Introduction to Computers & Programming

Comp 1330/ First Semester 2024/2025

Instructor: Saif Harbia

Faculty of Engineering and Technology
Department of Computer Science

Chapter 09

Recursion

Chapter Objectives:

1. How **recursion** is used as a problem solving tool.
2. Trace **recursive** functions.
3. Implement mathematical functions with **recursive** definitions as **C** functions.
4. Use **recursion** to solve problems involving arrays and strings.
5. Learn a **recursive** sort function

And more….

RECURSIVE FUNCTIONS

- A **recursive** function is:
  - A function that **calls itself.**
  - A function **f1** is also recursive if it calls a function **f2** , which under some circumstances calls **f1** (1).
- 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) (2)

9.1 THE NATURE OF RECURSION

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

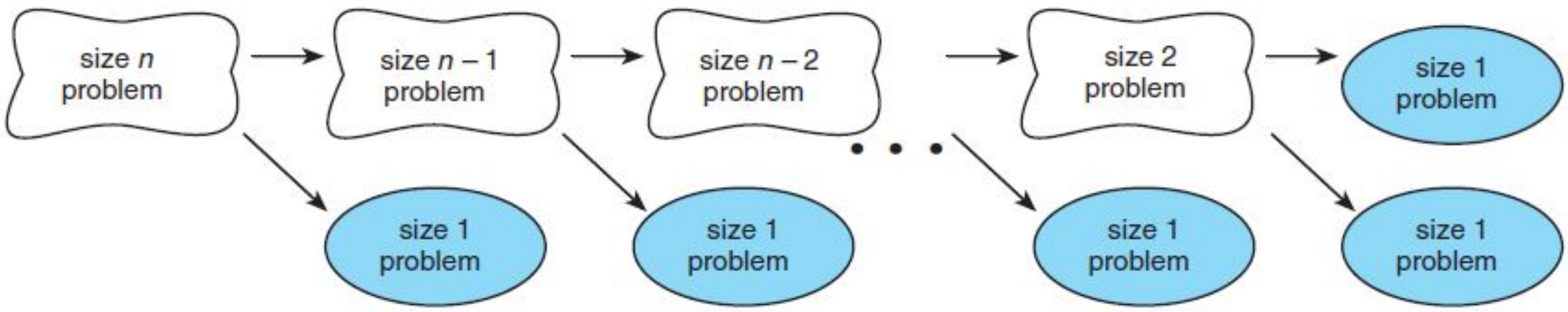
- One or more simple cases of the problem have a straightforward, **non-recursive** solution.
- The other cases can be redefined in terms of problems that are closer to the simple cases.
- By applying this redefinition process every time the recursive function is called, eventually the problem is reduced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

9.1 THE NATURE OF RECURSION

The recursive algorithms that we write will generally consist of an **if statement** with the following form (1):

```
if this is a simple case
   solve it
else
   redefine the problem using recursion
```

FIGURE 9.1 Splitting a Problem into Smaller Problems



9.1 THE NATURE OF RECURSION

- **EXAMPLE 9.1:** solve the problem of multiplying 6 by 3:

1. Multiply 6 by 2.
2. Add 6 to the result of problem 1.

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.

9.1 THE NATURE OF RECURSION

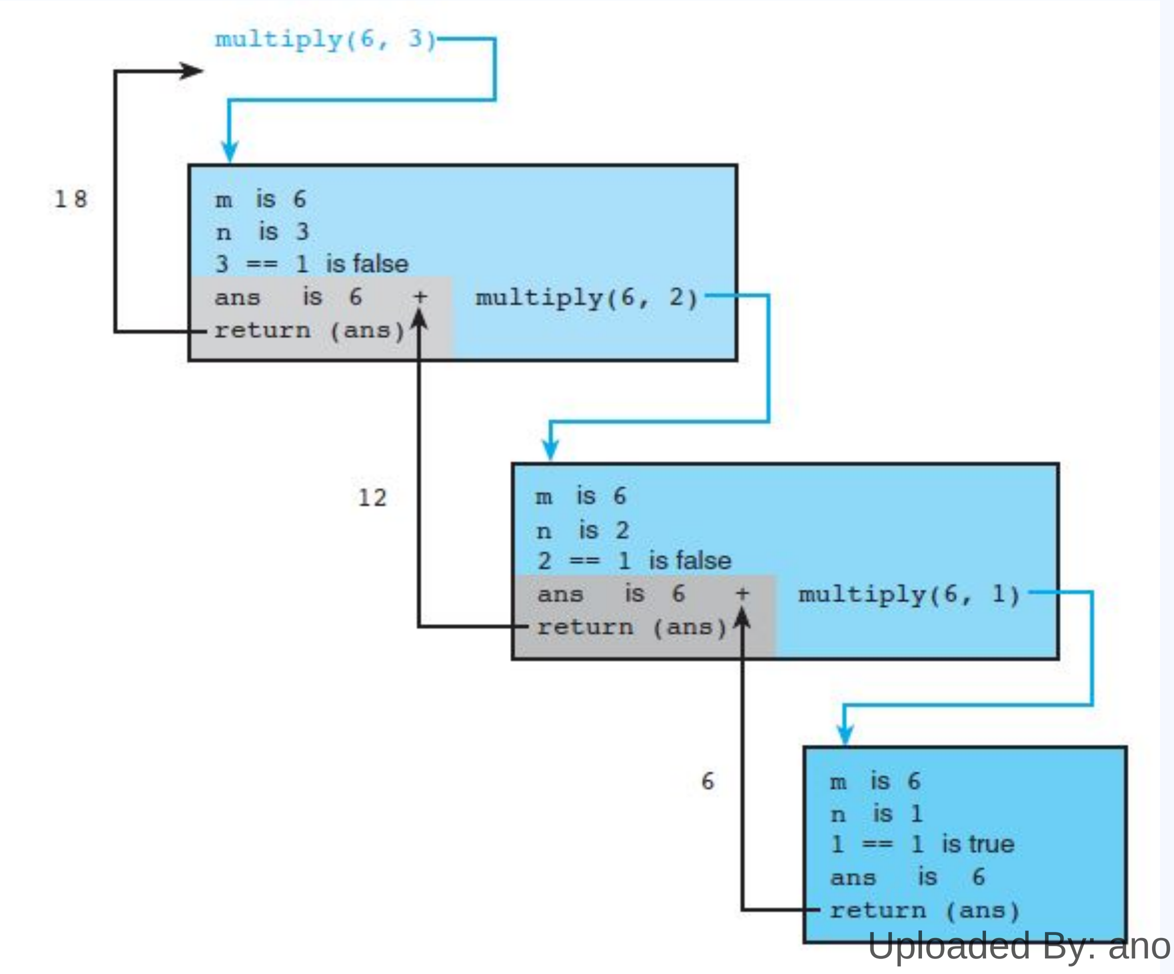
FIGURE 9.2 Recursive Function multiply

```
/*
 * 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);
}
```

- **EXAMPLE 9.2, Figure 9.4**

9.2 TRACING A RECURSIVE FUNCTION

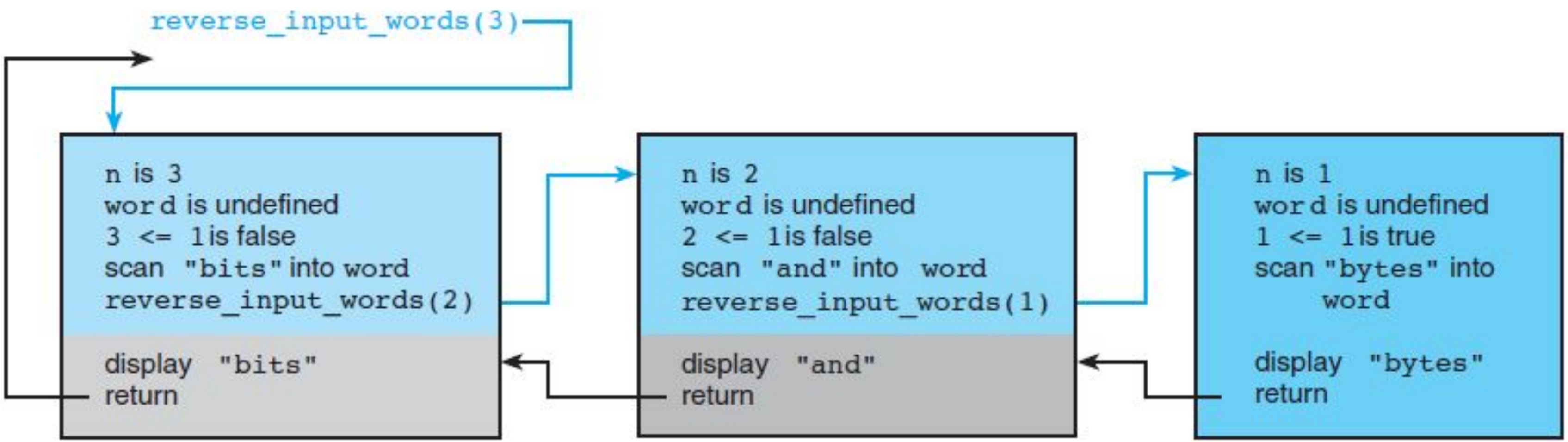
- *Figure 9.5*



TRACING A VOID FUNCTION THAT IS RECURSIVE

➢ **EXAMPLE 9.3, Figure 9.6**

FIGURE 9.7 Trace of reverse_input_words(3) When the Words Entered are "bits" "and" "bytes"



TRACING A VOID FUNCTION THAT IS RECURSIVE

FIGURE 9.8

Sequence of Events for Trace of reverse_input_words(3)

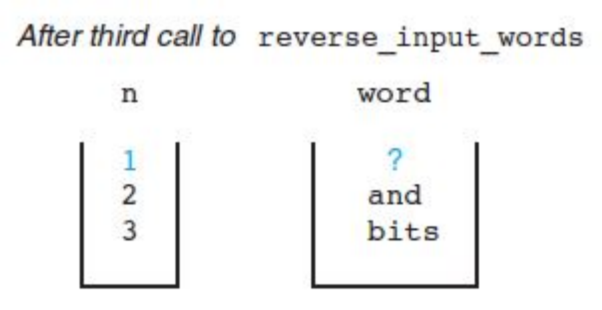
Call `reverse_input_words` with n equal to `3`.
    Scan the first word (`"bits"`) into `word`.
    Call `reverse_input_words` with n equal to `2`.
        Scan the second word (`"and"`) into `word`.
        Call `reverse_input_words` with n equal to `1`.
            Scan the third word (`"bytes"`) into `word`.
            Display the third word (`"bytes"`).
            Return from third call.
        Display the second word (`"and"`).
        Return from second call.
    Display the first word (`"bits"`).
    Return from original call.

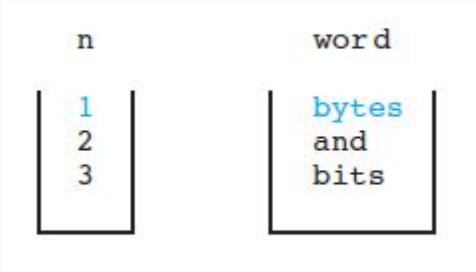
PARAMETER AND LOCAL VARIABLE STACKS

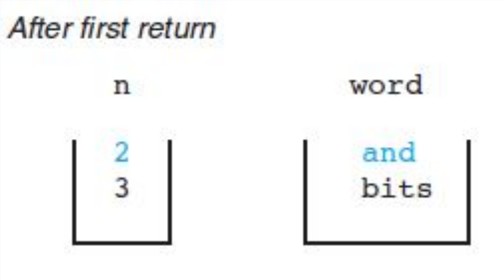
After first call to `reverse_input_words`

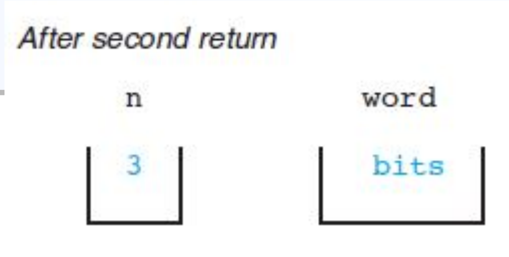
After second call to `reverse_input_words`

PARAMETER AND LOCAL VARIABLE STACKS









WHEN AND HOW TO TRACE RECURSIVE FUNCTIONS

- Doing a trace by hand of multiple calls to a recursive function is helpful in understanding how recursion works but less useful when trying to develop a recursive algorithm.
- During algorithm development, it is best to trace a specific case (1)
- Then the hand trace can check whether this value is manipulated properly to produce a correct function result for the case under consideration.
- The function can be made to trace itself by inserting debugging print statements showing entry to and exit from the function. (2)
- **FIGURE 9.9, p.531** (3)

9.3 RECURSIVE MATHEMATICAL FUNCTIONS

➢ **Example:** factorial of a number n (**n!**).

FIGURE 9.12 Iterative Function factorial

```
/*
 * Computes n!
 * Pre: n is greater than or equal to zero
 * /
int
factorial(int n)
{
      int i,                  /* local variables */
          product = 1;

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

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

9.3 RECURSIVE MATHEMATICAL FUNCTIONS

- **Factorial with recursion**

- **0! is 1 (1)**
- **n! is n X (n − 1)!, for n > 0**

4! is 4 X 3!

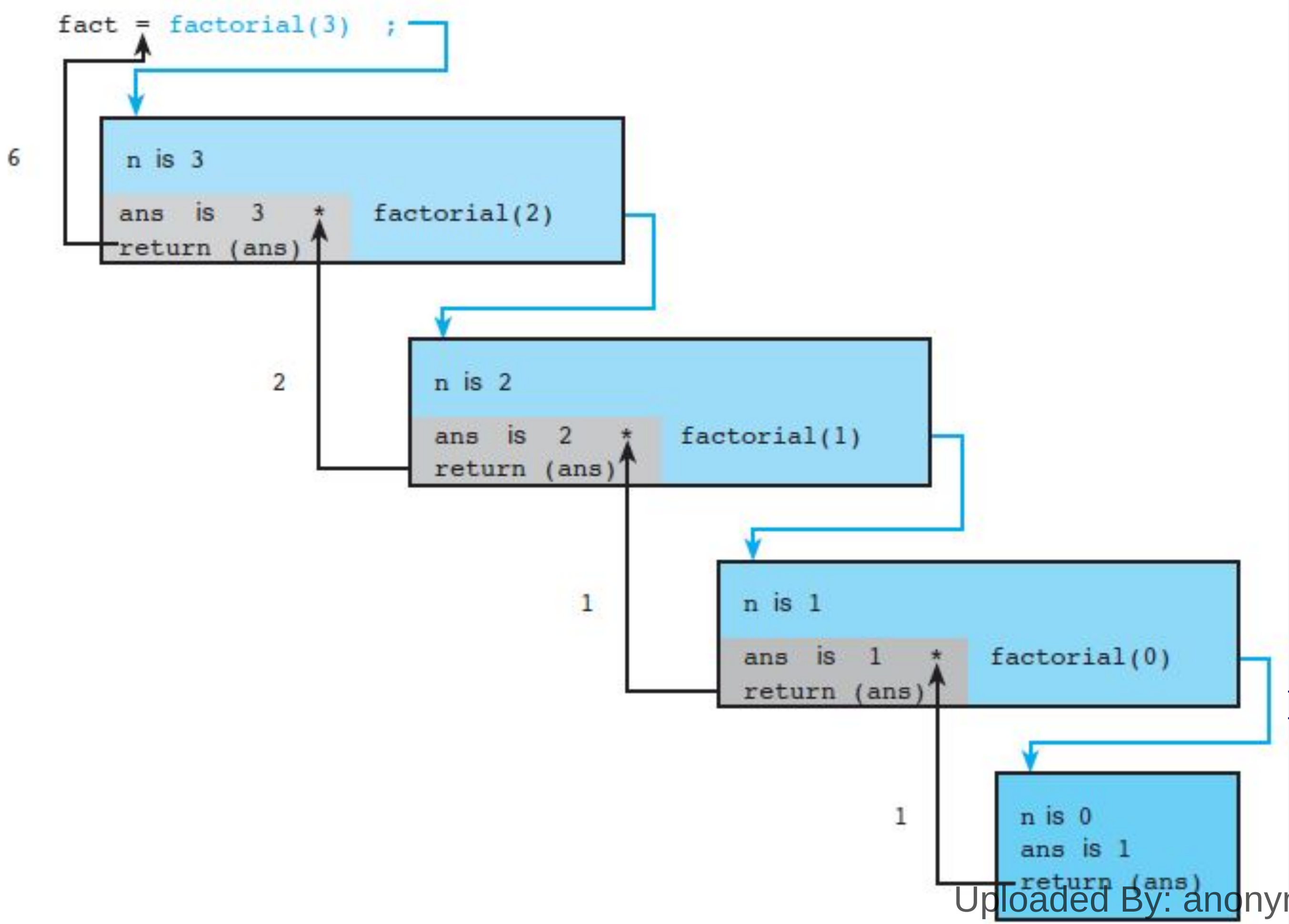
4 X 3 X 2 X 1 = 24

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);
}
```

9.3 RECURSIVE MATHEMATICAL FUNCTION

Figure9.11



➢ **EXAMPLE 9.6**: Finding the Greatest common divisor.

- The greatest common divisor of two integers is the largest integer that divides them both evenly.

➢ **Algorithm for finding gcd: (1)**

- **gcd(x , y) is y if y divides x evenly**
- **gcd(x , y) is gcd(y , remainder of x divided by y) otherwise**

➢ **Figure 9.14**.

CASE STUDY (Homework)
P.538 – 544

Finding Capital Letters in a String & Recursive Selection Sort

COMPARISON OF ITERATIVE AND RECURSIVE FUNCTIONS

- In general, if there are recursive and iterative solutions to the same problem, the recursive solution will require more time and space because of the extra function calls.

- Although recursion was not really needed to solve the simpler problems in this section, it was extremely useful in formulating algorithms to problems.

- For certain problems, recursion leads naturally to solutions that are much easier to read and understand than their iterative counterparts.

9.7 COMMON PROGRAMMING ERRORS

- The most common problem with a recursive function is that it may not terminate properly. (1)
- Frequently, a **run-time error message** noting **stack overflow** or an **access violation** is an indicator that a recursive function is not terminating.
- Make sure that you identify all simple cases and provide a terminating condition for each one.
- Also, be sure that each recursive step redefines the problem in terms of arguments that are closer to simple cases so that repeated recursive calls will eventually lead to simple cases only.

9.7 COMMON PROGRAMMING ERRORS

- The recopying of large arrays or other data structures can quickly consume all available memory. (1)

- It is also a good idea to introduce a nonrecursive function to handle preliminaries and call the recursive function when there is error checking. (2)

- Sometimes, it is difficult to observe the output produced when running recursive functions that you have made self-tracing. (3)

Refernces

Problem Solving and Program Design in C, 7th Ed., by Jeri R. Hanly and Elliot B. Koffman