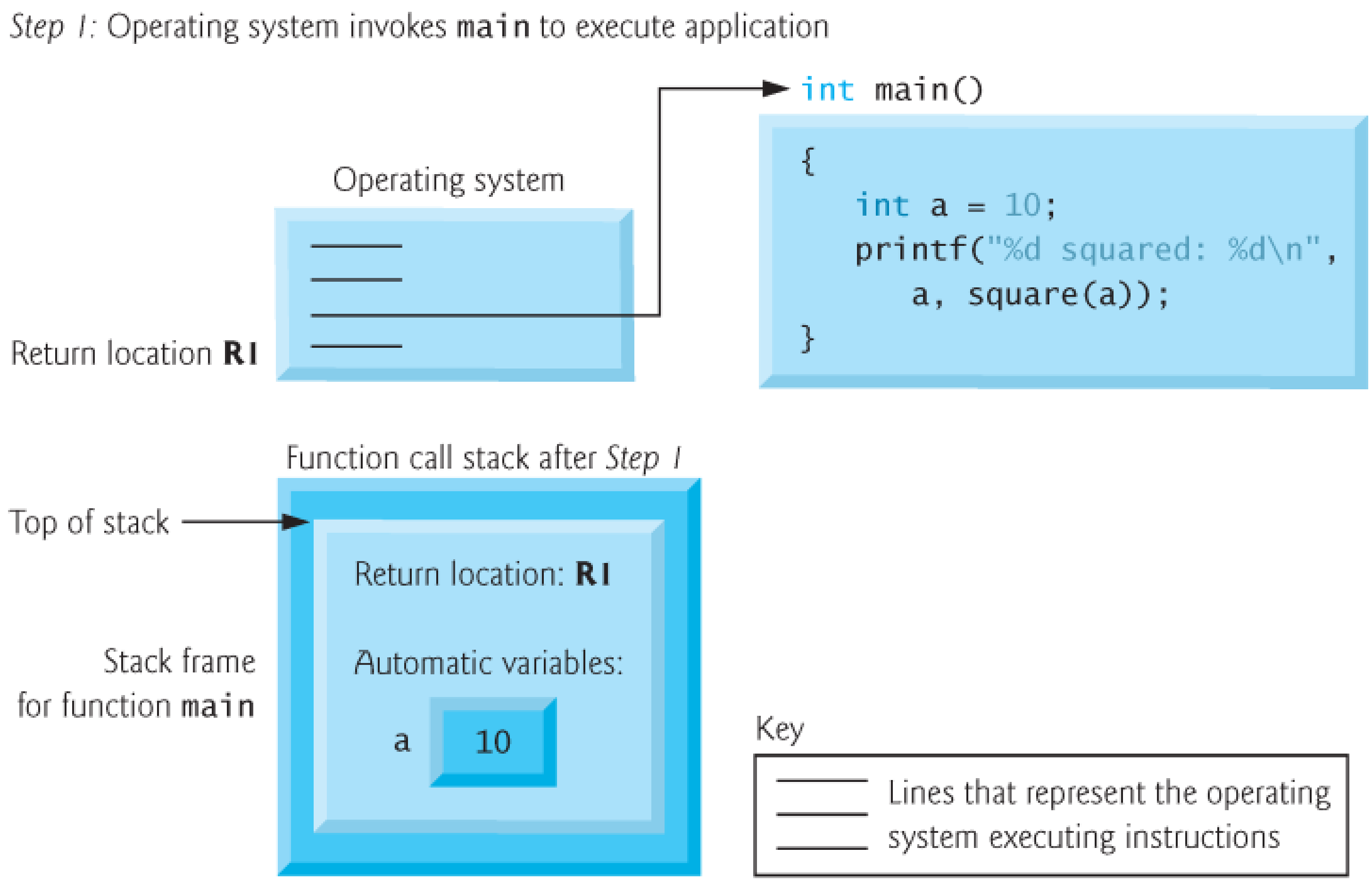

Fig. 5.7 | Function call stack after the operating system invokes `main` to execute the program.

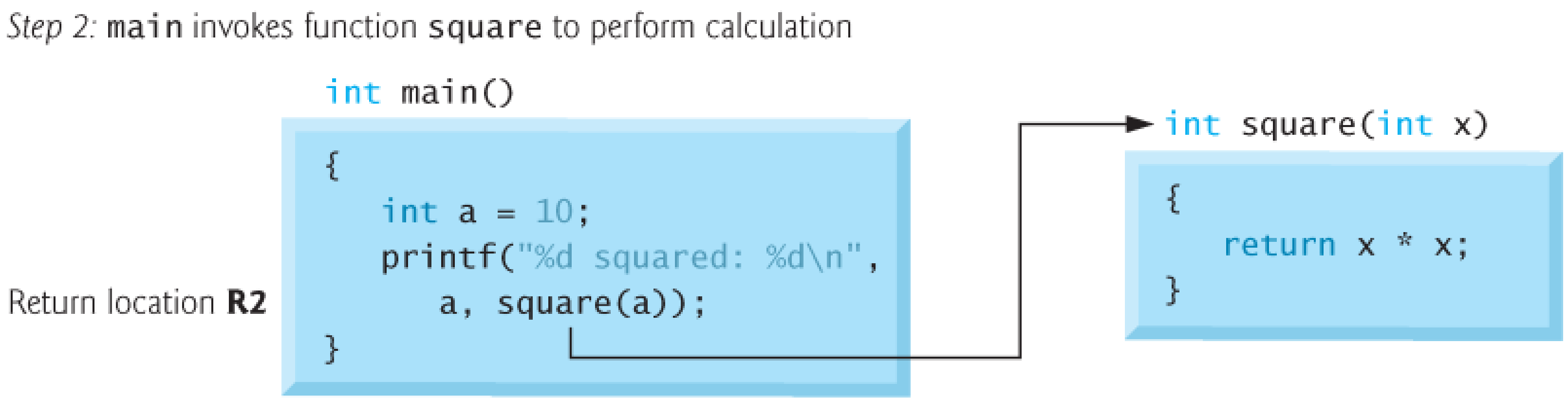

Fig. 5.8 | Function call stack after main invokes square to perform the calculation. (Part 1 of 2.)

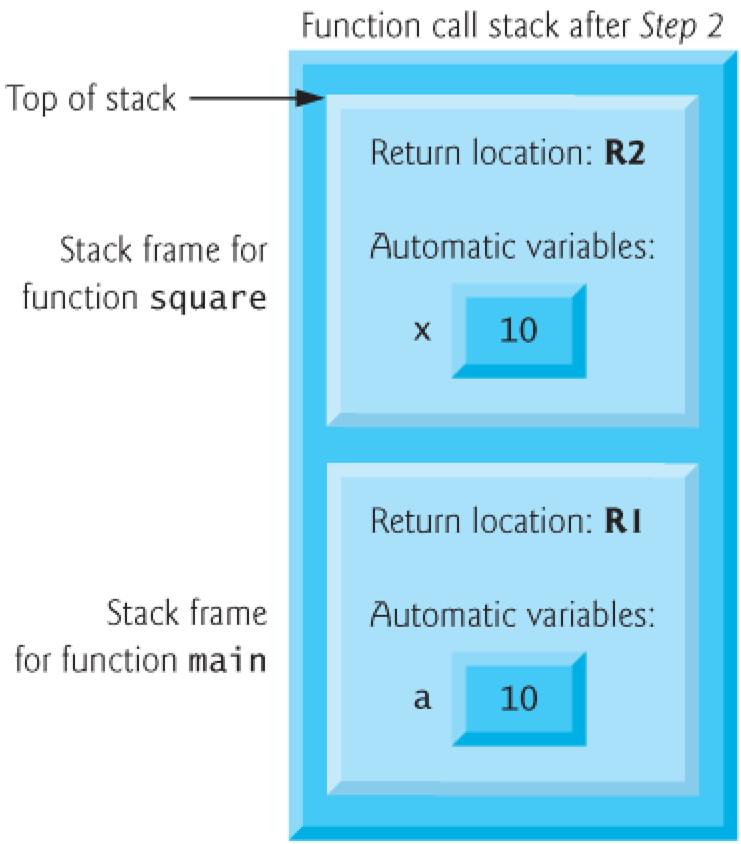

Fig. 5.8 | Function call stack after `main` invokes `square` to perform the calculation. (Part 2 of 2.)

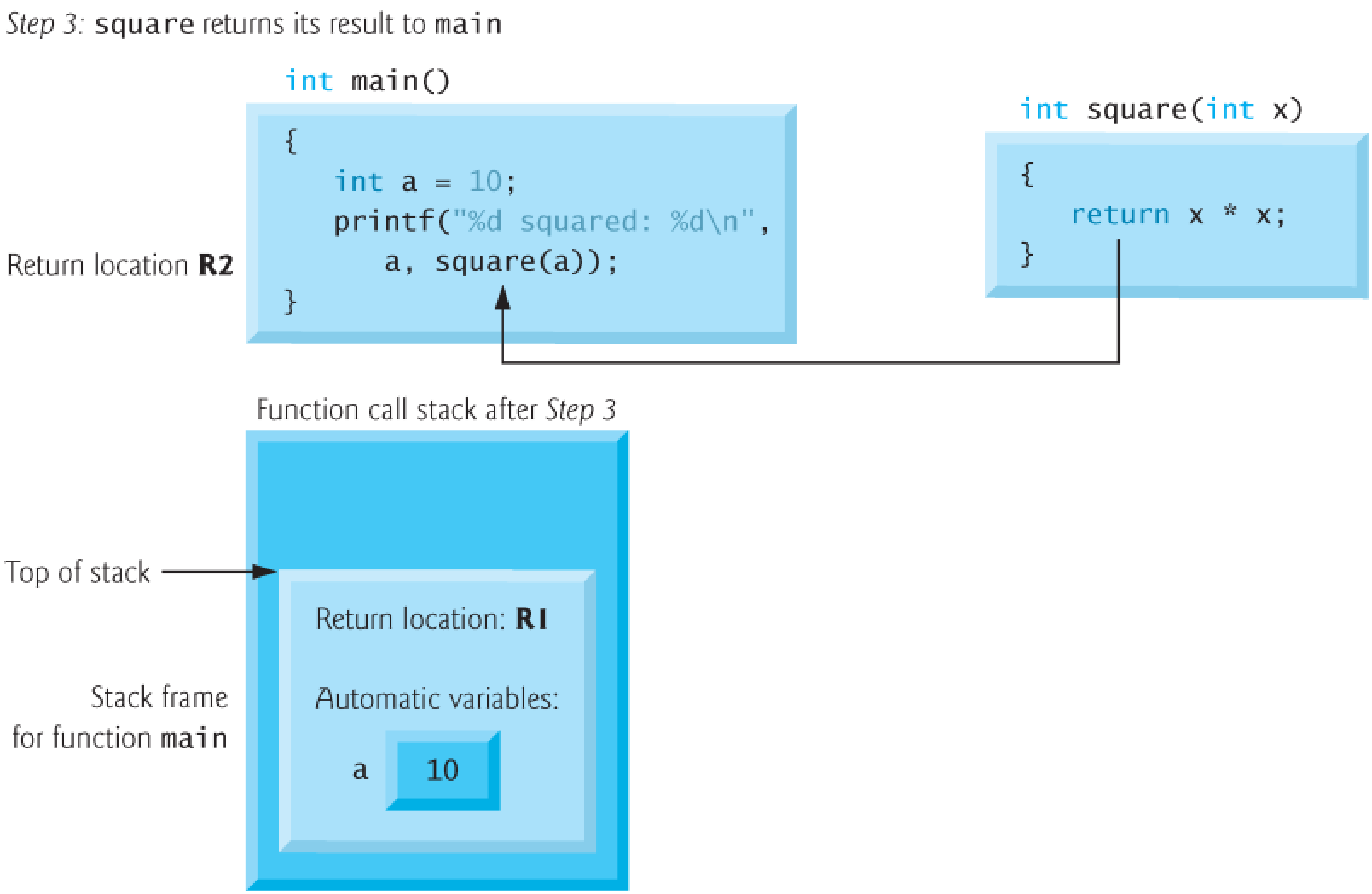

Fig. 5.9 | Function call stack after function `square` returns to `main`.

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

Fig. 5.10 | Some of the standard library headers. (Part 1 of 2.)

Header	Explanation
`<stdarg.h>`	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
`<stddef.h>`	Contains common type definitions used by C for performing calculations.
`<stdio.h>`	Contains function prototypes for the standard input/output library functions, and information used by them.
`<stdlib.h>`	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers and other utility functions.
`<string.h>`	Contains function prototypes for string-processing functions.
`<time.h>`	Contains function prototypes and types for manipulating the time and date.

Fig. 5.10 | Some of the standard library headers. (Part 2 of 2.)

```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
```

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

```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;
```

Fig. 5.12 | Rolling a six-sided die 60,000,000 times. (Part 1 of 3.)

```

            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
         }
      }

```

Fig. 5.12 | Rolling a six-sided die 60,000,000 times. (Part 2 of 3.)

```c
   // 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
```

Fig. 5.12 | Rolling a six-sided die 60,000,000 times. (Part 3 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

```

Fig. 5.13 | Randomizing the die-rolling program. (Part 1 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("");
       }
    }
}
```

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

Fig. 5.13 | Randomizing the die-rolling program. (Part 3 of 3.)

5.11 Example: A Game of Chance; Introducing enum

- 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 Fig. 5.15 shows several sample executions.

```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

```

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

```

Fig. 5.14 | Simulating the game of craps. (Part 2 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");
   }
}

```

Fig. 5.14 | Simulating the game of craps. (Part 3 of 4.)

```c
// 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
}
```

Fig. 5.14 | Simulating the game of craps. (Part 4 of 4.)

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
```

Fig. 5.15 | Sample runs for the game of craps. (Part 1 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
```

Fig. 5.15 | Sample runs for the game of craps. (Part 2 of 2.)

```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);

```

Fig. 5.16 | Scoping. (Part 1 of 4.)

```
   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);
}

```

Fig. 5.16 | Scoping. (Part 2 of 4.)

```c
// 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);
}
```

Fig. 5.16 | Scoping. (Part 3 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
```

Fig. 5.16 | Scoping. (Part 4 of 4.)

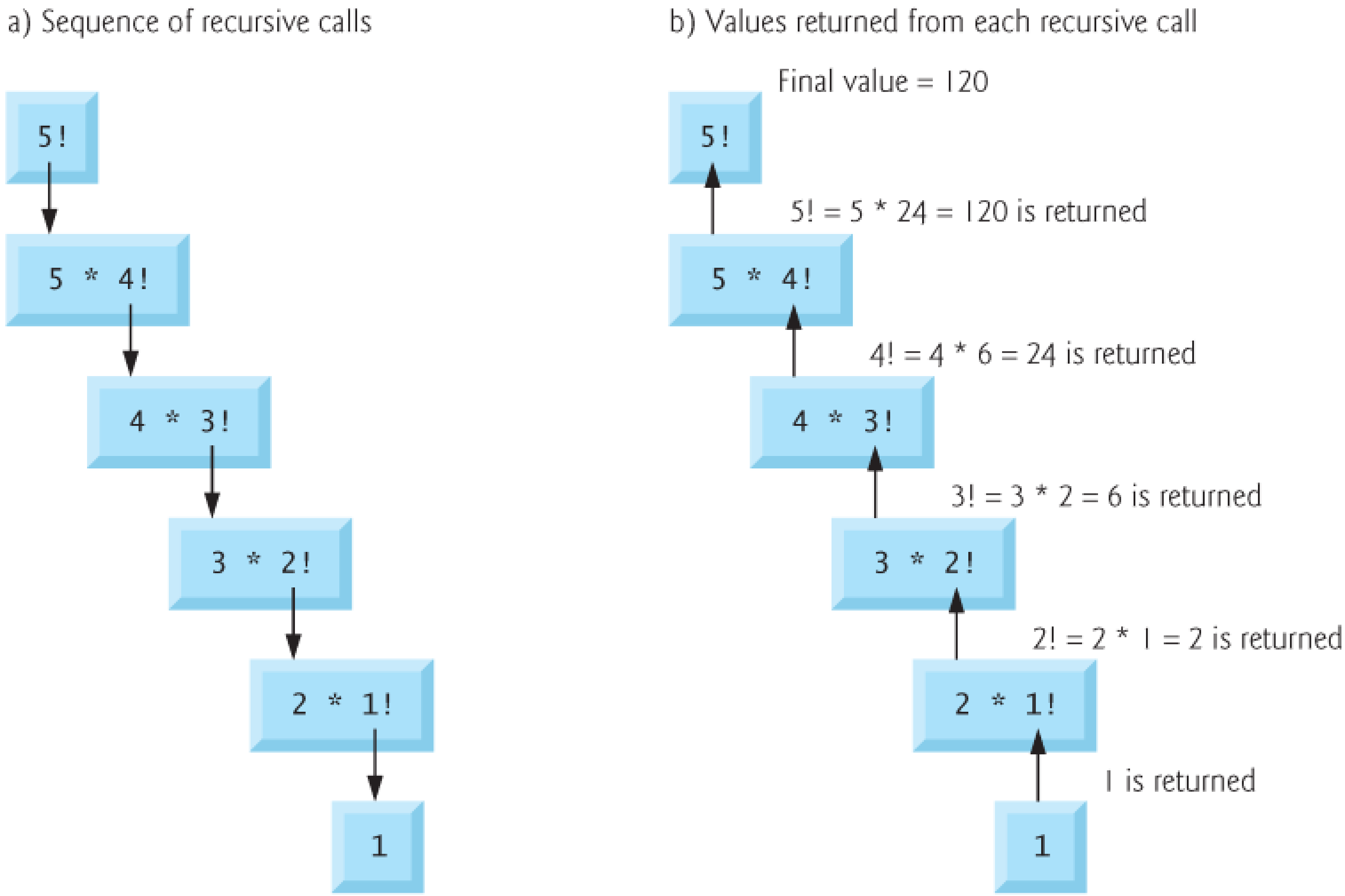

Fig. 5.17 | Recursive evaluation of 5!.

```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));
   }
}

```

Fig. 5.18 | Recursive factorial function. (Part 1 of 3.)

```
// 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));
   }
}
```

Fig. 5.18 | Recursive factorial function. (Part 2 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
```

Fig. 5.18 | Recursive factorial function. (Part 3 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);
}

```

Fig. 5.19 | Recursive fibonacci function. (Part 1 of 3.)

```cpp
// 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
```

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

Fig. 5.19 | Recursive fibonacci function. (Part 2 of 3.)

```
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
```

Fig. 5.19 | Recursive fibonacci function. (Part 3 of 3.)

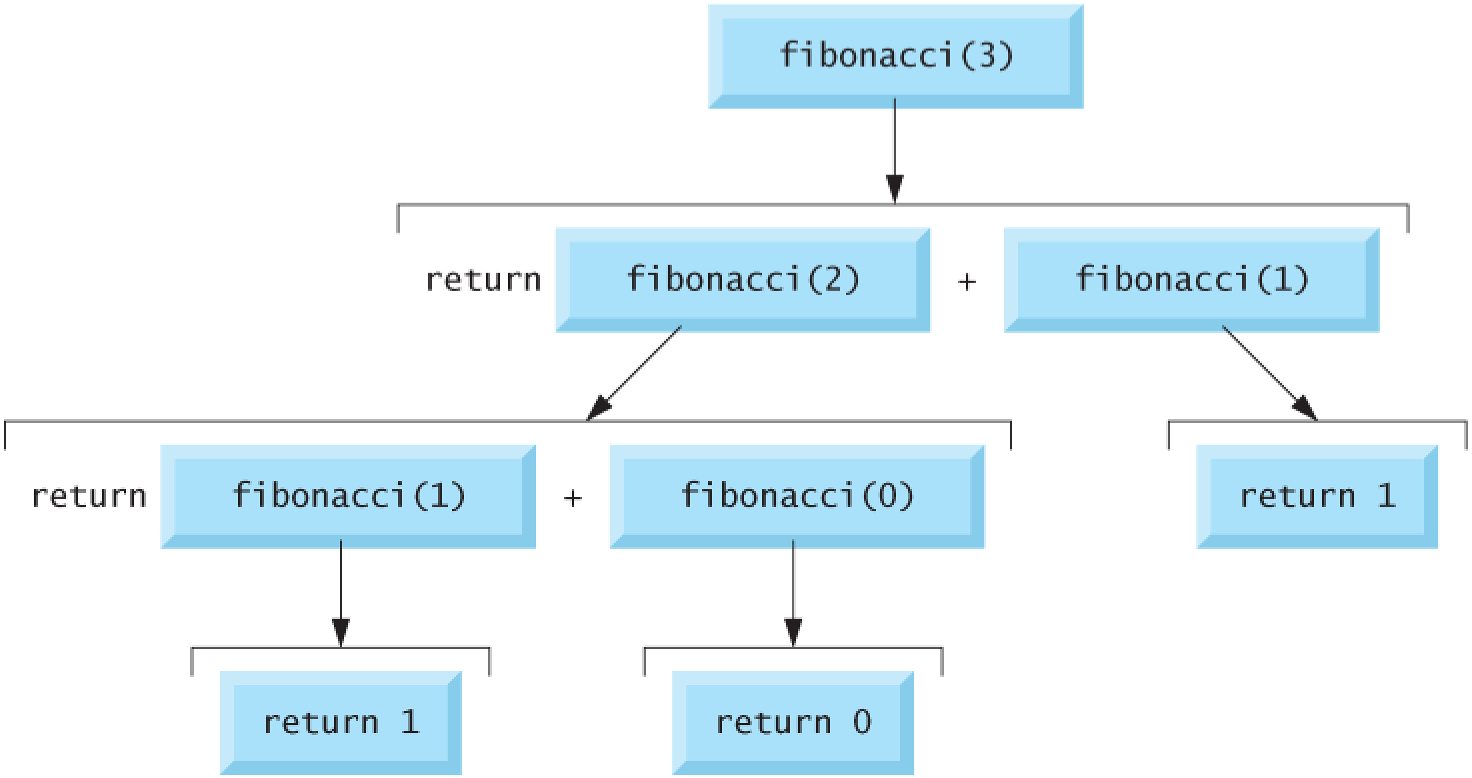

Fig. 5.20 | Set of recursive calls for `fibonacci(3)`.

5.15 Example Using Recursion: Fibonacci Series (Cont.)

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.

5.16 Recursion vs. Iteration

- 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*.

The _Bool Data Type

The _Bool Data Type

- The C standard includes a boolean type—represented by the keyword `_Bool`—which can hold only the values 0 or 1.
- Recall C’s convention of using zero and nonzero values to represent false and true—the value 0 in a condition evaluates to false, while any nonzero value evaluates to true.
- Assigning any non-zero value to a `_Bool` sets it to 1.
- The standard also includes the `<stdbool.h>` header, which defines `bool` as a shorthand for the type `_Bool`, and true and false as named representations of 1 and 0, respectively.

The `_Bool` Data Type (Cont.)

- At preprocessor time, `bool`, `true` and `false` are replaced with `_Bool`, `1` and `0`.
- The example uses a programmer-defined function, a concept we introduce in Chapter 5.
- Microsoft Visual C++ does not implement the `_Bool` data type.

5.17 Secure C Programming

Secure Random Numbers

- The C standard library does not provide a secure random-number generator.
- 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.

5.17 Secure C Programming (Cont.)

- In Section 5.10, we introduced the rand function for generating pseudorandom numbers.
- The guideline presents several platform-specific random-number generators that are considered to be secure.
- For example, Microsoft Windows provides the `CryptGenRandom` function, and POSIX based systems (such as Linux) provide a random function that produces more secure results.
- For more information, see guideline MSC30-C at `https://www.securecoding.cert.org`.