University of Setif 1
Faculty of Science
Department of Computer Science
Course: 2nd A. Academic Bachelor's degree
Academic year 2020/2021/2022/2023/2024

Module: Algorithmics and Data Structure (S3)

CH1 – REMINDER 2
Functions

Summary:

- Benefits of a modular program
- Defining a Function
- Calling a Function
- The settings
- Changing Arrays to Parameters
- Passing parameters by reference
- Input and output parameters
- Skip a table of any size
- Scope of variables
- Conclusion
- Corrected exercises

Why write functions

When you have a set of lines of code that need to be executed in different places in a program, instead of rewriting the same lines of code, it is interesting to create functions (modules).

Instead of writing a 500-line main() function, it is better to create 25 20-line functions
• the program is structured into modules.
• It is easier to test each function (module).

It is impossible to have more than 1000 lines of code in mind: most real programs have tens of thousands of lines and large applications have millions. Writing functions is absolutely mandatory.

Here are some **benefits of a modular program**:

• ***Better readability***
• ***Reduced risk of errors***
• ***Possibility of selective testing***
• ***Concealment of methods***
When using a module, you only need to know its effect, without having to worry about the details of its implementation.
• ***Reuse of existing modulesIt is easy to use modules that you have written yourself or that have been developed by others.***
• ***Easy maintenanceA 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 delegating the programming of the modules to different people or groups of people. Once developed, the modules can form a common basis for work.
• ***Module PrioritizationA program can first be solved globally at the main module level. The details can be transferred to subordinate modules, which can also be subdivided into sub-modules, and so on. In this way, we get a*** *hierarchy of modules*. Modules can be developed from the top down in the hierarchy *('top-down-development')* or from the bottom *up-development.*

In the definition of a function, we indicate:

- the name of the function- the type, number and names of the function's parameters- the type of the result provided by the function- the data local to the function- the instructions to be executed
Definition of a function in algorithmic language

```
function <FunctionNameFunction> (<NameBy1>, <NameBy2>, ...):<TypeRes>
```

```
    |  <Parameter declarations>
    |  <local statements>
    |  < instructions>
    ffunction
```

Defining a function in C

```
<ResType> <FunFunctionName> (<TypeBy1> <NameBy1>,
                             <TypeBy2> <NameBy2>, ... )
{
 Body of the function
 <local statements>
 < instructions>
}
```

Notice

That there is no semicolon behind the definition of the function's parameters.
Each time the function is called, the body of the function is executed.
The identifier is the name of the function.
The function can have settings.
The function can return a value of type type.

Calling a Function

When the function is called, the program executes all the instructions in the function body, and then resumes the calling program right after the function is called.

Example 1: An example of a function

```
#include <iostream>
using namespace std;
void b()
{
 cost<<"Hello"<<endl;
}
int main()
{
        cost<<"SALEM1"<<endl;
        b();
        cost<<" SALEM 2"<<endl;
        b();
        b();
        cost<<" SALEM 3"<<endl;
        b();
        system("pause");
        return 0;
}
```

Explanations

• In this program, we have created a function b that simply displays "Hello" on the screen. Function b is preceded by the void type: this means that the function does not return any value to the calling program.
• The main program (the main() function) displays "SALEM 1" on the screen, then calls function b, displays the message "SALEM 2" on the screen, then calls function b 2 times, displays the message "SALEM 3" and calls function b one last time.

Running Example 1

When you run the program, here is what you get on the screen:

```
SALEM 1
Hello
SALEM 2
Hello
Hello
SALEM 3
Hello
```

The settings

You can set up a function: the parameters make the function more general and therefore more easily reusable. Code reuse is one of the fundamental concepts of the C++ language. The notion of function is one of the first methods that allows us to approach the subject.

Example 2: A Function with Parameters

```
#include <iostream>
using namespace std;
void b(int i)
{
             int j;
            for(j=0; j<i; j++)
            cost<<"Hello"<<endl;
}
int main()
{
            cost<<" SALEM 1"<<endl;
            b(2);
            cost<<" SALEM 2"<<endl;
            b(3);
            system("pause");
            return 0;
}
```

Explanations

• This time the function b has an integer parameter i. This function displays "Hello" on the screen i times.

• In the main program, the main() function displays "SALEM 1", then calls the function b with parameter 2, **When calling b(2) of the function, we copy the value 2 into i and then we execute the body of the function with this value of i: we therefore display "Hello" 2 times.

• When calling b(3) the function, we copy the value 3 into i and then we execute the body of the function with this value of i: we therefore display "Hello" 3 times.

Execution

When you run the program, here is what you get on the screen:

```
SALEM 1
Hello
Hello
SALEM 2
Hello
Hello
Hello
```

The settings

You can set up a function: the parameters make the function more general and therefore more easily reusable. Code reuse is one of the fundamental concepts of the C++ language. The notion of function is one of the first methods that allows us to approach the subject.

Example 2: A Function with Parameters

```
#include <iostream>
using namespace std;
void b(int i)
            { int j;
            for(j=0; j<i; j++)
            cost<<"Hello"<<endl;
            }
int main()
            { cost<<"SALEM1"<<endl;
            b(2);
            cost<<" SALEM 2"<<endl;
            b(3);
            system("pause");
            return 0;
            }
```

•Explanations
• This time the function b has an integer parameter i. This function displays "Hello" on the screen i times.
• In the main program, the main() function displays "SALEM 1", then calls the function b with parameter 2,
**When calling b(2) of the function, we copy the value 2 into i and then we execute the body of the function with this value of i: we therefore display "Hello" 2 times.
• When calling b(3) the function, we copy the value 3 into i and then we execute the body of the function with this value of i: we therefore display "Hello" 3 times.

•Execution
When you run the program, here is what you get on the screen:

```
SALEM 1
Hello
Hello
SALEM 2
Hello
Hello
Hello
```

Function Environment
• A function can have its own local variables.
• The environment of a function is the set of variables accessible in the body of the function.
• It is possible to define global variables (see the definition of constants): it is strongly recommended not to use global variables other than constants.
• The environment for a function includes:
    • variables local to the function.
    • Function settings.
    • global variables.

A function that returns a value
A function can return a value of a certain type to the calling environment. void means that the function does not return anything. If we want the function to return a certain value, we will have to write another type instead of void. The return keyword allows you to:
• stop the execution of the function (it is usually placed at the end of the function)
• Return a value to the calling environment. If a function b returns an integer, the call will be written: a=b(...); a will retrieve the value returned by the return.

Example 3: A function that returns a value

```
#include<iostream>
using namespace std;
int b(int i, int j)
        { int k;
        k = i*i + j*j;
        return k;
        }
int main()
        { int a;
        a = b(3,4);
        cost<<"The result is: "<<a<<endl;
        return 0;
        }
```

•Explanations
When calling a=b(3,4);
• 3 and 4 are copied into variables i and j respectively.
• We execute the body of the function (which contains a local variable k)
• Return k stops the execution of the function by returning the value of k which is copied back to a.
• The environment of the main function includes only the variable a: the variables i, j and k cannot be used in the main function.
• The environment of function b includes the variables i, j and k: we cannot use a in function b.

• Separation of environments allows for better structure of applications.

• **Execution** :
When you run the program, here is what you get on the screen:

```
The result is: 25
```

Example 4: Another Function

```
#include <iostream>
using namespace std;
int b(int i)
        { int k, s=0;
        for(k=1; k<=i; k++)
        s = s+k*k;
        return s;
        }
int main()
        { int a; a = b(4);
        cost<<"The result is:"<<a<<endl;
        return 0;
        }
```

•Explanations
• We define a function b that has an integer parameter i and that calculates the sum of the squares of the first i integers.
• In the main function, we call b(4) and we get the value returned in the variable a. a therefore contains the sum of the first 4 squares.

•Execution
When you run the program, here is what you get on the screen:

```
The result is: 30
```

Example 5: A function manipulating reals

```
#include <iostream>
using namespace std;
Double B(Double X, Double Y)
        { double m; m = (x+y) / 2;
        return m;
        }
int main()
        { double a; a = b(3.2, 4.2);
        cost<<"The result is:"<<a<<endl;
        return 0;
        }
```

•Explanations:
• In this example, function b has 2 parameters of type double, named x and y. The function b returns a double. This function returns the average of x and y. The variable m is a variable local to function b.
• In the main program, we recover in a the average of 3.2 and 4.2 by calling the function b.

•Execution:
When you run the program, here is what you get on the screen:

```
The result is: 3.7
```

Type Compatibility

A function can have parameters of different types. When calling, the order and type of the parameters between the header and the function body must be respected.

Changing Arrays to Parameters

When we pass an array (of integers for example as a parameter), there is identification between the array of the calling environment and the parameter of the function. Any changes to the array in the function are reflected in the array of the calling environment.

Example 6: Passing Arrays as a Parameter

```
#include <iostream>
using namespace std;
const int n=4;
void enter(int t[n])
        { int i;
        for(i=0; i<n; i++)
                { cost<<"Type the value number "<<i<<": ";
                cin >> t[i];
                }
        }

void display(int t[n])
        { int i;for(i=0; i<n; i++)
        cost<<"The value number "<<i<<" is: "<<t[i]<<endl;
        }

int main()
        { int a[n];
        seize(a);
        poster(a);
        return 0;
        }
```

•Explanations

• When you call enter(a), there is identification of the array a and the parameter t of the enter function: any modification of t modifies the array a. The enter a function allows you to ask the user to enter all the boxes in an array of n boxes one by one.
• The function displays an array of n integers as parameters and displays all the boxes in that array.
• The program is structured: it is made up of a set of short functions whose role can be easily identified.
• The main() becomes a very short program!

•Execution

When you run the program, here is what you get on the screen:

```
Type the number 0 value: 4
Type the number 1 value: 6
Type the number 2 value: 5
Type the number 3 value: 7
The number 0 value is: 4
The number 1 value is: 6
The number 2 value is: 5
The number 3 value is: 7
```

Careful

A change to a parameter that is not an array in the body of a function is not reflected in the calling environment.

Example 7

```
#include <iostream>
using namespace std;

void enter(int n)
        { cost<<"Type an integer: ";
        cin>>n;
        }
```

```
void display(int n)
        { cost<<"The value of the integer is: "<<n<<endl;
        }

int main()
        { int x; x=54;
        seize(s);
        poster(s);
        return 0;
        }
```

Explanations
• In the main function, we define an integer variable x and set it to 54.
• We then call the enter function: we copy the value of x into n. In the body of the enter function, the user is asked to type a value that is put in x. But at no time is this value x put in the variable x: the value typed by the user is therefore lost.
• The value of x therefore remains at 54. When we call the display function, we display the value 54.

Execution
When you run the program, here is what you get on the screen:

```
Type an integer: 20
Integer value: 54
```

Passing parameters by reference

A parameter can be passed by reference (not by copy) by indicating in the function's header one & after the type. The parameter of the function and the variable of the calling environment are then identified.

Example 8: Passing parameters by reference

```
#include <iostream>
using namespace std;

void enter(int & n)
        { cost<<"Type an integer: ";
        cin>>n;
        }

void display(int n)
        { cost<<"The value of the integer: "<<n<<endl;
        }

int main()
        { int x; x=54;
        seize(s);
        poster(s);
        return 0;
        }
```

Explanations
• When calling enter(x), the variables x and n are identified: any change in n changes the value of x. When the user enters the value of n in the cin>>n statement; , there is a change in the content of variable x. We retrieve in x the value typed by the user
• We then call the display function which displays the value of x.

Execution
When you run the program, here is what you get on the screen:

```
Type an integer: 20
Integer Value: 20
```

Input and output parameters

• Some parameters will provide data to a function: the input parameters.

• Others will allow you to return a value to the calling environment: the output parameters.
• Technically, we will use the return or the passing of parameters by reference to send a value to the calling environment.
• Others may have the 2 roles: input/output parameters.

Example 9: Input and output parameters

```
#include <iostream>
using namespace std;

Void minmax(int i, int j, int & min, int & max)
        {if(i<j)
                {min=i; max=j;}
        else {min=j; max=i;};
        }

int main()
        { int a,b,w, x;
        cost<< "Type the value of a:"; cin >> a;
        cost<< "Type the value of b:"; cin >> b;
        minmax(a,b,w,x);
        cost<< "The smallest is: " << w << endl;
        cost<< "The largest is: " << x << endl;
        return 0;
        }
```

Explanations
• The minmax function allows you to retrieve the smallest and largest of 2 integers i and j.
• In the minmax function, i and j are the input parameters and min and max are the output parameters.
• In main, when we call the minmax function, we copy the values of a and b respectively into i and j and we identify the w and min variables and the x and max variables for the duration of the function call.
• In w we therefore recover the smallest of a and b and in x the largest of a and b.

Execution
When you run the program, here is what you get on the screen:

```
Type the value of a: 15
The value of b is: 11
The smallest is 11
The largest is worth 15
```

Skip a table of any size

• The parameter of a function can be an array of any size.
• In the calling environment, the size of the array will need to be determined.
• In general, when passing an array of any size, another integer parameter is passed that indicates the size of the array.

Example 10: Passing an array of any size

```
#include <iostream>
using namespace std;
void enter(int t[], int n)
        { int i;
        for(i=0; i<n; i++)
                { cost<< "Type integer number " << i << ": ";
                cin >> t[i];
                 }
        }

void display(int t[], int n)
        { int i;
        for(i=0; i<n; i++)
        cost<< "The integer number " << i << " is: " << t[i] << endl;
```

```
          }

int main()
{ int a[3], b[5];
          cost<< "TABLE ENTRY a" << endl;
          seize(a, 3);
          cost<< "TABLE ENTRY b" << endl;
          seize(b, 5);
          cost<< "TABLE DISPLAY to" << endl;
          fficher(a, 3);
          cost<< "TABLE DISPLAY b" << endl;
          display(b, 5);
          return 0;
}
```

Explanations

• In the enter function, the first parameter is int t[], so it is an array of integers of any size. The second parameter n indicates the size of the array. This function requires the user to enter all the boxes in the table one by one.
• The display function also has as parameters an array of integers of any size and a parameter n that indicates the size of the array. This function displays the contents of the table.
• In the main program, the contents of table a containing 3 cells and table b containing 5 cells are entered in
** However, it can be seen that in the main function, the size of the arrays a and b is well known.

Execution

When you run the program, here is what you get on the screen:

```
ENTERING THE TABLE a
Type integer number 0: '"123"'
Type integer number 1: '"456"'
Type integer number 2: '"789"'
ENTRY OF THE TABLE b
Type integer number 0: '"987"'
Type integer number 1: '"654"'
Type integer number 2: '"321"'
Type integer number 3: '"741"'
Type integer number 4: '"852"'

TABLE DISPLAY a
The integer 0 is : 123
Integer 1 is : 456
Integer 2 is 789
TABLE DISPLAY b
0 is : 987
Integer 1 is : 654
Integer 2 is: 321
Integer 3 is 741
The integer 4 is : 852
```

Scope of variables

• In the hand, you can access:
    • Main local variables
    • global constants
• Within a function, one can access
    • Function settings
    • global constants

• It is not recommended to define non-constant global variables!
• Exchanges between environments take place:
    • by returns
    • by parameter passes by reference

Conclusion

• It will be necessary to master the passage of parameters by value and by reference as well as the particular role of tables.
• It is essential to have a clear understanding of the role of the input and output parameters of a function.
• The functions will play a fundamental role in the structuring of our programs. Our programs will now be made up of a multitude of functions that will be small in size (10 to 30 lines) and that will all have a very specific role.
• This methodology is essential for creating long programs.

Training

NB: Adapt the proposed solutions to your development environment

Exercise 1

Write a program using a function F to display the table of values of the function defined by

$$f(x) = \sin(x) + \ln(x) - \sqrt{x}$$

where x is an integer between 1 and 10.

Solution

```
#include <stdio.h>
#include <math.h>
 main()
{
 /* Prototypes of functions called */
 double F(int X);
 /* Local variables */
 int I;
 /*Treatments*/
 printf("\tX\tF(X)\n");
 for (I=1; I<=10; I++)
      printf("\t%d\t%f\n", I, F(I));
 return 0;
}
double F(int X)
{ return sin(X)+log(X)-sqrt(X);
}
```

Exercise 2

Write the NDIGITS function of the int type that gets an integer value N (positive or negative) of the long type as a parameter and provides the number of digits of N as the result.

Write a small program that tests the NDIGITS function:

Example:

```
         Introduce an integer: 6457392
         The number 6457392 has 7 digits.
```

Solution 2

```
#include <stdio.h>
main()
{
 /* Prototypes of functions called */
 int N(long N);
 /* Local variables */
 long A;
 /*Treatments*/
 printf("Introduce an integer: ");
 scanf("%ld", &A);
 printf("The number %ld has %d digits.\n",A ,NDIGITS(A));
 return 0;
}

int NDIGITS(long N)
{/* Since N is transmitted by value, N can be */
 /* modified inside the function. */
 int I;
 /* Conversion of the sign if N is negative */
 if (N<0)
      N *= -1;
  /* Count the numbers */
 for (I=1; N>10; I++)
      N /= 10;
 return I;
}
```

Exercise 3 "Sorting by propagation"

Write the TRI_BULLE function that sorts an array of N integer elements in ascending order by applying the bubble method (propagation sorting - see exercise 7.15). Use the RANGER function from the exercise above.
Write a program to test the function TRI_BULLE.

Solution 3 "Sorting by propagation"

```
#include <stdio.h>
main()
{/* Prototypes of functions called */
 void LIRE_TAB (int *TAB, int *N, int NMAX);
 void TRI_BULLE(int *T, int N);
 void ECRIRE_TAB (int *TAB, int N);
 /* Local variables */
 int T[100]; /* Array of integers */
 int DIM;     /* Table size */
 /*Treatments*/
 LIRE_TAB (T, &DIM, 100);
 printf("Array given: \n");
 ECRIRE_TAB (T, DIM);
 TRI_BULLE(T, DIM);
  printf("Sorted array: \n");
 ECRIRE_TAB (T, DIM);
 return 0;
}
void TRI_BULLE(int *T, int N)
{
  /* Prototypes of functions called */
 int RANGER(int *X, int *Y);
 /* Local variables */
 int I,J;  /* common indices */
 int FIN;  /* position where the last permutation took place */
      /* allows you to skip a subset that has already been sorted. */
 /* Sorting of T by propagation of the maximum element */
 for (I=N-1; I>0; I=END)
    {
     END=0;
     for (J=0; J<I; J++)
          if (ARRANGE(T+J, T+J+1)) END = J;
    }
}
int RANGER(int *X, int *Y)
{
 . . .
}
void LIRE_TAB (int *TAB, int *N, int NMAX)
{
 . . .
}
void ECRIRE_TAB (int *TAB, int N)
{
 . . .
}
```

Exercise 4

Write the function ADDITION_MATRICE which performs the following addition of the matrices:

MAT1 = MAT1 + MAT2

Choose the necessary parameters and write a small program that tests the function ADDITION_MATRICE.

Solution 4

```
#include <stdio.h>
main()
{
 /* Prototypes of functions called */
 void ADDITION_MATRICE (int *MAT1, int *MAT2, int L, int C, int CMAX);
 void LIRE_DIM (int *L, int LMAX, int *C, int CMAX);
 void LIRE_MATRICE (int *MAT, int L, int C, int CMAX);
 void ECRIRE_MATRICE (int *MAT, int L, int C, int CMAX);
 /* Local variables */
 /* Matrices and their dimensions */
 int M1[30][30], M2[30][30];
 int L, C;
 /*Treatments*/
 LIRE_DIM (&L,30,&C,30);
 printf("*** Matrix 1 ***\n");
 LIRE_MATRICE ((int*)M1,L,C,30 );
 printf("*** Matrix 2 ***\n");
 LIRE_MATRICE ((int*)M2,L,C,30 );
 printf("Matrix given 1: \n");
 ECRIRE_MATRICE ((int*)M1,L,C,30);
 printf("Matrix given 2: \n");
 ECRIRE_MATRICE ((int*)M2,L,C,30);
 ADDITION_MATRICE( (int*)M1 , (int*)M2 ,L,C,30);
  printf("Result matrix: \n");
 ECRIRE_MATRICE ((int*)M1,L,C,30);
 return 0;
}
void ADDITION_MATRICE (int *MAT1, int *MAT2, int L, int C, int CMAX)
{
  /* Local variables */
 int I,J;
 /* Add items from MAT2 to MAT1 */
 for (I=0; I<L; I++)
      for (J=0; <C; J++)
           *(MAT1+I*CMAX+J) += *(MAT2+I*CMAX+J);
}
void LIRE_DIM (int *L, int LMAX, int *C, int CMAX)
{
 . . .
}
void LIRE_MATRICE (int *MAT, int L, int C, int CMAX)
{
 . . .
}
void ECRIRE_MATRICE (int *MAT, int L, int C, int CMAX)
{
 . . .
}
```

Exercise 5

Write the MULTI_MATRICE function that multiplies the matrix MAT1 by an integer X:

MAT1 = X * MAT1

Choose the necessary parameters and write a small program that tests the MULTI_MATRICE function.

Solution Exercise 5

```
#include <stdio.h>
main()
{
 /* Prototypes of functions called */
 void MULTI_MATRICE(int X, int *MAT, int L, int C, int CMAX);
 void LIRE_DIM (int *L, int LMAX, int *C, int CMAX);
 void LIRE_MATRICE (int *MAT, int L, int C, int CMAX);
 void ECRIRE_MATRICE (int *MAT, int L, int C, int CMAX);
 /* Local variables */
 int M[30][30]; /* Integer matrix */
 int L, C;      /* Dimensions of the matrix */
 int X;
 /*Treatments*/
 LIRE_DIM (&L,30,&C,30);
 LIRE_MATRICE ((int*)M,L,C,30 );
 printf("Enter the multiplier (integer): ");
 scanf("%d", &X);
 printf("Matrix given: \n");
 ECRIRE_MATRICE ((int*)M,L,C,30);
 MULTI_MATRICE (X,(int*)M,L,C,30);
  printf("Result matrix: \n");
 ECRIRE_MATRICE ((int*)M,L,C,30);
 return 0;
}
void MULTI_MATRICE(int X, int *MAT, int L, int C, int CMAX)
{
  /* Local variables */
 int I,J;
 /* Multiplication of elements */
 for (I=0; I<L; I++)
      for (J=0; <C; J++)
           *(MAT+I*CMAX+J) *= X;
}
void LIRE_DIM (int *L, int LMAX, int *C, int CMAX)
{
 . . .
}
void LIRE_MATRICE (int *MAT, int L, int C, int CMAX)
{
 . . .
}
void ECRIRE_MATRICE (int *MAT, int L, int C, int CMAX)
{
 . . .
}
```