Ministry of Higher Education and Scientific Research
Djilali BOUNAAMA University - Khemis Miliana(UDBKM)
Faculty of Science and Technology
Department of Mathematics and Computer Science

Chapter 2

Subprograms:
Functions and Procedures

MI-L1-UEF121 : Algorithms and Data Structures II

Noureddine AZZOUZA

n.azzouza@univ-dbkm.dz

Course Topics

1. Modularity
2. Passing parameters
3. Local variables and global variables
4. Functions
5. Procedures

Modularity


source_fuel
fuel_source
air_source
General
Fuel Properties
simulate_aero
aero_volume
aero_points
sources
colliders
influence_fields
settings
adaptivity
collider
collider
geometry
General
Common Properties
assign_material
out_geometry
geometry
set_surface
surface_material
set_displacement
displacement_material
set_volume
volume_material
material_reference
aero_solver_settings
combustion_settings
simulation_bounds
General
Aero Solver Globals
Accuracy
Diffusion Handling
"assign_material" Type: Rendering::Materials/assign_material

Modularity

Problem

Combination Calculation

Find the number of combinations of p objects among n such that

$$C_n^p = \frac{n!}{p!*(n-p)!}$$

```
Algorithm Calcul_ Combinaison ;
Var n,p,c : integer ;
      fact_ n, fact_ p, fact_ np: integer;
Begin
   //inputs
   Read (n,p);

    //manipulating data
    fact_ n = 1;
   For i ← 1 to n Do
        fact_ n ← fact_ n * i;

    fact_ p = 1;
   For i ← 1 à p Do
        fact_ p ← fact_ p * i;

    fact_ np = 1;
    For i ← 1 à (n-p) Do
        fact_ np ← fact_ np * i;

    c ← fact_ n/(fact_ p*fact_ np);

    //outputs
    Write ('number of combinations=',c);
End.
```

Problem

Repeating the factorial calculation

$fact\_n = 1;$
For $i \leftarrow 1$ to n **Do**
    $fact\_n \leftarrow fact\_n * i;$

$fact\_p = 1;$
For $i \leftarrow 1$ to p **Do**
    $fact\_p \leftarrow fact\_p * i;$

$fact\_np = 1;$
For $i \leftarrow 1$ to $n\text{-}p$ **Do**
    $fact\_np \leftarrow fact\_np * i;$

How to write the solution only once? Organize the code?

Modules / Subprograms

Definition

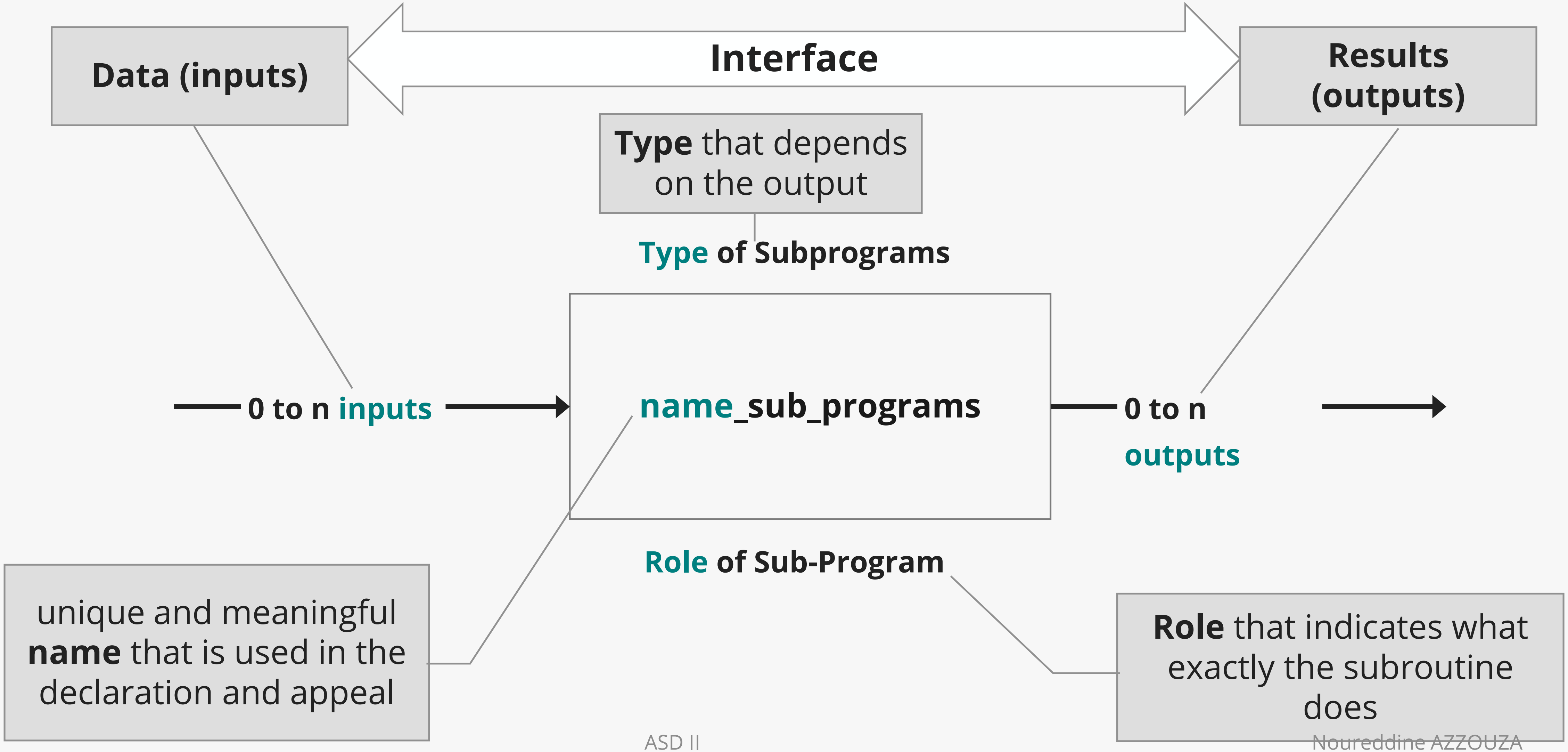
✓ A **subprogram** or module is a set of instructions with a well-defined interface that performs a specific task.

✓ The purpose of a subroutine is:

1. Receive input data
2. Carry out processing/transformation of this data
3. Return one or more results

✓ The **interface** consists of the inputs/outputs of the module. It makes it possible to establish the link between the module and its environment (main algorithm, other modules).

Structure


Data (inputs)
Interface
Results (outputs)
Type that depends on the output
Type of Subprograms
0 to n inputs
name_sub_programs
0 to n outputs
Role of Sub-Program
unique and meaningful **name** that is used in the declaration and appeal
Role that indicates what exactly the subroutine does

Modularity

Types

✓ Depending on the number and type of outputs, there are two (02) types of Subprograms (modules):

1. **Function** : When the module returns a single (1) result and this result is elementary (basic) type data, example:
   - Function that calculates the sum of an array of integers (return ***an integer***)
   - Function that checks if a number is prime (return ***a boolean***)
2. **Procedure** : When the module returns 0 to n results or the result is of structured type, examples:
   - Procedure that displays a matrix (return ***0 result***)
   - Procedure solves a 2nd degree equation (return ***2 results***)
   - Procedure that reverses the content of an array(return ***an array***)

Modularity

Qualities

✓ In order to decide whether a sequence of instructions deserves to be designed in the form of a subroutine or module, the following qualities must be checked:

1. **Reuse**: a module is designed so that it can be reused in several solutions. It must be generalized as much as possible.
2. **Independence**: avoid using global variables in a module so that it is independent of the main algorithm. Same thing for reads and writes.
3. **Simplicity**: keep your code readable and design a module that meets a specific task.

Modularity

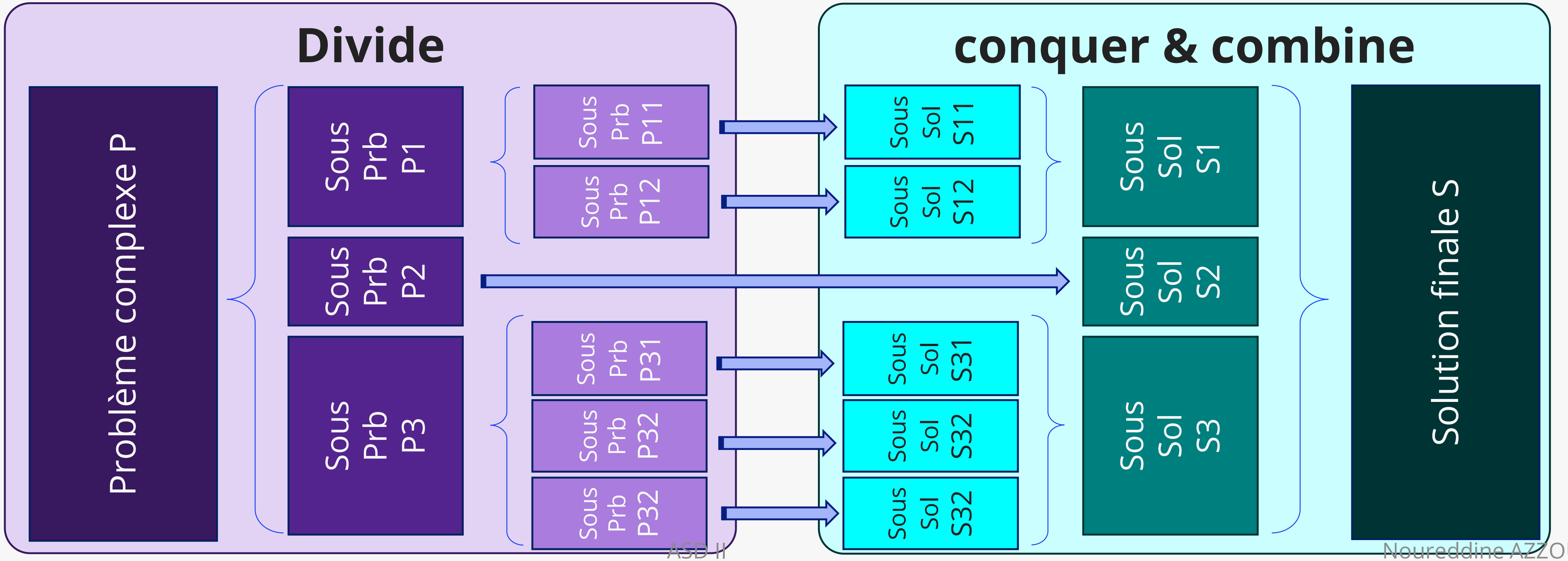
Definitions

- ✓ **Modularity**: "It is a way of thinking aimed at building algorithms starting from a very general level and gradually detailing each treatment, until arriving at the lowest level of description."
- ✓ **Modularity**: is a **Top-down Analysis** which **divides** (cuts) a problem into sub-problems to **conquer** them then **combine** the sub-solutions and obtain an overall result.
- ✓ **Modularity**: the basis of **structured programming** consists of solving a problem by building simple, readable and reusable **modules**.

Modularity

Objectifs / Goals

- ✓ Cut (divide) a complex problem into simple sub-problems which will be solved separately.
- ✓ Propose a solution to a (sub)problem once and only once



Modular Approach Stages

1st STEP: Understanding the problem

2nd STEP: Analysis and Design
- Modular Cutting/Split
- Construction of Modules

3rd STEP: Realization

Modular Breaking/Splitting

✓ Break the problem into coherent modules:

- ❑ Start extracting obvious modules that are easy to detect.
- ❑ Improve and enrich the breakdown as you progress in solving the problem

Build Modules

✓ To build a module, we start with its description:

- ❑ Draw the Module
- ❑ Give it a name
- ❑ Define your interface
- ❑ Specify its nature (function or procedure)
- ❑ Indicate its role

✓ If the module already exists, we do not build it. Its description is given only

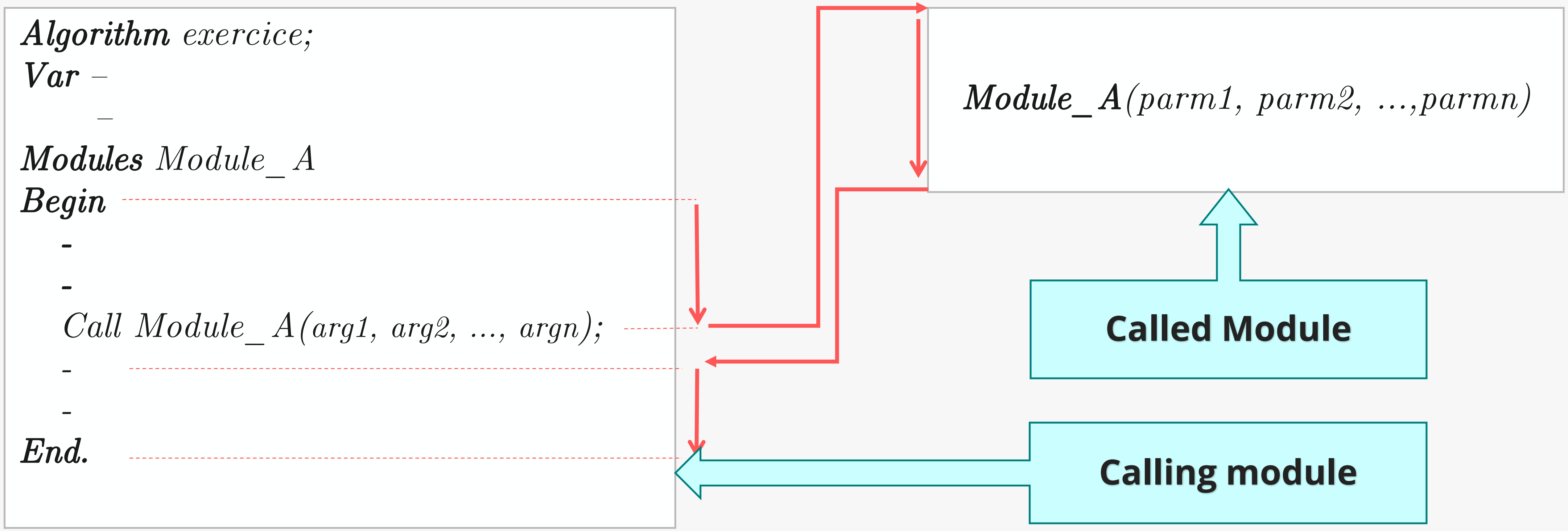
Modularity

Avantages / Benefits

- ✓ Cutting into coherent modules is done using a Top-Down Approach
- ✓ Simplifies design
- ✓ Independent and separate construction of modules
- ✓ Readability and ease of understanding of algorithms
- ✓ Ease of maintenance and code updating
- ✓ Reuse of modules (Subprograms) already designed

Communication

Communication between Modules



- ✓ When a **call** to a module is encountered,
  - ❑ Suspend the execution of the ***calling module*** (For example: Main algorithm)
  - ❑ Start and run the ***called module*** (For example: Module_A)
  - ❑ Resume execution of the calling module just after the calling instruction

Parameters and Arguments

- ✓ The variables used during the construction of the module (subroutine header) are called ***Parameters*** or ***formal parameters***
  - ❑ **Example** : les paramètres (ou paramètre formels) du module « **Module_A** » sont : $parm1,\ parm2,\ \ldots,parmn$
- ✓ The variables used when calling (using) a module are called ***Effective parameters*** or ***arguments***. They replace the formal parameters during the call.
  - ❑ **Example** : les arguments (ou paramètre effectifs) du module « **Module_A** » utilisés dans son appel à l'algorithme principale sont : $arg1,\ arg2,\ \ldots,\ argn$
- ✓ The ***number*** of arguments must match the number of parameters.
- ✓ The ***order*** of arguments must match the order of parameters.
- ✓ The ***type*** of the k_{th} argument must match the type of the k_{th} parameter.
- ✓ The correspondence between arguments and parameters is done using the ***order***.

Parameters Passing Modes

- ✓ It is the ***substitution*** of ***formal*** parameters by an ***effective*** parameters when calling a subprogram.
- ✓ On distingue deux mode de passage :
  - ❑ ***Passing by Value***: Any modification of the content of the parameter in the called program has no effect on the value of the effective parameter in the calling program.
  - ❑ ***Passing by Variable (by Reference)***: any modification of the content of the formal parameter automatically results in the modification of the effective parameter.
- ✓ formal parameters passed by variable are preceded by the keyword ***VAR*** in the header of the

$$type\_module\ nom\_fonction\ (Formal\ input\ parameters;\ \boldsymbol{VAR}\ Formal\ output\ parameters);$$

Passing by Value

- ✓ Copy the argument values to the start of the subprogram.
- ✓ This is in fact an assignment of the values of the arguments in the associated formal parameters.
- ✓ Can receive any expression (constant, variable, expression, function call, etc.)
- ✓ Peut recevoir n'importe qu'elle expression (constant, variable, expression, appel de fonct ...)
  - ❑ **Example** subroutine (function) which calculate the area of a rectangle.

$surf \leftarrow Surface\ (x,\ y);$ } **Call**

equivalent to

Module

```
Function Surface(long, larg:real):real;
Var      S: integer;
Begin
    S ← long * larg;
    Surface ← S;
End;
```

```
Function Surface(long, larg:real): real;
Var      S: integer;
Begin
    long ← x; larg ← y;
    S ← long * larg;
    Surface ← S;
End;
```

Passing by Variable

- ✓ In passing by variable (or reference) the parameter itself becomes the argument,
- ✓ that is, the parameter becomes an alias of the argument.
- ✓ Can only be linked to variables..
  - ❑ **Example** : subroutine (procedure) which swaps (exchanges) the values of two variables..

Echange (a, b); — **Call**

équivalent à

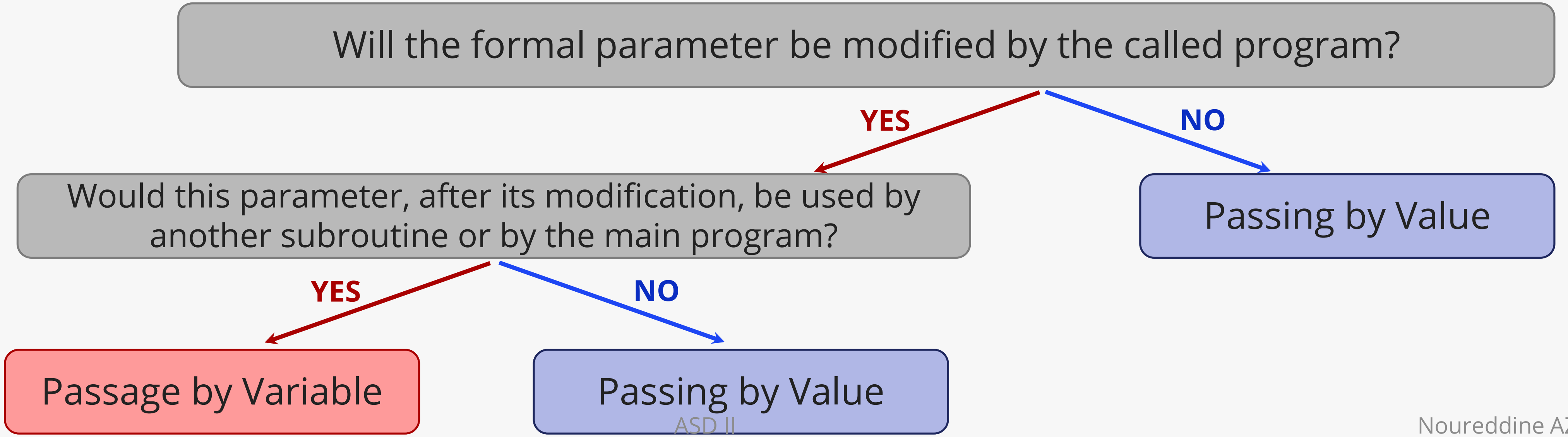
Module

```
Procedure Echange(VAR x, y:integer);
Var       z: integer;
Begin
    z ← x;
    x ← y;
    y ← z;
End;
```

```
Procedure Echange(VAR x, y:  integer);
Var       z: integer;
Begin
    z ← a;
    a ← b;
    b ← z;
End;
```

Params Passing Modes

- ***Passing by Value***: is adopted when we want the module to return the **same value** that the parameter had at the input, or the parameter **is not used** in other modules (useless to find the final result).
- ***Passage by Variable (by Reference)***: is adopted when the input parameter **is modified** during the execution of a subroutine and it is the **modified content** of the parameter that we want.



Notes

- It is recommended to use:

- A **passing by values** for the **input** parameters of a **function**.

- A **passing per variable** for all the **output** parameters of a **procedure**.

Local Variables and Global Variables

There are two categories of variables:

- ***Local Variables***: which are defined in a module and which can only be manipulated in this module.
- ***Global Variables***: which are defined in a calling module and can be manipulated in this module and in all modules called by this module.

- **The scope**: of a variable is the set of modules where this variable is accessible (or defined).

Local Variables and Global Variables

```
Module_Appelant
Var       A, B, X: real;
    Module_1
    Var       X: integer;
              T: boolean;
        Module_2
        Var       C: integer;
    Module_3
    Var       X, Y, Z: integer;
Begin
    -
    -
End;
```

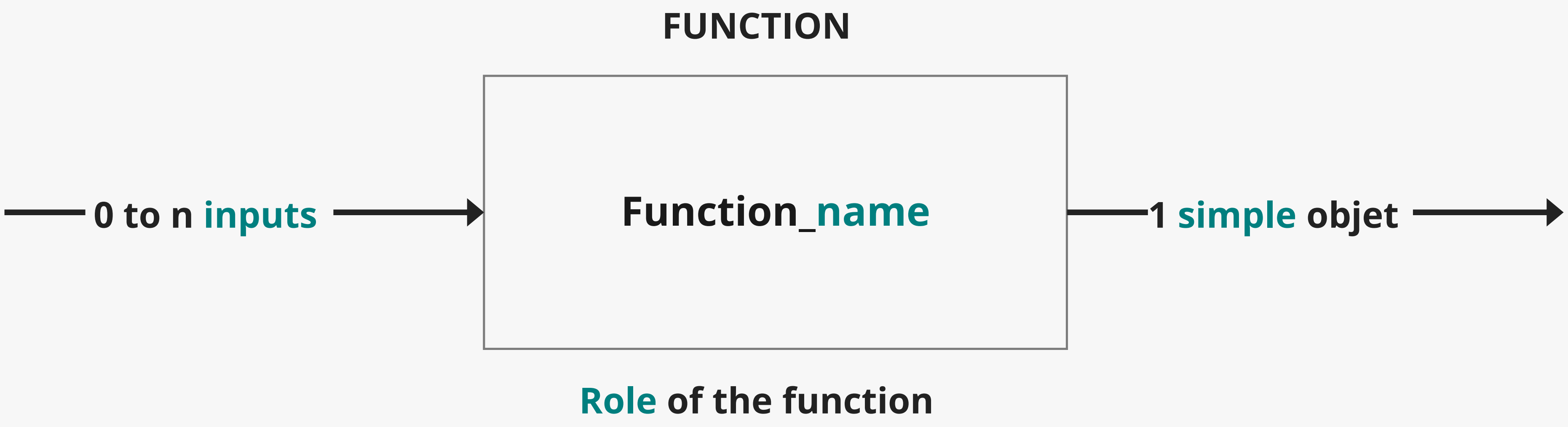
Variable	Scope (Portée)
A,B	*Module_Appelant , Module_1, Module_2, Module_3*
T	*Module_1, Module_2*
C	*Module_2*
X:déclaré au Module_Appelant	*Module_Appelant*
X:déclaré au Module_1	*Module_1, Module_2*
X:déclaré au Module_3	*Module_3*
Y, Z	*Module_3*

Functions


$password;
$database;
$charset;
$link = null;
public
self::$link
throw new MysqlException(

Definition and Description

- ✓ A function is a sub-programme (subroutine or module) which returns a single result (single output) of simple (elementary) type: integer, real, boolean, character. It can receive 0 to n input parameters.



Description of a fuonction

Structure and Syntax

Header

Function fonction_ name (List of formal input parameters :Type): Return Type;

Body

Type –
Const –
Var –

Déclaration des données (objets) locales

Begin
-
-
-
-
-
- *fonction_ name ← Result;*
End;

Traitements

Properties & Notes

- ✓ The body of a function can contain all **declarations** (Type, Const, Var, etc.) and algorithmic **structures** (Assignment, Repetition, Conditional, etc.).
- ✓ The calculated result (return value) must be **passed in the function name**. This assignment is located – in most cases – at the end of the function.
- ✓ **Formal parameters** describe the input parameters used in the function as well as their **type** and their passing **mode**.
- ✓ In Functions, formal parameters are used in **passing-by-value** mode.

Calls

- ✓ Calling a function can be used as:
  - ❑ ***Expression*** in an **assignment,**
  - ❑ ***Operand*** in a **condition**
  - ❑ ***Argument*** in a **procedure** or **function call**
- ✓ **Examples:**
  - ❑ X ← Prime (a)
  - ❑ if Prime (a) = True then write (a, ‘is prime’)
  - ❑ Res ← Prime (Fact(n))

1st Step : Split Modules

Function : Fact

Fact (n) = n!

= n*(n-1)*….*3*2*1

FUNCTION

n: integer → **Fact** → integer

Role : Return the factoriel of **n** (n!)

Function : Comb

Comb (n,p) $= C_n^p = \frac{n!}{p! * (n-p)!}$

= Fact(n)/Fact(p)*Fact(n-p)

FUNCTION

n: integer →
p: integer → **Comb** → integer

Role : return the num of combinaison of **p** objets from **n**

Example: Calcul of Combinaisons

2nd Step: Construction of modules

```
Function Fact (n: integer): integer;
Var       F, i: integer;
Begin
    F ← 1;
    for i ← 1 to n Do
            F ← F * i;

     Fact ← F;
End;
```

```
Function Comb (n , p: integer): integer;
Functions :   Fact;
Begin
    Comb ← Fact(n)/ Fact(p)* Fact(n-p);
End;
```

Functions

3rd Step: Main Algorithm

```
Algorithm Calcul_comb;
Var          x,y,c: integer;
Functions : Comb;
Begin
    Read(x,y);
    c ← Comb(x,y);
    Write ('Le nombre de combinaison = ', c);
End;
```

Function declaration

PASCAL

FUNCTION nom_fonction (Input parameters): type_retour;
var $\{$ *Déclaration des données locales* $\}$
begin
 - $\{$ *Instructions* $\}$
 -
 -
 nom_fonction ← valeur_retour;
fin;

```pascal
function Fact(n: integer) : integer;
 var F,i : integer;
 begin
   F := 1;
   for i := 2 to n do
       F := F*i;

   Fact := F;
 end;
```

C

type_retour nom_fonction (Input parameters)
{ $\{$ *Déclaration des données locales* $\}$
 - $\{$ *Instructions* $\}$
 -
 -
 -
 return valeur_retour;
}

```c
int Fact (int n)
{
    int F , i;
    F = 1;
    for (i=2; i<=n; i++){
        F = F*i;
    }

    return F;
}
```

Function declaration

PASCAL

- ✓ In PASCAL, Functions and procedures must be declared **before** the main program.
- ✓ In general, each called module must be constructed **before** the calling module for it to be **recognized**.
- ✓ In this example:
  - ❑ A ***Call*** to a Fact function (line **17**)
  - ❑ **n**: ***parameter*** (***formal*** parameter) (line **5**)
  - ❑ **x**: ***argument*** (***effective*** parameter)(line **17**)

```pascal
1  program factoriel;
2   uses Crt;
3   var x : integer;
4
5   function Fact(n: integer) : integer;
6    var F,i : integer;
7    begin
8      F := 1;
9      for i := 2 to n do
10         F := F*i;
11
12     Fact := F;
13   end;
14
15 begin
16   Readln(x);
17   Writeln(x,'! =', Fact(x));
18 end.
```

Function declaration

Programmation C / PASCAL

C

- ✓ In C language, Functions and procedures can be declared **before** and **after** the *main* function.
- ✓ If the function is placed **before** the *main* , the compiler checks the parameters and executes the function.
- ✓ If the function is placed **after** the *main* , we need to define a **prototype** of the function for it to be recognized.

```c
#include <stdio.h>

int Fact (int n)
{
    int F , i;
    F = 1;
    for (i=2; i<=n; i++){
        F = F*i;
    }

    return F;
}

int main()
{
    int x;
    scanf("%d", &x);
    printf("%d! = %d", x, Fact(x));

    return 0;
}
```

Function declaration

C

- ✓ **A prototype** is a function declaration so that it can be used (called) even before it is coded.
- ✓ The prototype is placed at the **beginning of the program** (just after the libraries declaration).
- ✓ A prototype is declared as a function

$$type\_retour\ nom\_fonction\ (Input\ parameters)$$

```c
#include <stdio.h>

int Fact (int n);

int main()
{
    int x;
    scanf("%d", &x);
    printf("%d! = %d", x, Fact(x));

    return 0;
}

int Fact (int n)
{
    int F , i;
    F = 1;
    for (i=2; i<=n; i++){
        F = F*i;
    }

    return F;
}
```

Procedures


string sInput;
int iLength, iN;
double dblTemp;
bool again = true;
while (again) {
iN = -1;
again = false;
getline(cin, sInput);
system("cls");
if (iLength < 4) {
again = true;
continue;
} else if (sInput[iLength - 3] !=
again = true;
continue;
} while (++iN < iLength) {
if (isdigit(sInput[iN])) {
continue;
} else if (iN == (iLength -

Procedures

Definition and Description

✓ A procedure is a sous-programme (subroutine or module) which returns 0 to n results (multiple output) of simple or compound type. It can receive 0 to n input parameters.

PROCEDURE

0 to n inputs → **Procedure_name** → **0 to n outputs** →

Role of the procédure

Description of a procedure

Structure & Syntax

Header

Procedure procedure_ name (List of formels input and output parameters :Type);

Body

Type –
Const –
Var –

Déclaration des données (objets) locales

Begin

-
-
-
-
-

Traitements

End;

Properties & Notes

- ✓ The body of a procedure can contain all **declarations** (Type, Const, Var, etc.) and algorithmic **structures** (Assignment, Repetition, Conditional, etc.).
- ✓ The **formal parameters** describe the **input** and **output** parameters used in the procedure as well as their **type** and their **passing mode**.
- ✓ In Procedures, formal **output** parameters must always be described in a **passing-by-variable** mode.

Procedures

Calls

- ✓ Calling a procedure is a **primitive action**. It is composed of the name of the procedure followed in parentheses by the list of effec tive input and output parameters separated by commas.
- ✓ As for functions, the **number**, **order**, and **type** of the effective paramete rs must be identical to those of the formal parameters.
- ✓ **Examples:**
  - ❑ remplir_tab (n, T)
  - ❑ echange (x, y)

Example: Convert a time T (in seconds) to hours, minutes and seconds

1st Step : Split Modules

Procedure: convert_temps

convert_temps (T, h, m, s)

1. On divise T sur 360 : le quotient est h
2. Le reste de cette division est divisé sur 60
   a. Le quotient est m
   b. Le reste est s

PROCEDURE

Role : Converts a time T (in seconds) into
h hours, m minutes and s seconds

Example: Convert a time T (in seconds) to hours, minutes and seconds

2nd Step: Construction of modules

```
Procedure convert_temps (T: integer; VAR h, m, s : integer);
Var      R: integer;
Begin
    h ← T DIV 3600;
    R ← T MOD 3600;
    m ← R DIV 60;
    s ← R MOD 60;
End;
```

Functions

3rd Step: Main Algorithm

```
Algorithm Convert;
Var          A, x, y, z: integer;
Procedures : convert_temps;
Begin
    Read(A);
    convert_temps(A, x, y, z);
    Write (A,'=',x,'heures et ',y,' minutes et ',z,'secondes');
End;
```

Exemple: Convertir un temps T (en secondes) en heures, minutes et secondes

Programmation C / PASCAL

PASCAL

PROCEDURE *nom_procedure* *(Input/output parameters);*

var {*Déclaration des données locales*}

begin

-
-
-
-

{*Instructions*}

fin;

```
procedure convert_temps(T: integer; VAR h,m,s: integer);
 var R : integer;
 begin
   h := T DIV 3600;
   R := T MOD 3600;
   m := R DIV 60;
   s := R MOD 60;
 end;
```

C

void *nom_fonction* *(Input/output parameters)*

{ {*Déclaration des données locales*}

-
-
-
-
-

{*Instructions*}

}

```
void convert_temps (int T, int *h, int *m, int *s)
{
    int R;

    *h = T / 3600;
    R = T % 3600;
    *m = R / 60;
    *s = R % 60;
}
```

Declaration of a Procedure

PASCAL

✓ Dans cet exemple:

- ❑ Un ***Appel*** d'une procédure convert_temps (ligne **5**)
- ❑ **T** : ***paramètre*** (paramètre ***formel*** d'entrée) (ligne **5**)
- ❑ **h,m,s** : ***paramètre*** (paramètre ***formel*** de sortie) (ligne **5**)
- ❑ **A** : ***argument*** (paramètre ***effectif***) d'entrée (ligne **17**)
- ❑ **x,y,z** : ***argument*** (paramètre ***effectif***) de sortie (ligne **17**)

```
1  program Convert;
2   uses Crt;
3   var A,x,y,z : integer;
4
5   procedure convert_temps(T: integer; VAR h,m,s: integer);
6    var R : integer;
7    begin
8      h := T DIV 3600;
9      R := T MOD 3600;
10     m := R DIV 60;
11     s := R MOD 60;
12   end;
13
14 begin
15   Readln(A);
16   convert_temps(A, x, y, z);
17   Writeln(A,'=',x,'heures et ',y,' minutes et ',z,'secondes');
18 end.
```

Declaration of a Procedure

C

- ✓ In C language, the return type of procedures is specified as ***void***.
- ✓ When declaring the procedure, formal output parameters are preceded by *****.
- ✓ When calling this procedure, the effective output parameters are preceded by **&**.

```c
#include <stdio.h>

void convert_temps (int T, int *h, int *m, int *s);

int main()
{
    int A, x, y, z;
    scanf("%d", &A);
    convert_temps(A, &x, &y, &z);
    printf("%d = %d heures et %d minutes et %d secondes", A, x, y, z);

    return 0;
}

void convert_temps (int T, int *h, int *m, int *s)
{
    int R;

    *h = T / 3600;
    R = T % 3600;
    *m = R / 60;
    *s = R % 60;
}
```