MODULAR PROGRAMMING

1. Overview

The learning objectives of this lab session is to:

- Understand the concepts of modules, modular programming, and variable scope.
- Learn how to develop applications using modules

2. Brief theory reminder

2.1. The notion of a module

A *source* module is a part of the source code of the program, which is compiled separately from the rest of the source code.

An *object* module is the result of the compilation of a source module.

A module contains related functions, as they are developed for solving a subproblem. Actually, the modules of a program correspond to the subproblems which result from a top-down design for solving a complex problem.

Modular programming is the programming style which is based on using modules. By using modular programming, a programmer can valuate the possibilities of hiding data and procedures to protect against unauthorized access form other modules. Thus, static data declared outside the functions of a module may be accessed by all those functions, but they are not accessible by functions of other modules.

It is strongly recommended to write modular solutions for complex programs, as this facilitates teamwork.

Modularization of a program also speeds up implementation and testing.

Tuned modules can be later used for solving other problems.

When solving complex problems, an executable can be obtained in the following ways:

Write a number of source files, each containing a module. Every module must be debugged and tuned separately. Using the construct:

#include ”file_specifier”

1. One can include the source code either in the module which contains the **main()** function**,** or in a file which contains only module inclusion directives, also including the module which contains **main().** This method actually produces a single source program, which will be compiled, linked and then executed. As the whole code is recompiled all the time, we do not advise for the use of this method.
2. Write separate source modules, and compile them separately, resulting in a number of object files (typically with the extension **.obj).** Link them and obtain the executable. Source modules can be separately compiled and then linked together by using projects in Dev-C++. This is our recommended method.

2.2. The scope of variables

2.2.1. Global variables

Global variables are defined at the beginning of a source file, i.e. before any function definition. These variables are visible from the point of definition to the end of the source file. If a program uses more than one source file, a global variable defined in one source file can be used in all the other source files if it is declared extern. Declaration of an extern variable can be done:

- After a function header, but then the scope of the variable is only that function.
- At the beginning of the source file, i.e. before the first function. Then its scope is the whole file (all functions in that file have access to it).

Note: It is recommended that the extern variables to be declared as **extern** in every function where they are used, thus avoiding the errors which can arise as a result of moving a function to a different module.

Global variables are allocated at compile time, in a special area.

2.2.2. Local variables

Variables declared in a function or in a composite statement are visible only within the unit where they were declared. Local variables can be:

a) **automatic** – variables which are allocated on the stack, at execution time. They are discarded upon return form a function, or at the end of the composite statement (block). The syntax for declaring them is the usual one (e.g. int a, b, c; double x; etc.);

b) **static** – variables which are allocated at compile time in a special area. Their declaration uses the keyword **static** before the type of the variable. Example:

static int x, y, z;

A static variable can be declared:

- At the beginning of a source file (i.e. before the first function declaration). In this case, the static variable is visible in the whole source file, but it cannot be declared extern (and thus made visible) in other files.
- In the body of a function, case in which its scope is only that function, or in a block (composite statement), and then its scope is that block.

register – variables which are allocated in the registers of the processor. Their type can be only integer, character, and pointer. Its is advised to declare as register variables frequently used variables in a function. The number of register variables is limited. If many were declared, the ones which cannot be allocated in registers, are allocated on the stack., as automatic variables. Modern compilers do their own optimization of register usage, making this declaration obsolete. A register variable declaration uses the keyword **register:**

register type variable;

Register variable allocation scope is the function where they were declared.

2.3. Modularized program example

The following program computes the reciprocal matrix and the determinant of a square matrix with elements of type **double.** The program was divided into modules as follows:

File L5Ex1_1.cpp – contains a function for reading the dimensions and elements of a matrix, and a matrix display function, i.e. functions performing I/O operations:

```
/* Read and display of a n*m element matrix with elements of type double */

#include <stdio.h>

#define MAXN 10

void display_matrix(int n, int m, double a[MAXN][MAXN], char ch)
{
    int i, j;
```

```
    printf("\n    MATRIX  %c\n",ch);
    for( i=0; i<n; i++)
    {
        for(j=0;j<m;j++) printf("%8.2lf ",a[i][j]);
        printf("\n");
     }
}

void read_matrix(int *n, int *m, double a[MAXN][MAXN])
{
    int i,j;
    printf("\nInput of size and elements of a matrix\n");
    printf("\n\tNumber of rows, n=");
    scanf("%d", n);
    printf("\n\tNumber of columns, m=");
    scanf("%d", m);
    printf("\n\tThe elements of the matrix\n");
    for (i=0;i<*n;i++)
        for(j=0; j<*m; j++)
        {
            printf("a[%d,%d]=", i, j);
            scanf("%lf",&a[i][j]);
        }
    printf("\n");
}
```

File L5Ex1_2.cpp – contains a multiplication function for two matrices:

```
  /*  Compute the product of matrices a[n][m] and b[m][p], of type double.
      The result is stored in matrix c[n[][p] */
   #define MAXN 10

void multiply_matrix(int n, int m, int p, double a[MAXN][MAXN],
                          double b[MAXN][MAXN], double c[MAXN][MAXN])
{
    int i, j, k;
    double s;
    for(i=0; i<n; i++)
        for(j=0; j<p; j++)
        {
            s=0.0;
            for(k=0; k<m; k++)
            s=s+a[i][k]*b[k][j];
            c[i][j]=s;
        }
}
```

File L5Ex1_3.cpp – contains a function used to compute the reciprocal matrix of a square matrix and the value of its determinant.

```
/* Computes the reciprocal matrix of square matrix a of size n and
    The value of its determinant, det_a, with precision eps */

#include <math.h>
#define MAXN 10
```

```
void  reciprocal_matrix(int n, double a[MAXN][MAXN], double eps,
            double b[MAXN][MAXN], double *det_a, int *err)
{
     int i, j, k, maxpos;
     double amax,aux;
     /* Initialize matrix b with the unit matrix */
     for(i=0; i<n; i++)
         for(j=0; j<n; j++)
             if(i==j) b[i][j]=1.0;
             else b[i][j]=0.0;
     /* Initialization of the value of the determinant */
     *det_a=1.0;
     /* Set to 0 all the elements under the main diagonal,
         and to 1 those on the diagonal */
     k=0; /* k=row number */
     *err=0;
     while( (k<n) && (*err==0) )
     {
         /* Calculate the pivot element */
         amax=fabs(a[k][k]);
         maxpos=k;
         for(i=k+1; i<n; i++)
             if (fabs(a[i][k]) > amax)
             {
                 amax=fabs(a[i][k]);
                 maxpos=i;
             }
     /* Interchange row k cu rwo maxpos in matrices a and b */
     if(k!=maxpos)
    {
         for(j=0; j<n; j++)
         {
             aux=a[k][j];
             a[k][j]=a[maxpos][j];
             a[maxpos][j]=aux;
             aux=b[k][j];
             b[k][j]=b[maxpos][j];
             b[maxpos][j]=aux;
         }
         *det_a=-*det_a;
    }
    if( fabs(a[k][k]) < eps) *err=1;
    else
    {
         *det_a =*det_a*a[k][k];
         aux=a[k][k];
         for(j=0; j<n; j++)
         {
             a[k][j]=a[k][j] / aux;
             b[k][j]=b[k][j] / aux;
          }
         for(i=0; i<n; i++)
             if(i != k)
             {
                 aux=a[i][k];
                 for(j=0; j<n; j++)
                 {
```

```
                    a[i][j]=a[i][j]-a[k][j]*aux;
                    b[i][j]=b[i][j]-b[k][j]*aux;
                }
            }
        }
        k++;
    }
}
```

File L5Ex1_4.cpp – contains the **main()** function.

```
/*  Program to compute the reciprocal matrix of a square matrix and Its determinant
value  */
#include <stdio.h>
#include <stdlib.h>

#define MAXN 10

extern void display_matrix(int n, int m, double a[MAXN][MAXN], char ch);
extern void read_matrix(int *n, int *m, double a[MAXN][MAXN]);
extern void multiply_matrix(int n, int m, int p, double a[MAXN][MAXN],
                            double b[MAXN][MAXN], double c[MAXN][MAXN]);
extern void  reciprocal_matrix(int n, double a[MAXN][MAXN], double eps,
              double b[MAXN][MAXN], double *det_a, int *err);

int main()
{
    int i, j, n, m, err;
    double eps, det_a, a[MAXN][MAXN], a1[MAXN][MAXN],
             b[MAXN][MAXN], c[MAXN][MAXN];

    system("cls");
    read_matrix(&n, &m, a);
    display_matrix(n, m, a, 'A');
    printf("Type Enter");
    getchar();
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)  a1[i][j]=a[i][j];
    eps=1.0e-6;
    reciprocal_matrix(n, a1, eps, b, &det_a, &err);
    if(err == 1) printf("\nMATRIx IS SINGULAR");
    else
    {
        printf("\nReciprocal matrix B=A^(-1)\n");
        display_matrix(n, n, b, 'B');
        printf("\nDeterminant of matrix A IS %8.4lf", det_a);
        multiply_matrix(n, n, n, a, b, c);
        printf("\nCHECK THAT C=A*B RESULTS IN THE UNIT MATRIX.");
        display_matrix(n, n, c, 'C');
        printf("Type Enter");
        getchar();
    }
    return 0;
}
```

In order to build the program, place all files in a directory, open a new project in Codeblocks, remove the predefined main and include all the above files in the project. Then build it.

3. Lab Tasks

3.1. Compile, execute and analyze the example program..

Write modular programs for solving the following problems:

3.2. Given an arithmetic expression in postfix form, with operands only integers, and only the operators **+**, **–**, *****, **/** write a program to evaluate such an expression.

3.3. Write programs to calculate the greatest common divisor and the smallest common multiple for two polynomials.

3.5. Implement sets and the operations on sets.

3.6. Implement a text editor which allows a number of operations on text (insert, delete/append characters or lines, cut, paste, etc.).

3.7. Read the number of points, ***n***, and their coordinates, (***x***, ***y***) in the plane ***x***0***y***, obtained as a result of experiments. Approximate the function ***y***=***f***(***x***) by a polynomial of degree ***m*** (***m*** is also read from the standard input), such a way that the sum of the squares of the approximation errors is minimal. Hint: use the least squares method (see Numerical Recipes…).

3.8. Read the number of towns in a region, ***n***, and their coordinated (***x***, ***y***) in the plane ***x***0***y*** . Find a way of connecting them by telephone lines of minimal length, provided that every pair of towns is connected either directly, or by intermediate(s).