FUNCTIONS in C

1. Overview

The learning objective of this lab session is to:

- Understand the structure of a C function
- Understand function calls both using arguments passed by value and by reference
- Understand function prototypes
- Acquire hands on experience with program development using programmer made functions

2. Brief theory reminder

A program contains one or more functions, one of which is the main function – its name is **main**. The other functions have programmer-given names.

2.1. The structure of a function

The structure of a function is:

```
type name (formal_parameter_list)
{
  local_declarations
  statements
}
```

The first line in a function definition is called a function *header* and the rest is called a function *body*.

In C/C++ there are two categories of functions:

1. Functions which return a value, specified by the statement **return expression**; the returned value has the type specified in the header by **type.**
2. Functions which return no value, the **type** being replaced by the keyword **void**.

A formal parameter list may contain:

- zero parameters, case in which the function header reduces to:
  type name() or **type name**(**void**)
- one or more formal parameters, separated by commas. A formal parameter is specified by **type name**

Example:

int solve_system(int n, double a[10][10], double b[10], double x[10], int error)

A function *prototype* can be obtained by writing a semicolon after a construction identical to the function header, or obtained by eliminating the name(s) of the formal parameter(s).

Example:

```
int factorial (int n);
int factorial (int);
```

The standard library functions have their prototypes in various files with the extension **.h**, such as **stdio.h**, **conio.h**, **math.h**, etc. The standard library functions are stored in object format (extension

.obj) and are added to programs at link edition. The prototypes of the standard functions are included into programs prior to their invocation, via the **#include** directive.

2.2. Function invocation

A function which does not return a value is invoked as follows:

name(effective_parameter_list);

The correspondence between the formal and the effective (actual) parameters is by position.

A function which returns a value is invoked as follows:

- Either using an invocation statement as above, case in which the returned value is discarded,
- or as an operand of an expression, the returned value is then used in evaluating the expression.

It is recommended that the type of the formal parameters and of the effective (actual) parameters coincides. If that is not satisfied, in C, the type of the effective (actual) parameters is automatically converted to the type of the formal parameter. In C++, a more complex verification method is used for function calls. That is why the use of the explicit conversion operator (type cast operator), **(type)**, is recommended.

Example:

f((double)n)

The return from a function is executed either after the execution of the last statement of the function body, or when a return statement is encountered.

The **return** statement has the following formats:

return;

or

return expression;

Notes.

1. If the type of the expression in the return statement is not the one specified in the function header, an automatic conversion to the type specified in the function header is performed.
2. The first of the two formats is used for functions returning no value.

The passing of the effective (actual) parameters can be done:

- By **value** (call by value);
- By **reference** (call by reference).

In the case of the call by value, the value of the effective parameter is passed for the formal parameter. In this case, the called function cannot change the effective parameter, as it does not have access to its value. The program example **L4Ex1.c** illustrates this fact:

```
/*Program L4Ex1.c  */

#include <stdio.h>
/* CALL BY VALUE */
/* Procedure for swapping a and b */
void swap(int a, int b)
{
    int aux;

    printf("\nValues at entry: a=%d b=%d\n", a, b);
```

```
    aux=a;
    a=b;
    b=aux;
    printf("\nValues at exit: a=%d b=%d\n", a, b);
}
int main()
{
    int a, b;

    a=2;
    b=3;
    printf("\nIn the main function, prior to invoking swap: a=%d b=%d\n", a, b);
    swap(a, b);
    printf("\nIn the main function, after the return from swap: a=%d b=%d\n", a, b);
    return 0;
}
```

One can notice that the variables **a** and **b** retain their original values.

In order to allow for the swap to occur, it is necessary to use pointers, as below:

```
 /*Program L4Ex2.c  */

#include <stdio.h>
/* CALL BY VALUE USING POINTERS */
/* Procedure for swapping a and b */
void swap(int *a, int *b)
{
    int aux;

    printf("\nValues at entry: a=%d b=%d\n", *a, *b);
    aux=*a;
    *a=*b;
    *b=aux;
    printf("\nValues at exit: a=%d b=%d\n", *a, *b);
}
int main()
{
    int a, b;

    a=2;
    b=3;
    printf("\nIn the main function, prior to invoking swap: a=%d b=%d\n", a, b);
    swap(&a, &b);
    printf("\nIn the main function, after the return from swap: a=%d b=%d\n", a, b);
    return 0;
}
```

One can notice that the values of **a** and **b** were swapped. This way of parameter passing is still by value, i.e. an address was passed by value to a pointer.

In the case of call by reference, the addresses of the parameters are passed, not their values. This mode of parameter passing is valid in C++. In this case, the formal parameters are defined to be of type reference.

This former example becomes:

```
/*Program L4Ex3.cpp  */

#include <stdio.h>
/* CALL BY REFERENCE */
/* Procedure for swapping a and b */
void swap(int& a, int& b)
{
    int aux;

    printf("\nValues at entry: a=%d b=%d\n", a, b);
    aux=a;
    a=b;
    b=aux;
    printf("\nValues at exit: a=%d b=%d\n", a, b);
}
void main()
{
    int a, b;

    a=2;
    b=3;
    printf("\nIn the main function, prior to invoking swap: a=%d b=%d\n", a, b);
    swap(a, b);
    printf("\nIn the main function, after the return from swap: a=%d b=%d\n", a, b);
}
```

One can notice that the values of **a** and **b** were interchanged.

Important note: when an effective (actual) parameter is the name of an array, then its value is the address of the first element of the array; as a consequence, the formal parameter receives as a value this address. In this case, the invoked function can change the values of the elements of the array whose name was given as an affective (actual) parameter.

For an example of function usage, here is a program which performs some operations on a pair of polynomials.

```
/*Program L4Ex4.c  */

/* Operations on polynomials
   A polynomial is of the form P(x)=p[0]+p[1]*x+ p[2]*x^2 +…p[n]* x^n  */

#include <conio.h>
#include <stdio.h>
# define MAXDEGREE 20

/* Multiply two polynomials, a(x) of degree n, and b(x) of degree m,
   i.e.  c(x)=a(x)*b(x)   */
void multiply(int n, float a[], int m, float b[], int *p, float c[])
{
    int i, j;
    *p=n+m;
    for(i=0; i<=n+m; i++) c[i]=0.0;
    for(i=0; i<=n; i++)
    for(j=0;j<=m;j++) c[i+j]+=a[i]*b[j];
 }

/* Divide polynomial a(x), of degree n, by polynomial b(x) of degree m
```

```
   i.e. a(x)=b(x)*quotient(x)+remainder(x) */
void divide(int n, float a[], int m, float b[],
            int *quotient_deg, float quotient[],
            int *remainder_deg, float remainder[])
{
    int i, j, k;
    if (n<m)
    {
        *quotient_deg=0;
        quotient[0]=0.0;
        *remainder_deg=m;
        remainder=quotient;
    }
    else
    {
        *quotient_deg=n-m;
        *remainder_deg=m-1;
        for(i=n-m, j=n; i>=0; i--, j--)
        {
            quotient[i]=a[j]/b[m];
            for (k=m; k>=0; k--) a[i+k]=a[i+k]-quotient[i]*b[k];
            a[j]=0;
        }
       for(i=0; i<=m-1; i++) remainder[i]=a[i];
    }
}

/* Read the degree, n,  and coefficients of  polynomial a */
void read_polynomial(int *n, float a[])
{
    int i;
    printf("\nInput the degree of polynomial a: ");
    scanf("%d", n);
    for(i=0; i<=*n; i++)
    {
        printf("\na[%d]=", i);
        scanf("%f", &a[i]);
    }
    printf("\n");
}

/* Evaluate polynomial a, of degree n,  for a given value of x */
float evaluate_polynomial(float x, int n, float a[])
{
    int i;
    float v;
    v=0.0;
    for(i=n;i>=0;i--) v=v*x+a[i];
    return v;
}

/* Show a polynomial a, of degree n, named as c  */
void show_polynomial(int n, float a[], char c)
{
    int i;
    printf("\n%c[x]=%g", c, a[0]);
    for(i=1; i<=n; i++) printf("+%g*x^%d", a[i], i);
```

```
    printf("\n");
}

int main()
{
    int n, m, r_deg, quotient_deg, remainder_deg;
    float x, v, p[MAXDEGREE+1], q[MAXDEGREE+1], r[MAXDEGREE+1],
          quotient[MAXDEGREE+1], remainder[MAXDEGREE+1];

    clrscr();
    read_polynomial(&n, p);
    show_polynomial(n, p, 'P');
    read_polynomial(&m, q);
    show_polynomial(m, q, 'Q');
    printf("\nInput a value for the polynomial variable, x=");scanf("%f", &x);
    v=evaluate_polynomial(x, n, p);
    printf("Value of polynomial P on point x=%f is %f", x, v);
    getchar();

    multiply(n, p, m, q, &r_deg, r);
    printf("\nR[x]=P[x]*Q[x]\n");
    show_polynomial(r_deg, r, 'R');
    getchar();

    divide(n, p, m, q, &quotient_deg, quotient, &remainder_deg, remainder);
    printf("\nDivision  P[x]/Q[x] yields quotient C[x] and remainder R[x]\n");
    show_polynomial(quotient_deg, quotient, 'C');
    show_polynomial(remainder_deg, remainder, 'R');
    getchar();

    printf("\nWARNING. The polynomial P is changed.\n");
    show_polynomial(n, p, 'P');
    return 0;
}
```

3. Lab Tasks

3.1. Analyze the effective parameter passing modes used in the programs given as examples.

Write programs for solving the following problems, using functions and different parameter passing modes. Avoid using global variables.

3.2 Read the degree of a polynomial and its coefficients, all integers, from the standard input. The polynomial is of the form

$$P(x) = p_0 + p_1 x^1 + ... + p_n x^n ,$$

where p_0 is not zero.
The polynomial has only simple integer roots. Find its roots.

3.3. Read a positive integer, ***n***, and the pairs of integers $(x_i, y_i), \quad i = \overline{1,n}$, where these pairs represent a binary relation, ***R***, over a set ***M***. Provided that each element of ***M*** occurs in at least one pair, find the set, ***M***. Verify whether the relation, ***R***, is an equivalence relation (i.e. it is reflexive, symmetric, and transitive).

3.4. Two character strings representing very large decimal integers are given. Write a program to perform arithmetic operations with such very large decimal integers.

3.5. Write the functions for addition, subtraction and multiplication of two matrices, and then compute ***A=B*C−2*(B+C)***, where ***B*** and ***C*** are two quadratic matrices of order ***n***.

3.6. Write a function to perform arithmetic operations on sparse matrices. (A sparse matrix is a large matrix where most of the elements are zero).

3.7. Given three positive integers, representing a year, a month and a day-of-the-month respectively, write a function to compute the number of the day in the year, and the number of days to the end of that year.

3.8. Write a function which has a string of characters representing a Roman numeral as a parameter, and returns the corresponding radix (base) 10 Arabian number.

3.9. Write the complementary function, which converts a base 10 Arabian number to a Roman numeral.

3.10. Read an integer, multiple of 100, representing an amount of money expressed in RON from the standard input. Write a function to determine the minimum number of banknotes needed to pay that amount.

3.12. Write a function to check whether a character string is a substring of another character string. The function should return the position at which the substring starts if true, or –1 otherwise.

3.13. Write a function to check if its parameter (positive integer) is a perfect square. Then apply this function to a vector of positive integers, and extract all perfect squares and place them in another vector.

3.14. Write a function which returns the highest perfect square which is less or equal to its parameter (a positive integer).