C Functions

Form ever follows function.

—Louis Henri Sullivan

E pluribus unum.
(One composed of many.)

—Virgil

O! call back yesterday, bid time return.

—William Shakespeare

Call me Ishmael.

—Herman Melville

When you call me that, smile!

—Owen Wister

Answer me in one word.

—William Shakespeare

There is a point at which methods devour themselves.

—Frantz Fanon

Life can only be understood backward; but it must be lived forward.

—Soren Kierkegaard

OBJECTIVES

In this chapter you will learn:

- To construct programs modularly from small pieces called functions.
- The common math functions available in the C Standard Library.
- To create new functions.
- The mechanisms used to pass information between functions.
- Simulation techniques using random number generation.
- How to write and use recursive functions, i.e., functions that call themselves.

Outline

Outline

5.1 Introduction

- **Divide and conquer**
  - **Construct a program from smaller pieces or components**
    - **These smaller pieces are called modules**
  - **Each piece more manageable than the original program**

5.2 Program Modules in C

- Functions
  - Modules in C
  - Programs combine user-defined functions with library functions
    - C standard library has a wide variety of functions
- Function calls
  - Invoking functions
    - Provide function name and arguments (data)
    - Function performs operations or manipulations
    - Function returns results
  - Function call analogy:
    - Boss asks worker to complete task

      Worker gets information, does task, returns result

      Information hiding: boss does not know details

Good Programming Practice 5.1

Familiarize yourself with the rich collection of functions in the C Standard Library.

Software Engineering Observation 5.1

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

Portability Tip 5.1

Using the functions in the C Standard Library helps make programs more portable.

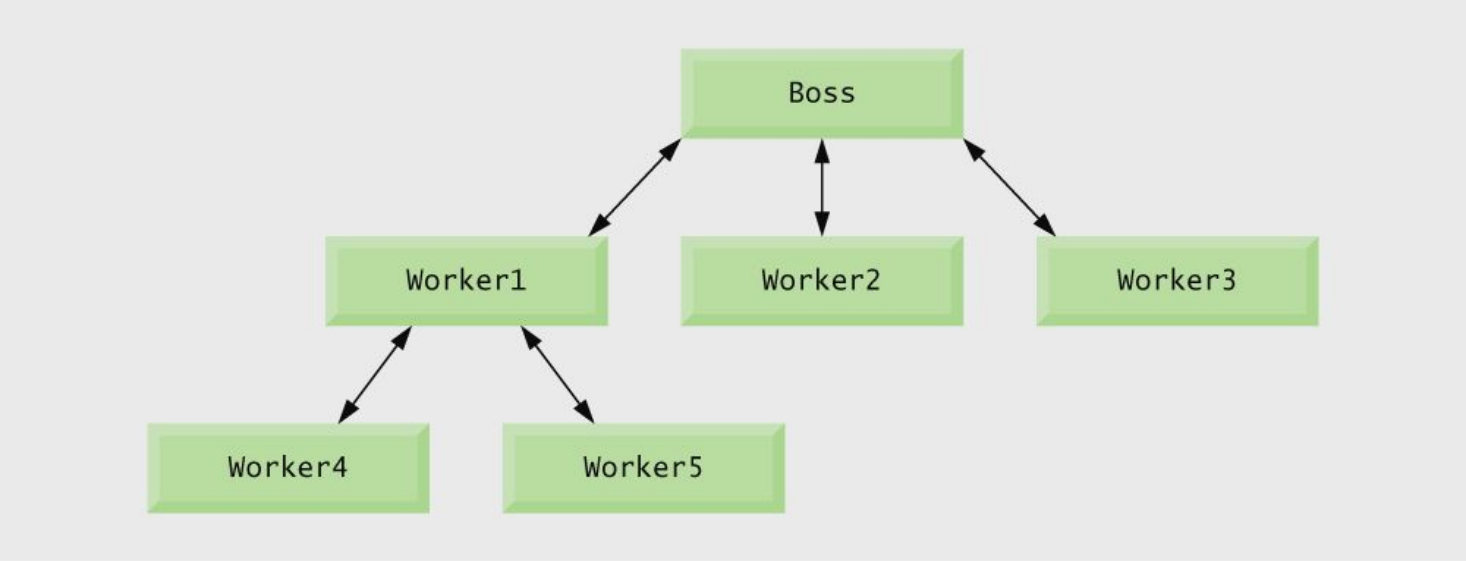

Fig. 5.1 | Hierarchical boss function/worker function relationship.

5.3 Math Library Functions

- **Math library functions**
  - perform common mathematical calculations
  - `#include <math.h>`
- **Format for calling functions**
  - `FunctionName( argument );`
    - If multiple arguments, use comma-separated list
  - `printf( "%.2f", sqrt( 900.0 ) );`
    - Calls function `sqrt`, which returns the square root of its argument
    - All math functions return data type `double`
  - Arguments may be constants, variables, or expressions

Error-Prevention Tip 5.1

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

Function	Description	Example
`sqrt( x )`	square root of x	`sqrt( 900.0 )` is `30.0` `sqrt( 9.0 )` is `3.0`
`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	`fabs( 5.0 )` is `5.0`
		`fabs( 0.0 )` is `0.0`
		`fabs( -5.0 )` is `5.0`

`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.)

5.4 Functions

- **Functions**
  - **Modularize a program**
  - **All variables defined inside functions are local variables**
    - **Known only in function defined**
  - **Parameters**
    - **Communicate information between functions**
    - **Local variables**
- **Benefits of functions**
  - **Divide and conquer**
    - **Manageable program development**
  - **Software reusability**
    - **Use existing functions as building blocks for new programs**
    - **Abstraction - hide internal details (library functions)**
  - **Avoid code repetition**

Software Engineering Observation 5.2

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

Software Engineering Observation 5.3

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

Software Engineering Observation 5.4

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

5.5 Function Definitions

- **Function definition format**

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

  - **Function-name: any valid identifier**
  - **Return-value-type: data type of the result (default `int`)**
    - **`void` – indicates that the function returns nothing**
  - **Parameter-list: comma separated list, declares parameters**
    - **A type must be listed explicitly for each parameter unless, the parameter is of type `int`**

5.5 Function Definitions

- **Function definition format (continued)**

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

  - **Definitions and statements: function body (block)**
    - **Variables can be defined inside blocks (can be nested)**
    - **Functions can not be defined inside other functions**
  - **Returning control**
    - **If nothing returned**

      `return;`

      or, until reaches right brace
    - **If something returned**

      `return` *expression* `;`

Good Programming Practice 5.2

Place a blank line between function definitions to separate the functions and enhance program readability.

Outline

fig05_03.c

```
1  /* Fig. 5.3: fig05_03.c
2     Creating and using a programmer-defined function */
3  #include <stdio.h>
4
5  int square( int y ); /* function prototype */
6
7  /* function main begins program execution */
8  int main( void )
9  {
10    int x; /* counter */
11
12    /* loop 10 times and calculate and output square of x each time */
13    for ( x = 1; x <= 10; x++ ) {
14       printf( "%d  ", square( x ) ); /* function call */
15    } /* end for */
16
17    printf( "\n" );
18
19    return 0; /* indicates successful termination */
20
21 } /* end main */
22
23 /* square function definition returns square of parameter */
24 int square( int y ) /* y is a copy of argument to function */
25 {
26    return y * y; /* returns square of y as an int */
27
28 } /* end function square */
```

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

Function prototype indicates function will be defined later in the program

Call to **square** function

Function definition

Common Programming Error 5.1

Omitting the return-value-type in a function definition is a syntax error if the function prototype specifies a return type other than `int`.

Common Programming Error 5.2

Forgetting to return a value from a function that is supposed to return a value can lead to unexpected errors. The C standard states that the result of this omission is undefined.

Common Programming Error 5.3

Returning a value from a function with a `void` return type is a syntax error.

Good Programming Practice 5.3

Even though an omitted return type defaults to `int`, always state the return type explicitly.

Common Programming Error 5.4

Specifying function parameters of the same type as `double x, y` instead of `double x, double y` might cause errors in your programs. The parameter declaration `double x, y` would actually make `y` a parameter of type `int` because `int` is the default.

Common Programming Error 5.5

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

Common Programming Error 5.6

Defining a function parameter again as a local variable within the function is a syntax error.

Good Programming Practice 5.4

Include the type of each parameter in the parameter list, even if that parameter is of the default type `int`.

Good Programming Practice 5.5

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

Common Programming Error 5.7

Defining a function inside another function is a syntax error.

Good Programming Practice 5.6

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

Software Engineering Observation 5.5

A function should generally be no longer than one page. Better yet, functions should generally be no longer than half a page. Small functions promote software reusability.

Software Engineering Observation 5.6

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

Software Engineering Observation 5.7

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

Software Engineering Observation 5.8

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

Outline

fig05_04.c

(1 of 2)

```
1  /* Fig. 5.4: fig05_04.c
2     Finding the maximum of three integers */
3  #include <stdio.h>
4
5  int maximum( int x, int y, int z ); /* function prototype */
6
7  /* function main begins program execution */
8  int main( void )
9  {
10    int number1; /* first integer */
11    int number2; /* second integer */
12    int number3; /* third integer */
13
14    printf( "Enter three integers: " );
15    scanf( "%d%d%d", &number1, &number2, &number3 );
16
17    /* number1, number2 and number3 are arguments
18       to the maximum function call */
19    printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );
20
21    return 0; /* indicates successful termination */
22
23 } /* end main */
24
```

Function prototype

Function call

Outline

fig05_04.c

(2 of 2)

```
25 /* Function maximum definition */
26 /* x, y and z are parameters */
27 int maximum( int x, int y, int z )
28 {
29    int max = x;      /* assume x is largest */
30
31    if ( y > max ) { /* if y is larger than max, assign y to max */
32       max = y;
33    } /* end if */
34
35    if ( z > max ) { /* if z is larger than max, assign z to max */
36       max = z;
37    } /* end if */
38
39    return max;       /* max is largest value */
40
41 } /* end function maximum */
```

Function definition

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

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

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

5.6 Function Prototypes

- **Function prototype**
  - **Function name**
  - **Parameters – what the function takes in**
  - **Return type – data type function returns (default `int`)**
  - **Used to validate functions**
  - **Prototype only needed if function definition comes after use in program**
  - **The function with the prototype**

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

    - **Takes in 3 `int`s**
    - **Returns an `int`**
- **Promotion rules and conversions**
  - **Converting to lower types can lead to errors**

Good Programming Practice 5.7

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

Good Programming Practice 5.8

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

Common Programming Error 5.8

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

Data type	printf conversion specification	scanf conversion specification
Long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
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 | Promotion hierarchy for data types.

Common Programming Error 5.9

Converting from a higher data type in the promotion hierarchy to a lower type can change the data value.

Common Programming Error 5.10

Forgetting a function prototype causes a syntax error if the return type of the function is not `int` and the function definition appears after the function call in the program. Otherwise, forgetting a function prototype may cause a runtime error or an unexpected result.

Software Engineering Observation 5.9

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

5.7 Function Call Stack and Activation Records

- **Program execution stack**
  - **A stack is a last-in, first-out (LIFO) data structure**
    - **Anything put into the stack is placed "on top"**
    - **The only data that can be taken out is the data on top**
  - **C uses a program execution stack to keep track of which functions have been called**
    - **When a function is called, it is placed on top of the stack**
    - **When a function ends, it is taken off the stack and control returns to the function immediately below it**
  - **Calling more functions than C can handle at once is known as a "stack overflow error"**

5.8 Headers

- **Header files**
  - Contain function prototypes for library functions
  - `<stdlib.h>` , `<math.h>` , etc
  - Load with `#include <filename>`

    `#include <math.h>`
- **Custom header files**
  - Create file with functions
  - Save as `filename.h`
  - Load in other files with `#include "filename.h"`
  - Reuse functions

Standard library header	Explanation
<assert.h>	Contains macros and 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 is running. The notion of locale enables the computer system to handle different conventions for expressing data like dates, times, dollar amounts and large numbers throughout the world.

Fig. 5.6 | Some of the standard library headers. (Part 1 of 3.)

Standard library header	Explanation
<math.h>	Contains function prototypes for math library functions.
<setjmp.h>	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.
<signal.h>	Contains function prototypes and macros to handle various conditions that may arise during program execution.
<stdarg.h>	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
<stddef.h>	Contains common definitions of types used by C for performing certain calculations.

Fig. 5.6 | Some of the standard library headers. (Part 2 of 3.)

Standard library header	Explanation
`<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. 5.6 | Some of the standard library headers. (Part 3 of 3.)

5.9 Calling Functions: Call-by-Value and Call-by-Reference

- **Call by value**
  - Copy of argument passed to function
  - Changes in function do not effect original
  - Use when function does not need to modify argument
    - Avoids accidental changes
- **Call by reference**
  - Passes original argument
  - Changes in function effect original
  - Only used with trusted functions
- **For now, we focus on call by value**

5.10 Random Number Generation

- `rand` function
  - Load `<stdlib.h>`
  - Returns "random" number between 0 and `RAND_MAX` (at least 32767)

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

  - Pseudorandom
    - Preset sequence of "random" numbers
    - Same sequence for every function call
- Scaling
  - To get a random number between 1 and n

```
1 + ( rand() % n )
```

    - `rand() % n` returns a number between 0 and `n - 1`
    - Add 1 to make random number between 1 and n

```
1 + ( rand() % 6)
```

      number between 1 and 6

Outline

fig05_07.c

```c
/* Fig. 5.7: fig05_07.c
   Shifted, scaled integers produced by 1 + rand() %6 */
#include <stdio.h>
#include <stdlib.h>

/* function main begins program execution */
int main( void )
{
   int i; /* counter */

   /* loop 20 times */
   for ( 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 ) {
         printf( "\n" );
      } /* end if */

   } /* end for */

   return 0; /* indicates successful termination */

} /* end main */
```

Generates a random number between 1 and 6

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

Outline

fig05_08.c

(1 of 3)

```
1  /* Fig. 5.8: fig05_08.c
2     Roll a six-sided die 6000 times */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* function main begins program execution */
7  int main( void )
8  {
9     int frequency1 = 0; /* rolled 1 counter */
10    int frequency2 = 0; /* rolled 2 counter */
11    int frequency3 = 0; /* rolled 3 counter */
12    int frequency4 = 0; /* rolled 4 counter */
13    int frequency5 = 0; /* rolled 5 counter */
14    int frequency6 = 0; /* rolled 6 counter */
15
16    int roll; /* roll counter, value 1 to 6000 */
17    int face; /* represents one roll of the die, value 1 to 6 */
18
19    /* loop 6000 times and summarize results */
20    for ( roll = 1; roll <= 6000; roll++ ) {
21       face = 1 + rand() % 6; /* random number from 1 to 6 */
22
23       /* determine face value and increment appropriate counter */
24       switch ( face ) {
25
26          case 1: /* rolled 1 */
27             ++frequency1;
28             break;
29
```

Outline

fig05_08.c

(2 of 3)

```
30         case 2: /* rolled 2 */
31            ++frequency2;
32            break;
33
34         case 3: /* rolled 3 */
35            ++frequency3;
36            break;
37
38         case 4: /* rolled 4 */
39            ++frequency4;
40            break;
41
42         case 5: /* rolled 5 */
43            ++frequency5;
44            break;
45
46         case 6: /* rolled 6 */
47            ++frequency6;
48            break; /* optional */
49      } /* end switch */
50
51   } /* end for */
52
```

Outline

```
53    /* display results in tabular format */
54    printf( "%s%13s\n", "Face", "Frequency" );
55    printf( "    1%13d\n", frequency1 );
56    printf( "    2%13d\n", frequency2 );
57    printf( "    3%13d\n", frequency3 );
58    printf( "    4%13d\n", frequency4 );
59    printf( "    5%13d\n", frequency5 );
60    printf( "    6%13d\n", frequency6 );
61
62    return 0; /* indicates successful termination */
63
64 } /* end main */
```

```
Face    Frequency
   1         1003
   2         1017
   3          983
   4          994
   5         1004
   6          999
```

fig05_08.c

(3 of 3)

5.10 Random Number Generation

- `srand` function
  - `<stdlib.h>`
  - Takes an integer seed and jumps to that location in its "random" sequence

    ```
    srand( seed );
    ```

  - `srand( time( NULL ) );/*load <time.h> */`
    - `time( NULL )`

      Returns the number of seconds that have passed since January 1, 1970

      “Randomizes" the seed

Outline

fig05_09.c

(1 of 2)

```
1  /* Fig. 5.9: fig05_09.c
2     Randomizing die-rolling program */
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  /* function main begins program execution */
7  int main( void )
8  {
9     int i;           /* counter */
10    unsigned seed; /* number used to seed random number generator */
11
12    printf( "Enter seed: " );
13    scanf( "%u", &seed ); /* note %u for unsigned */
14
15    srand( seed ); /* seed random number generator */
16
17    /* loop 10 times */
18    for ( i = 1; i <= 10; i++ ) {
19
```

Seeds the **rand** function

Outline

fig05_09.c

(2 of 2)

```
20       /* pick a random number from 1 to 6 and output it */
21       printf( "%10d", 1 + ( rand() % 6 ) );
22
23       /* if counter is divisible by 5, begin a new line of output */
24       if ( i % 5 == 0 ) {
25          printf( "\n" );
26       } /* end if */
27
28    } /* end for */
29
30    return 0; /* indicates successful termination */
31
32 } /* end main */
```

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

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

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

Common Programming Error 5.11

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

5.11 Example: A Game of Chance

- Craps simulator
- Rules
  - Roll two dice
    - 7 or 11 on first throw, player wins
    - 2, 3, or 12 on first throw, player loses
    - 4, 5, 6, 8, 9, 10 - value becomes player's "point"
  - Player must roll his point before rolling 7 to win

Outline

fig05_10.c

(1 of 4)

```
1  /* Fig. 5.10: fig05_10.c
2     Craps */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h> /* contains prototype for function time */
6
7  /* enumeration constants represent game status */
8  enum Status { CONTINUE, WON, LOST };
9
10 int rollDice( void ); /* function prototype */
11
12 /* function main begins program execution */
13 int main( void )
14 {
15    int sum;        /* sum of rolled dice */
16    int myPoint;    /* point earned */
17
18    enum Status gameStatus; /* can contain CONTINUE, WON, or LOST */
19
20    /* randomize random number generator using current time */
21    srand( time( NULL ) );
22
23    sum = rollDice(); /* first roll of the dice */
24
25    /* determine game status based on sum of dice */
26    switch( sum ) {
27
```

enum (enumeration) assigns numerical values to **CONTINUE**, **WON** and **LOST**

Outline

fig05_10.c

(2 of 4)

```
28       /* win on first roll */
29       case 7:
30       case 11:
31          gameStatus = WON;
32          break;
33
34       /* lose on first roll */
35       case 2:
36       case 3:
37       case 12:
38          gameStatus = LOST;
39          break;
40
41       /* remember point */
42       default:
43          gameStatus = CONTINUE;
44          myPoint = sum;
45          printf( "Point is %d\n", myPoint );
46          break; /* optional */
47    } /* end switch */
48
```

Outline

fig05_10.c

(3 of 4)

```
49    /* while game not complete */
50    while ( gameStatus == CONTINUE ) {
51       sum = rollDice(); /* roll dice again */
52
53       /* determine game status */
54       if ( sum == myPoint ) { /* win by making point */
55          gameStatus = WON; /* game over, player won */
56       } /* end if */
57       else {
58
59          if ( sum == 7 ) { /* lose by rolling 7 */
60             gameStatus = LOST; /* game over, player lost */
61          } /* end if */
62
63       } /* end else */
64
65    } /* end while */
66
67    /* display won or lost message */
68    if ( gameStatus == WON ) { /* did player win? */
69       printf( "Player wins\n" );
70    } /* end if */
71    else { /* player lost */
72       printf( "Player loses\n" );
73    } /* end else */
74
75    return 0; /* indicates successful termination */
76
77 } /* end main */
```

Outline

fig05_10.c

(4 of 4)

```
78
79 /* roll dice, calculate sum and display results */
80 int rollDice( void )
81 {
82    int die1;     /* first die */
83    int die2;     /* second die */
84    int workSum; /* sum of dice */
85
86    die1 = 1 + ( rand() % 6 ); /* pick random die1 value */
87    die2 = 1 + ( rand() % 6 ); /* pick random die2 value */
88    workSum = die1 + die2;     /* sum die1 and die2 */
89
90    /* display results of this roll */
91    printf( "Player rolled %d + %d = %d\n", die1, die2, workSum );
92
93    return workSum; /* return sum of dice */
94
95 } /* end function rollDice */
```

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

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

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

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

Outline

fig05_11.c

Common Programming Error 5.12

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

Good Programming Practice 5.9

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

5.12 Storage Classes

- **Storage class specifiers**
  - **Storage duration – how long an object exists in memory**
  - **Scope – where object can be referenced in program**
  - **Linkage – specifies the files in which an identifier is known (more in Chapter 14)**
- **Automatic storage**
  - **Object created and destroyed within its block**
  - **`auto`: default for local variables**

```
auto double x, y;
```

  - **`register`: tries to put variable into high-speed registers**
    - **Can only be used for automatic variables**

```
register int counter = 1;
```

Performance Tip 5.1

Automatic storage is a means of conserving memory, because automatic variables exist only when they are needed. They are created when the function in which they are defined is entered and they are destroyed when the function is exited.

Software Engineering Observation 5.10

Automatic storage is an example of the *principle of least privilege*—allowing access to data only when it is absolutely needed. Why have variables stored in memory and accessible when in fact they are not needed?

Performance Tip 5.2

The storage-class specifier `register` can be placed before an automatic variable declaration to suggest that the compiler maintain the variable in one of the computer's high-speed hardware registers. If intensely used variables such as counters or totals can be maintained in hardware registers, the overhead of repeatedly loading the variables from memory into the registers and storing the results back into memory can be eliminated.

Performance Tip 5.3

Often, `register` declarations are unnecessary. Today's optimizing compilers are capable of recognizing frequently used variables and can decide to place them in registers without the need for a `register` declaration.

5.12 Storage Classes

- **Static storage**
  - **Variables exist for entire program execution**
  - **Default value of zero**
  - **`static`: local variables defined in functions.**
    - **Keep value after function ends**
    - **Only known in their own function**
  - **`extern`: default for global variables and functions**
    - **Known in any function**

Software Engineering Observation 5.11

Defining a variable as global rather than local allows unintended side effects to occur when a function that does not need access to the variable accidentally or maliciously modifies it. In general, use of global variables should be avoided except in certain situations with unique performance requirements (as discussed in Chapter 14).

Software Engineering Observation 5.12

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

Common Programming Error 5.13

Using multiple storage-class specifiers for an identifier. Only one storage-class specifier can be applied to an identifier.

5.13 Scope Rules

- **File scope**
  - **Identifier defined outside function, known in all functions**
  - **Used for global variables, function definitions, function prototypes**
- **Function scope**
  - **Can only be referenced inside a function body**
  - **Used only for labels (`start:`, `case:`, etc.)**

5.13 Scope Rules

- **Block scope**
  - **Identifier declared inside a block**
    - **Block scope begins at definition, ends at right brace**
  - **Used for variables, function parameters (local variables of function)**
  - **Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block**
- **Function prototype scope**
  - **Used for identifiers in parameter list**

Common Programming Error 5.14

Accidentally using the same name for an identifier in an inner block as is used for an identifier in an outer block, when in fact you want the identifier in the outer block to be active for the duration of the inner block.

Error-Prevention Tip 5.2

Avoid variable names that hide names in outer scopes. This can be accomplished simply by avoiding the use of duplicate identifiers in a program.

Outline

fig05_12.c

(1 of 4)

```
1  /* Fig. 5.12: fig05_12.c
2     A scoping example */
3  #include <stdio.h>
4
5  void useLocal( void );        /* function prototype */
6  void useStaticLocal( void ); /* function prototype */
7  void useGlobal( void );       /* function prototype */
8
9  int x = 1; /* global variable */
10
11 /* function main begins program execution */
12 int main( void )
13 {
14    int x = 5; /* local variable to main */
15
16    printf("local x in outer scope of main is %d\n", x );
17
18    { /* start new scope */
19       int x = 7; /* local variable to new scope */
20
21       printf( "local x in inner scope of main is %d\n", x );
22    } /* end new scope */
23
```

Global variable with file scope

Variable with block scope

Variable with block scope

Outline

fig05_12.c

(2 of 4)

```
24    printf( "local x in outer scope of main is %d\n", x );
25
26    useLocal();       /* useLocal has automatic local x */
27    useStaticLocal(); /* useStaticLocal has static local x */
28    useGlobal();      /* useGlobal uses global x */
29    useLocal();       /* useLocal reinitializes automatic local x */
30    useStaticLocal(); /* static local x retains its prior value */
31    useGlobal();      /* global x also retains its value */
32
33    printf( "\nlocal x in main is %d\n", x );
34
35    return 0; /* indicates successful termination */
36
37 } /* end main */
38
39 /* useLocal reinitializes local variable x during each call */
40 void useLocal( void )
41 {
42    int x = 25;  /* initialized each time useLocal is called */
43
44    printf( "\nlocal x in useLocal is %d after entering useLocal\n", x );
45    x++;
46    printf( "local x in useLocal is %d before exiting useLocal\n", x );
47 } /* end function useLocal */
48
```

Variable with block scope

Outline

fig05_12.c

(3 of 4)

```
49 /* useStaticLocal initializes static local variable x only the first time
50    the function is called; value of x is saved between calls to this
51    function */
52 void useStaticLocal( void )
53 {
54    /* initialized only first time useStaticLocal is called */
55    static int x = 50;
56
57    printf( "\nlocal static x is %d on entering useStaticLocal\n", x );
58    x++;
59    printf( "local static x is %d on exiting useStaticLocal\n", x );
60 } /* end function useStaticLocal */
61
62 /* function useGlobal modifies global variable x during each call */
63 void useGlobal( void )
64 {
65    printf( "\nglobal x is %d on entering useGlobal\n", x );
66    x *= 10;
67    printf( "global x is %d on exiting useGlobal\n", x );
68 } /* end function useGlobal */
```

Static variable with block scope

Global variable

```
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
```

Outline

fig05_12.c

(4 of 4)

5.14 Recursion

- **Recursive functions**
  - **Functions that call themselves**
  - **Can only solve a base case**
  - **Divide a problem up into**
    - **What it can do**
    - **What it cannot do**

      What it cannot do resembles original problem

      The function launches a new copy of itself (recursion step) to solve what it cannot do
  - **Eventually base case gets solved**
    - **Gets plugged in, works its way up and solves whole problem**

5.14 Recursion

- **Example: factorials**
  - `5! = 5 * 4 * 3 * 2 * 1`
  - **Notice that**
    - `5! = 5 * 4!`
    - `4! = 4 * 3! ...`
  - **Can compute factorials recursively**
  - **Solve base case (`1! = 0! = 1`) then plug in**
    - `2! = 2 * 1! = 2 * 1 = 2;`
    - `3! = 3 * 2! = 3 * 2 = 6;`

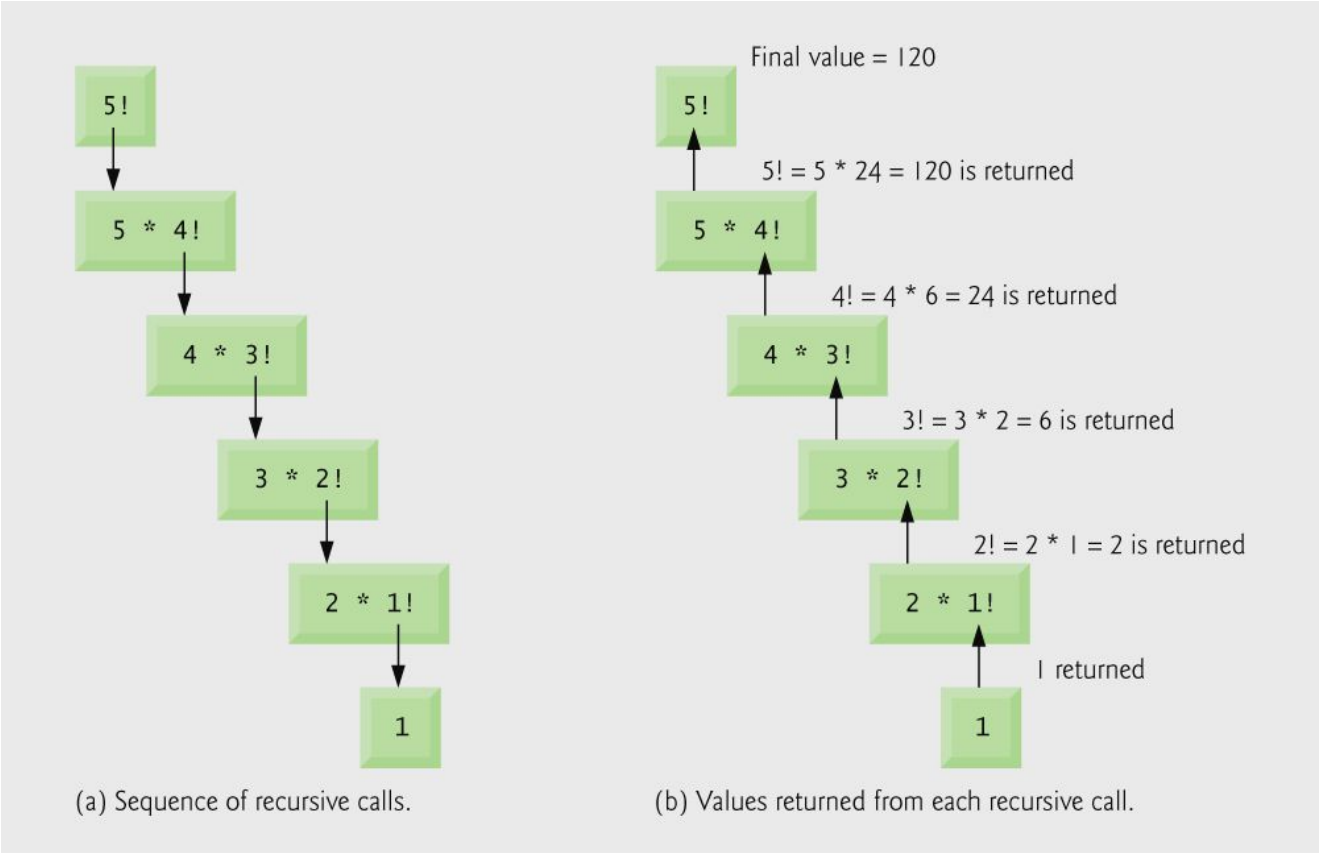

Fig. 5.13 | Recursive evaluation of 5!.

Outline

fig05_14.c

(1 of 2)

```
1  /* Fig. 5.14: fig05_14.c
2     Recursive factorial function */
3  #include <stdio.h>
4
5  long factorial( long number ); /* function prototype */
6
7  /* function main begins program execution */
8  int main( void )
9  {
10    int i; /* counter */
11
12    /* loop 11 times; during each iteration, calculate
13       factorial( i ) and display result */
14    for ( i = 0; i <= 10; i++ ) {
15       printf( "%2d! = %ld\n", i, factorial( i ) );
16    } /* end for */
17
18    return 0; /* indicates successful termination */
19
20 } /* end main */
21
```

Outline

fig05_14.c

(2 of 2)

```
22 /* recursive definition of function factorial */
23 long factorial( long number )
24 {
25    /* base case */
26    if ( number <= 1 ) {
27       return 1;
28    } /* end if */
29    else { /* recursive step */
30       return ( number * factorial( number - 1 ) );
31    } /* end else */
32
33 } /* end function factorial */
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Common Programming Error 5.15

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

Common Programming Error 5.16

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

5.15 Example Using Recursion: Fibonacci Series

- **Fibonacci series: 0, 1, 1, 2, 3, 5, 8...**
  - **Each number is the sum of the previous two**
  - **Can be solved recursively:**
    - `fib( n ) = fib( n - 1 ) + fib( n – 2 )`
  - **Code for the `fibonacci` function**

```
long fibonacci( long n )
{
  if (n == 0 || n == 1)  // base case
    return n;
  else
    return fibonacci( n - 1) +
      fibonacci( n – 2 );
}
```

Outline

fig05_15.c

(1 of 4)

```
1  /* Fig. 5.15: fig05_15.c
2     Recursive fibonacci function */
3  #include <stdio.h>
4
5  long fibonacci( long n ); /* function prototype */
6
7  /* function main begins program execution */
8  int main( void )
9  {
10    long result; /* fibonacci value */
11    long number; /* number input by user */
12
13    /* obtain integer from user */
14    printf( "Enter an integer: " );
15    scanf( "%ld", &number );
16
17    /* calculate fibonacci value for number input by user */
18    result = fibonacci( number );
19
20    /* display result */
21    printf( "Fibonacci( %ld ) = %ld\n", number, result );
22
23    return 0; /* indicates successful termination */
24
25 } /* end main */
26
```

Outline

fig05_15.c

(2 of 4)

```
27 /* Recursive definition of function fibonacci */
28 long fibonacci( long n )
29 {
30    /* base case */
31    if ( n == 0 || n == 1 ) {
32       return n;
33    } /* end if */
34    else { /* recursive step */
35       return fibonacci( n - 1 ) + fibonacci( n - 2 );
36    } /* end else */
37
38 } /* end function fibonacci */
```

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

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

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

(continued on next slide...)

(continued from previous slide...)

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

```
Enter an integer: 4
Fibonacci( 4 ) = 3
```

```
Enter an integer: 5
Fibonacci( 5 ) = 5
```

```
Enter an integer: 6
Fibonacci( 6 ) = 8
```

(continued on next slide...)

Outline

fig05_15.c

(3 of 4)

(continued from previous slide...)

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

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

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

```
Enter an integer: 35
Fibonacci( 35 ) = 9227465
```

Outline

fig05_15.c

(4 of 4)

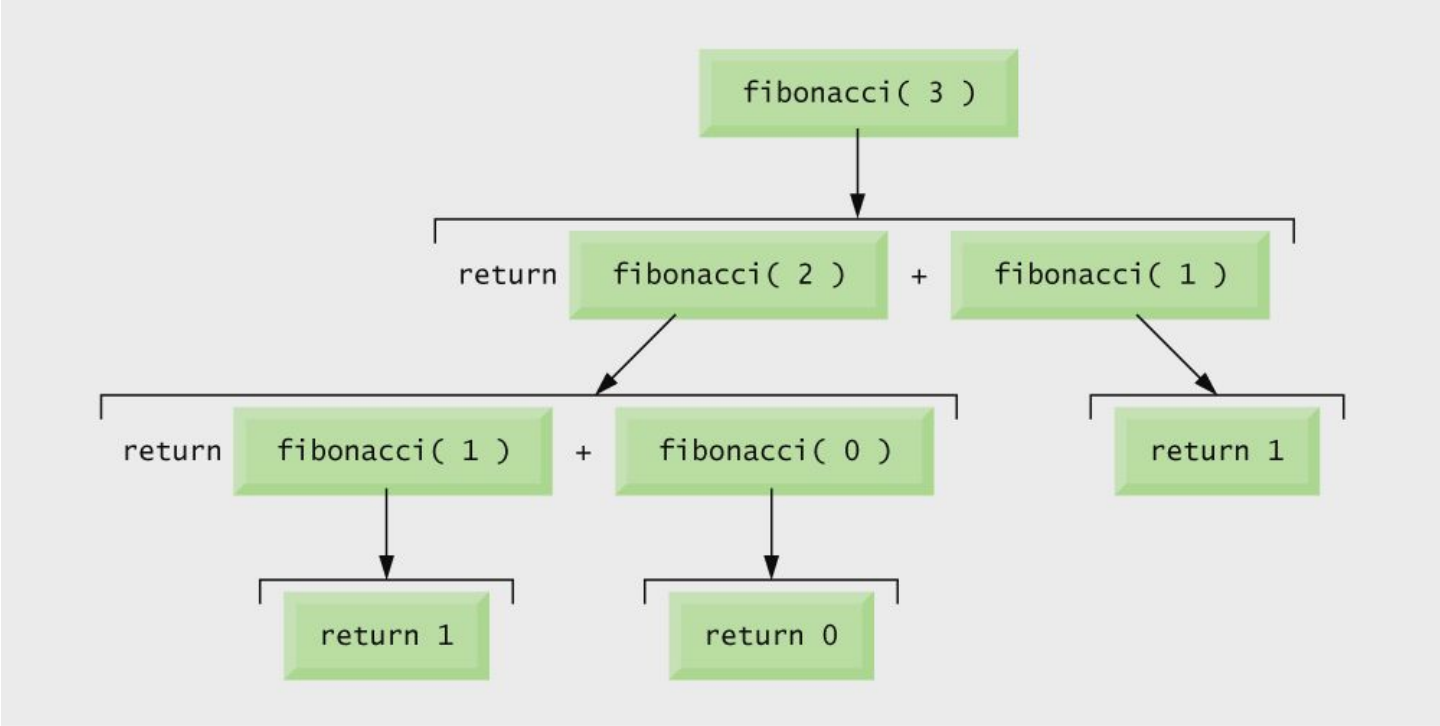

Fig. 5.16 | Set of recursive calls for **`fibonacci(3)`**.

Common Programming Error 5.17

Writing programs that depend on the order of evaluation of the operands of operators other than `&&`, `||`, `?:`, and the comma (`,`) operator can lead to errors because compilers may not necessarily evaluate the operands in the order you expect.

Portability Tip 5.2

Programs that depend on the order of evaluation of the operands of operators other than `&&`, `||`, `?:`, and the comma (`,`) operator can function differently on systems with different compilers.

Performance Tip 5.4

Avoid Fibonacci-style recursive programs which result in an exponential "explosion" of calls.

5.16 Recursion vs. Iteration

- **Repetition**
  - **Iteration: explicit loop**
  - **Recursion: repeated function calls**
- **Termination**
  - **Iteration: loop condition fails**
  - **Recursion: base case recognized**
- **Both can have infinite loops**
- **Balance**
  - **Choice between performance (iteration) and good software engineering (recursion)**

Software Engineering Observation 5.13

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). 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 is easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution may not be apparent.

Performance Tip 5.5

Avoid using recursion in performance situations. Recursive calls take time and consume additional memory.

Common Programming Error 5.18

Accidentally having a nonrecursive function call itself either directly, or indirectly through another function.

Performance Tip 5.6

Functionalizing programs in a neat, hierarchical manner 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). So, although monolithic programs may perform better, they are more difficult to program, test, debug, maintain, and evolve.

Chapter	Recursion examples and exercises
Chapter 5	Factorial function
	Fibonacci function
	Greatest common divisor
	Sum of two integers
	Multiply two integers
	Raising an integer to an integer power
	Towers of Hanoi
	Recursive `main`
	Printing keyboard inputs in reverse
	Visualizing recursion
Chapter 6	Sum the elements of an array
	Print an array
	Print an array backward
	Print a string backward
	Check if a string is a palindrome
	Minimum value in an array
	Selection sort
	Quicksort
	Linear search
	Binary search

Fig. 5.17 | Recursion examples and exercises in the text. (Part 1 of 2.)

Chapter	Recursion examples and exercises
Chapter 7	Eight Queens
	Maze traversal
Chapter 8	Printing a string input at the keyboard backward
Chapter 12	Linked list insert
	Linked list delete
	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
Chapter 16	Selection sort
	Quicksort

Fig. 5.17 | Recursion examples and exercises in the text. (Part 2 of 2.)