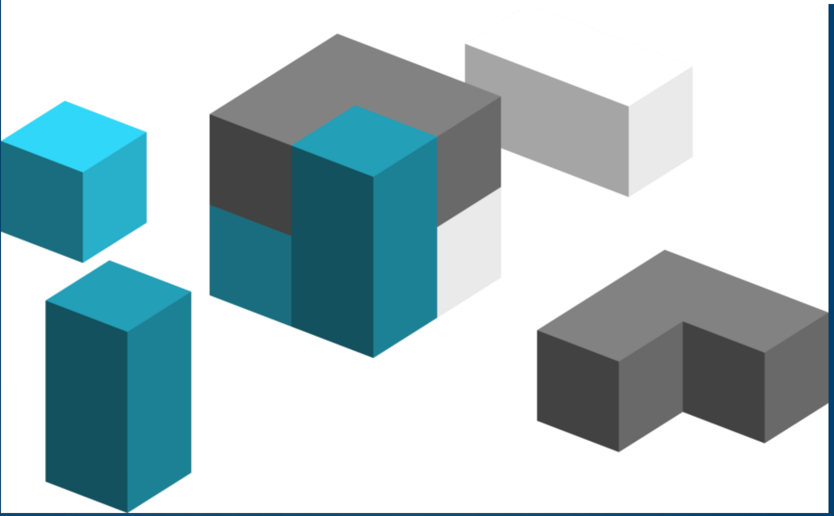


Data Structure & Algorithms 1

CHAPTER 3:

MODULAR PROGRAMMING

Sep – Dec 2023

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Fundamental Concepts of Modularity

- In a program, you may often find that a particular sequence of actions is repeated multiple times. In such cases, it's wise to write this sequence only once and reuse it as needed.
- Furthermore, you may notice that certain groups of actions relate to different tasks. It's advisable to represent each of these tasks separately in a subroutine, improving the clarity and readability of the program (or algorithm).

→ As a result, a program can be seen as a main program along with a collection of subroutines, enhancing organization and efficiency.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Fundamental Concepts of Modularity

Modularity, the cornerstone of structured programming, is simply the act of breaking down a problem into a set of reusable modules. It serves two fundamental objectives:

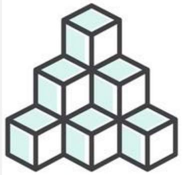
- **Decomposing Complexity**: It transforms a complex problem into "**n**" simpler problems that can be resolved independently.
- **Solution Reusability**: The idea is to find a solution for a problem just once. Once a module, designed for a specific task, is constructed, tested, and documented, it becomes a reusable asset.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Fundamental Concepts of Modularity

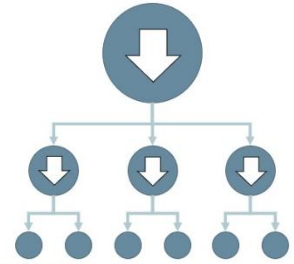
In summary, to save time and write shorter, well-structured algorithms, we group one or more blocks performing specific tasks into a **MODULE**. We assign a **NAME** to this module, and subsequently, we can "call" it whenever needed, simply by referencing its name. There's no need to reconstruct it.

The advantages of modularity make it not just interesting but highly recommended to systematically employ modular design in constructing our solutions.



Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Fundamental Concepts of Modularity



- Algorithm design naturally follows the top-down approach through progressive refinement:
  1. We start by breaking down our problem into logically coherent modules.
  2. Next, we separately construct each module, whether they are caller modules (calling other modules) or callee modules (called by other modules), along with the main algorithm. We treat them as if they were **independent** problems. As mentioned earlier, these modules may even be assigned to different developers.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Fundamental Concepts of Modularity

- When using a module, **we no longer need to know** how it is constructed **but** only what it does.

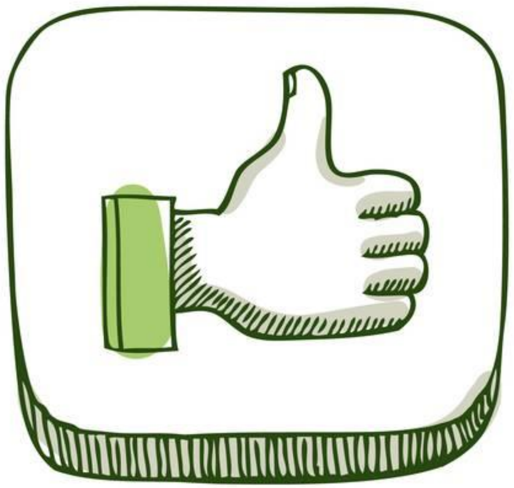
- The role you assign to each module is crucial, because if it is inadequately defined, ambiguous, or incomplete, your module becomes entirely unusable and, as a result, USELESS.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Fundamental Concepts of Modularity

Advantages

- Improved readability
- Reduced risk of errors
- Selective testing possibilities
- Reuse of existing modules
- Ease of modification
- Promote collaborative work
- …..



Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Fundamental Concepts of Modularity

```
Algorithm Algo_1

BEGIN
    Bloc A
    Bloc B
    Bloc C
    Bloc A
    Bloc B
    Bloc C
END
```

MODULE A

MODULE B

MODULE C

```
Algorithm Algo_1
Modules A, B, C
BEGIN

    Call Module A

    Call Module B

    Call Module C

    Call Module A

    Call Module B

    Call Module C

END
```

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Fundamental Concepts of Modularity (calling a module)

```
Algorithm Algo_1
Variables…
   Module A
BEGIN

   -

   -

   Call Module A

   -

   -

END
```

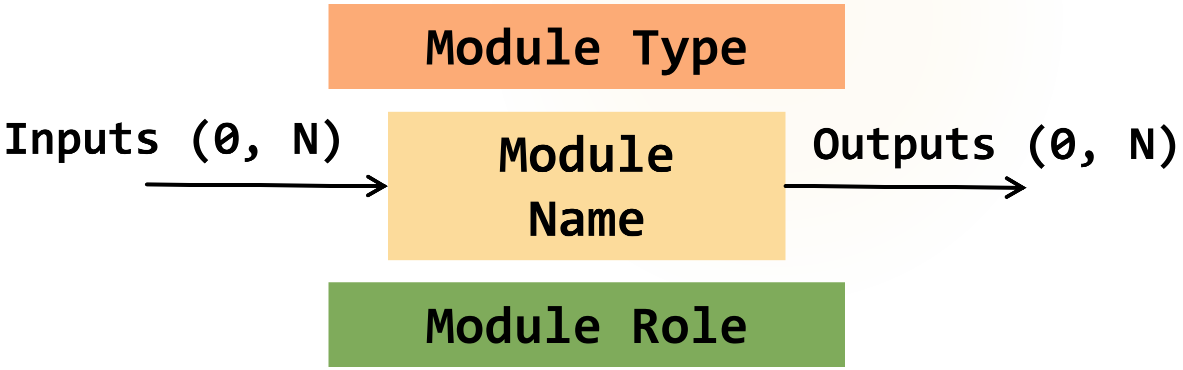
```
Module A
Variables...
BEGIN
  -
  -
  -

END
```

When a module **call** is encountered, the execution of the calling module is **suspended** until the called module is entirely executed, and then the execution of the calling module **resumes** immediately after the call.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Fundamental Concepts of Modularity (calling a module)

A module is considered as a black box that performs a specific task.
From a syntax perspective, a module follows the same structure as an algorithm. It is defined by **its name, its role, its type, and its interfaces** which means the data it receives as input and the data it returns to the caller.



Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Fundamental Concepts of Modularity (calling a module)

The interface consists of the inputs / outputs of the module; it is what allows establishing the **link** between the **module** and its **environment**.

Function

Integer N → **NbDigits** → Integer

Role: return the # digits in N

```
Algorithm Example_1
Variable Integer N, NbD
Integer Function NbDigits (Integer N)
   The body of NbDigits function…
BEGIN
   READ(N)
   NbD = NbDigits(N)
   WRITE (NbD)
END
```

```
Integer Function NbDigits (Integer N)
Variable Integer i
BEGIN
    i = 0
    WHILE (N > 0) DO
        i = i + 1
        N = N DIV 10
    END WHILE
    NbDigits = i
END
```

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

How to proceed?

First step: understanding the problem

Second step: problem analysis and design

1. break down the problem.
2. construct the modules.

Third step: implementation

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

Dividing the Problem

- Identify the modules to be built. Some modules are evident and can be quickly detected, while others may not be apparent at first.
- Don't waste your time and energy trying to list all the necessary modules.
- Start with the obvious modules and expand your division as you see fit.
- Sometimes the division is implicit, meaning that a careful reading of the problem will help you identify the modules to build, while in other cases, it may require analysis and creativity.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

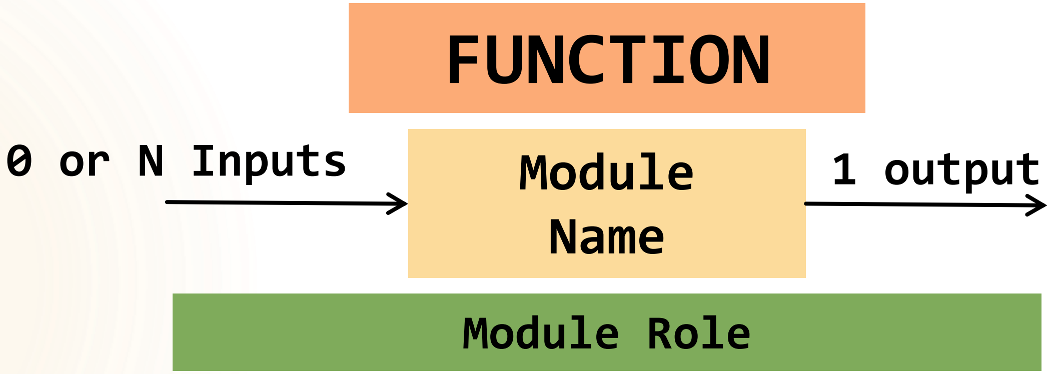
Module Quality

- **Reusability**: Always aim to make your modules as general as possible for later reuse.
- **Independence**: Avoid using <u>read</u> or <u>write</u> operations in a module, and the use of global variables. (best way to prevent side effects)
- **Simplicity**: A module should have a **SINGLE** clear task to perform.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

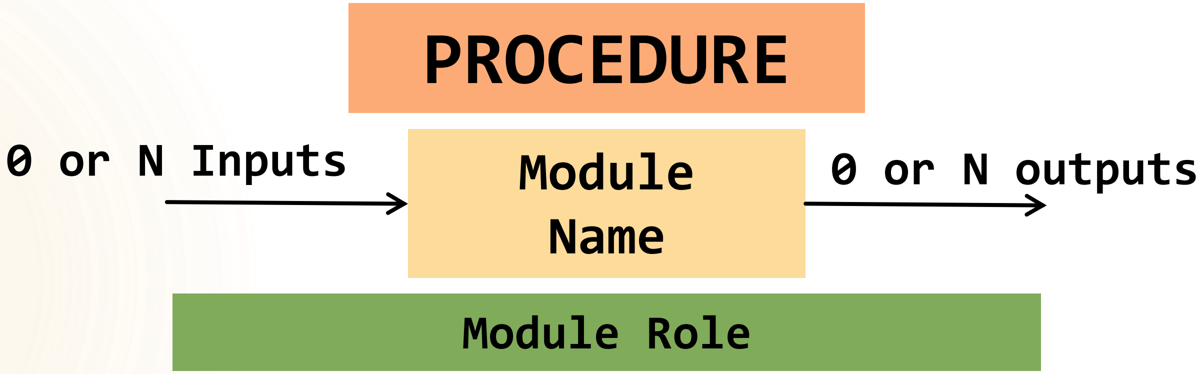
When a module has only one output, and it's a basic data element, it's called a FUNCTION. Otherwise, it's a PROCEDURE, meaning it can have zero to many outputs of any type.



Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

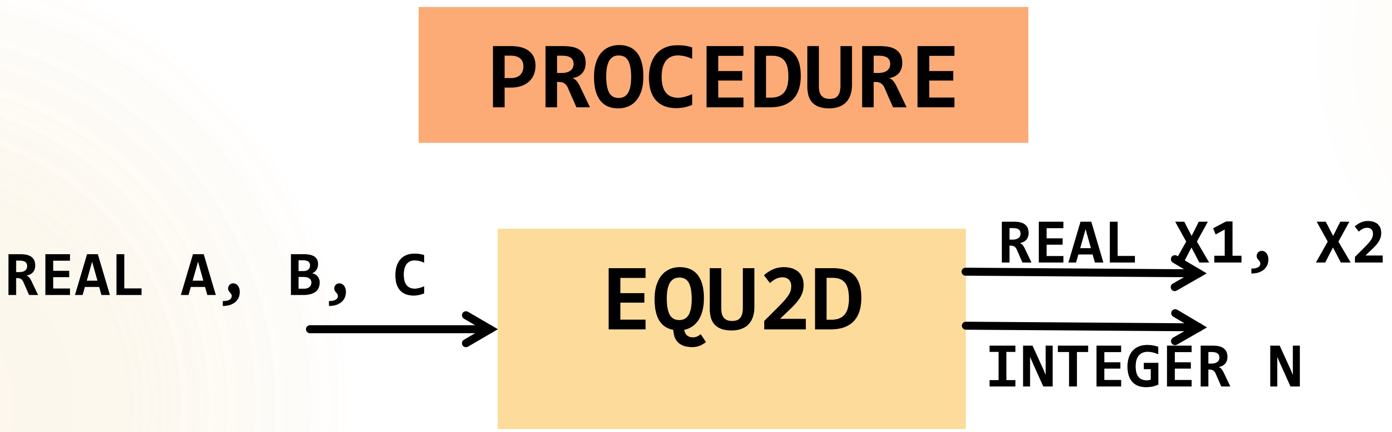
When a module has only one output, and it's a basic data element, it's called a FUNCTION. Otherwise, it's a PROCEDURE, meaning it can have zero to many outputs of any type.



Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

EXAMPLE 1:



Role: Find the number of solutions (N) and the roots (X1 and X2) of a quadratic equation with coefficients A, B, and C.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

EXAMPLE 2:

FUNCTION

INTEGER N → **FACTO** → INTEGER

Role: Calculte the factorial of an integer N

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

FUNCTION STRUCTURE

Function Head → **Result-Type FUNCTION Function_name (Input formal parameters)**

Environment → **Declarations**

Function Body →

BEGIN

-
-
-

Function_name = Result of the function

END

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

FUNCTION STRUCTURE

- The body of the function can contain all the declarations and algorithmic structures.
- The **result** must always be **transmitted** in the **name of the function**, and this assignment is typically the final action of the function.
- The list of formal parameters describes the objects provided to the function, including their types and their passing mode (this concept is discussed later)."

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

FUNCTION Call

The call to a function is made by referencing its name to the right of the assignment symbol in a condition or in a procedure or function call.

```
Algorithm Example_Prime
Variable Integer N
         Boolean Res
Boolean Function PRIME (Integer N)
   The body of PRIME function…
BEGIN
   READ(N)
   Res = PRIME(N)
   IF Res == True Then
       WRITE (N, ‘ is Prime’)
   ELSE
       WRITE (N, ‘ is not Prime’)
   END IF
END
```

```
Boolean Function PRIME (Integer N)
Variable Integer i
BEGIN
    i = 2
    WHILE (N MOD i <> 0) AND (i <= N DIV 2) DO
        i = i + 1
    END WHILE
    PRIME = ((N == 2) OR (i > N DIV 2))
END
```

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

Functions in C++

- **Local variables**
  - Known only in the function in which they are defined
  - All variables declared in function definitions are local variables
- **Parameters**
  - Local variables passed to function when called
  - Provide outside information

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

Functions in C++

- Function prototype
  - Tells compiler argument type and return type of function
  - **int square(int);**
    - Function takes an **int** and returns an **int**
  - Explained in more detail later

- Calling/invoking a function
  - **square(x);**
  - Parentheses an operator used to call function
    - Pass argument x
    - Function gets its own copy of arguments
  - After finished, passes back result

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

Functions in C++

- Format for function definition

```
return-value-type  function-name( parameter-list )
{
    declarations and statements
}
```

  - Parameter list
    - Comma separated list of arguments
      - Data type needed for each argument
    - If no arguments, use **void** or leave blank
  - Return-value-type
    - Data type of result returned (use **void** if nothing returned)

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

Functions in C++

- Example function

```
int square( int y )
{
   return y * y;
}
```

- **return** keyword
  - Returns data, and control goes to function's caller
    - If no data to return, use **return;**
  - Function ends when reaches right brace
    - Control goes to caller
- Functions cannot be defined inside other functions

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

Functions in C++

EXAMPLE

```cpp
// Finding the maximum of three floating-point numbers.
//
    #include <iostream>

    using std::cout;
    using std::cin;
    using std::endl;

    double maximum( double, double, double ); // function prototype

   int main()
   {
      double number1;
      double number2;
      double number3;

      cout << "Enter three floating-point numbers: ";
      cin >> number1 >> number2 >> number3;

      // number1, number2 and number3 are arguments to
      // the maximum function call
      cout << "Maximum is: "
           << maximum( number1, number2, number3 ) << endl;

      return 0;  // indicates successful termination
```

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

Functions in C++

EXAMPLE

```cpp

} // end main

// function maximum definition;
// x, y and z are parameters
double maximum( double x, double y, double z )
{
   double max = x;   // assume x is largest

   if ( y > max )    // if y is larger,
      max = y;       // assign y to max

   if ( z > max )    // if z is larger,
      max = z;       // assign z to max

   return max;       // max is largest value

} // end function maximum
```

```
Enter three floating-point numbers: 99.32 37.3 27.1928
Maximum is: 99.32

Enter three floating-point numbers: 1.1 3.333 2.22
Maximum is: 3.333

Enter three floating-point numbers: 27.9 14.31 88.99
Maximum is: 88.99
```

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Function declaration and use

Result type = Function type

```
Integer FUNCTION  LeastCommonMultiple (Integer A, B)
Integer FUNCTION  Factorial (Integer N)
Boolean FUNCTION  Prime (Integer N)
```

Formal parameters

Function call

Functions call:

```
WRITE ('The LCM of ' i ' and ', j, ' is: ', LeastCommonMultiple(i, j))
WRITE ('The Factorial of ' N, ' is: ‘, Factorial(N))
IF Prime (i * j + 25) THEN …
```

Actual parameters

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Function declaration and use

CAUTION ⚠

- The number of actual parameters must be equal to that of formal parameters.
- The order of actual parameters must match the order of formal parameters.
- There must be type compatibility between formal and actual parameters.

```
Integer FUNCTION  LeastCommonMultiple (Integer A, B)

READ (i, j)                          // i, j and Res must be of type Integer
Res = LeastCommonMultiple(i, j)
WRITE ('The LCM of ' i ' and ', j, ' is: ‘, Res)
```

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

PROCEDURE

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

PROCEDURE STRUCTURE

Procedure Head → **Procedure** Procedure_name (Input/output formal parameters)

Environment → **Declarations**

Function Body →

BEGIN

-
-
-
-
-

END

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

PROCEDURE STRUCTURE

- The body of the procedure can contain all the declarations and algorithmic structures.
- The list of formal parameters describes the input and **output** parameters including their types and their passing mode
- The parameter passing is identical to that of functions, but **ALL** output parameters must be defined with a **pass-by-variable** method.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

PROCEDURE Call

Calling a procedure is done by stating its name followed by the list of actual parameters, just like an **instruction**.

The calling of a procedure is, in fact, a primitive action, consisting of the procedure name followed by the list of actual input and output parameters enclosed in parentheses, separated by commas.
Just like with functions, the number, order, and type of actual parameters **must match** those of the formal parameters.

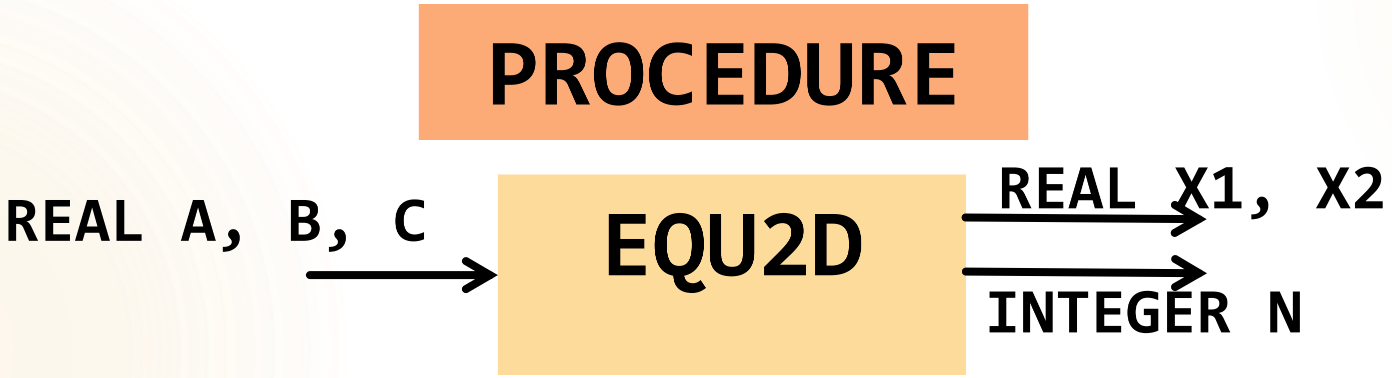
Call Example: `EQU2D(e, f, g, Nr, S1, S2)`

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

Procedure Declaration (Example):

Write an algorithm that solves a second-degree equation with coefficients A, B, and C read from the keyboard.



Role: Find the number of solutions (N) and the roots (X1 and X2) of a quadratic equation with coefficients A, B, and C.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

Procedure Declaration (Example):

Analysis:

- Calculation of Delta = BB - 4A*C
- IF (Delta < 0) THEN
  N = 0
- ELSE
  - IF (Delta == 0) THEN
    N = 1
    X1 = -B / (2*A)
    X2 = X1
  - ELSE
    Ns = 2
    X1 = (-B - SQRT(Delta)) / (2*A)
    X2 = (-B + SQRT(Delta)) / (2*A)
  - END IF
- END IF

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

Procedure Declaration (Example):

```
PROCEDURE EQU2D(Real A, B, C; Var Integer N; Var Real X1, X2)
Variable Real Delta
BEGIN
    Delta = B*B - 4*A*C
    IF (Delta > 0) THEN
        X1 = (-B - SQRT(Delta)) / (2*A)
        X2 = (-B + SQRT(Delta)) / (2*A)
        N = 2
    ELSE
        IF (Delta == 0) THEN
            X1 = -B / (2*A)
            x2 = X1
            N = 1
        ELSE
            N = 0
        END IF
    END IF
END
```

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

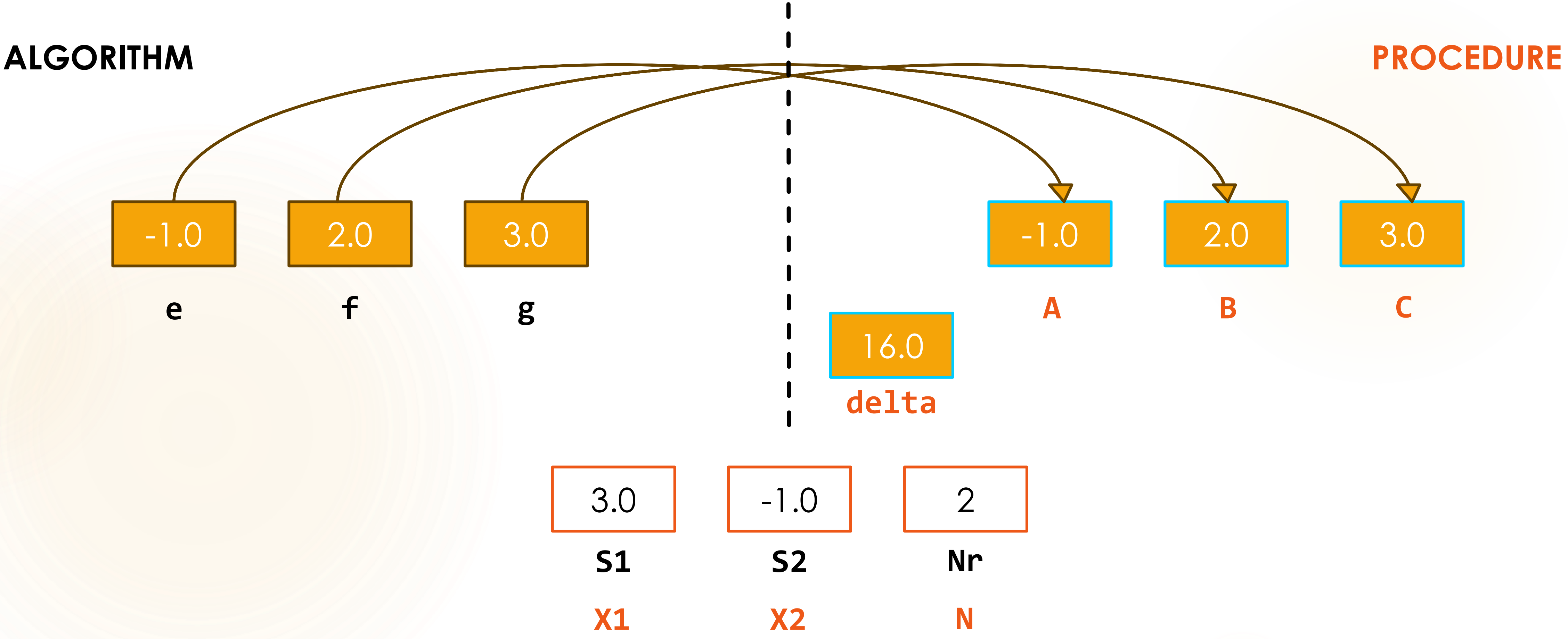
Procedure Declaration (Example):

```
Algorithm Example_Procedure
Variable Real e, f, g, S1, S2
         Integer Nr
         //Procedure declaration
Procedure EQU2D(Real A, B, C; Var Integer N; Var Real XI, X2)
   … The body of EQU2D procedure here …
BEGIN
   WRITE (‘Give the parameters: A, B and C: ’)
   READ(e, f, g)
   EQU2D(e, f, g, Nr, S1, S2)
   CASE OF Nr
      0: WRTIE (‘No Solution’)
      1: WRITE (‘One double root X1 = X2 = ‘, S1)
      2: WRITE (Two distinct roots, X1 = ’, S1, ‘X2 = ’, S2)
   ENDCASE
END
```

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Approach and Formalism

Procedure Declaration (Example):



Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

MODULAR DECOMPOSITION EXAMPLE

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Decomposition:
Example

Consider the sequence: `1, 11, 21, 1112, 3112, ...`
What is the next element? You've found it!
Now, let's build an analysis to obtain an element from the previous one.

Justification:
We observe that each element is derived from the previous one. It consists of the number of **1s** followed by **1** and then the number of **2s** followed by **2**. These elements are concatenated as we progress.

`1, 11, 21, 1112, 3112, 211213 ...`

This sequence is formed by concatenating the counts of 1s and 2s to the previous element.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Decomposition:
Example

ANALYSIS

- Let N be an integer representing an element of the sequence.
- For each digit C (from 1 to 9):
  - **Count** the number of times (NB) the digit C appears in N.
  - **Compose** the desired number N2.
- Display N2.
- Additional instructions may be required.

We will need two modules:

- A module that calculates the number of occurrences of a given digit in a number (**FREQDG**).
- A module that concatenates two numbers (**CONCAT**).

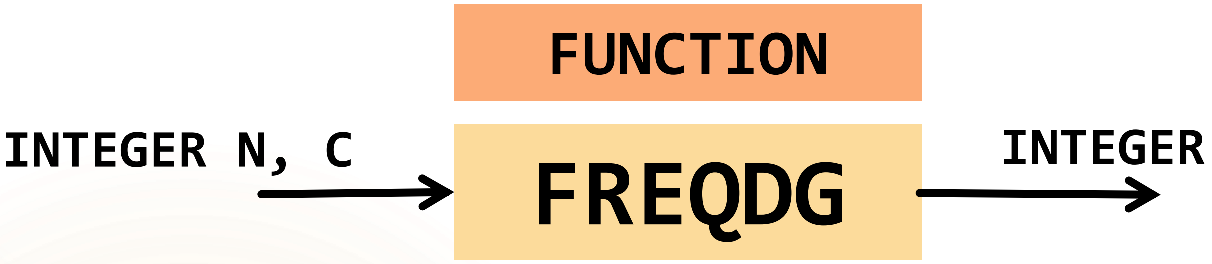
Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Decomposition:
Example

```
Algorithm Sequence
Variable Integer N, C, N2, N3, Nb
Integer Function FREQDG (Integer N , C)
… //Function Body here
Integer Function CONCAT (Integer A , B)
… //Function Body here
BEGIN
    READ(N)
    N2 = 0
    FOR C FROM 1 TO 9 DO
        Nb = FREQDG(N,C)
        IF (Nb <> 0) THEN
            N3 = CONCAT(Nb, C)
            N2 = CONCAT(N2, N3)
        END IF
    END FOR
    WRITE(N2)
    N = N2
END
```

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Decomposition:
Example



Role: Provide the number of times the digit C appears in N

ANALYSIS:

The algorithm decomposes N digit by digit until a quotient of 0 is obtained. For each digit obtained (N Mod 10), if it is equal to C, it is counted.

```
Integer Function FREQDG (Integer N , C)
Variable Integer Cpt, ND, i
BEGIN
    Cpt = 0
    ND = NbDigit(N)
    FOR i FROM 1 TO ND DO
        IF (N Mod 10 == C) THEN
            Cpt = Cpt + 1
        END IF
        N = N Div 10
    END FOR
    FREQDG = Cpt
END
```

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Decomposition:
Example

FUNCTION

INTEGER A, B → **CONCAT** → INTEGER

Role: Concatenate integer A with integer B

```
  1230000
+    4567
-------------
= 1234567
```

ANALYSIS:
When you want to concatenate N1 (123) and N2 (4567), you can achieve this by adding **4 zeros** to N1 and then adding N2 (**4** is the number of positions in N2).

Result = N1 * 10^Np + N2 = N1 * N3 + N2, where N3 = 10^Np.
In this context, Np represents the number of positions in N2 that determine how many zeros need to be added to N1 for concatenation.

From the analysis we need to construct the following 2 modules: **CONCAT, NbDigits**

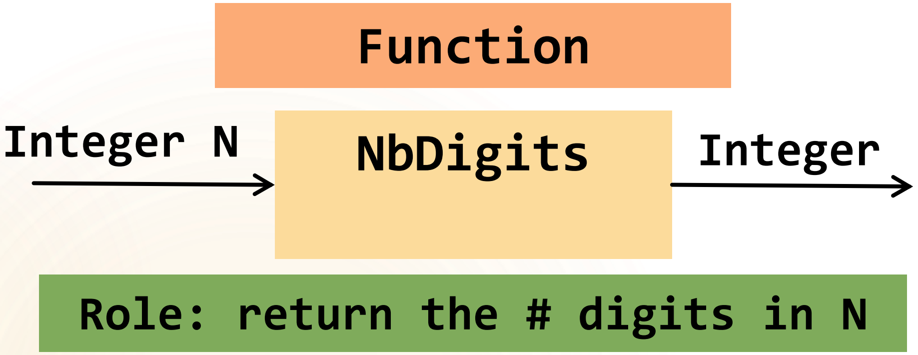
Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Modular Decomposition:
Example

```
Integer Function NbDigits(Integer N)
  //….. Function Body

Integer Function CONCAT(Integer A, B)
Variables Integer NP, P1
BEGIN
    NP = NbDigits (B)
    P1 =  10 ** NP
    CONCAT = A* P1 + B
END
```

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

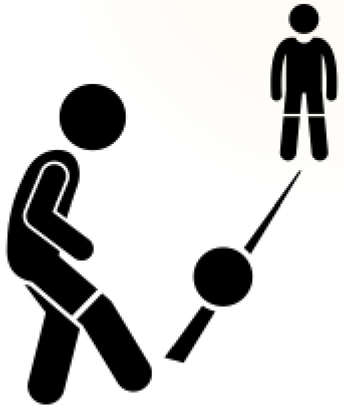
Modular Decomposition:
Example




```
Integer Function NbDigits (Integer N)
Variable Integer i
BEGIN
    i = 0
    WHILE (N > 0) DO
        i = i + 1
        N = N DIV 10
    END WHILE
    NbDigits = i
END
```

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

PRAMETER PASSING



Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Parameter Passing

Parameter passing is a concept that needs to be **mastered**; otherwise, there is a risk of losing control over the functioning of modules, especially in terms of the **flow** of our algorithms.

There are two modes of parameter passing:

1. **Value passing mode**
2. **Reference or variable passing mode**

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Parameter Passing

- When a parameter is provided to a module, and we want the module to return the same value that the parameter had at the input, we will perform a **value passing** of parameters.

- But when a parameter is modified during the execution of a module, and we want to have the modified content of the parameter at the output of the module, we will perform a **variable (or reference) passing**. In this case, the formal parameter must be preceded by the word "**VAR**."

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Parameter Passing

In general

It is necessary to use:

- Variable passing for ALL output parameters of a procedure-type module.
- Value passing for input parameters of a function.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Parameter Passing

```
PROCEDURE EQU2D(Real A, B, C; Var Integer N; Var Real X1, X2)
```

The **absence** of "**VAR**" indicates that the parameter passing mode is a **value passing** mode.

6 Formal parameters

"**VAR**" indicates that the parameter passing mode is a **variable passing** mode.

Procedure Call:

```
EQU2D (i, j, k, Nr, r1, r2)

EQU2D (-2.0, 3.0, 1, Nr, r1, r2)
```

6 Actual parameters

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Parameter Passing
in C++

Call by value

- Copy of data passed to function
- Changes to copy do not change original

Call by reference

- Function can directly access data
- Changes affect original

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Parameter Passing
in C++

- Reference parameter
  - Alias for argument in function call
    - Passes parameter by reference
  - Use & after data type in prototype
    - `void myFunction( int &data )`
    - Read "**data** is a reference to an **int**"
  - Function call format the same
    - However, original can now be changed

- Function call with omitted parameters
  - If not enough parameters, rightmost go to their defaults
  - Default values
    - Can be constants, global variables, or function calls
- Set defaults in function prototype

```
int myFunction( int x = 1, int y = 2, int z = 3 );
```

  - `myFunction(3)`
    - **x = 3**, **y** and **z** get defaults (rightmost)
  - `myFunction(3, 5)`
    - **x = 3**, **y = 5** and **z** gets default

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Math Library in C++

Method	Description	Example
ceil(x)	rounds *x* to the smallest integer not less than *x*	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
cos(x)	trigonometric cosine of *x* (*x* in radians)	cos(0.0) is 1.0
exp(x)	exponential function e^x	exp(1.0) is 2.71828 exp(2.0) is 7.38906
fabs(x)	absolute value of *x*	fabs(5.1) is 5.1 fabs(0.0) is 0.0 fabs(-8.76) is 8.76
floor(x)	rounds *x* to the largest integer not greater than *x*	floor(9.2) is 9.0 floor(-9.8) is -10.0
fmod(x, y)	remainder of *x/y* as a floating-point number	fmod(13.657, 2.333) is 1.992
log(x)	natural logarithm of *x* (base *e*)	log(2.718282) is 1.0 log(7.389056) is 2.0
log10(x)	logarithm of *x* (base 10)	log10(10.0) is 1.0 log10(100.0) is 2.0
pow(x, y)	*x* raised to power *y* (x^y)	pow(2, 7) is 128 pow(9, .5) is 3
sin(x)	trigonometric sine of *x* (*x* in radians)	sin(0.0) is 0
sqrt(x)	square root of *x*	sqrt(900.0) is 30.0 sqrt(9.0) is 3.0
tan(x)	trigonometric tangent of *x* (*x* in radians)	tan(0.0) is 0

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

LOCAL & GLOBAL OBJECT

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Local and Global Objects

A **block** corresponds to a **region** of a program.
In a program, blocks can be nested within each other.
This structure, known as the block structure, defines levels of blocks:

- The **main** program forms block **level 0**.
- **Procedures** and **functions** declared directly within the main program each form a block of **level 1**.
- **Procedures** and **functions** declared within level 1 procedures or functions each form a block of **level 2**.

Level 0 Main Program
|
Level 1 Procedures Functions in Main Program
|
Level 2 Procedures Functions in Level 1
|
...

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Local and Global Objects

- Procedures and functions declared within procedures or functions of level N each form a block of level N+1.

- The level number is referred to as the depth of the level. A block of level 5 is deeper than a block of level 2.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Local and Global Objects

Scope of an object

An object is known and usable:

- In its defining block as soon as the object has been declared.
- In all blocks at deeper levels if they are declared within and after its defining block.

The **region** of a program where an object **can be used** is called the **object's scope**.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Local and Global Objects

Scope of an object

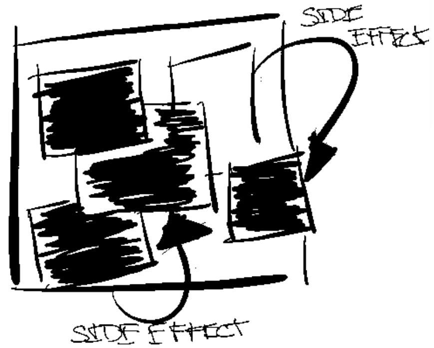
An object is known and usable:

- An object is local to a block if it is declared in the declarative part of the block.

- An object is global to a block if it is declared outside the block, and the block is within the scope of that identifier.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

SIDE EFFECT



Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Side Effects

A side effect is when a function relies on, or modifies, something outside its parameters to do something.

For example, a function which reads or writes from a variable outside its own arguments, a database, a file, or the console can be described as having side effects.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Side Effects

```
Algorithm Side_Effect
Variable Integer A, B, E
Integer Function Test(Integer C)
   BEGIN
       A = A * B
       B = A - C
       Test = A + B
   END
BEGIN
   A = 1
   B = 5
   E = 1
   WRITE (Test(E))
   WRITE (Test(E))
   WRITE (Test(E))
END
```

Caller Module

A	B	E
1	5	
5	4	
20	19	1
380	379	

Callee Module

C	Test
1	9
1	39
1	759

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024

Side Effects

To avoid side effects, simply follow the following rule:

- Whenever possible, do not use global variables within a module. Otherwise, only modify local objects.

Copyright © Dr. Noureddine Lasla/ Dr. Fouzia Anguel, 2023/2024