5

C Functions

OBJECTIVES

In this chapter you will learn:

- How to write and use recursive functions, i.e., functions that call themselves.

Outline

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;`

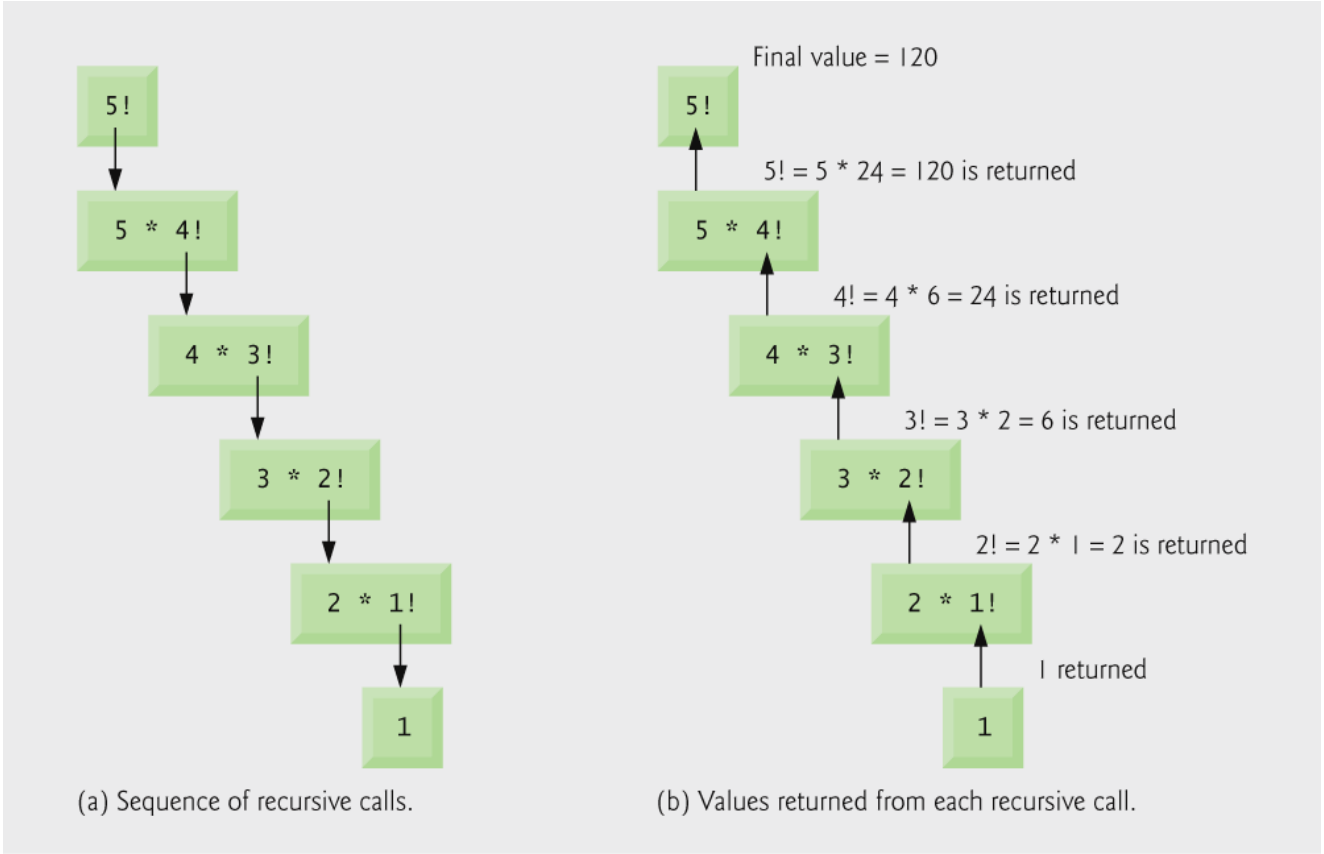

Fig. 5.13 | Recursive evaluation of 5!.

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

```
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
```

fig05_14.c

(2 of 2)

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)

```c
/* Fig. 5.15: fig05_15.c
   Recursive fibonacci function */
#include <stdio.h>

long fibonacci( long n ); /* function prototype */

/* function main begins program execution */
int main( void )
{
   long result; /* fibonacci value */
   long number; /* number input by user */

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

   /* calculate fibonacci value for number input by user */
   result = fibonacci( number );

   /* display result */
   printf( "Fibonacci( %ld ) = %ld\n", number, result );

   return 0; /* indicates successful termination */

} /* end main */

```

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)

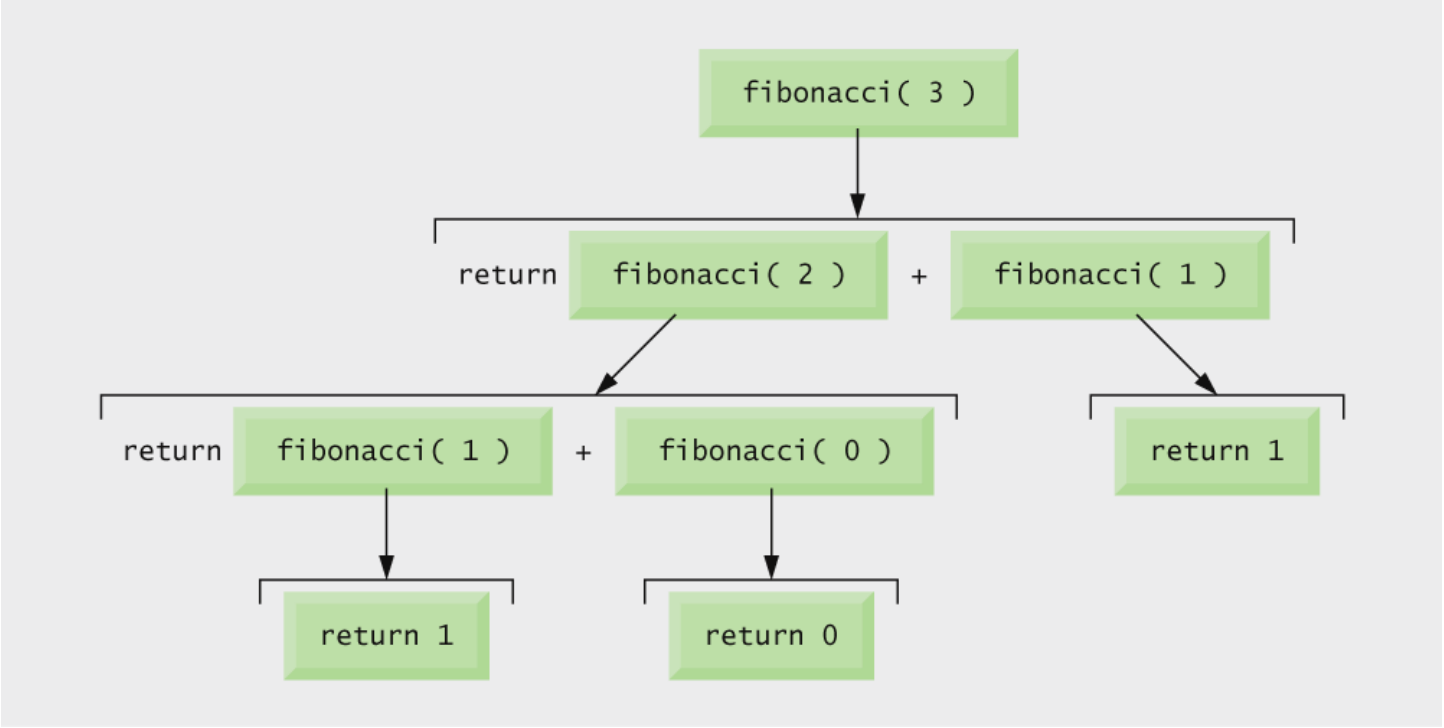

Fig. 5.16 | Set of recursive calls for `fibonacci(3)`.

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.