

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

Université de M'sila
Faculté des Mathématiques et de l'Informatique
Département d'informatique



جامعة المسيلة
كلية الرياضيات والإعلام الآلي
قسم الإعلام الآلي

Chapter 1:

Sub -programs: Functions and Procedures

Algorithms and data structure 2

Presented byr : Dr. Benazi Makhoulouf
Academic year : 2023/2024

Chapter 01 Content:

1. Introduction
2. Definitions
3. Local and global variables
4. Parameter Passing
5. Recursivity

Introduction

A program is a set of sequential instructions to solve a specific problem. In order to find the solution method (algorithm), the problem must be divided into different sub-problems whose solution is less complicated. Partial problems can be solved using subroutines.

Subprograms:

- A subroutine is a set of independent instructions that have a name and are called for execution. The caller is either the main program or another subroutine.
- When the program, during its execution, reaches the instruction that calls the subroutine, the execution context becomes the contents of the subroutine. Once the subroutine has finished executing, the program returns to the instruction immediately following the invocation.
- Subroutines can be called procedures, functions, methods, routines, or macros.

Procedure

A **procedure** is a subroutine that does not return any values in its name, but it can return results via arguments.

The procedure name can be used as a standalone statement. For example:

For example:

- Displaying numbers on the screen below a certain limit
- Showing table values on the screen
- Solving a quadratic equation

algorithm	C
SomeProc	SomeProc ();
OtherProc (x)	OtherProc (x);

Function

A **function** is a subroutine that necessarily returns a result in its name because its name is considered a variable that carries a certain value. Therefore, the function call can be used as a variable in assignment operations and other expressions.

For example:

- - Calculating the square of a number
- - Calculating the area of a rectangle
- - Solving a first-order equation
- - Finding the sum of an array
- - Checking if a number is prime or not

Algorithm	VS
<code>Y ← SomeFunction (X) * 5</code>	<code>Y = SomeFunction (X) * 5;</code>

Advantages of using subroutines

- **Readability:** Utilizing subroutines organizes and simplifies the program, enhancing comprehension of the code.
- **Speed in programming:** Avoiding the repetition of the same sequence of instructions within the program improves efficiency.
- **Reduced program size:** Subroutines contribute to a more compact program.
- **Facilitates maintenance:** Subroutines make it easier to maintain and update the program.
- **Reuse:** They can be stored in libraries for convenient reuse in other programs.

Declarations

Procedure

- **Algorithm**

```
procedure proc_name ( parameter list )  
local variables  
Begin  
instructions  
End .
```

- **In C**

```
void proc_name ( parameter list ){  
local variables;  
instructions ;  
}
```


- `proc_name` : valid identifiers.
- List of parameters (optional): called “formal parameters” a set of variables by which data is transmitted and results are retrieved,
 - separated by a comma «,»
 - are between two parentheses
 - are of the form `paramname : type` in **algorithm** and `type paramname` in **C** such as: `(a:integer , b:real)` in **algorithm** and `(int a, float b)` in **C**
 - Parentheses () are required even if they contain no arguments in **C** such as: `proc()` .
- Local declarations (optional): A list of local variables of the form:
 - `var varLoc : type`
- Instructions: a set of instructions of any type, which will be executed when the subroutine is called.
- all variables declared in the “parameter list” or local declaration, which are called local variables, and variables declared in the main program, called global variables

Declarations

Function

- **Algorithm**

```
function func_name (parameter list): type
```

```
    local variables
```

```
Begin
```

```
    instructions
```

```
End .
```

- **In C**

```
type function_name ( parameter list ){
```

```
    local variables;
```

```
    instructions ;
```

```
}
```

- **Result_Type** : When the program is a function, the type of value that the function will return to the program that called it must be specified, and a value must be assigned to the function name. This is usually the last statement in the function.
 - In algorithm it is of the form **function_name ← expression** where the name of the function acts as a special variable which contains the return value by the function.
 - In C, `void` indicates that the function returns nothing, ie a procedure either leaves the Type empty or uses the word **void** depending on the version.
 - the word **return** is used to assign a value to the function name.
 - the **return** statement causes the subroutine to exit and resume the program that called it at the statement immediately following the invocation. It can return a value to the program that called the subroutine if it was a function.
 - **Format** : **return** [`<expression>`] ;

Example :

- **return** 5*x;
- **return** ;

Important notes:

1. To determine the arguments, consider what input is provided to the subroutine and what it returns as a result.
2. The list of parameters in the subroutine's definition must match in number type and order with those used in its invocation.
3. The first line of a function or procedure declaration, including function type, function name, type, order, and number of arguments (excluding their names), is called a header or prototype.
4. Arguments are not grouped if they are of the same type, as in `(x, y: integer)`. Instead, use `(x: integer, y: integer)` or `(int x, int y)`.
5. Any return type other than `void` indicates that the program is a function, not a procedure.
6. `void main()` or simply `main()` is a procedure, while `int main()` is a function and should include a “**return** statement”.
7. `scanf()` and `printf()` are two functions declared in the `stdio` library.

Ordering and placement

- In the algorithm, function declarations are placed after the declaration of variables and before the **begin** of the main program.
- In a C program, functions are declared before the `main()` function.
- The order of subroutines is crucial because each function must be defined before it can be used.
- For example, if function `f1()` calls function `f2()`, then function `f2()` must be defined before function `f1()`.

The invocation

- To **call** and execute a procedure, we use its **name** as a separate instruction and assign values and/or variables to the arguments in parentheses, referred to as effective parameters.
- **Parentheses** () can be omitted in the absence of any arguments, but in **C**, they are required.
- Similarly, when **calling** a **function**, its **name** is treated as a **variable** carrying a certain value. Therefore, the function call can be used as a variable in assignment operations and other expressions.
- It's important to note that the actual parameters must correspond in number, type, and order with the formal parameters.

Examples

```
Algorithm Test  
var z: real
```

```
procedure displaysNbrs ( n:integer )  
  var i:integer  
Begin  
  for i←1 to n do  
    Write(i)  
  endfor  
End of procedure
```

```
function sumNbrs ( x:integer , y:integer): integer  
Begin  
  sumNbrs ← x+y  
END function
```

```
Begin  
  displaysNbrs (5)  
  z←sumNbrs (5, 3)  
Write("the sum is ", z)  
END .
```

Examples

```
#include <stdio.h >
float z;
```

```
void displayNbrs (int n)
{
    int i ;
    for (i=1; i<=n; i++)
        printf("%d\ t",i );
}
```

```
int sumNbrs (int x, int y)
{
    return x+y ;
}
```

```
int main() {
    displayNbrs (5);
    z = sumNbrs (5, 3);
    printf("the sum is %d", z);
    return 0 ;
}
```


Local variables and global variables

1. Global Variables:

- - Declared outside any subprogram.
- - Usable anywhere in the program.
- - No need to pass as a parameter in subroutines.
- - Created during program loading.
- - Deleted at the end of program execution.

2. Local Variables:

- - Specific to the subroutine or block where defined.
- - Created upon function call.
- - Deleted when the function execution concludes.

• Recommendations:

- It is recommended to use local variables and parameters instead of global variables to avoid errors and ensure function independence."

Example Algorithm

```
Algorithm glob_loc  
  Var glob , b: integer
```

```
Procedure tst  
  Var b, loc : integer  
Begin  
  glob←11  
  b←22  
  loc←33  
  Write("in tst : glob =", glob , "b=", b, " loc =", loc )  
End
```

```
Begin  
  glob←1  
  b←2  
  Write("before tst : glob =", glob , "b=", b)  
  tst  
  Write("after tst : glob =", glob , "b=", b)  
end
```

Example C

```
#include <stdio.h >
int glob , b ;
```

```
tst () {
    int b, loc ;
    glob =11;
    b=22;
    loc =33;
    printf("in tst : glob =%db=%d loc =%d", glob , b, loc );
}
```

```
int main() {
    glob =1;
    b=2;
    printf("before tst : glob =%db=%d", glob , b);
    tst();
    printf("after tst : glob =%db=%d", glob , b);
    return 0 ;
}
```

The screen:

Before tst : glob =1 b=2

In tst : glob =11 b=22 loc =33

After tst : glob =11 b=2

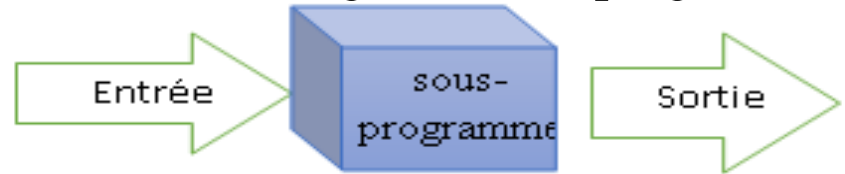
Explanation :

avant d'appeler tst	Durant l'appel de tst	après avoir appelé tst
<p>glob b</p> <p>1 2</p>	<p>glob b</p> <p>11 2</p> <div style="border: 1px solid black; padding: 5px; margin-left: 150px;"><p>tst b loc</p><p>22 33</p></div>	<p>glob b</p> <p>11 2</p>

4 Passing parameters

Parameters: are **Variables** facilitating information exchange between programs.

Two Ways to Pass Parameters:




1. Passage by Value:

- Value of the original variable is copied into the (formal) parameter.
- Copy is used as a local variable; original variable remains unmodified.
- Can use a constant value, expression, or variable.
- Used exclusively for entering information.

2. Passage by Reference, Address, or Variable:

- Not only the value is passed, but also the place of the original variable (address).
- Formal and effective variables become a single entity.
- Any change to the parameter in the subroutine affects the original variable.
- Only variables (not constants or expressions) can be passed.
- Primarily used to receive results.
- Also used for entering information, especially for large variables like tables and matrices, to avoid copying.

Passing by reference, address or variable:

- The word '**var**' is used before declaring the argument name to indicate that the pass is a pass by variable or pass by reference.
- To pass arguments by address in C, we use pointers, which will be covered in the third chapter of this course.
- In the declaration of the formal parameter, the name is preceded by '~~*~~' to indicate a pointer, and during the use within the function. However, when calling the function, the effective variable is preceded by '&'.

- In C++, the symbol '&' is used for references during the declaration only.
- **Example:**

	Algorithm	C	C++
Declaration	f(var x:integer)	int f(int *x)	int f(int &x)
Use	x ← 5	*x=5;	x=5;
Call	f(a)	f(&a);	f(a);

Examples

```
Algorithm pass_value  
var a, c: real
```

```
Procedure square (x: real, y: real)
```

```
Begin
```

```
y ← x*x
```

```
END
```

```
beginning
```

```
c ← 0
```

```
a ← 3
```

```
write ("before square c=", c)
```

```
square(a,c) // we can use square(3,c)
```

```
write (" after square c=", c)
```

```
END
```

```
the screen
```

```
before square c=0
```

```
after square c=0
```

Examples

```
Algorithm pass_variable _  
  var a, c: real
```

```
Procedure square (x: real, var y: real)
```

```
Begin
```

```
  y ← x*x
```

Van

```
END
```

```
begin
```

```
  c ← 0
```

```
  a ← 3
```

```
  write ("before square c=", c)
```

```
  square(a,c) // we can use square(3,c)
```

```
  write (" after square c=", c)
```

```
END
```

the screen

```
before square c=0
```

```
after square c=9
```


Examples

```
#include <stdio.h >
void square (float x, float y){
    y= x*x;
}
```

```
int main() {
    float a, c;
    c=0;
    a=3;
    printf ("before square c=%f ", c);
    square(a,c); // we can use square(a,5)
    printf (" after square c=%f", c);
    return 0 ;
}
```

the screen

```
before square c=0
after square c=0
```

Examples

```
#include <stdio.h >
void square (float x, float *y){
    *y= x*x;
}
```

```
int main() {
    float a, c;
    c=0;
    a=3;
    printf ("before square c=%f ", c);
    square(a, &c); // we can use square(a,5)
    printf (" after square c=%f", c);
    return 0 ;
}
```

the screen

```
before square c=0
after square c=9
```

Converting a procedure into a function:

Any procedure that returns a single result can be converted to a function,

1. change the word **Procedure** to **function**
2. transform the “**result parameter**” into a local variable
3. define the type of the function to be the type of this parameter
4. At the end of the function, assign the value of this parameter to the function name.

Example Algorithm

```
Procedure abs( x:real, var y:real)
```

```
Begin
```

```
  if x<0 then
```

```
    y← -x
```

```
  else
```

```
    y←x
```

```
  end if
```

```
END
```

```
//Call
```

```
abs(-5, z)
```

```
function abs(x: real): real
```

```
  var y: real
```

```
Begin
```

```
  if x<0 then
```

```
    y← -x
```

```
  else
```

```
    y←x
```

```
  end if
```

```
  abs←y
```

```
END
```

```
//Call
```

```
z←abs (-5)
```

Example C

```
void abs(float x, float *y) {
    if (x<0)
        *y = -x;
    else
        *y = x;
}
//call
abs(-5, &z);
```

```
float abs (float x) {
    float y;
    if (x<0)
        y = -x;
    else
        y = x;
    return y ;
}
//call
z = abs(-5);
```

The variable y can be omitted

As we can omit **else** which comes after **return** .

```
float abs (float x) {
    if (x<0)
        return -x;
    return x;
}
```

End of part 1 of Chapter 01

5 Recursion

1. A recursive program is any program that calls itself.
 2. the defined program is used to define itself.
 3. A recursive program is one that does part of the work and then calls itself to complete the rest.
- Recursion is a simple and elegant way to solve certain problems of a recurring nature.
 - **Note:** Any **for** or **while loop** can be transformed into a recursive program .

Stop Condition

- Since the recursive program calls itself, it is necessary to provide a condition to stop the recursion, which is the case where the program does not call itself
- It is best to test the stopping condition first, and then, if the condition is not met, to call the program back as the call leads to the stopping condition.

- **Example :**

```
void displays (int i)
{
    printf("% d",i );
    display (i +1);
}
```

```
void displays (int i)
{
    if (i<10) {
        printf("% d",i );
        display (i +1);
    }
}
```

The general form of the recursive program:

```
procedure Recursive
begin
if (stop condition) then
  <breakpoint instructions>
else
  <instructions>
  Recursive call(parameters
                    changed)
  <Instructions>
End if
END
```

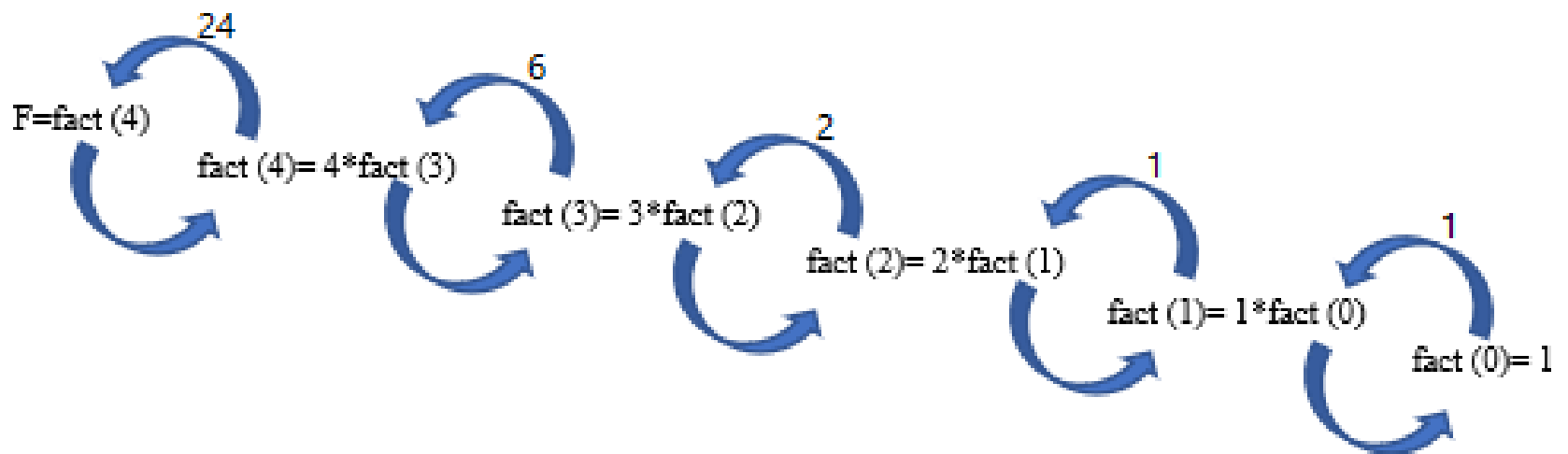
```
void recursive (parameters) {
  if (stop condition)
    <breakpoint instructions>
  else
    {
      <instructions>
      Recursive call (parameters
                    changed)
      <Instructions>
    }
}
```


Example

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fact}(n - 1) & \text{if } n > 0 \end{cases}$$

```
Function fact (n: Integer): Integer
begin
if (n = 0) then
fact←1
else
fact←n*fact (n-1)
fsi ;
END .
```

```
int fact (int not){
if (n == 0) return 1;
    return n* fact (n-1);
}
```



The execution stack:

- A memory location designated to hold parameters and local variables and where the result is stored for each running subroutine.
- Programming in recursive mode is typically easier and more readable, but it consumes a significant amount of memory. For example, when calculating $4!$, we reserve a place in the stack to store the result, another for the parameter $n=4$, and then additional places for the results of $3!$, the parameter $n=3$, and so on until $0!$ is calculated. Subsequently, the parameter $n=0$ is deleted, and then the parameters and results are removed in the reverse order of their creation.

Mutual recursion

- Mutual recursion is a situation where one program calls another program, and in turn, the second program calls back the first program.

• **Example** `#include <stdio.h >`

```
float f2 ( int n );
```

```
float f1 ( int n) {  
    if (n <= 0 ) return 0 ;  
    return 1. / n + f2(n - 2 );  
}
```

```
float f2 ( int n) {  
    if (n <= 0 ) return 0 ;  
    return - 1. / n + f1(n - 2 );  
}
```

```
void main () {  
printf("%f \n ", 4 *f1( 2 * 100 + 1 )  
* 4 );  
}
```

Mutual recursion

- **Important note:** In C language, when function f1 calls function f2, and f2 is not yet defined, the header of function f2 (the first line without its body) must be added before defining function f1. The actual definition of f2 can come later..