PROGRAMMING LANGUAGES

Laboratory 8

MATRICES

I. THEORETICAL BACKGROUND

1. Statements of matrix

A matrix, as any simple variable, must be declared before being used. Declaration of array, in the simplest form, contains the common type of its elements, the name of the array and upper borders for each index, included between square brackets:

type name [lim1][lim2]...[limn];

where:

type - is a keyword for predefined types

limi - is the upper border of the i-th index, this means that the i-th index can have the following values: 0, 1, 2, ..., lim i-1.

The borders limi (i=1, 2, ..., n) are constant expressions. By constant expression we understand an expression that can be evaluated at compiling, when it is found by the compiler.

The elements of an array can be referenced using indices.

In C language, the array name is a symbol which has a value that is the address of the first element.

At the declaration of an array, the compiler allocates a memory area required to preserve the values of its elements. The matrix name can be used in different expressions and its value is the beginning address of the memory area where it was allocated.

Examples:

a. int vect[10];

Statement defines a vect array of 10 elements of the type int. This matrix uses 10 * 2 = 20 bytes.

vect is a symbol whose value is the address of its first element, that is the address of vect[0]. So vect[0] has the value of the first element of the array, and vect has as value, the address of this element.

b. char tab[100]

tab is a one-dimensional char array, which has 100 elements. It just allocates 100 bytes and tab has as value, the address of tab[0].

c. double mat[10] [20]

mat is a two dimensional array of type double. It is a matrix of 10 lines and 20 columns fiecare. The compiler reserves for this matrix 10 * 20 * 8 = 1600 bytes.

The elements of this matrix are:

```
mat[0][0]  mat[0][1]  ...  mat[0][19]
mat[1][0]  mat[1][1]  ...  mat[1][19]
....................................

mat[9][0]  mat[9][1]  ...  mat[9][19]
```

mat has as value the address of the mat[0][0] element.

2. The sscanf and sprintf standard functions

The standard library of C and C + + languages contains the sscanf and sprintf functions, which are analogous to the scanf and printf functions.

Note: sscanf and sprintf functions have prototypes in the stdio.h file.

They have an extra parameter to call, which is the address of a memory area that can store characters of ASCII code. The other parameters are identical to those correspondents in scanf and printf.

The first parameter of these functions can be the name of an array of type char, because this kind of name has as value the beginning address of a memory area that its allocated.

The sprintf function is used, like the printf function also, to convert various kinds of data from theirs internal formats to external formats represented by sequences of characters. The difference is that, this time, those characters are not displayed on the standard terminal, but are stored in the memory area defined by the first parameter of sprintf function. They are kept as a string and can be subsequently displayed from that area with the puts function. Therefore, a call to printf function can always be replaced with a call to sprintf function, followed by a call of the function puts. Such a replacement is useful when you want to display the same data multiple times. In this case we call the function sprintf only once to make the necessary conversions from the internal format into external format, preserving conversion results in an array of char type. Further data can be displayed by calling puts function whenever posting is required.

The sprintf function, like printf function, returns the number of bytes for the string character resulting from the conversions.

Example:

```
int day,  month, year;
char data[20];
...
sprintf (data,"%02d/%02d/%d", day, month , year);
puts (data);
...
puts (data);
```

The function sscanf, like the scanf function, performs conversions from external format to internal format. The difference is that this time, characters are not read from the corresponding keyboard buffer, but they come from a memory area whose address is defined by the first parameter of the sscanf function. These characters may come in the respective area when one call the function gets. Thus, the call to scanf can be replaced by the call of gets followed by the call of sscanf function. Such replacements are useful when you want to eliminate typing errors occurring in data.

The sscanf function, like the scanf function, returns the number of fields converted correctly according to its format specifiers from the control parameter. When they found an error, both functions interrupt their execution and they return the number of fields correctly treated. By analyzing the return value it can be established if all the fields have been processed correctly or if an error is occurred.

In case of an error the data can be corrected. For this purpose it is necessary to eliminate the characters starting at the wrong position.

If you use the sequence:

```
gets
sscanf
```

the abandonment of the wrong characters is done automatically by recalling the gets function. When using the scanf function is necessary to advance to the newline character found in the buffer area of keyboard or to clean the area by special functions.

Example:

```
char tab[255];
int day, month, year;
...
gets (tab);
sscanf (tab, "%d %d %d", &day, &month , &year);
```

Note that the gets function returns the value NULL when is meet the end of file.

II. ASSIGNMENT WORKFLOW

1. Write a program (using the sscanf routine) that reads a positive integer of type long and then displays it. For typing of non-numerical characters it will display an error message and will require introducing an integer again.

A possible solution:

```
#include<stdio.h>
#include<stdlib.h>

void main( )
{
  long n, i;
  int j;
  char tab[255];
  do
    {  printf ("\n Type a positive integer: ");
       if (gets (tab)==NULL)
        { printf ("It was typed EOF\n");
          exit (1);
        }
       if (sscanf (tab,"%ld",&n)!=1  ||  n<=0 )
```

```
        /* without non-numerical characters and decimal values */
         { printf ("There is no positive integer \n");
           j=1;
         }
        else
           j=0;
    }
  while (j);
  printf ("%ld", n);
  getch( );
}
```

Note: When typing some non-numerical characters, if they are preceded by numeric characters, the non-numerical ones will be ignored.

2. Modify the previous program in order to replace the "else" instruction with a "continue" statement properly placed. Also, the last printf has to be substituted by a sprintf without affecting the initial development of the program.

3. Starting from the program written in section 2., fill it out so it can determine if the number introduced is prime and to display an appropriate message.

Tip: A simple algorithm is to compute successive division of the number n by

2, 3, 4, ..., k where k*k <= n.

If at least one of these numbers divides n, it follows that n is not prime.

4. Write a declared function (in a .c extension file):

void ordasc (double tab[], int n)

which sorts the n elements of the tab string, in ascending order.

Tip: The string is covered and if

tab[i] > tab[i+1]

their order will be reversed. Resume covering as many times until no longer makes any inversion.

5. Complete the above file with a program that reads a series of numbers and displays them in ascending order.

6. Modify the above program so that it will require a user password of one character. If there is not the correct key then force out the program.

III. SOLUTIONS

2. L8_2.C

```
#include<stdio.h>
#include<stdlib.h>

void main( )
{
  long n, i;
  int j;
  char tab[255];
  char data[20];
  do
    {  printf ("\n Type a positive integer:");
       if (gets (tab)==NULL)
        { printf ("It was typed EOF\n");
          exit(1);
        }
       if (sscanf (tab,"%ld", &n)!=1  ||  n<=0)
        { printf ("No positive integer \n ");
          j=1;
          continue;
        }
       j=0;
    }
  while(j);
  sprintf (data,"%ld", n);
  puts (data);
  getch( );
}
```

3. L8_3.C

```
#include<stdio.h>
#include<stdlib.h>

void main( )
{
  long n, i;
  int j;
  char tab[255];

  do
    {  printf ("\n Type a positive integer: ");
       if (gets (tab)==NULL)
        { printf ("It was typed EOF\n");
          exit(1);
        }
       if (sscanf (tab, "%ld", &n)!=1  ||  n<=0)
        { printf ("No positive integer \n ");
          j=1;
          continue;
        }
       j=0;
    }
  while(j);
  for (j=1, i=2; i*i<=n && j; i++)
     if (n%i==0)     /* number is not prime */
       j=0;
  printf("Number  is: %ld", n);
  if(j==0)
  printf (" not ");
  printf (" prime\n");
  getch( );
}
```

4. L8_4.C

```
void ordasc (double tab[], int n)
 /* Sort the elements of tab in ascending order */
 {
   int i, ind;
   double t;
   ind=1;
   while (ind)
     {
       ind=0;
       for (i=0; i<n-1; i++)
        if (tab[i] > tab[i+1])
         { t=tab[i];
           tab[i]=tab[i+1];
           tab[i+1]=t;
           ind=1;
         }   /*end if*/
     }  /*end while*/
 }
```

5. L8_5.C

```
#include <stdio.h>
#define MAX 1000

void ordasc (double tab[], int n);

void main( )
{
  double v[MAX];
  int m, s, i;
  printf ("\n Enter the number of elements =  ");
  scanf ("%d", &m);
  printf ("\n Enter the string elements:  \n");
  for (s=0; s<m; s++)
    scanf ("%lf", &v[s]);
    ordasc(v, m);
    for (i=0; i<m; i++)
    {  printf ("v[%d]=%g\n", i, v[i]);
       if( (i+1)%23==0)
         {  printf ("Press a key to continue \n");
                getch( );
         }
    }
    getch( );
}


 void ordasc(double tab[], int n)
 /* Sort the elements of tab in ascending order */
 {
   int i, ind;
   double t;
   ind=1;
   while (ind)
     {
       ind=0;
       for (i=0; i<n-1; i++)
        if (tab[i] > tab[i+1])
         { t=tab[i];
           tab[i]=tab[i+1];
           tab[i+1]=t;
           ind=1;
         }   /*end  if*/
     }  /*end while*/

 }
```

6. L8_6.C

```
#include <stdio.h>
#define MAX 1000

void ordasc (double tab[], int n);

void main( )
{
  double v[MAX];
  int m, s, i;
  char par;

  printf ("\nParola: ");
  scanf ("%c", &par);
  if (par != 's')
        exit(1);
  printf ("\n Enter the number of elements =  ");
  scanf ("%d", &m);
  printf ("\n Enter the string elements:  \n");
  for (s=0 ; s<m ; s++)
    scanf ("%lf", &v[s]);
    ordasc (v, m);
    for (i=0; i<m; i++)
    {  printf ("v[%d]=%g\n", i, v[i]);
       if( (i+1)%23==0)
         {  printf ("Press a key to continue \n ");
               getch( );
         }
    }
    getch( );
}


 void ordasc (double tab[], int n)
 /* Sort the elements of tab in ascending order */
 {
   int i, ind;
   double t;
   ind=1;
   while(ind)
     {
       ind=0;
       for (i=0; i<n-1; i++)
        if(tab[i] > tab[i+1])
         { t=tab[i];
           tab[i]=tab[i+1];
           tab[i+1]=t;
           ind=1;
         }   /*end if*/
     }  /*end while*/
 }
```