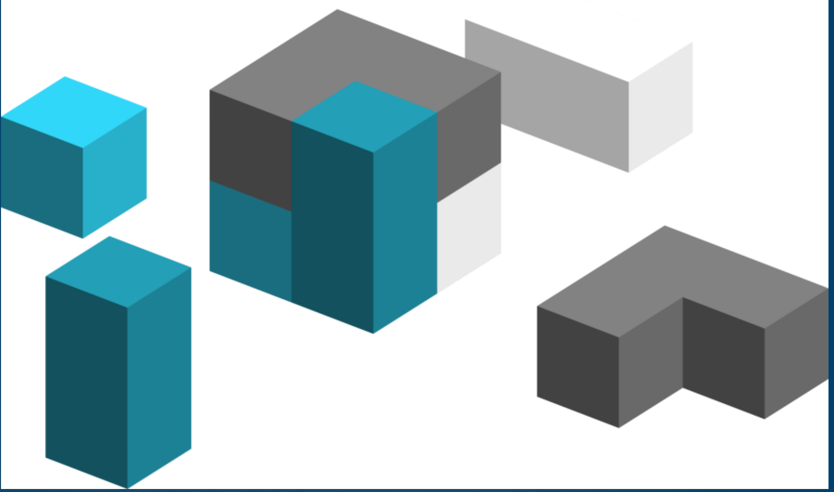
ensia The National School of Artificial Intelligence
المدرسة الوطنية العليا للذكاء الاصطناعي



Data Structure & Algorithms 1

CHAPTER 3: MODULAR PROGRAMMING

Sep – Dec 2023

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.

Fundamental Concepts of Modularity

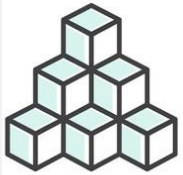
Modularity, the cornerstone of structured programming, is simply the act of breaking down a problem into a set of reusable modules. It serves two fundamental objectives:

- **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.

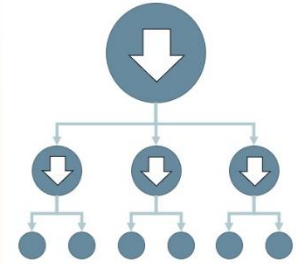
Fundamental Concepts of Modularity

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. <u>There's no need to reconstruct it</u>.

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



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.

Fundamental Concepts of Modularity

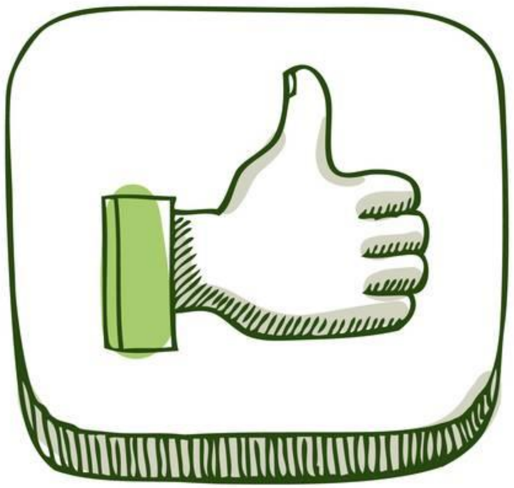
- When using a module, **we no longer need to know** how it is constructed **but** only what it does.

- 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.

Fundamental Concepts of Modularity

Advantages

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



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
```

Fundamental Concepts of Modularity (calling a module)

```
Algorithm Algo_1
Variables…
   Module A
BEGIN

   -

   -

   Call Module A

   -

   -

END
```

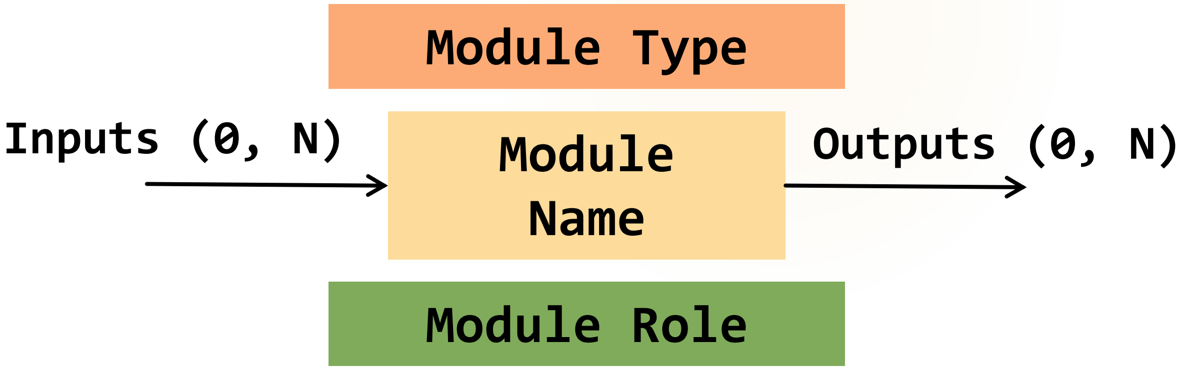
```
Module A
Variables…
BEGIN
   -
   -
   -

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.

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.



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
```

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

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.

Modular Approach and Formalism

Module Quality

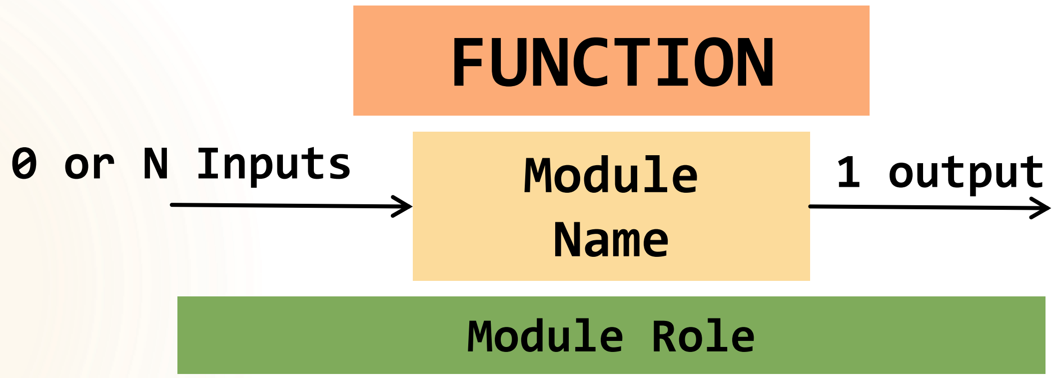
- **Reusability**: Always aim to make your modules as general as possible for later reuse.

- **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.

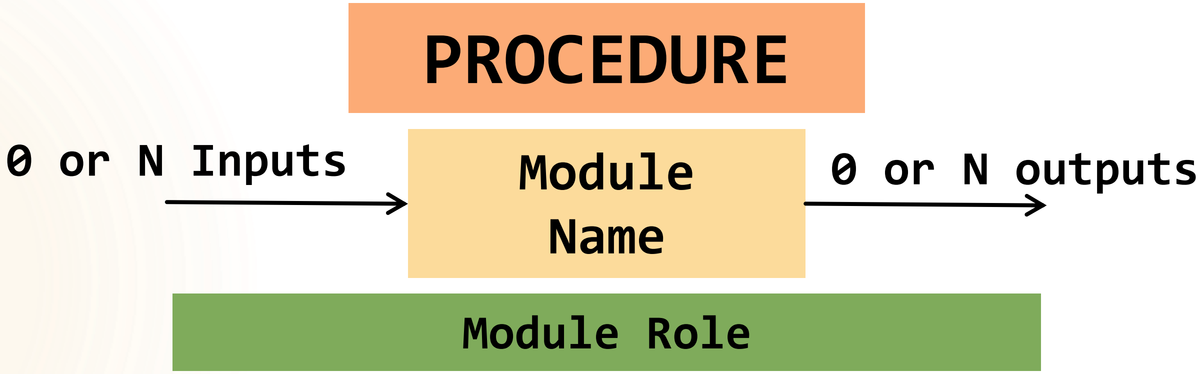
Modular Approach and Formalism

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.



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.



Modular Approach and Formalism

EXAMPLE 1:

PROCEDURE

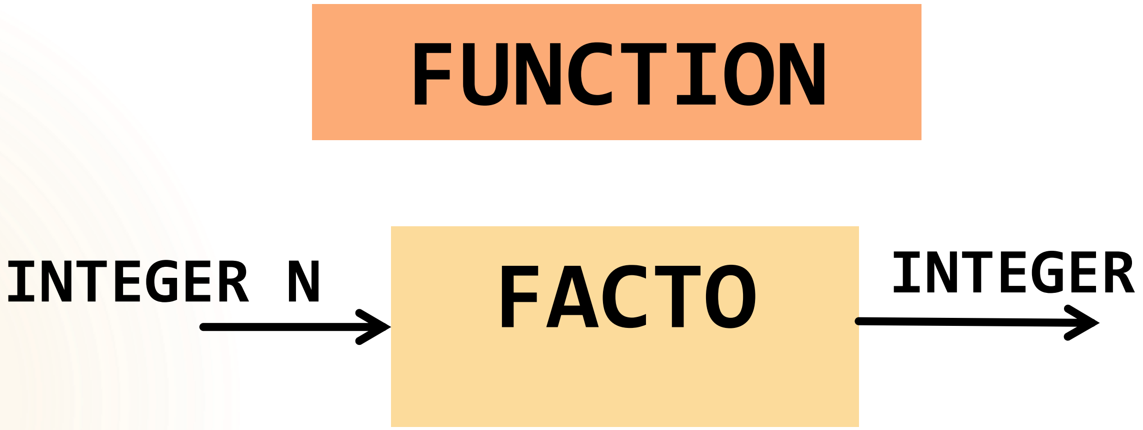
REAL A, B, C → **EQU2D** → REAL X1, X2

→ INTEGER N

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

Modular Approach and Formalism

EXAMPLE 2:



Role: Calculte the factorial of an integer N

Modular Approach and Formalism

FUNCTION STRUCTURE

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

Environment → **Declarations**

Function Body →

BEGIN

-
-
-

Function_name = Result of the function

END

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)."

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 NbDigits 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 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
```

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

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

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)

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

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
```

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
```