BIRZEIT UNIVERSITY

Recursion

PROBLEM SOLVING AND PROGRAM DESIGN In C
7th EDITION
Jeri R. Hanly, Elliot B. Koffman

By: Mamoun Nawahdah (PhD)
2013/2014

Recursion

- A function that calls **itself** is said to be **recursive**.
- A function **f1** is also **recursive** if it calls a function **f2**, which under some circumstances calls **f1**, creating a **cycle** in the sequence of calls.
- The ability to invoke itself enables a recursive function to be repeated with different parameter values.
- You can use recursion as an alternative to iteration (looping).

The Nature of Recursion

❖ Problems that lend themselves to a recursive solution have the following characteristics:

- One or more **simple cases** of the problem have a straightforward, non recursive solution.
- The other cases can be redefined in terms of problems that are closer to the simple cases.
- By applying this redefinition process every time the recursive function is called, eventually the **problem is reduced** entirely to simple cases, which are relatively easy to solve.

The Nature of Recursion

❖ The recursive algorithms that we write will generally consist of an **if** statement with the following form:

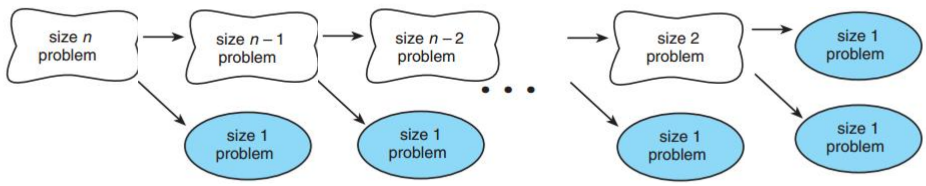
if this is a **simple case**

solve it

else

redefine the problem using recursion

Illustration



Example

- Solve the problem of **multiplying** 6 by 3, assuming we only know addition:
- **Simple case**: any number multiplied by 1 gives us the original number.
- The problem can be split into the two problems:
  1. Multiply 6 by 2. ✗
     - 1.1 Multiply 6 by 1. ✓
     - 1.2 Add **6** to the result of problem 1.1. ✓
  2. Add **6** to the result of problem 1. ✓

FIGURE 9.2 Recursive Function multiply

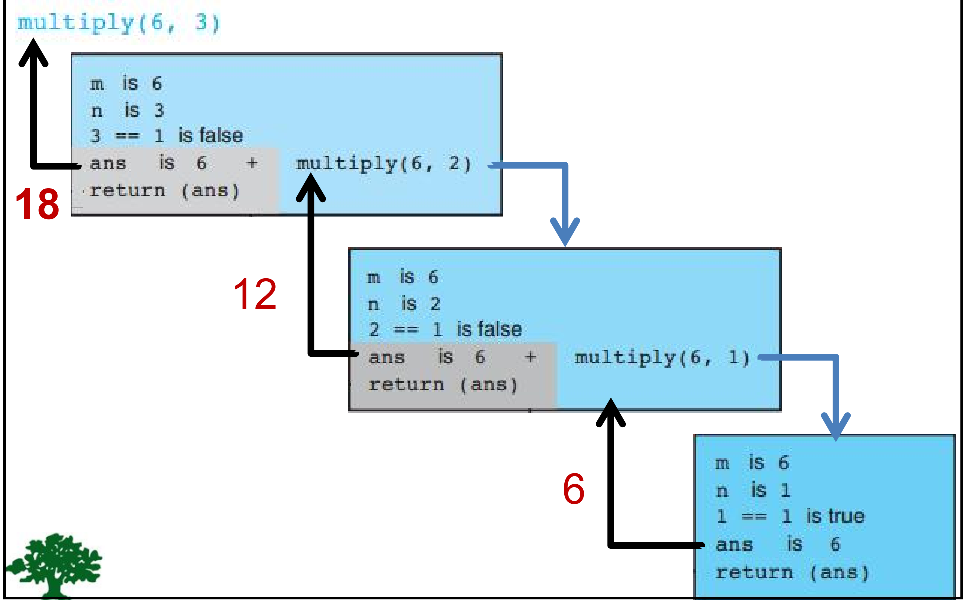
```
/*
 * Performs integer multiplication using + operator.
 * Pre:   m and n are defined and n > 0
 * Post:  returns m * n
 */
int
multiply(int m, int n)
{
    int ans;

    if (n == 1)
          ans = m;    /* simple case */
    else
          ans = m + multiply(m, n - 1); /* recursive step */

    return (ans);
}
```

The simplest case is reached when n == 1

Tracing a Recursive Function

- Hand tracing an algorithm's execution provides us with valuable insight into how that algorithm works.
- By drawing an **activation frame** corresponding to each call of the function.
- An activation frame shows the parameter values for each call and summarizes the execution of the call.

multiply(6, 3)



Self-Check

- Using diagrams (similar to previous slide) show the specific problems that are generated by the following call.

  multiply(5, 4)

- Write a recursive function **add** that computes the **sum** of its two integer parameters. Assume add does not know general addition tables but does know how to **add or subtract 1**.

Recursive Mathematical Functions

❖ Many mathematical functions can be defined recursively.

❖ An example is the **factorial** of **n** (**n!**):

- **0! is 1**

The simplest case

- **n! is n * (n– 1)!, for n> 0**

❖ Thus 4! is 4 *3!, which means 4 *3 *2 *1, or 24.

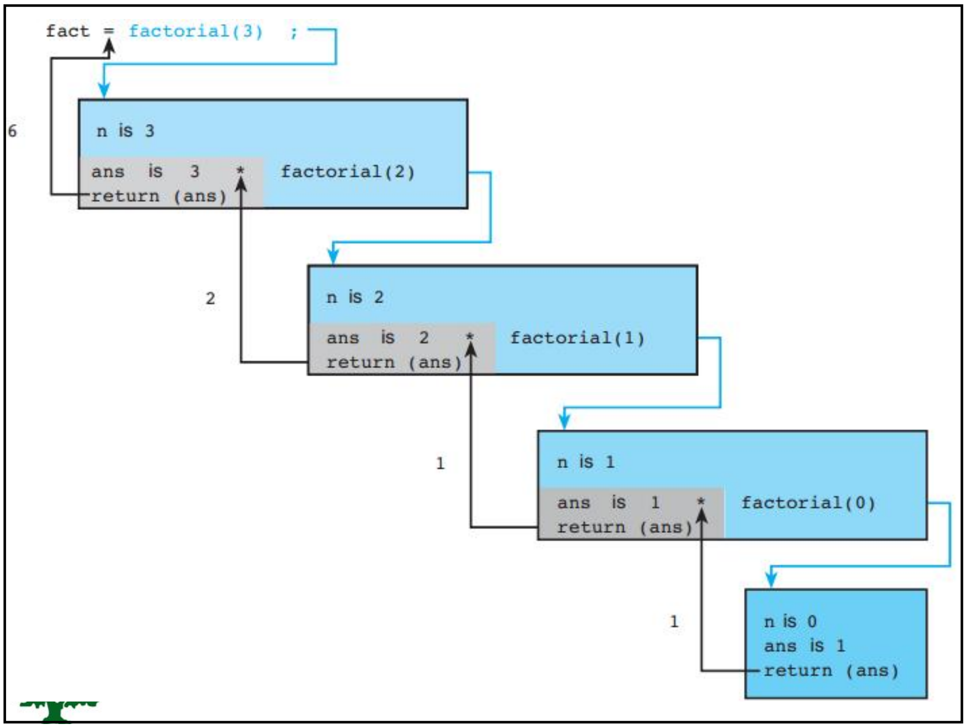
FIGURE 5.7 Function to Compute Factorial

```
/*
 * Computes n!
 * Pre: n is greater than or equal to zero
 */
int
factorial(int n)
{
    int i,          /* local variables */
        product;    /* accumulator for product computation */

    product = 1;
    /* Computes the product n x (n-1) x (n-2) x . . . x 2 x 1 */
    for (i = n; i > 1; --i) {
         product = product * i;
    }

    /* Returns function result */
    return (product);
}
```

FIGURE 9.10 Recursive factorial Function

```
/*
 * Compute n! using a recursive definition
 * Pre: n >= 0
 */
int
factorial(int n)
{
      int ans;

      if (n == 0)
            ans = 1;
      else
            ans = n * factorial(n - 1);

      return (ans);
}
```

The simplest case



Fibonacci Numbers

- The Fibonacci sequence is defined as:
  - Fibonacci 1 is 1
  - Fibonacci 2 is 1
  - Fibonacci n is Fibonacci n−2 + Fibonacci n−1, for n> 2

The simplest cases

FIGURE 9.13 Recursive Function fibonacci

```
/*
 * Computes the nth Fibonacci number
 * Pre: n > 0
 */
int
fibonacci(int n)
{
      int ans;

      if (n == 1 || n == 2)
            ans = 1;
      else
            ans = fibonacci(n - 2) + fibonacci(n - 1);

      return (ans);
}
```

Self Check

- Write and test a recursive function that returns the value of the following recursive definition:
  - f(x) = 0 if x = 0
  - f(x) = f(x - 1) + 2 otherwise
- What set of numbers is generated by this definition?

Example

```
#include <stdio.h>
int sum(int n);
int main(){
     int num,add;
     printf("Enter a positive integer:\n");
     scanf("%d",&num);
     add=sum(num);
     printf("sum=%d",add);
}
int sum(int n){
     if(n==0)
         return n;
     else
         return n+sum(n-1);
}
```

Visualization of Recursion

```
Enter a positive integer:
5
15
```

```
sum(5)
=5+sum(4)
=5+4+sum(3)
=5+4+3+sum(2)
=5+4+3+2+sum(1)
=5+4+3+2+1+sum(0)
=5+4+3+2+1+0
=5+4+3+2+1
=5+4+3+3
=5+4+6
=5+10
=15
```