Recursive Functions in C

Dr Bhanu

Recursion

- We saw programs generally structured as functions that call one another in a disciplined, hierarchical manner.
- For some types of problems, it's useful to have functions call themselves.
- A recursive function is a function that calls itself either directly or indirectly through another function.
- Recursion is a complex topic discussed at length in upper-level computer science courses.

Recursion

- Recursive problem-solving approaches have a number of elements in common.
- A recursive function is called to solve a problem.
- The function actually knows how to solve only the simplest case(s), or so-called base case(s).

Recursion

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

Recursion

- Because this new problem looks like the original problem, the function launches (calls) a fresh copy of itself to go to work on the smaller problem—this is referred to as a recursive call and is also called the recursion step.
- The recursion step also includes the keyword `return`, because its result will be combined with the portion of the problem the function knew how to solve to form a result that will be passed back to the original caller, possibly `main`.
- The recursion step executes while the original call to the function is still open, i.e., it has not yet finished executing.

Recursion

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

Recursion

- The factorial of a nonnegative integer *n*, written *n!* (and pronounced "*n* factorial"), is the product
    - $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

  with 1! equal to 1, and 0! defined to be 1.
- For example, 5! is the product 5 * 4 * 3 * 2 * 1, which is equal to 120.
- The factorial of an integer, `number`, greater than or equal to 0 can be calculated iteratively (nonrecursively) using a `for` statement as follows:

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

Recursion

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

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

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

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

- The evaluation of 5! would proceed as shown in Fig. 5.13.

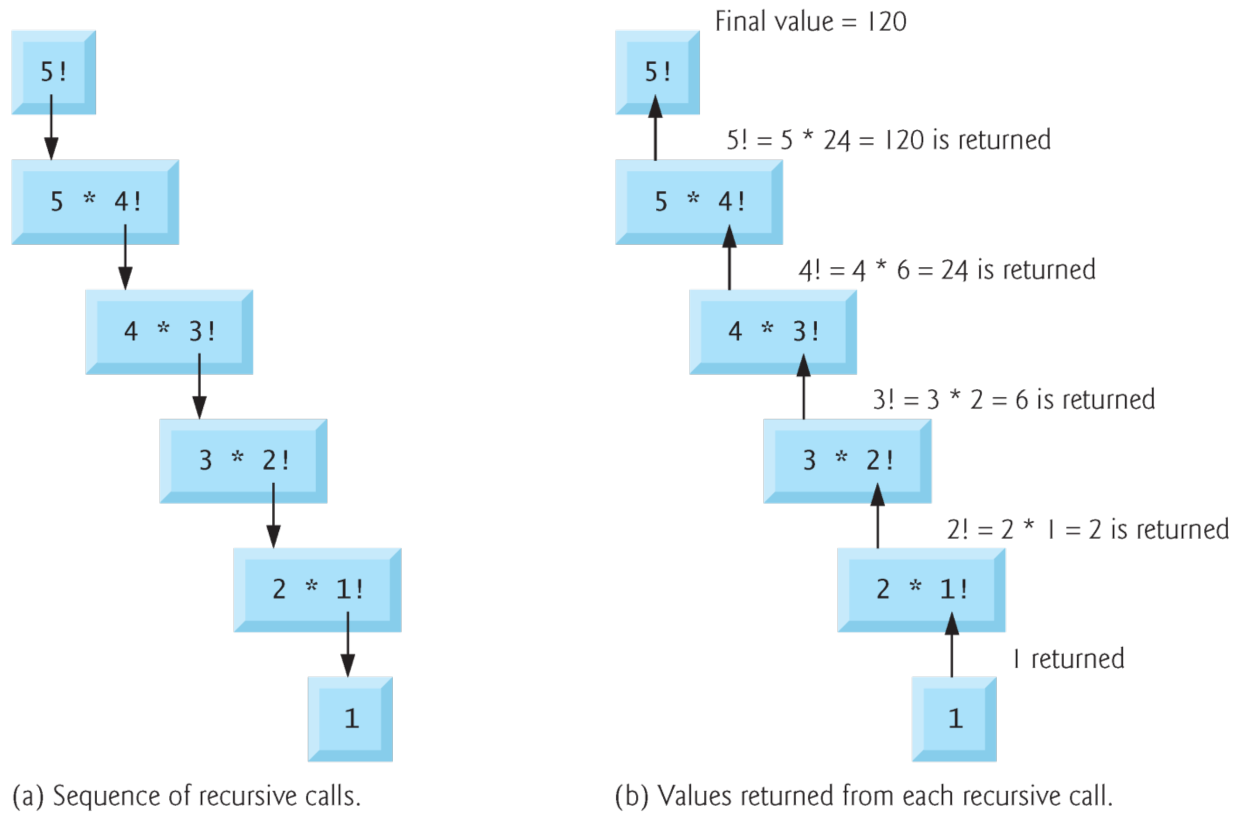
5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
(a) Sequence of recursive calls.
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 returned
1
(b) Values returned from each recursive call.

Fig. 5.13 | Recursive evaluation of 5!.

Recursion

- Figure 5.13(a) shows how the succession of recursive calls proceeds until 1! is evaluated to be 1, which terminates the recursion.
- Figure 5.13(b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.
- The recursive `factorial` function first tests whether a terminating condition is true, i.e., whether `number` is less than or equal to 1.

Recursion

- If `number` is indeed less than or equal to 1, `factorial` returns 1, no further recursion is necessary, and the program terminates.
- If `number` is greater than 1, the statement

```
    return number * factorial( number - 1 );
```

- expresses the problem as the product of `number` and a recursive call to `factorial` evaluating the factorial of `number - 1`.
- The call `factorial( number - 1 )` is a slightly simpler problem than the original calculation `factorial( number )`.

Recursion

- Function `factorial` (line 22) has been declared to receive a parameter of type `long` and return a result of type `long`.
- The C standard specifies that a variable of type `long int` is stored in at least 4 bytes, and thus may hold a value as large as +2147483647.
- Factorial values become large quickly.

```c
/* Fig. 5.14: fig05_14.c
   Recursive factorial function */
#include <stdio.h>

long factorial( long number ); /* function prototype */

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

   /* loop 11 times; during each iteration, calculate
      factorial( i ) and display result */
   for ( i = 0; i <= 10; i++ ) {
      printf( "%2d! = %ld\n", i, factorial( i ) );
   } /* end for */

   return 0; /* indicates successful termination */
} /* end main */

```

Fig. 5.14 | Calculating factorials with a recursive function. (Part 1 of 2.)

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

Fig. 5.14 | Calculating factorials with a recursive function. (Part 2 of 2.)

Recursion

- The conversion specifier `%ld` is used to print `long` values.
- Unfortunately, the `factorial` function produces large values so quickly that even `long int` does not help us print many factorial values before the size of a `long int` variable is exceeded.
- `double` may ultimately be needed by the user desiring to calculate factorials of larger numbers.

Common Programming Error 5.14

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.

Example Using Recursion: Fibonacci Series

- The Fibonacci series
    - 0, 1, 1, 2, 3, 5, 8, 13, 21, …
- begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.
- The series occurs in nature and, in particular, describes a form of spiral.
- The ratio of successive Fibonacci numbers converges to a constant value of 1.618….

Example Using Recursion: Fibonacci Series

- This number, too, repeatedly occurs in nature and has been called the golden ratio or the golden mean.
- Humans tend to find the golden mean aesthetically pleasing.
- Architects often design windows, rooms, and buildings whose length and width are in the ratio of the golden mean.
- Postcards are often designed with a golden mean length/width ratio.

Example Using Recursion: Fibonacci Series

- The Fibonacci series may be defined recursively as follows:

  fibonacci(0) = 0
  fibonacci(1) = 1
  fibonacci(*n*) = *fibonacci*(*n* – 1) + *fibonacci*(*n* – 2)

- Figure 5.15 calculates the *n*th Fibonacci number recursively using function `fibonacci`.
- Notice that Fibonacci numbers tend to become large quickly.
- Therefore, we've chosen the data type `long` for the parameter type and the return type in function `fibonacci`.
- In Fig. 5.15, each pair of output lines shows a separate run of the program.

```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 */

```

Fig. 5.15 | Recursively generating Fibonacci numbers. (Part 1 of 4.)

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

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

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

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

Fig. 5.15 | Recursively generating Fibonacci numbers. (Part 2 of 4.)

```
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
```

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

Fig. 5.15 | Recursively generating Fibonacci numbers. (Part 3 of 4.)

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

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

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

Fig. 5.15 | Recursively generating Fibonacci numbers. (Part 4 of 4.)

Example Using Recursion: Fibonacci Series

- The call to `fibonacci` from `main` is not a recursive call (line 18), but all subsequent calls to `fibonacci` are recursive (line 33).
- Each time `fibonacci` is invoked, it immediately tests for the base case—`n` is equal to 0 or 1.
- If this is true, `n` is returned.
- Interestingly, if `n` is greater than 1, the recursion step generates two recursive calls, each of which is for a slightly simpler problem than the original call to `fibonacci`.
- Figure 5.16 shows how function `fibonacci` would evaluate `fibonacci(3)`.

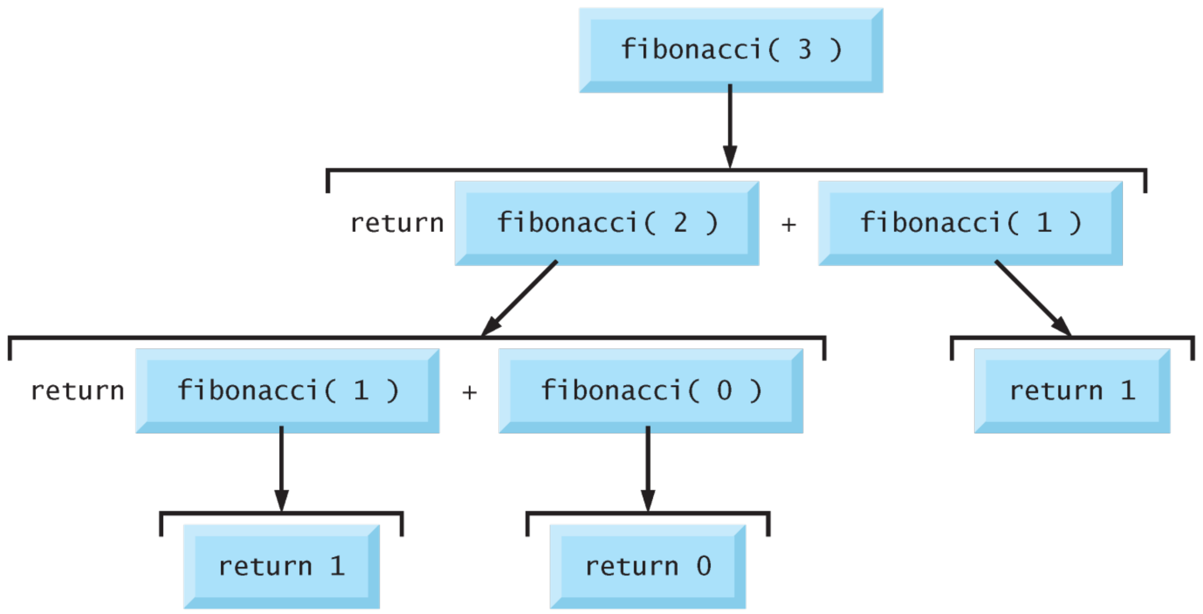

Fig. 5.16 | Set of recursive calls for `fibonacci( 3 )`.

Example Using Recursion: Fibonacci Series

- 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; i.e., 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.

Example Using Recursion: Fibonacci Series

- Computer scientists refer to this as exponential complexity.
- Problems of this nature humble even the world’s most powerful computers!
- Complexity issues in general, and exponential complexity in particular, are discussed in detail in the upper-level computer science curriculum course generally called “Algorithms.”

Performance Tip 5.4

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

Recursion vs. Iteration

- we studied two functions that can easily be implemented either recursively or iteratively.
- In this section, we compare the two approaches and discuss why you might choose one approach over the other in a particular situation.
- Both iteration and recursion are based on a control structure: Iteration uses a repetition structure; recursion uses a selection structure.
- Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated function calls.

Recursion vs. Iteration

- Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.
- Iteration with counter-controlled repetition and successive recursion gradually approach termination: Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion keeps producing simpler versions of the original problem until the base case is reached.

Recursion vs. Iteration

- Both iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case.
- Recursion has many negatives.
- It repeatedly invokes the mechanism, and consequently the overhead, of function calls.
- This can be expensive in both processor time and memory space.

Recursion vs. Iteration

- Each recursive call causes another copy of the function to be created; this can consume considerable memory.
- So why choose recursion?

Performance Tip 5.5

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

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
	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
Appendix F	Selection sort
	Quicksort

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

Recursion vs. Iteration - Observations

- Good software engineering is important.
- High performance is important.
- Unfortunately, these goals are often at odds with one another.
- Good software engineering is key to making more manageable the task of developing the larger and more complex software systems we need.
- High performance is key to realizing the systems of the future that will place ever greater computing demands on hardware.
- Where do functions fit in here?

Performance Tip 5.6

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're more difficult to program, test, debug, maintain, and evolve.

Property	Recursion	Iteration
Definition	Function calls itself.	A set of instructions repeatedly executed.
Application	For functions.	For loops.
Termination	Through base case, where there will be no function call.	When the termination condition for the iterator ceases to be satisfied.
Usage	Used when code size needs to be small, and time complexity is not an issue.	Used when time complexity needs to be balanced against an expanded code size.
Code Size	Smaller code size	Larger Code Size.
Time Complexity	Very high(generally exponential) time complexity.	Relatively lower time complexity(generally polynomial-logarithmic).