Chapter 1: subprograms : functions and procedures

1 Introduction

A program is a set of sequential instructions for solving 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 sub-programs.

2 Definitions

2.1 Sub-programs :

Is a set of independent instructions that have a name and are called for execution. The caller is either the main program or another subprogram. When the program, during its execution, reaches the instruction that calls the procedure, the execution context becomes the contents of the subprogram, and once it has finished executing the subprogram, it returns to executing the instruction immediately following the invocation.
The subprograms are also known as procedures, functions, methods and routines.

2.1.1 Procedure :

Procedure is a sub-program which returns no values in its name, but can return results via arguments. The procedure name can be used as a complete instruction, for example:

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

2.1.2 Function:

Function is a sub-program that necessarily returns a result in its name, as its name is considered to be a variable that carries a certain value. Consequently, the function call can be used as a variable in assignment operations and other expressions. For example:

Algorithm	C
Y← SomeFunction(X)*5	Y= SomeFunction(X)*5 ;

Note: Any procedure that returns a single result as an argument can be converted into a function.

2.1.3 Advantages of using subroutines :

- Readability: the use of subroutines organizes and simplifies the program, making it easier to understand the program code.
- Programming speed: don't repeat the same sequence of instructions several times within the program.
- Reduce program size
- Facilitates the maintenance process
- Reuse: it can be stored in libraries for reuse in other programs.

2.2 Declarations

Procedure: the declaration of a procedure takes the following form:

algorithm	C

procedure proc_name (parameter list)
    local variables
begin
    instructions
end.
	void proc_name(parameter list){
	    local variables ;
	    instructions ;
	}

Function: the declaration is similar to the declaration of a procedure, except that the type of the result value returned must be specified. It takes the following form:

Algorithm	C

`function func_name (parameter list) : type`
`   local variables`
`begin`
`   instructions`
`end.`
	`type func_name (parameter list){`
	`   local variables ;`
	`   instructions ;`
	`}`

- `proc_name`, `func_name`: valid identifiers.
- Parameter list (optional): a set of variables through which data is transmitted and results are retrieved, separated by a comma ",", and which are enclosed in parenthesis () and are of the form paramName: type, such as (a:integer, b:real) and are called "formal parameters".
  - In C, the list of arguments takes the form of type paramName (int a, float b) Parentheses () are required even if they contain no arguments.
- 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 subprogram is called. Where all variables declared in the parameter list or in the local declaration, which are called local variables, and variables declared in the main program, called global variables, can be used.
- 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 generally the function's last instruction, and is of the form func_name ← expression where the function name acts as a special variable that contains the return value by the function.
  - In the C language, you can dispense with the result type if the subprogram is a procedure, but some versions use the word **void**, which means that the function returns nothing, and the word **return** is used to assign a value to the function name.

- **return**: the **return** instruction exits the sub-program and returns it to the program that called it at the instruction immediately following the invocation. It can return a value to the program that called the sub-program if it was a function.

Format :

```
return [ <expression> ] ;
```

Example:

`return 5*x ;`	If a function
`return ;`	if it's a procedure (i.e. a void function)

Important notes:

- To find the arguments, we ask what we're giving the subprogram as input and what it's returning as output.
- The list of parameters in the definition part of the sub-program must be identical in number and type to that used in the sub-program invocation.
- The first line of a function or procedure declaration, i.e. function type, function name, type, order and number of arguments, except their names, is called the header or prototype.
- Arguments are not grouped together if they are of the same type, as in (x, y:integer), but we put (x:integer, y:integer) (int x, int y)
- Any return type other than void indicates that the program is a function and not a procedure.
- void main() or simply main() is a procedure, while int main() is a function, so you need to use return.
- scanf() and printf() are two functions declared in the stdio library

2.3 Where to declare subprograms :

In the algorithm, it is located after the declaration of variables and before the begin of the main program.

In a C program, it is declared before the main() function.

Note: The order of subroutines is important, as each function must be defined before it can be used. In other words, if function f1() calls function f2(), then function f2() must be defined before function f1().

2.4 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 brackets, called effective parameters. Parentheses can be omitted in the absence of any arguments, but in C, they are mandatory.

The same goes for calling a function, where its name is considered a variable that carries a certain value, so the function call can be used as a variable in assignment operations and other expressions.

The parameters must correspond in number, type and order with the formal parameters.

2.5 Examples

Examples of procedures

- If numbers below a certain limit are displayed on the screen, it takes the upper limit and returns nothing.
  `procedure displayNbs(n : integer)`
- Display array values on screen takes an array and returns nothing
  `procedure displayTab(t :real array, n :integer)`
- Solve a quadratic equation that takes three coefficients and returns two solutions
  `procedure eq2(a : integer, b : integer, c : integer, var x1 : integer, var x2 : integer)`

Examples of functions

- Square a number Takes a number and returns its square
  `function square(x :real) : real`
- The area of a rectangle takes two numbers and returns the area
  `function area(long :real, wide :real) : real`
- Solving a first-order equation takes two coefficients and returns a solution
  `function eq1(a :real, b :real) : real`
- The sum of an array takes an array and returns the sum
  `function sum(t :array of real numbers, size :integer) : real number`
- whether the number is prime or not
  `function isPrime(x : integer) : Boolean`

Example Algorithm

algorithm Test	Program name
var z : real	Global variable
procedure displayNbs(n:entire)	The name of the procedure that takes an integer variable n as argument
var i:integer	local variable
Begin	The begin of the procedure

for i←1 **to** n **do**
Write(i)
endfor
	Procedural instructions

End procedure	end of procedure
function sumNbrs (x:integer, y:integer) : integer	The name of the function that takes two integer variables and returns an integer result. x and y are not grouped even if they are of the same type.
Begin	The begin of the
sumNbrs ←x+y	The function name acts as a variable and takes the result of the sum
End function	end of function
Begin	Begin of main program
displayNbs(5)	Call the displayNbrs procedure, where 5 is assigned to n, and the procedure displays the numbers from 1 to 5.
z←sommeNbrs (5, 3)	Calling sumNbrs, the program assigns the value 5 to x and the value 3 to y, then calculates the sum and assigns it to z

Write("the sum is ", z)	It displays the sum is 8
End.	End of main program

Examples C

#include <stdio.h>	tilizing the stdio library
float z ;	Global variable
void displayNbs (int n)	The name of the procedure that takes an integer variable n as an argument
{	The begin of the procedure
int i ;	local variable
for (i=1; i<=n; i++) printf("%d\t",i);	Procedural instructions
}	end of procedure
int sumNbrs (int x, int y)	The name of the function that takes two integer variables and returns an integer result. x and y are not grouped even if they are of the same type.
{	The begin of the
return x+y ;	The function name acts as a variable and takes the result of the sum
}	end of function
int main()**{**	Begin of main function
displayNbs (5);	Call the displayNbrs procedure, where 5 is assigned to n, and the procedure displays the numbers from 1 to 5.
Z=sumNbrs (5, 3);	Calling sumNbrs, the program assigns the value 5 to x and the value 3 to y, then calculates the sum and assigns it to z
printf("sum is %d", z);	It displays the sum is 8
return 0 ;**}**	End of main function

3 Local and global variables

A **global variable is** a variable declared outside the body of any sub-program, and therefore usable anywhere in the program. Since a variable is global, it is not necessary to pass it as a parameter to use it in subprograms. As for its lifetime, i.e. its existence in memory, it is created when the program is loaded into memory, and is only deleted at the end of program execution.

A local variable is a variable that can only be used in the subprogram or block where it is defined. The variable is created when the function is called and deleted when execution is complete.

- We recommend using local variables and parameters rather than global variables to avoid errors and ensure function independence.

Example Algorithm:

algorithm glob_loc	
Var glob, b : integer	global variables
Procedure tst	
Var b, loc : integer	local variables
Begin	
glob←11	Global variables are accessible within the
b←22	Local variable b hides global variable b
loc←33	
Write("in tst: glob=", glob, "b=", b, "loc=", loc)	
End	
Begin	
glob←1	
b←2	Variable b is a global variable

`Write("before tst : glob=", glob, "b=", b)`	Local variables such as loc are not accessible
`tst`	Procedure call
`Write("after tst : glob=", glob, "b=", b)`	
end	

Example in C

`#include <stdio.h>`	
`int glob, b ;`	global variables
`tst(){`	
`int b, loc ;`	local variables
`glob=11;`	Global variables are accessible within the
`b=22;`	Local variable b hides global variable b
`loc=33;`	
`printf("in tst: glob=%d b=%d loc=%d", glob, b, loc);`	
`}`	
`int main(){`	
`glob=1;`	
`b=2;`	Variable b is a global variable
`printf("before tst : glob=%d b=%d", glob, b);`	
`//Local variables such as loc are not accessible`	
`tst();`	Procedure call
`printf("after tst : glob=%d b=%d", glob, b);`	
`return 0 ;}`	

Screen :

```
before tst :   glob=1     b=2
in tst:        glob=11    b=22   loc=33
after tst :    glob=11    b=2
```

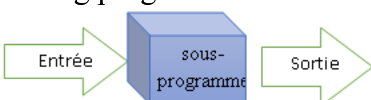
Explanation:

before calling tst	During tst call	after calling tst
glob b 1 2	glob b 11 2 tst b loc 22 33	glob b 11 2

Before the call, there are only two variables glob and b, but when the tst procedure is called, the processor reserves two more variables, loc and b. The procedure can access global variables, but the local variable b hides the global variable b, and when the procedure is terminated, the processor deletes all local variables.

4 Passing parameters

Arguments are the variables through which information can be exchanged between programs, i.e. the input of data from the calling program to the subprogram and/or the output of results from the subprogram to the calling program.



There are two ways of passing parameters or arguments

Passage by value :

In this mode, the value of the original variable is copied into the (formal) parameter, and this copy is used (a local variable), leaving the original variable unchanged. In this mode, a constant value or expression can be passed, and need not be a variable.

This mode is only used to enter information into the sub-program and is not used to receive results.

Passage by reference, address or variable :

Not only is the value passed, but the place of the original variable (address) is passed to the formal variable, so they become a single variable, and any modification of the parameter in the sub called program results in the modification of the original variable that was passed as a parameter.

In this mode, it's not possible to pass a constant value or an expression, but it must be a variable, so it's called pass by variable.

This mode is used to enter information for the sub-program, especially large variables such as arrays and matrices, to avoid copying. It is also used to receive results.

In algorithm the word "**var**" is used before declaring the name of the argument to indicate that the pass is a pass by variable or pass by reference.

To pass arguments with address in C, we use the pointers we'll see in the third chapter of this course, where the name of the formal parameter is preceded by * when declared and when used, but when the function is called, this variable is preceded by "**&**".

Declaration	int f(int *x)
Usage	*x=5;
Call	f(&a);

In C++, pointer management is masked by using the "**&**" symbol in the declaration only, and this is called a reference.

Declaration	int f(int &x)
Usage	x=5;
Call	f(a);

Note: We don't use the word **var** (* in C) to enter data and display results.

Example Algorithm :

by value	passage by reference, address or variable
algorithm Passage_value	**algorithm** Variable_passage
var a, c: real	**var** a, c: real
Procedure square (x: real, y: real)	**Procedure** square (x: real, **var** y: real)
Begin	**Begin**
y← x*x	y← x*x
end	**end**
begin	**begin**
c←0	c←0
a←3	a←3
write("before square c=", c)	write("before square c=", c)
square(a ,c)	square(a,c)
// we can use square(3,c)	write("after square c=", c)
write("after square c=", c)	**end**
end	
the screen	
before square c=0	before square c=0
after square c=0	after square c=9

Example C:

passage by value	passage by reference, address or variable
#include <stdio.h>	#include <stdio.h>
void square(float x, float y){	**void** square(float x, float ***y**) {
y= x*x;	*y=x*x;

<table>
<tr><td><pre>}
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 ;}</pre></td><td><pre>}
int main(){
   float a, c;
   c←0;
   a←3;
 printf ("before square c=%f ", c);
 square(a,&c);
 // square(a,5) cannot be used
  printf ("after square c=%f", c);
return 0 ;}</pre></td></tr>
<tr><td colspan="2">the screen</td></tr>
<tr><td>before square c=0
after square c=0</td><td>before square c=0
after square c=9</td></tr>
</table>

Switching from a procedure to a function :

Any procedure that returns a single result can be converted into a function, where we change the word Procedure into function and transform the argument that the procedure returns into a local variable and define the type of the function as the type of this argument and before terminating the function, we assign the value of the variable to the name of the function.

For example, the sub-program that calculates the absolute value of a real number :

<table>
<tr><th>In the form of a procedure</th><th>In the form of a function</th></tr>
<tr><td><pre>Procedure abs (x: real, var y: real)
Begin
   if x<0 then
      y← -x
   else
      y← x
   end if
end</pre></td><td><pre>function abs (x: real) : real
   var y: real
Begin
   if x<0 then
      y← -x
   else
      y← x
   end if
   abs←y
end</pre></td></tr>
<tr><td colspan="2">call</td></tr>
<tr><td>abs(-5, z)</td><td>z←abs (-5)</td></tr>
</table>

In C

<table>
<tr><td><pre>void abs (float x, float *y){
   if (x<0)
      *y= -x;
   else
      *y= x;
}</pre></td><td><pre>float abs (float x){
   float y;
   if (x<0)
      y= -x;
   else
      y= x;
   return y ;
}</pre></td></tr>
<tr><td>The variable y can be omitted
You can omit else, which comes after return.</td><td><pre>float abs (float x){
   if (x<0)
      return -x;
   return x;
}</pre></td></tr>
<tr><td colspan="2">call</td></tr>
<tr><td>abs(-5, &z);</td><td>z=abs(-5);</td></tr>
</table>

5 Recursivity

The recursion is a simple and elegant way of solving certain problems of a recurring nature.

A recursive program is any program that recalls itself. Whereas a defined program is used to define itself. In concrete terms, a recursive program is one that does part of the work and then recalls itself to complete the rest.

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 for stopping the recursion, which is the case when the program doesn't call itself or it will never stop.
It is preferable to test the stop condition first, then, if the condition is not met, to call the program back as the call leads to the stop condition.

Example:

```
Procedure display (i :integer)
begin
   write(i)
   display (i +1)
end.
```

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

For example, we invoke `display`(1), so it displays 1, then it invokes `display` for i=i+1=2, so it displays 2, then to infinity, so the algorithm must have a stop condition, by Example:

```
Procedure display (i :integer)
begin
  if (i<10) then
    write(i)
    display (i +1)
  endif
end.
```

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

The general form of the recursive program :

```
procedure Recursive (parameters)
begin
   if (stop condition) then
      <stop point instructions>
   else
      <instructions>
      Recursive call (parameters changed)
      <Instructions>
   endif
end
```

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

Example:

1. Factorial

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

The function can be written as a recursive relationship:

$$b_0 = 1$$

$$b_n = nb_{n-1}$$

iterative	recursive

```
Function fact (n : Integer) : Integer
var i, f: Integer
begin
   f←1
   for i←2 to n do
      f← f * i
   ffor
   fact←f
end.
```

```
Function fact (n : Integer) : Integer
begin
   if (n = 0) then
      fact←1
   else
      fact←n*fact (n-1)
   endif
end.
```

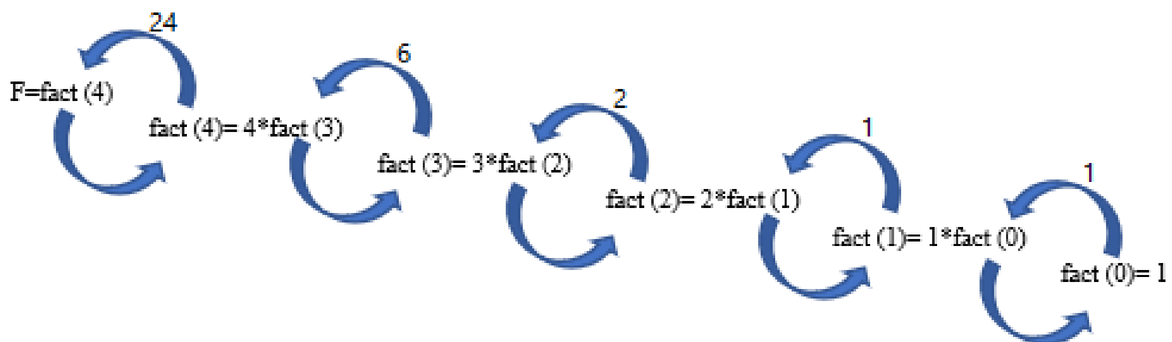
```
int fact (int n){
   int f=1;
   for (i=2 ;i<= n ; i++)
      f← f * I;
   return f;
}
```

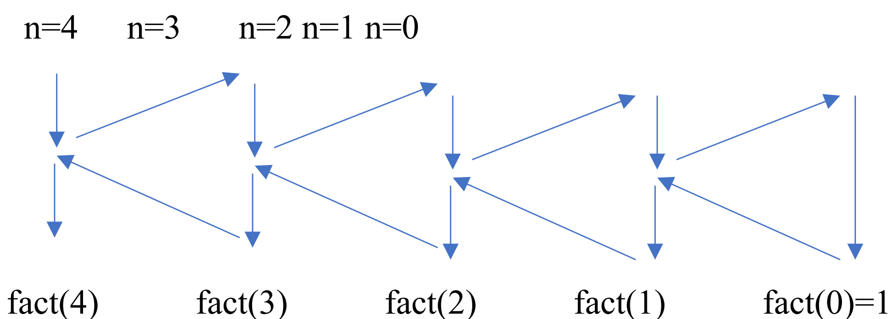
```
int fact (int n){
   if (n == 0) return 1;
   return n*fact(n-1);
}
```

How does it work?

We call the function fact for n=4 to calculate 4!

We call F=fact(4) which in turn calls fact(3) which calls fact(2) until it calls fact(0) which terminates and returns 1 allowing fact(1) to be calculated which allows fact(2) to be calculated until fact(4) is calculated fact(4). See below.





The execution stack :

A memory location designated to hold parameters and local variables, and where the result is stored for each running sub-program.

Usually, programming in recursive mode is easier and more readable, but it consumes a lot of memory, for example to calculate 4! We reserve a place in the stack for the result, another for the parameter n=4, then another place for the result of 3! And the parameter n = 3 and so on until 0! is calculated The parameter n=0 is deleted, then the parameters and results are deleted in the reverse order in which they were created.

Mutual recursive: a recursive program can call itself directly or indirectly, because it calls another program, which in turn calls the first program.

Example:

To calculate π, we use the following relationship π/4=1-1/3+1/5-1/7+1/9... We create two recursive functions, the first adding 1/n, calling the second for n=n-2, then subtracting 1/n which in turn calls the first to add and so on until n becomes zero.

```
function f1(n: integer)
begin
    if n<=0 then
        f1←0
    else
        f1←1/n+f2(n-2)
    endif
end

function f2(n: integer)
begin
    if n<=0 then
        f2←0
    else
        f2←-1/n+f1(n-2)
    endif
end
```

```c
#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);
}
```

The f1 function calculates π/4, and to calculate π, we multiply the result by 4.

Important note: Since function f1 calls function f2, which is not yet defined in C, the header of function f2 must be added without its body (the first line) before defining function f1, knowing that its definition comes after .