PROGRAMMING LANGUAGES

Laboratory 5

PROCEDURAL PROGRAMMING. FUNCTIONS

I. THEORETICAL BACKGROUND

1. Procedural programming, functions, calling and returning functions

All high level programming languages use procedural programming. If the execution of a certain instruction set is desired, with different values or in different places, these instructions are grouped in a subroutine which will be called each time it is needed. The compiler jumps to this subroutine and then comes back to the program line following the one where the subroutine was called and therefore it is different from the jumps accomplished with the „goto" instruction. The instructions sequence organised in this way, received various names in different programming languages: subprogram, subroutine, procedure, function, etc.

All programming languages use two procedure categories:
a) Procedures that define a return value.
b) Procedures that don't define a return value.

Procedures from the a) category are usually called functions. The return value is also called returning value or value returned by the function.

In the C language, all subroutines are called functions. As said before, theoretically any function returns a value. In reality though, the returned values are sometimes ignored and recent definitions admit function declarations with return type void. This means they don't return any value.

A function can be declared and defined. When it is declared, a function becomes available in the program, so as for other functions to be able to call it. When defining it though, the explicite function code is specified.

The "C" language integrates a series of frequently used functions. They are kept in a special file, with OBJ format, that is a compiled file and it is added to the editing phase in the program. These functions are called standard library functions. Their prototypes are located in various header files.

Examples of such functions are the reading and displaying functions which realises input-output operations (printf, scanf, etc.), routines for elementary mathematical functions evaluation (sqrt, sin, cos, atan, log, pow), etc.

The prototypes of standard functions are included in the program before their calls using the preprocessor directive **# include**.

Examples of files containing standard function prototypes:

stdio.h	(printf, scanf, gets, puts, etc.)
conio.h	(putch, getch, etc.)
math.h	(sqrt, sin, cos, etc.)

1.1. Declaring functions

There are two styles used when declaring functions:

a) The classic style – specifies only the name of the functions and the type of the return value:

type name_function( );

No information regarding the parameters are available, therefore no error checks are possible.

b) The modern style – data regarding the function parameters are also introduced.
This construction is named function prototype:

```
type name_function (inf_p, inf_p, etc.);
```

where "inf_p" may have the following forms: "type" or "type name".

Obs.: The b. method is preferable because the compiler is able to make checks regarding the number and the type of the parameters when calling functions.

1.2. Defining functions

a) The classic style define a function in the following way:

```
type name_function (name parameters)
defining_parameters
{
...
}
```

b) The modern style – parameter definitions appear in the first line:

```
type name_function (inf_p, inf_p, etc.)
{
...
}
```

where "inf_p" contains all the information regarding the parameters (type and name). The first line will therefore be identical to the function prototype, with only one exception: when defining the function, the character ";" doesn't appear.

Obs. The two styles presented above define evolution states of the C programming language.
The modern style is strongly recommended.

2. The format specifiers (overview and a few adds)

A format specifier starts with a percent character (%), which may be followed by characters such as:

- **The minus character "-" (optional)** determines the left bordering of the corresponding data. If it doesn't exist, the data are bordered implicitely to the right of the field where are written.

- **The string of decimal digits (optional)** defines the minimum dimension of the field in which the characters are displayed. If the data is shorter than the specified field, it will be written in that specific field to the right or to the left depending on the absence or presence of the minus.

- **A point followed by a number (optional)** defines the precision. If the data is float type, the precision defines the number of decimals. If the data is a character string, the precision indicates the number of characters which are written.

- **One or two letters** define the conversion type applied.

Obs. A format specifier starts with the character % and ends with a letter.
If we want to display a percent character (%), we write: %%

The first character is ignored and the second one will be displayed at the terminal.

In the following, we will indicate the main conversions performed using format specifiers:

%c => permits displaying a character.
%s => permits displaying a string of characters.

E.g.:

Calling:
```
printf ("*%10s*", "abc");
```
displays:
```
*       abc*
```

Calling:
```
printf ("*%-10s*","abc");
```
displays:
```
*abc       *
```

Calling:
```
printf ("*%10.5s*", "Program");
```
displays:
```
*     Progr*
```

%d => displays an integer (int).

E.g.:

Calling:
```
printf ("*%10d*", 123);
```
displays:
```
*       123*
```

Calling:
```
printf("*%-10d*", 123);
```
displays:
```
*123       *
```

Calling:
```
printf ("*%010d*", 123);
```
displays:
```
*0000000123*
```

%u => Displays an unsigned integer.

%o => the unsigned int data are converted and displayed in octal

E.g.:

Calling:
```
printf ("*%10o*", 123);
```
displays:

```
*       173*
```

%x or %X => the unsigned int data are converted and displayed in hexadecimal
E.g.:

Calling:

```
printf ("*%10x*", 123);
```

displays:

```
*       7b*
```

When the upper-case X is used, the hexa digits above 9 are displayed using upper-case letters.

Calling:

```
printf ("*%10x*", 123);
```

displays:

```
*       7B*
```

If the letter "l" precedes one of the letters d, o, x, X or u, conversions from the type long or unsigned long are made.

%f   => Displays a float or double value.

%e or %E   => Displays a float or double value with exponential format.

%g or %G   => Chooses between %f and %e (or E) so as the displayed expression to require a minimum number of characters

If the letter "L" precedes one of the letters f, e, E, g or G, the displayed data is of long double type.

%p   => Displays a pointer.

II. ASSIGNMENT WORKFLOW

1. Write a program based on the principles of procedural programming. The main function will call three functions:
   - the function "taking-over" which will read from the keyboard two variables x and y and will store them at the &x, &y addresses.
   - the function "computing" which will determine the result of the division x/y if y!=0.
   - the function "display" which will display the obtained result.

The program will be created using the modern style.

We remind two unary operators necessary for the "taking-over" function:

a) * => the indirection operator. The expression following it is a **pointer** (a variable that contains the address of another variable) and the result is a **lvalue** (an expression refering to an object).
E.g.: y = *px   /* y represents the content of the address pointed by px */

b) & => the operator that obtains a pointer. The operand is a lvalue and the result is a pointer.

```
E.g.:   px = &x ;
        y = *px ;      <=>     y = x
Obs.:   & is applied only for variables.
        Formats of this kind are wrong:        & (x+1)
                                               &3
```

A possible solution:

```
#include<stdio.h>

 /* Declaring the functions */

 void taking-over (float*p1,float*p2);
 float computing (float dividend, float divisor);
 void display (float result);

 const float INFINIT = 3.4e+38;

 /* The main function*/
void  main()
 {
    float x, y, res;
    taking-over (&x, &y);
    res=computing(x, y);
    display(res);
    getch( );
  }   /*end main*/

  /*Defining the functions*/

  void taking-over (float*p1, float*p2)
  {
     printf (" \nIntroduce the first number:  ");
     scanf ("%f", p1);
     printf ("\nIntroduce the second number:  ");
     scanf ("%f",p2);
   }

   float computing (float dividend, float divisor)
   {
      if (divisor==0.0)
          return (INFINIT);
      else
        return(dividend/divisor);
   }

   void display (float res)
   {
      if (res==INFINIT)
          printf ("\nUndefined result\n");
      else
          printf ("\nResult=  %f\n", res);
    }
```

2. Modify the program from above so as to eliminate the function "taking-over" without changing the result of the program.

3. Modify the program from above so as to eliminate the function "display" without changing the result of the program.

4. Modify the program from above so as to contain only the main function, without changing the result of the program.

5. Write a program where the main function calls another function named factorial, which determines and returns n! If n is in the interval [1, 170] and -1, otherwise. In the main program, the result will be stored in the variable fact and then displayed in the following form:

```
fact = factorial (x);
printf ("\nFactorial format(e):  %e\n    ", fact);
printf ("\nFactorial format(f):  %f\n    ", fact);
printf ("\nFactorial format(.0f):  %.0f\n", fact);
printf ("\nFactorial format(d):  %d\n    ", fact);
printf ("\nFactorial format(ld):  %ld\n  ", fact);
printf ("\nFactorial format(g):  %g\n    ", fact);
```

III. SOLUTIONS:

2. L5_2.C

```
#include<stdio.h>

 /* Declaring the functions */

 float computing (float dividend, float divisor);
 void display (float result);

 const float INFINIT = 3.4e+38;

 /* The main function*/
void  main( )
 {
    float x, y, res;
    printf (" \nIntroduce the first number:  ");
    scanf ("%f", &x);
    printf ("\nIntroduce the second number:  ");
    scanf ("%f", &y);
    res=computing(x, y);
    display(res);
    getch( );
  }   /*end main*/

  /* Defining the functions */

   float computing(float dividend, float divisor)
   {
      if (divisor==0.0)
        return(INFINIT);
      else
        return(dividend/divisor);
   }

   void display (float res)
   {
      if (res==INFINIT)
          printf ("\nUndefined result\n");
      else
          printf("\nDivision result =  %f\n", res);
    }
```

3. L5_3.C

```
#include<stdio.h>

 /* Declaring the functions */

 void display (float result);

 const float INFINIT = 3.4e+38;

 /* The main function*/
```

```
void  main()
 {
    float x, y, res;
    printf (" \nIntroduce the first number:  ");
    scanf ("%f", &x);
    printf ("\nIntroduce the second number:  ");
    scanf ("%f", &y);
    if (y==0.0)
        res=INFINIT;
    else
        res=x/y;
    display (res);
    getch( );
  }   /*end main*/

  /* Defining the functions */

   void display (float res)
   {
      if (res==INFINIT)
          printf ("\nUndefined result \n");
      else
          printf ("\nResult =  %f\n", res);
    }
```

4. L5_4.C

```
#include<stdio.h>
 const float INFINIT = 3.4e+38;
void  main( )
 {
    float x, y, res;
    printf (" \nIntroduce the first number:  ");
    scanf ("%f", &x);
    printf ("\nIntroduce the second number:  ");
    scanf ("%f",&y);
    if (y==0.0)
          printf ("\nInfinite result \n");
    else
      {  res=x/y;
         if (res==INFINIT)
             printf ("\nUndefined result \n");
          else
             printf ("\nDivision result =  %f\n", x/y);
       }
     getch( );
  }
```

5. L5_5.C

```
#include<stdio.h>

 /* Declaring the functions */
double factorial (int n);

 /* the main function*/
```

```
void  main( )
 {
    int x;
    double fact;
    printf ("\nIntroduce an integer number : ");
    scanf ("%d", &x);
    fact = factorial (x);
    printf ("\nFactorial format(e):  %e\n    ", fact);
    printf ("\nFactorial format(f):  %f\n    ", fact);
    printf ("\nFactorial format(.0f):  %.0f\n", fact);
    printf ("\nFactorial format(d):  %d\n    ", fact);
    printf ("\nFactorial format(ld):  %ld\n  ", fact);
    printf ("\nFactorial format(g):  %g\n    ", fact);
    getch( );
  }

double factorial (int n)
/* calculate and return n! for n in the interval [0, 170], otherwise returns -1 */
  {
    double f;
    int i;
    if ( n<0  ||  n>170)
       return (-1.0);
    for ( i=2, f=1;  i<=n; i++)
       f*=i;
    return (f);
   }
```