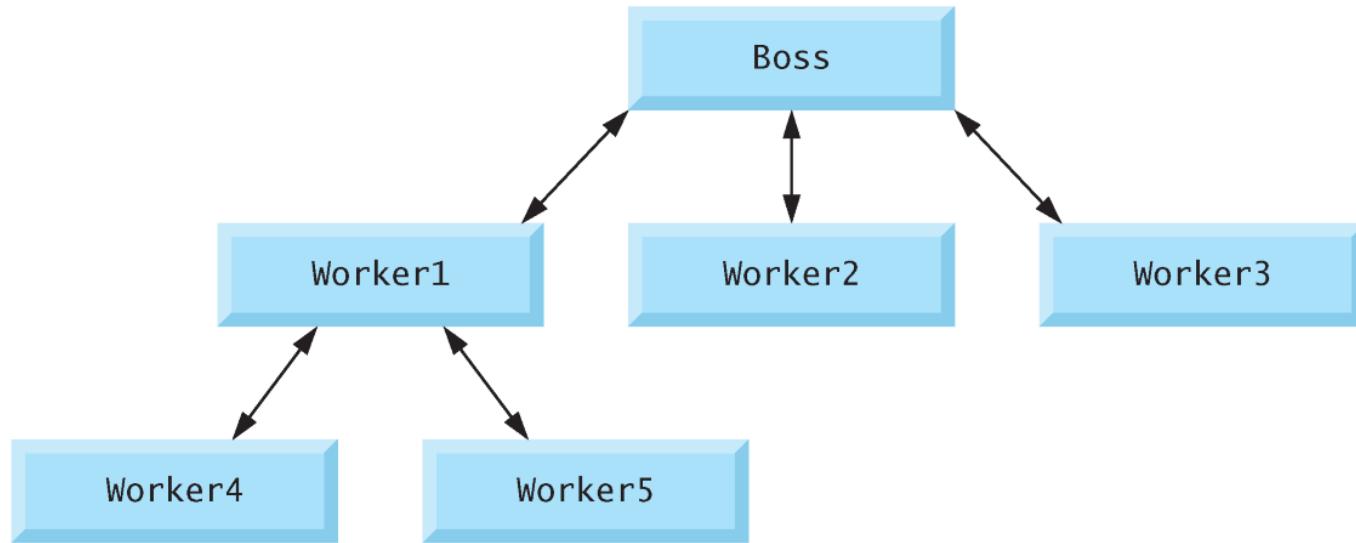


# 02\_05\_C Functions

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

**Fig. 5.2** | Commonly used math library functions. (Part 1 of 2.)

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

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

```
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 int main(void)
8 {
9     // Loop 10 times and calculate and output square of x each time
10    for (int x = 1; x <= 10; ++x) {
11        printf("%d ", square(x)); // function call
12    }
13
14    puts("");
15 }
16
17 // square function definition returns the square of its parameter
18 int square(int y) // y is a copy of the argument to the function
19 {
20     return y * y; // returns the square of y as an int
21 }
```

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

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

---

```
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 int main(void)
8 {
9     int number1; // first integer entered by the user
10    int number2; // second integer entered by the user
11    int number3; // third integer entered by the user
12
13    printf("%s", "Enter three integers: ");
14    scanf("%d%d%d", &number1, &number2, &number3);
15
16    // number1, number2 and number3 are arguments
17    // to the maximum function call
18    printf("Maximum is: %d\n", maximum(number1, number2, number3));
19 }
20
```

---

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

---

```
21 // Function maximum definition
22 // x, y and z are parameters
23 int maximum(int x, int y, int z)
24 {
25     int max = x; // assume x is largest
26
27     if (y > max) { // if y is larger than max,
28         max = y; // assign y to max
29     }
30
31     if (z > max) { // if z is larger than max,
32         max = z; // assign z to max
33     }
34
35     return max; // max is largest value
36 }
```

---

**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
<i>Floating-point types</i>		
long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
<i>Integer types</i>		
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.

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

```
10 squared: 100
```

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

Header	Explanation
<assert.h>	Contains information for adding diagnostics that aid program debugging.
<cctype.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.)

```
1 // Fig. 5.11: fig05_11.c
2 // Shifted, scaled random integers produced by 1 + rand() % 6.
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main(void)
7 {
8     // Loop 20 times
9     for (unsigned int i = 1; i <= 20; ++i) {
10
11         // pick random number from 1 to 6 and output it
12         printf("%10d", 1 + (rand() % 6));
13
14         // if counter is divisible by 5, begin new line of output
15         if (i % 5 == 0) {
16             puts("");
17         }
18     }
19 }
```

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  $1 + \text{rand()} \% 6$ .

---

```
1 // Fig. 5.12: fig05_12.c
2 // Rolling a six-sided die 60,000,000 times.
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main(void)
7 {
8     unsigned int frequency1 = 0; // rolled 1 counter
9     unsigned int frequency2 = 0; // rolled 2 counter
10    unsigned int frequency3 = 0; // rolled 3 counter
11    unsigned int frequency4 = 0; // rolled 4 counter
12    unsigned int frequency5 = 0; // rolled 5 counter
13    unsigned int frequency6 = 0; // rolled 6 counter
14
15    // Loop 60000000 times and summarize results
16    for (unsigned int roll = 1; roll <= 60000000; ++roll) {
17        int face = 1 + rand() % 6; // random number from 1 to 6
18
19        // determine face value and increment appropriate counter
20        switch (face) {
21
22            case 1: // rolled 1
23                ++frequency1;
24                break;
```

---

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

---

```
25
26     case 2: // rolled 2
27         ++frequency2;
28         break;
29
30     case 3: // rolled 3
31         ++frequency3;
32         break;
33
34     case 4: // rolled 4
35         ++frequency4;
36         break;
37
38     case 5: // rolled 5
39         ++frequency5;
40         break;
41
42     case 6: // rolled 6
43         ++frequency6;
44         break; // optional
45     }
46 }
47
```

---

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

```
48     // display results in tabular format
49     printf("%s%13s\n", "Face", "Frequency");
50     printf("    1%13u\n", frequency1);
51     printf("    2%13u\n", frequency2);
52     printf("    3%13u\n", frequency3);
53     printf("    4%13u\n", frequency4);
54     printf("    5%13u\n", frequency5);
55     printf("    6%13u\n", frequency6);
56 }
```

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.)

---

```
1 // Fig. 5.13: fig05_13.c
2 // Randomizing the die-rolling program.
3 #include <stdlib.h>
4 #include <stdio.h>
5
6 int main(void)
7 {
8     unsigned int seed; // number used to seed the random number generator
9
10    printf("%s", "Enter seed: ");
11    scanf("%u", &seed); // note %u for unsigned int
12
13    srand(seed); // seed the random number generator
14
```

---

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

---

```
15 // loop 10 times
16 for (unsigned int i = 1; i <= 10; ++i) {
17
18     // pick a random number from 1 to 6 and output it
19     printf("%10d", 1 + (rand() % 6));
20
21     // if counter is divisible by 5, begin a new line of output
22     if (i % 5 == 0) {
23         puts("");
24     }
25 }
26 }
```

---

**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.

---

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

---

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

---

```
21 // determine game status based on sum of dice
22 switch(sum) {
23
24     // win on first roll
25     case 7: // 7 is a winner
26     case 11: // 11 is a winner
27         gameStatus = WON;
28         break;
29
30     // lose on first roll
31     case 2: // 2 is a loser
32     case 3: // 3 is a loser
33     case 12: // 12 is a loser
34         gameStatus = LOST;
35         break;
36
37     // remember point
38     default:
39         gameStatus = CONTINUE; // player should keep rolling
40         myPoint = sum; // remember the point
41         printf("Point is %d\n", myPoint);
42         break; // optional
43     }
44 }
```

---

**Fig. 5.14** | Simulating the game of craps. (Part 2 of 4.)

```
45 // while game not complete
46 while (CONTINUE == gameStatus) { // player should keep rolling
47     sum = rollDice(); // roll dice again
48
49     // determine game status
50     if (sum == myPoint) { // win by making point
51         gameStatus = WON;
52     }
53     else {
54         if (7 == sum) { // lose by rolling 7
55             gameStatus = LOST;
56         }
57     }
58 }
59
60 // display won or lost message
61 if (WON == gameStatus) { // did player win?
62     puts("Player wins");
63 }
64 else { // player lost
65     puts("Player loses");
66 }
67 }
68 }
```

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

---

```
69 // roll dice, calculate sum and display results
70 int rollDice(void)
71 {
72     int die1 = 1 + (rand() % 6); // pick random die1 value
73     int die2 = 1 + (rand() % 6); // pick random die2 value
74
75     // display results of this roll
76     printf("Player rolled %d + %d = %d\n", die1, die2, die1 + die2);
77     return die1 + die2; // return sum of dice
78 }
```

---

**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 I 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.)

---

```
1 // Fig. 5.16: fig05_16.c
2 // Scoping.
3 #include <stdio.h>
4
5 void useLocal(void); // function prototype
6 void useStaticLocal(void); // function prototype
7 void useGlobal(void); // function prototype
8
9 int x = 1; // global variable
10
11 int main(void)
12 {
13     int x = 5; // local variable to main
14
15     printf("local x in outer scope of main is %d\n", x);
16
17     { // start new scope
18         int x = 7; // local variable to new scope
19
20         printf("local x in inner scope of main is %d\n", x);
21     } // end new scope
22
23     printf("local x in outer scope of main is %d\n", x);
24 }
```

---

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

---

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

---

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

---

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

---

**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

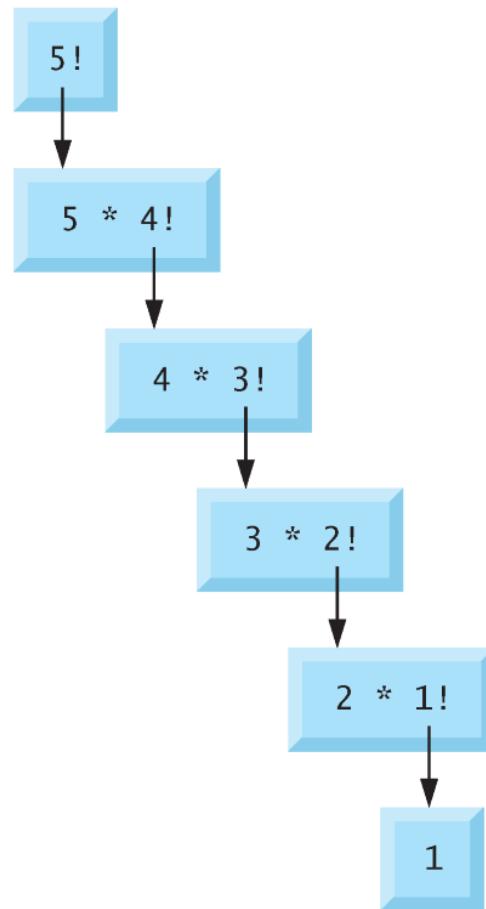
local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

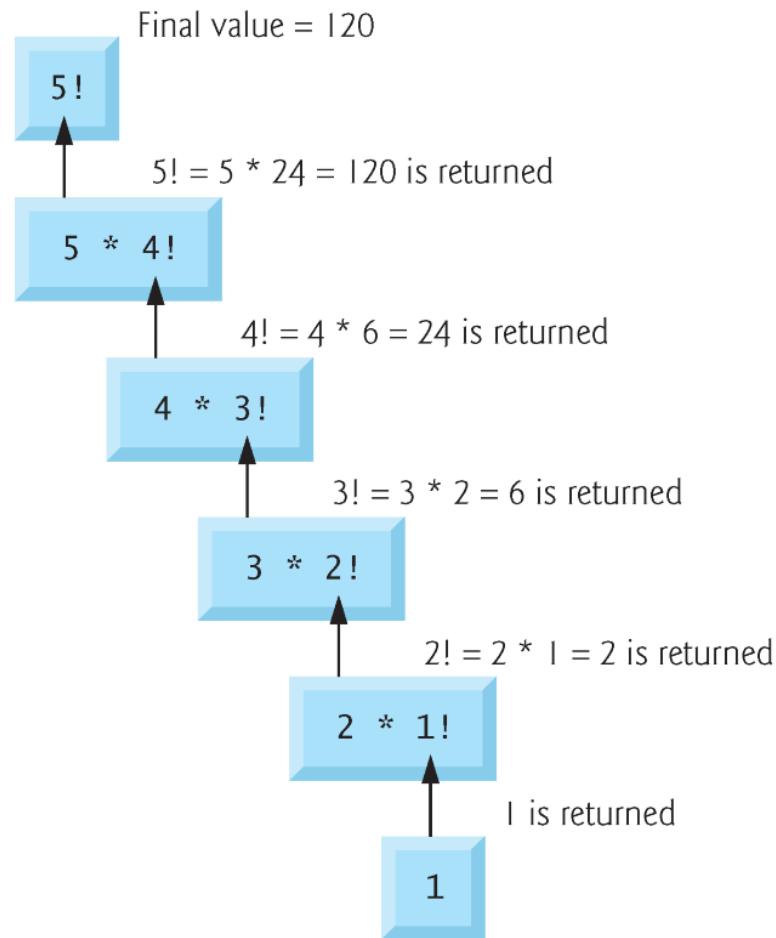
local x in main is 5
```

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

a) Sequence of recursive calls



b) Values returned from each recursive call



**Fig. 5.17** | Recursive evaluation of 5!.

---

```
1 // Fig. 5.18: fig05_18.c
2 // Recursive factorial function.
3 #include <stdio.h>
4
5 unsigned long long int factorial(unsigned int number);
6
7 int main(void)
8 {
9     // during each iteration, calculate
10    // factorial(i) and display result
11    for (unsigned int i = 0; i <= 21; ++i) {
12        printf("%u! = %llu\n", i, factorial(i));
13    }
14 }
```

---

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

---

```
16 // recursive definition of function factorial
17 unsigned long long int factorial(unsigned int number)
18 {
19     // base case
20     if (number <= 1) {
21         return 1;
22     }
23     else { // recursive step
24         return (number * factorial(number - 1));
25     }
26 }
```

---

**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.)

---

```
1 // Fig. 5.19: fig05_19.c
2 // Recursive fibonacci function
3 #include <stdio.h>
4
5 unsigned long long int fibonacci(unsigned int n); // function prototype
6
7 int main(void)
8 {
9     unsigned int number; // number input by user
10
11    // obtain integer from user
12    printf("%s", "Enter an integer: ");
13    scanf("%u", &number);
14
15    // calculate fibonacci value for number input by user
16    unsigned long long int result = fibonacci(number);
17
18    // display result
19    printf("Fibonacci(%u) = %llu\n", number, result);
20 }
21
```

---

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

```
22 // Recursive definition of function fibonacci
23 unsigned long long int fibonacci(unsigned int n)
24 {
25     // base case
26     if (0 == n || 1 == n) {
27         return n;
28     }
29     else { // recursive step
30         return fibonacci(n - 1) + fibonacci(n - 2);
31     }
32 }
```

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

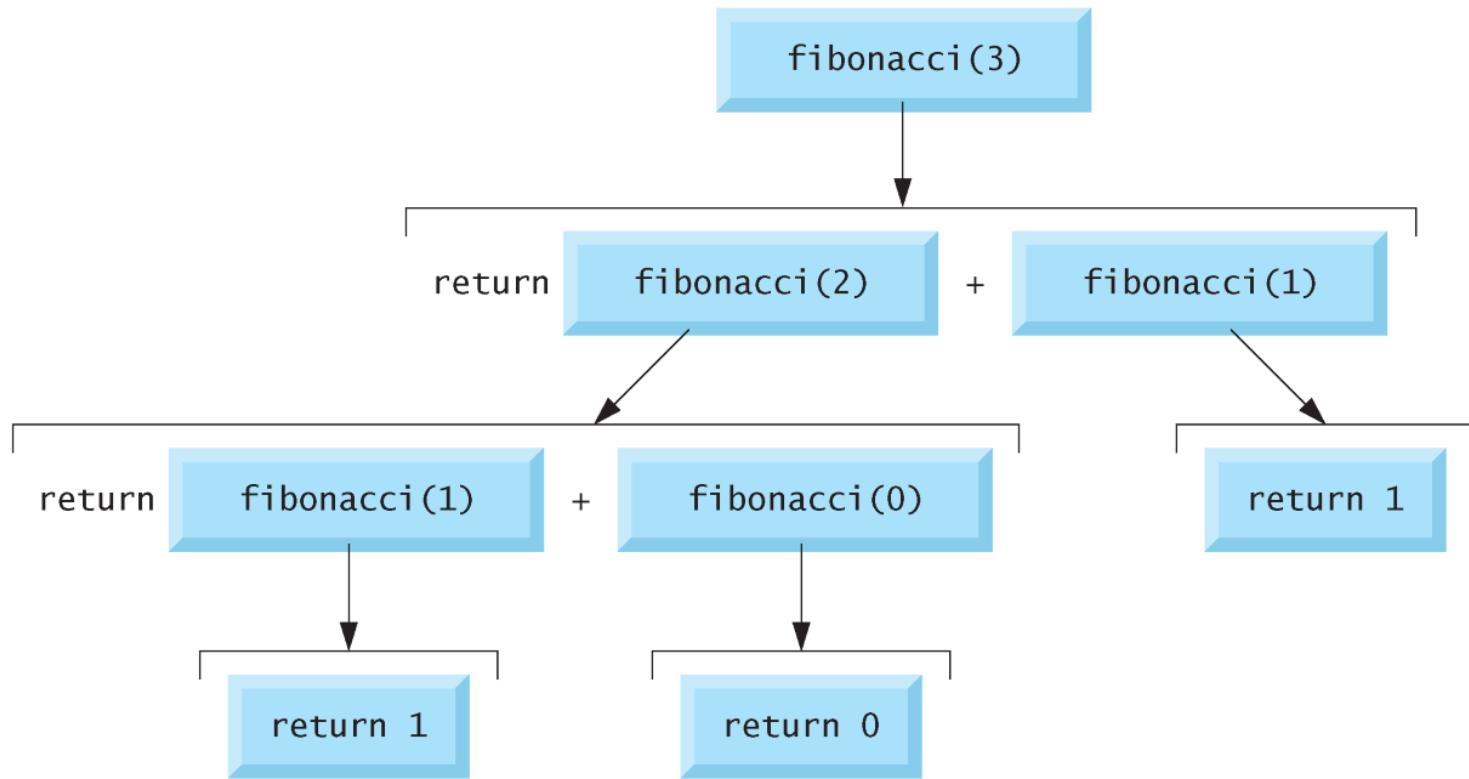
Enter an integer: 10  
Fibonacci(10) = 55

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)`.

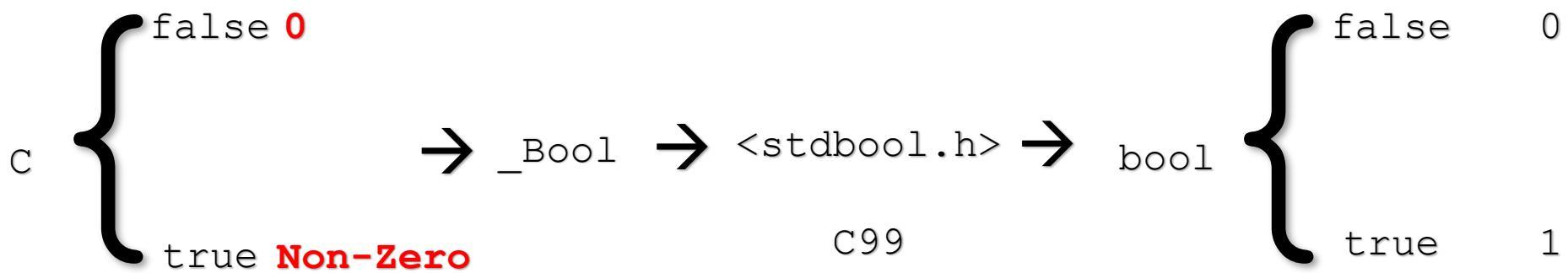
## The `_Bool` Data Type

### *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

## *The \_Bool Data Type*



```
#include <stdbool.h>
#include <stdio.h>
int main()
{
bool a = true;
if(a)
    printf("Its ture");
return 0;
}
```

C Copy

Output:

Its true

C Copy

```
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
    bool a=true, b=false;
    printf("%d\n", a&&b);
    printf("%d\n", a| |b);
    printf("%d\n", !b);
}
```

Output:

0

1

1

# Bool Array

```
#include <stdbool.h>
int main()
{
    bool arr[2] = {true, false};
    printf("Value at index 1 of array is %d",arr[1]);
    return 0;
}
```

C Copy

Output:

```
Value at index 1 of array is 0
```

## 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.