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

Chapter : 5

Subprograms:
Functions and Procedures

MI-L1-UEF121 : Algorithms and Data Structures I

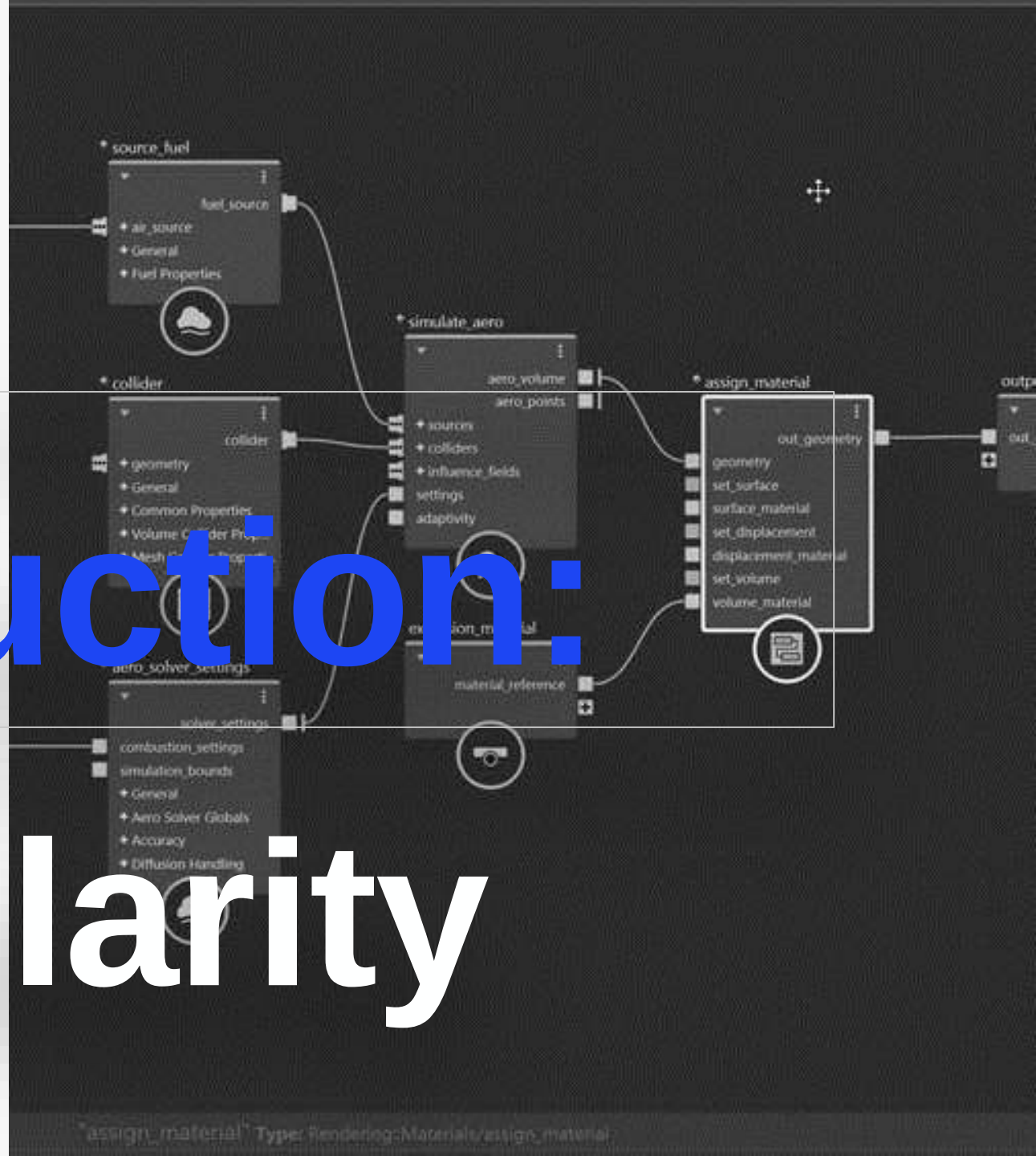
Ali Khalfi

Khalfiali.udbkm@gmail.com

Course Topics

1. Introduction
2. Passing parameters
3. Local variables and global variables
4. Functions
5. Procedures
6. Recursion

Introduction: 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 ← 1 to n Do
      fact_n ← fact_n * i;
```

```
fact_p = 1;
For i ← 1 to p Do
      fact_p ← fact_p * i;
```

```
fact_np = 1;
For i ← 1 to n-p Do
      fact_np ← fact_np * i;
```

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

Modules / Subprograms

Definition

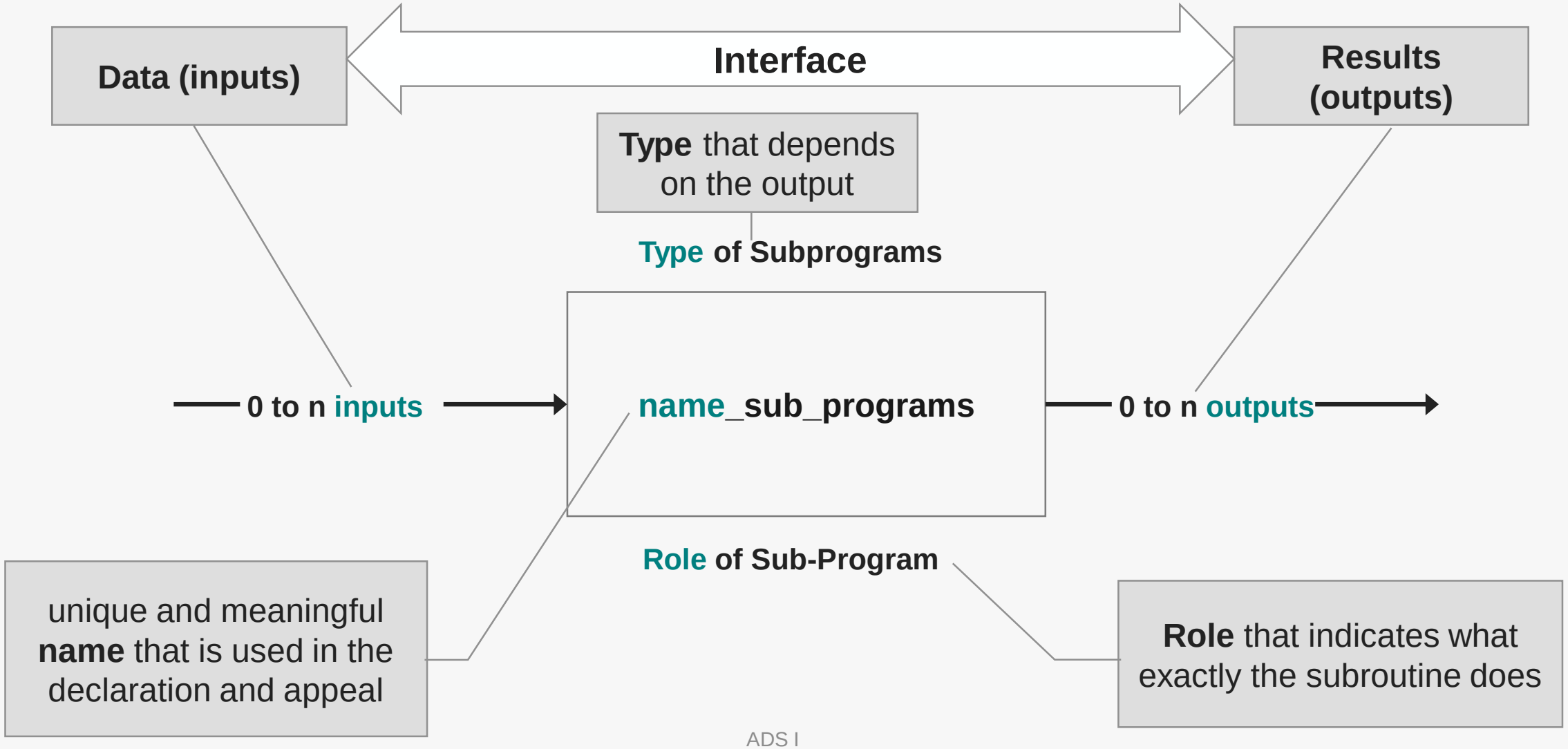
- ✓ A **subprogram** or module is a set of instructions with a well-defined interface that performs a specific task.

- ✓ 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

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***)

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:

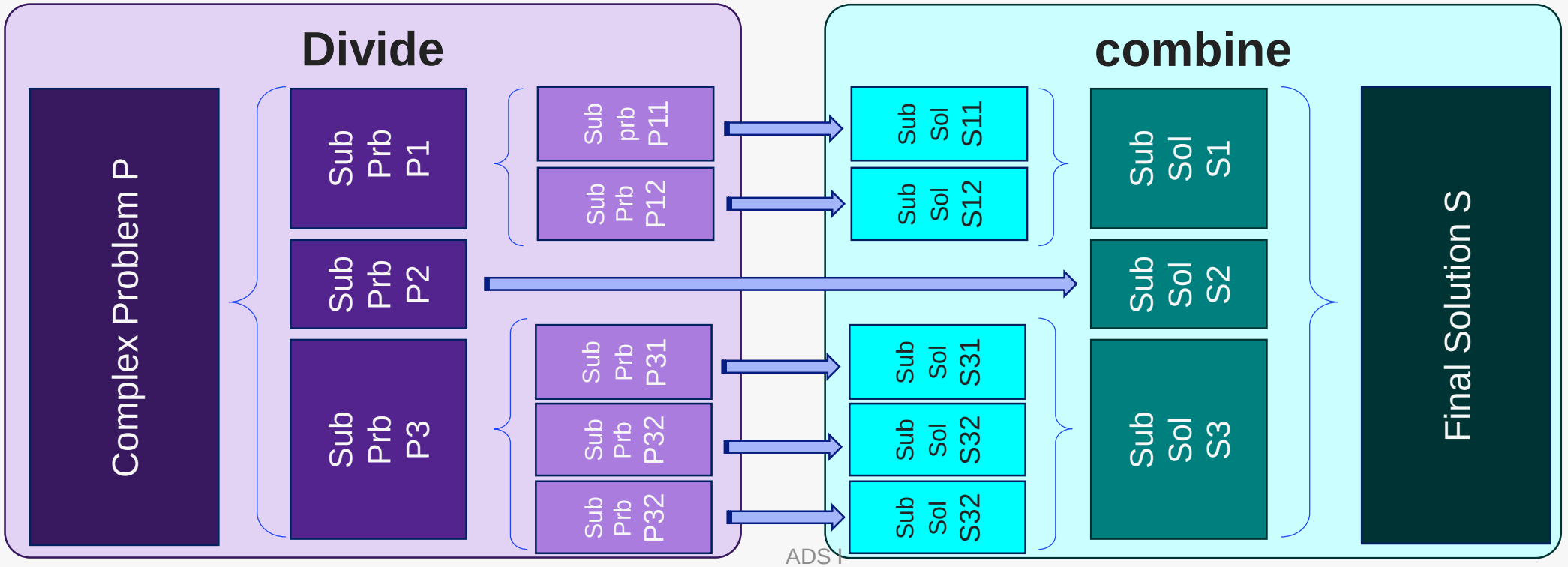
1. **Reuse**: a module is designed so that it can be reused in several solutions. It must be generalized as much as possible.
2. **Independence**: avoid using global variables in a module so that it is independent of the main algorithm. Same thing for reads and writes.
3. **Simplicity**: keep your code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

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**.

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

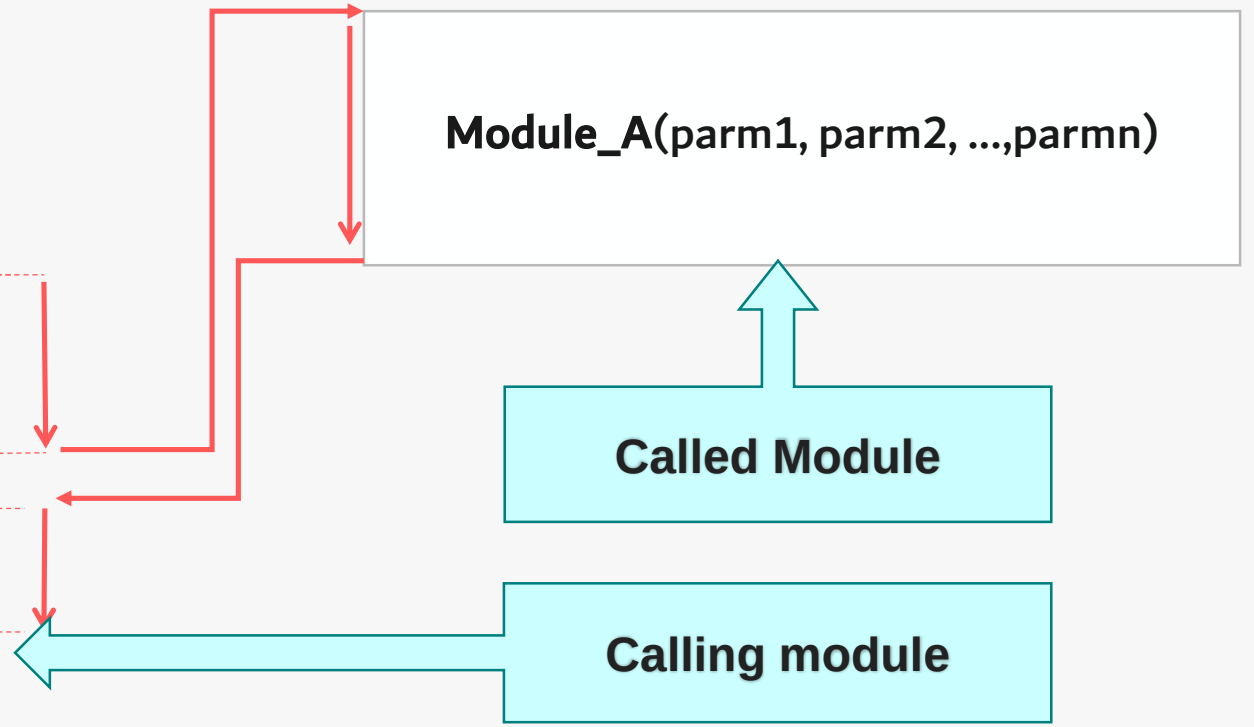
- ✓ To build a module, we start with its description:
  - ❑ Draw the Module
  - ❑ Give it a name
  - ❑ Define your interface
  - ❑ Specify its nature (function or procedure)
  - ❑ Indicate its role
- ✓ If the module already exists, we do not build it. Its description is given only

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 between Modules

```
Algorithm exercice;
Var –
    –
Modules Module_A
Begin
    -
    -
    Call Module_A(arg1, arg2, ..., argn);
    -
    -
End.
```



- ✓ 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** : The parameters (or formal parameters) of the module "Module_A" are: **parm1, parm2, …, parmn**
- ✓ The variables used when calling (using) a module are called ***Effective parameters*** or ***arguments***. They replace the formal parameters during the call.
  - ❑ **Example** : the arguments (or effective parameters) of the module "Module_A" used in its call to the main algorithm are: **arg1, arg2, …, 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.
- ✓ There are two modes of 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 module :

```
module_type module_name (Formal input parameters; 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.)
  - ❑ **Example** subroutine (function) which calculate the area of a rectangle.

```
surf ← 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

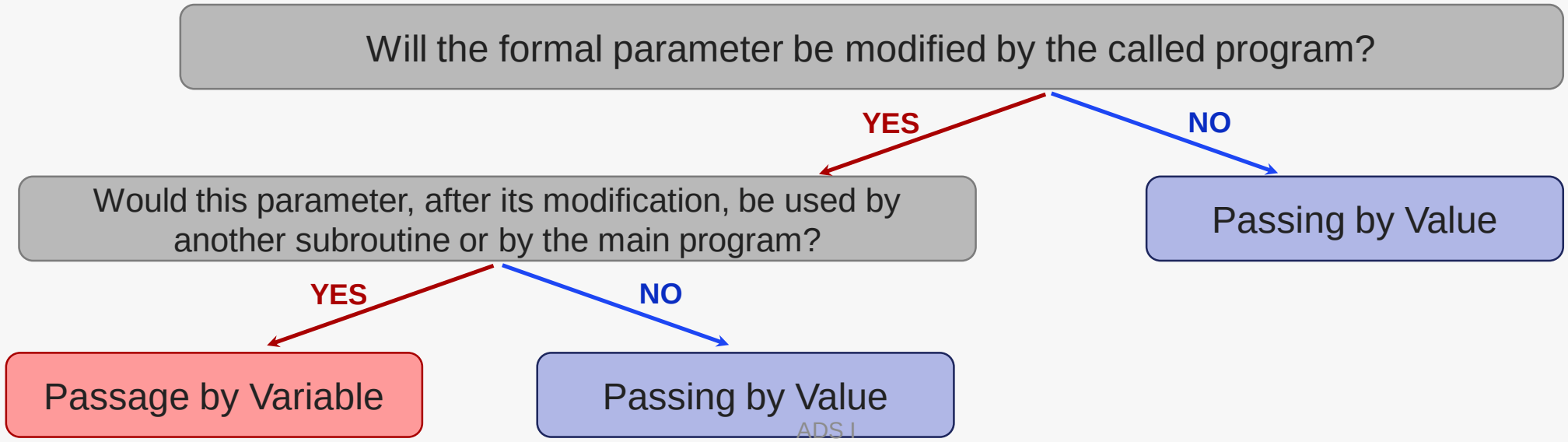
Module

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

```
Procedure Swap(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

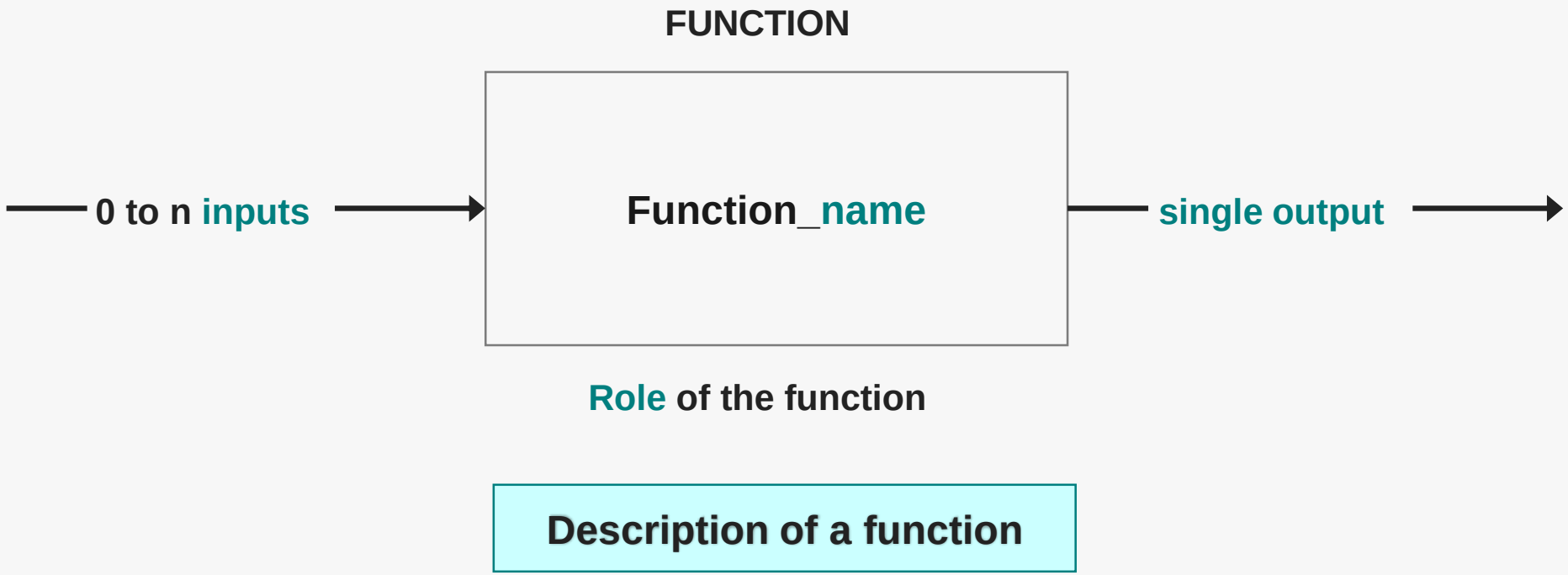

```
Module_Caller
Var        A, B, X: real;
    Module_1
    Var        X: integer;
               T: boolean;
        Module_2
        Var        C: integer;
    Module_3
    Var        X, Y, Z: integer;
Begin
    -
    -
End;
```

Variable	Scope (Portée)
A,B	Module_Caller, Module_1, Module_2, Module_3
T	Module_1, Module_2
C	Module_2
X: declared to theModule_Caller	Module_Caller
X: declared to the Module_1	Module_1, Module_2
X: declared to the Module_3	Module_3
Y, Z	Module_3

Functions

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.



Structure and Syntax

Header

Body

Function function_name (List of formal input parameters :Type)**:** Return Type;

Type –
Const –
Var –

$\left\{ \begin{array}{c} Locale\ Data\ Declaration \\ (objects) \end{array} \right\}$

Begin

-
-
-
-
-

$\left\{ Traitements \right\}$

- function_name ← Result;

End;

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.

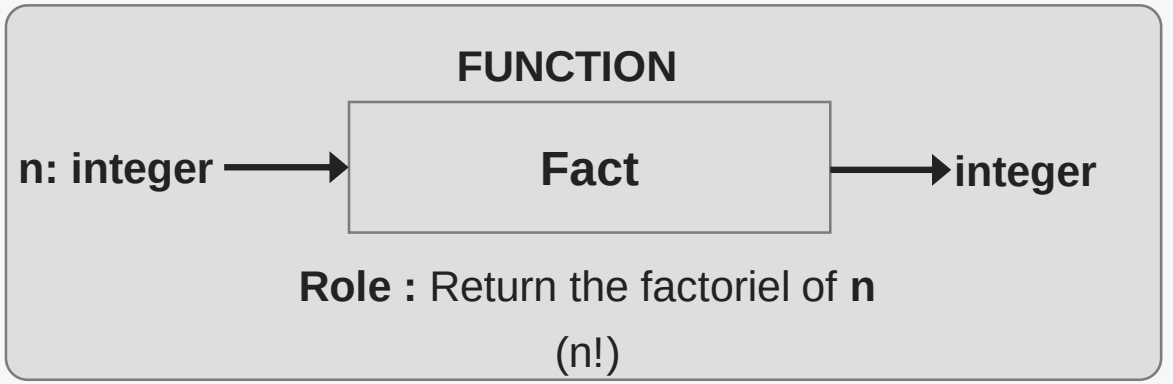
Calls

- ✓ 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

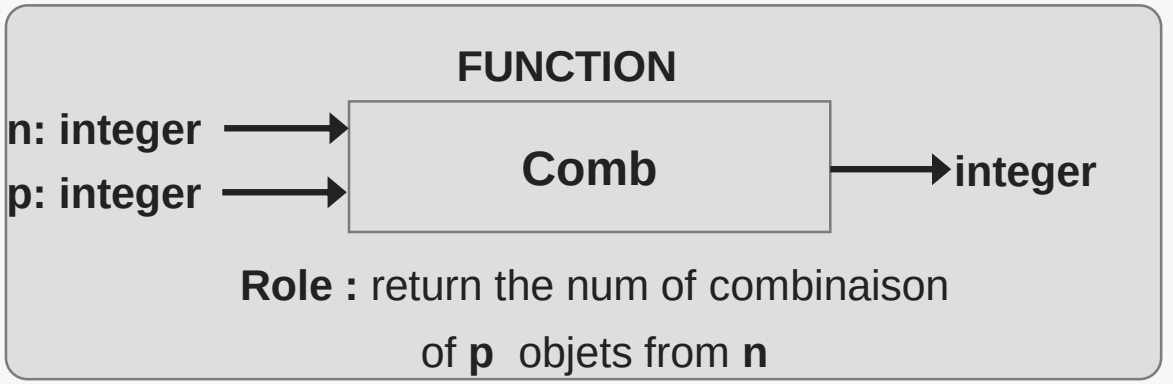
Fact (n) = n!
= n*(n-1)*....*3*2*1



Function : Comb

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

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



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;
```

3rd Step: Main Algorithm

```
Algorithm Calcul_comb;
Var             x,y,c: integer;
Functions : Comb;
Begin
    Read(x,y);
    c ← Comb(x,y);
    Write ('The number of combinations= ', c);
End;
```

Function declaration

Algorithm

```
FUNCTION name_function (Input parameters): retrun_type;
var          { Local data declaration }
begin
   -
   -         { Instructions }
   -
    name _function ← valeur_retour;
fin;
```

```
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

```
return _type nam_function (Input parameters)
{            { Local data declaration }
   -
   -         { Instructions; }
   -
   -
    return retrun_value;
}
```

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

    return F;
}
```

Algorithm

- ✓ In Algorithm, Functions and procedures must be declared **before** the main algorithm.
- ✓ 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
  - ❑ **n**: ***parameter*** (***formal*** parameter)
  - ❑ **x**: ***argument*** (***effective*** parameter)

```
Algorithm factorial;
Var              x: integer;
Function fact(n:integer) : integer;
Var f,i: integer;
Begin
    f ← 1;
    for i<-2 to n do
          f ← f*i;
    Endfor;
Fact ← f;
End;

begin
read (x);
Write (x,'! =', fact(x));
End.
```

Function declaration

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.

Program C

```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

retrun _type name_fonction (Input parameters)

Program C

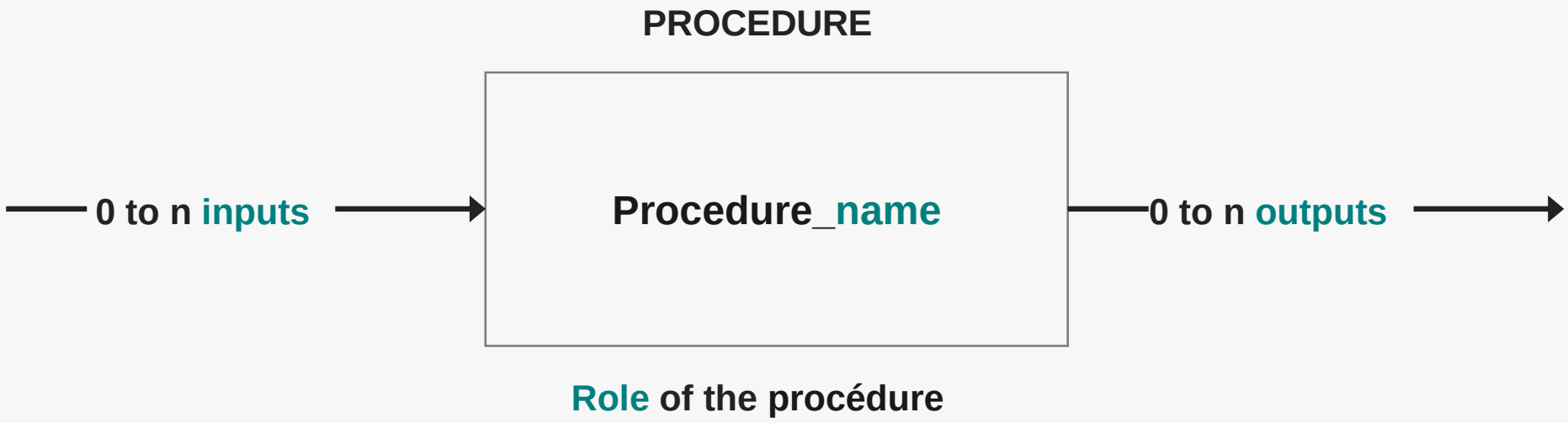
```c
#include <stdio.h>

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;

Definition and Description

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



Structure & Syntax

Header

Body

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

Type –
Const –
Var –

$\left\{ \textit{Local declaration data } (\textit{objects}) \right\}$

Begin
-
-
-
-
-
End;

$\left\{ \textit{Traitements} \right\}$

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.

Calls

- ✓ Calling a procedure is a **primitive action**. It is composed of the name of the procedure followed in parentheses by the list of effective inp ut and output parameters separated by commas.
- ✓ As for functions, the **number**, **order**, and **type** of the effective parameters must be identical to those of the formal parameters.
- ✓ **Example:**
  - ❑ swap (x, y)

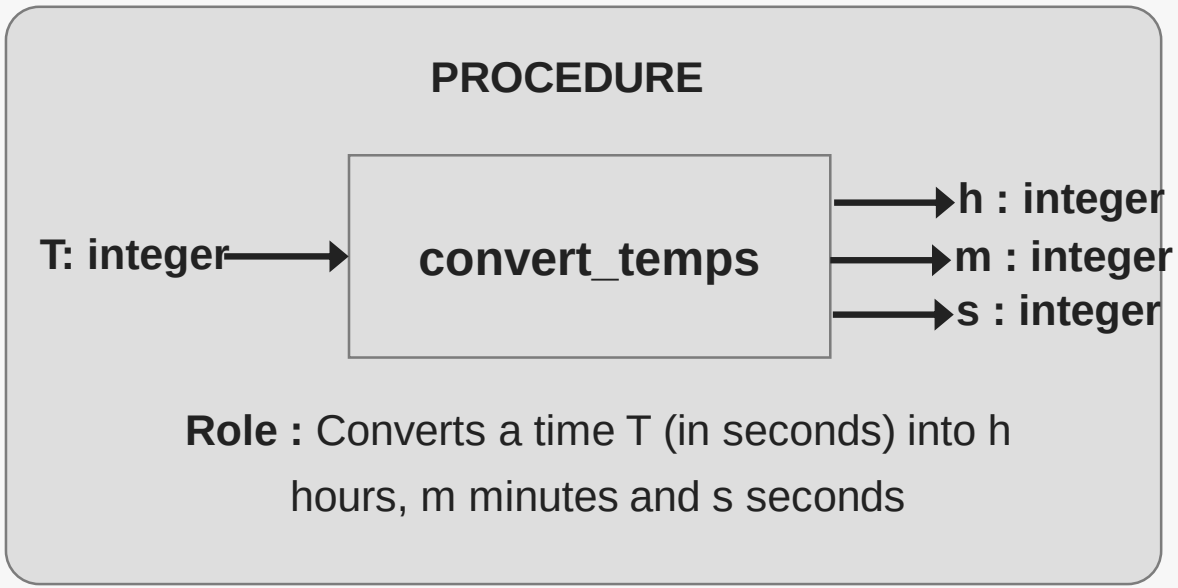
1st Step : Split Modules

Procedure

Procedure: convert_temps

convert_temps (T, h, m, s)

1. we divide T by 360 : the quotient is h
2. the remainder of this division is divided by 60
   a. the quotient is m
   b. the remainder is s



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

2nd Step: Construction of modules

```
Procedure convert_time (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;
```

Procedure

3rd Step: Main Algorithm

```
Algorithm Convert;
Var             A, x, y, z: integer;
Procedures : convert_time;
Begin
    Read(A);
    convert_time(A, x, y, z);
    Write (A,'=',x,'hours ',y,' minutes ',z,'seconds');
End;
```

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

Algorithm

```
PROCEDURE procedure_name (Input/output parameters);
var        { Local data declaration }
begin
   -
   -       { Instructions }
   -
   -
fin;
```

C

```
void procedure_name (Input/output parameters)
{          { Local data declaration }
   -
   -       { Instructions }
   -
   -
   -
}
```

```
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;
```

```
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

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 **&**.

Program C

```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;
}
```

Recursion

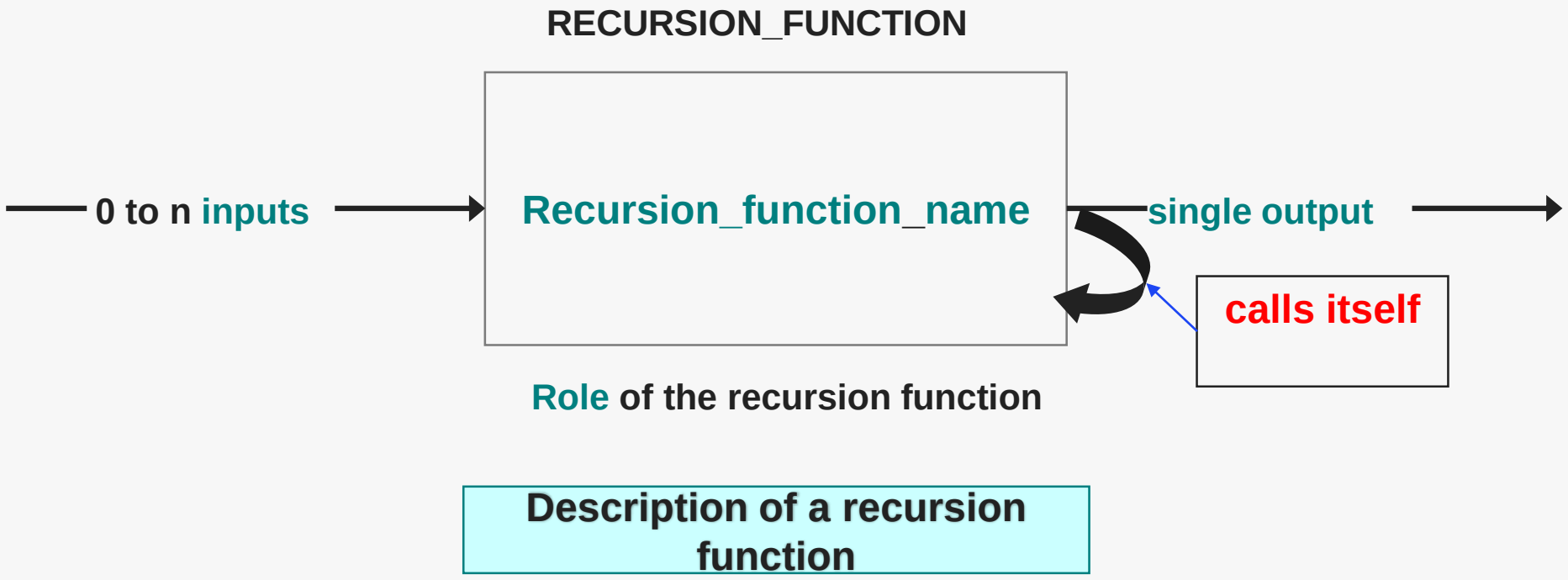

string sInput;
int iLength, iN;
double dblTemp;
bool again = true;
while (again) {
iN = -1;
again = false;
getline(cin, sInput);
system("cls");
istringstream(sInput) >> dblTemp;
iLength = sInput.length();
if (iLength < 4) {
again = true;
continue;
} else if (sInput[iLength - 3] !=
again = true;
continue;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

Key Analogy

- ✓ Imagine you're standing in a line of people, and you need to know how many people are in front of you. You can't see the front. So, you ask the person in front of you, "How many people are in front of you?" That person does the same, and so on, until the question reaches the first person in line, who sees no one and answers "0". This answer is then passed back down the line, with each person adding 1, until it gets back to you

Definition and Description

- ✓ recursion is a programming technique where a function **calls itself** directly or indirectly in order to solve a problem. The core idea is to break down a complex problem into smaller, more manageable sub-problems of the same type.



Structure and Syntax

Recursive function

Header

Body

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

Type –
Const –
Var –

$$\left\{ \begin{array}{c} Locale\ Data\ Declaration \\ (objects) \end{array} \right\}$$

Begin

-
-
-
-
-
-
-

$$\left\{ \begin{array}{c} Traitements \\ \text{recursion\_function\_name <– recursion\_function\_name (List of effectives parameters);} \end{array} \right\}$$

calls itself

End;

The Two Essential Parts of a Recursive Function

For a recursive function to work correctly and not run forever, it must have two fundamental components:

- ✓ **Base Case:** The condition under which the function stops calling itself. This is the simplest, smallest instance of the problem, which can be solved directly without recursion. Without a base case, the function will call itself indefinitely, leading to a stack overflow error.
- ✓ **Recursive Case**: The part of the function where it calls itself with a modified, smaller version of the original problem. The goal is to eventually reduce the problem to the base case.

Classic Example: Calculating Factorial

The factorial of a non-negative integer n (denoted as n!) is the product of all positive integers less than or equal to n.

5! = 5 * 4 * 3 * 2 * 1 = 120

By definition, 0! = 1 (this is our base case).

We can observe that: n! = n * (n-1)!

This is a recursive definition!

Classic Example: Calculating Factorial

- ✓ Recursive function must be declared **before** the main algorithm.
- ✓ a recursive function **calls itself**
- ✓ **Calling** of the recursive function in the main algorithm

```
Algorithm factorial;
Var                x: integer;
Function fact(n:integer) : integer;
Begin
        if (n=0) then fact ← 1;
                 else fact ← n*fact(n-1);
        Endif;
End;

// Main algorithm
Begin
read (x);
Write (x,'! =', fact(x));
End.
```

Classic Example: Calculating Factorial

✓ **How it works for factorial(5):**

1. factorial(5) returns 5 * factorial(4)
2. factorial(4) returns 4 * factorial(3)
3. factorial(3) returns 3 * factorial(2)
4. factorial(2) returns 2 * factorial(1)
5. factorial(1) returns 1 * factorial(0)
6. factorial(0) hits the base case and returns 1.

✓ **Now, the returns "unwind" back up:**

1. 7. 1 * factorial(0) = 1 * 1 = 1
2. 8. 2 * factorial(1) = 2 * 1 = 2
3. 9. 3 * factorial(2) = 3 * 2 = 6
4. 10. 4 * factorial(3) = 4 * 6 = 24
5. 11. 5 * factorial(4) = 5 * 24 = 120

Classic Example: Calculating Factorial (C Program)

```
#include <stdio.h>
// Recursive function to calculate factorial
long factorial(int n) {
    // 1. Base Case
    if (n == 0) {       return 1;    }
    // 2. Recursive Case
    else {       return n * factorial(n - 1);    }}
int main() {
    int number = 5;
    long result = factorial(number);
    printf("The factorial of %d is %ld\n", number, result);
// Output: 120
    return 0;   }
```

Another Classic Example: The Fibonacci Sequence

✓ The Fibonacci sequence is defined as:

F(0) = 0 (Base Case 1)

F(1) = 1 (Base Case 2)

F(n) = F(n-1) + F(n-2) for n > 1 (Recursive Case)

```
Algorithm Fib;
Var     x: integer;
Function Fib(n:integer) : integer;
Begin
        if (n=0) then Fib ← 0;
                else if n=1 then Fib ← 1
                        else  Fib ← Fib(n-1)+Fib(n-2);
                    Endif;
         Endif;

// Main algorithm
Begin
X ← 6;
Write ('The Fibonacci number is' , x , Fib(x)); //ouput :8
End.
```

Another Classic Example: The Fibonacci Sequence

```
#include <stdio.h>
// Recursive function to find the nth Fibonacci number
int Fib(int n) {
    // Base Cases
    if (n == 0) {    return 0;    }
                else if (n == 1) {    return 1;    }
    // Recursive Case
                        else {   return Fib(n - 1) + Fib(n - 2); }
                    }
int main() {
    int n = 6;
    printf("The %dth Fibonacci number is %d\n", n, Fib(n)); // Output: 8
    return 0;
}
```

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

Chapter : 5

Subprograms:
Functions and Procedures

MI-L1-UEF121 : Algorithms and Data Structures I

Ali Khalfi

Khalfiali.udbkm@gmail.com