Chapter 1 : Subroutines: Procedures and Functions

1. Introduction

Solving a computer problem is broken down into 4 Phases: Analysis, algorithm writing, programming, compilation and execution. So, An algorithm is a solution to a class of problems. Sometimes the problem to be solved is too sophisticated (complicated), i.e.: It becomes difficult to have a global vision to solve this problem.
The program (algorithm) becomes large (in a single block) , and generally very difficult to understand, find errors, develop and read . In this case, it is advisable to break down the problem into sub-problems, then find a solution to each.

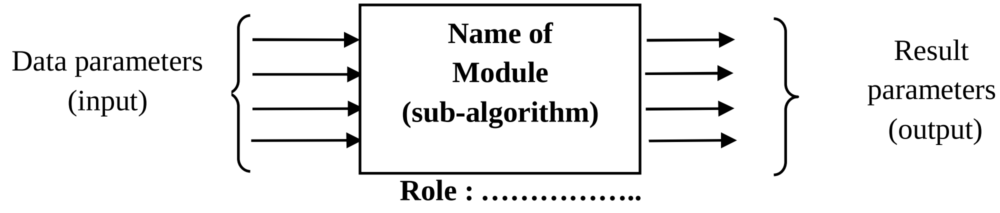
Example : **(problem and sub-problems)**

We want to create a program allowing us to read the exam notes, tutorials and practical work of students in the introductory algorithmic module, calculate their averages and say for each student whether they are admitted or postponed in this module?

- ➢ This problem can be broken down into 3 sub-problems:
  - ✓ ***Sub-problem 1*** : reading student notes
  - ✓ ***Sub-problem 2*** : calculate student averages
  - ✓ ***Sub-problem 3*** : say if student is admitted or deferred

2. Subprograms *(modules)* :

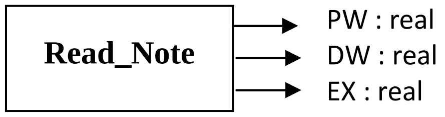
- ➢ A subroutine (sub-algorithm) is a program (algorithm) which describes the solution to a sub-problem.
- ➢ Schematically, a module (subprogram) is represented by a black box which has inputs, outputs and a very specific role as follows:



Example 1:

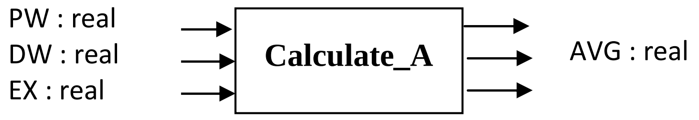
We want to write an algorithm allowing us to read the grades (exam, **D**irected and **p**ractical **w**ork) of students in the Algorithmics and Data Structures 1 module, calculate their averages and say for each student whether they are **admitted** or **adjourned** for this module.
Do the necessary modular division?

Module 1:



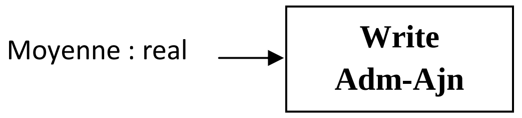
Role : read a student's notes

Module 2:



Role : Calculate the average from the grades

Module 3:

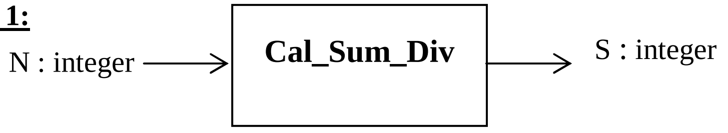


Role : Say if the student is admitted or adjourned

Example 2:

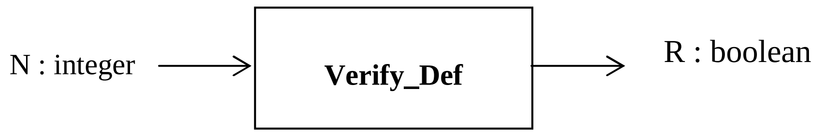
deficient number is a natural integer number **n** which is strictly greater than the sum of its strict divisors. We want to write an algorithm that reads an integer **X** and displays all deficient numbers **less than** X.
Do the necessary modular division?

Module 1:



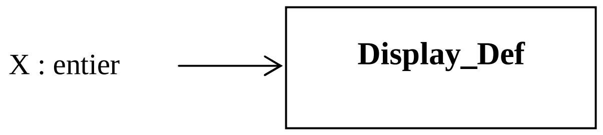
Role: calculate the sum of the strict divisor of N

Module 2:



Role : verify if an integer N is deficient or not

Module 3:



Role : Read an integer X and display all deficient number less than X

Noticed :

- When developing a modular layout we are not looking for the answer to the question ***how to do it?*** But sooner ***What to do*** ? ;that means identification of the precise role of each module.
- There are two types of subprograms (modules): **procedures** and **functions.**

2.1. Procedures :

- A procedure is a subprogram (sub-algorithm), which can be called (used) in another program (algorithm) or in different places of the same program (algorithm).
- A procedure can be called as an instruction in a program (algorithm) through its **name**.

a) Declaration of a procedure:

- A procedure is defined in the **declarative part of the algorithm**.
- A procedure consists **of a head**, **variable declarations** (if they exist), and **a body** .

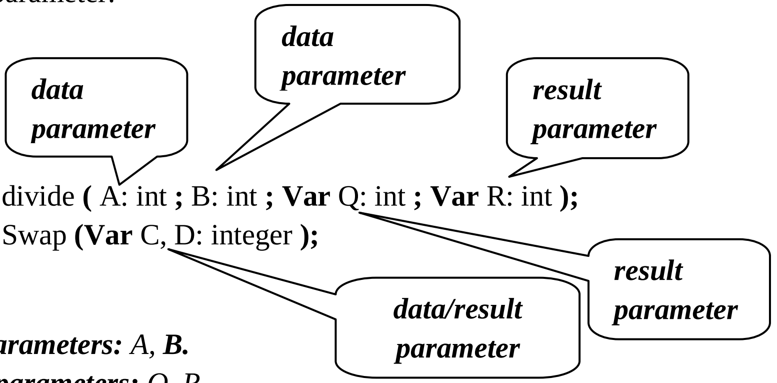
Syntax:

```
Procedure <procedure name> (List of parameters)
<Declaration part>
Begin
<Body of the procedure>
End ;
```

The parameters of a procedure : The parameters are ***variables*** . Each parameter is described by:

- A name,
- A type,
- Transmission mode: data parameter, result parameter, or data/result parameter.

Example :



Procedure divide **(** A: int **;** B: int **; Var** Q: int **; Var** R: int **);**
Procedure Swap **(Var** C, D: integer **);**

The data parameters: *A,* ***B.***
The result parameters: *Q, R.*
The data/result parameters: *C, D.*

For the previous example:
Procedure calculate_A (DW, TW, EX: real; **Var** Avg: real);
Procedure verify_def(N: integer; **Var** R: boolean);

Noticed :

Var keyword indicates that the parameters are outputs (results), and which can also be inputs.

Example :

Solution 1: without using a procedure	**Solution 2: Using a procedure**

Algorithm Addition
A, B, som: real;
Begin
Read (A, B);
som ←A+B ;
 	
Write (som);
END.
	Algorithm Addition
	A, B, sum: real;
	procedure add (X, Y: real; **Var** S: real)
	Begin
	S ←X+Y ;
	END;
	Begin // main program
	Read (A, B);
	add (A, B, sum); Call for the procedure
	Write (sum);
	END.

b) Calling a procedure:

- ➢ A procedure is called by its name:
  - ✓ The parameters indicated in the declaration of the subroutines are called **"Formal Parameters".**
  - ✓ The parameters specified in the subroutine call are called
  - ✓ **"Actual parameters"**.
- ➢ During the call, the order of the effective parameters must conform to that of the formal parameters.

  Example : previous example

  X, Y: **formal parameters.**
  A, B: **effective parameters**.

2.2. Functions:

- ➢ A function is a special case of procedures, unlike the procedure; the function must have a ***type***, because it must return a ***value*** as output.

a) Declaration of a function:

```
Function <function name> (List of input parameters): Type ;
        <Declaration part>
Begin
        <Function body>
        Return (output value);
End ;
```

b) Function call:

Calling a function is done in the same way as a procedure except that the function name directly contains the return value (output value).

Examples: function Square (X: real) **:** real; returns the square of the number

For the previous examples:
function AVG (DW, TW, EX: real): real;
Function verify_def (N: integer): boolean;

Write an algorithm that adds two real numbers?

Solution 1: classic solution	**Solution 2: Using the function**

Algorithm Addition
A, B, sum: real;
Begin
Read (A, B);
sum ←A+B ;
 	
Write (sum);
END.
	Algorithm Addition
	A, B, sum: real;
	function add (X, Y: real): real;
	S: real;
	Begin
	S ←X+Y ;
	Return (S);
	End;
	Begin
	Read (A, B);
	sum ← add (A,B);
	Write (sum);
	END.

Noticed :

- The head of a function always ends with **the type** of the value returned by the function.
- The body of a function always ends with the ***Return*** (output value) instruction which returns the output value to the ***calling program***.

3. Global variables and local variables:

- A ***global variable*** is a variable declared in the main program (algorithm) and can be used by one or more procedures or functions.
- A ***local variable*** is a variable declared and used in a subprogram (procedure or function).

Example :

```
util-Proc algorithm
A, B: integer;            Global variables
Procedure Swap (Var x, y: real);
Z: real;                  Local variables
Begin
Z ←x ;
x ←y ;
y ←Z ;
End;
Begin
Read (A, B);
Swap (A, B);
Write (A, B);
END.
```

Global variables: A, B.
Local variables : Z.

4. Passing the parameters:

- It should be remembered that a ***variable*** in a program is a memory space intended to store a ***value***.
- A variable has a ***name*** (identifier) and a memory ***address***.

Variable1 (address1)	Value1
Variable2 (address2)	Value2
. . .	. . .
Variablen (addressn)	Value n

- The identifiers represent the parameters in the ***declaration*** of the subroutines are not the same at ***the call***.
- There are two types of parameter passing (transmissions):
  a) Passage by value
  b) Passing by variable (or by addresses)

4.1. The passage by value:

- The values of the effective parameters are copied into the formal subroutine parameters without affecting the original values.
- In this case, the local variables in the called subroutine are used.
- Modifying local variables in the subroutine does not modify the variables passed as parameters, because these modifications only apply to a copy of these variables.

Example :

```
Algorithm P_Value
A, B: real;
Procedure Swap (X, Y: real);
Z: real;
Begin
Z ←X
X ←Y ;      } Permutation des valeurs
Y ←Z ;
End;
Begin
A ←3;
B ←5;
```

```
Swap (A, B);
Write (A, B);
END.
```

A = 3
B = 5

Parameters are passed by value:

- ➢ The values of the effective parameters 'A' and 'B' are copied into the formal parameters 'X' and 'Y' when calling: A → X, B → Y
- ➢ Modifications (permutation) are made only on local variables (X, Y, Z).
- ➢ Finally, the values of the formal parameters 'X' and 'Y' are restored in the effective parameters 'A' and 'B'.

4.2. The passage by address:

- ➢ This technique consists of not just passing the values of the actual parameters, but of passing the variables themselves (its location in memory).
- ➢ There is therefore no more copying, any modification of the formal parameters in the “called subroutine” results in the modification of the effective parameters (variables passed as parameters).

<u>Example :</u>

```
Algorithm P_Variable
A, B: integer;
Procedure Swap ( Var x, y: real);
Z: real;
Begin
  Z ←x ;
  x ←y ;
  y ←Z ;
End;
Begin //main program
A ←3;
B ←5;
Swap (A, B);
Write (A, B);
END.
```

- ➢ The addresses of the effective parameters 'A' and 'B' are passed to the procedure during the call.
- ➢ Modifications (permutation) are made to variables A, B.

5. Differences between procedures and functions

Procedure	Function
<u>The head:</u> Procedure <Proc_Name> (list of parameters);	**<u>The head:</u>** Function <Function_Name> (list of parameters) : <Type>;
<u>Usage example</u> : (call) Proc_name (parameter list);	**<u>Usage example</u> : (call)** ***Assignment*** **:** x←f_name (parameter list); ***Write*** **:** Write (f_name (parameter list));
<u>Passing parameters:</u> - Passing by value - Passage by address	**<u>Passing parameters:</u>** - Passing by value

6. Benefits of using procedures and functions

- ➢ Here are some advantages of modular programming:
  - ✓ ***Minimizing code duplication***
  - ✓ ***Better readability***
  - ✓ ***Reduced risk of errors***
  - ✓ ***Possibility of selective tests:*** (module by module)
  - ✓ ***Reuse of existing modules:*** It is easy to use modules that you have written yourself or that have been developed by other people.
  - ✓ ***Ease of maintenance:*** A module can be changed or replaced without having to touch the other modules of the program.
  - ✓ ***Promoting teamwork:*** A program can be developed as a team by dividing and assigning modules to different people or groups of people.

***Example* 1:** Write an algorithm that read three non-zero positive numbers A, B, C then calculates and displays the following sum: (A! + B!+ C!)!

Solution1	**Solution2**
Algorithm example1	**Algorithm** example1
A, B, factA, factB, factC, Sum, fact :	A, B, C: integers;
integers;	function Factorial (N: integer;): integer;
Begin	factN, i: int;
Read (A, B);	**Begin**
factA ← 1;	factN ← 1;
For i **=** 1 **to** A **Do**	**For** i **=** 1 **to N do**
factA ← factA * i;	factN ← factN* i;
End for;	**End for;**
factB ← 1;	Return (fact);
For i **=** 1 **to** B **do**	**End**;
factB ← factB * i;	**Begin**
End for;	Read (A, B);
factC ← 1;	Write (factorial (factorial (A) +factorial
For i **=** 1 **to** C **do**	(B) factorial (C)));
factC← factC * i;	**END.**
End for;	
Sum ← factA + factB + factC;	
fact ← 1 ;	
For i **= 1 to** sum **do**	
fact ← fact * i;	
End for	
Write (fact);	
END.	

7. Nested calls

7.1definition

The nested call consists of call a function in another function.

Example: let be the following functions:

$$\begin{cases} F(x) = 3\,x^{2} \\ G(x) = 7\,x \\ H(x) = F(x) + G(x) \end{cases}$$

Q) Write an algorithm that reads a real number **a** and displays: F(a), G(a) and H(a).

Solution:

Algorithm example	**Function** H(x3: integer;):entire;
A: integers;	S: entire;
Function F(x1: integer;):integer;	**Begin**
S: integer;	S←F(x3)+G(x3);
Begin	Return (S) ;
S←3*x1*x1;	**END**;
Return (S);	**Begin** // main program
END;	Read (A);
Function G(x2: integer;):integer ;	write (F (A),G(A),H(A));
S: integer;	**END.**
Begin	
S←7*x2;	
Return (S);	
END;	

Nested call