RECURSION

1. Overview

The learning objective of this lab session is to:

- Understand the notion of recursion, the advantages and the shortcomings of recursive functions relative to the non-recursive ones, based on some simple examples.
- Learn how to proper develop recursive functions

2. Brief theory reminder

2.1. The recursion mechanism

An object is recursive if is defined by itself. A function is recursive if it is a self-calling function.

The recursion may be:

- direct – when in the context of a function it exist a self-call statement
- indirect – when in the context of a function it exist a call statement to another function, that calls the first function

At each function call or function self-call, the parameters and the "auto" variables are allocated on the stack, in an independent zone. These data have distinct values at each new self-call. The static and global variables are stored all the time in the same memory location. Consequently, any updates of their values are made only at their fixed address, so these variables conserve their values from a self-call to another.

Upon return from a called function either in the main program or in the calling or self-calling function, execution continues with the next statement relative to the calling or self-calling statement. This address is also stored in the stack. Upon return, the stack rolls back to the status it was before the call or the self-call, meaning that the automatic variables and the parameters will also have their previous values.

A very important thing is to stop the self-calling cycle. So, it is necessary to have an end condition, otherwise the recursive call leads to an infinite loop. In practice, it is necessary to have a finite depth of the recursion; moreover this depth must be small enough, because each recursive call requires allocations on the stack for:

- all the parameters of the recursive function
- all the automatic variables of the function
- the return address.

That means that the stack may rapidly grow, occupying in the end the entire memory area allocated to the stack.

A classic example of a recursive process is the computation of the factorial, defined as follows:

$$fact(n) = \begin{cases} 1 & if \quad n = 0 \\ n * fact(n-1) & if \quad n > 0 \end{cases}$$

Note that in the definition of the function ***fact***, there is a zone that is not self-definition: ***fact*(n)=1** if ***n*=0**. This is the initialization statement.

In C/C++, the corresponding program sequence is:

```
double fact(int n)
{
    if (n==0) return 1.0;
    return n*fact(n-1);
```

```
}
```

The linear recursion is characterized by the fact that on different branches of the program there may not appear more then one recursive call, so on each level we may have only one recursive call.

The linear recursion may be transformed in iteration, saving both memory and computation time, by eliminating the context-saving operations caused by each recursive call of the function.

The main advantage of the recursion resides in the possibility to write programs in a more compact end clear manner. The most often recursively designed computation processes are backtracking and divide and conquer principle based methods.

2.2. Examples

2.2.1. Reading and printing, mirrored, n words (character strings), each of them having a space at the end:

```
/* Program L7Ex1.c */

#include <stdio.h>
/* The program reads and prints, mirrored, n words, separated by spaces,
   having a space and a newline character after the last word */

void reverse(void)
{
    char c;
    scanf("%c", &c);
    if (c!=' ')
    {
        printf("%c", c);
        reverse();
    }
    printf("%c", c);
}

int main(void)
{
    int n, i;
    printf("\nThe number of the words=");
    scanf("%d", &n);
    for(i=1; i<=n; ++i)
    {
        reverse();
        printf("\n");
    }
    printf("\nEnd of the program.\n");
    return 0;
}
```

The function ***reverse*** reads and then prints the characters one by one, until a space is encountered. At each self-call, the local variable ***c*** is stored on the stack. When space is encountered, the cycle of the self-calls is stopped, and a space, followed by the characters written in reverse order are printed.

2.2.2. Finding the minimum of a string of *n* integers:

```
/* Program L7Ex2.c */

#include <stdio.h>
#include <limits.h>

/* The program computes the minimum of a array of integer numbers*/

#define NMAX 100
#define MAXIMUM INT_MAX
int array[NMAX];
int minim(int x, int y)
{
   if (x <= y) return x;
   else return y;
}

int term_min(int arraySize)
{
   if (arraySize  >= 0) return minim(array[arraySize], term_min(arraySize - 1));
   else return MAXIMUM;
}

int main(void)
{
   int i, n;
   printf("\nInput the number of the elements of the array n=");
   scanf("%d",&n);
   printf("\nInput the values of the elements\n");
   for (i=0; i<n; ++i)
   {
      printf("array[%d]=", i);
      scanf("%d", &array[i]);
   }
   printf("\nInput array:\n");
   for (i=0; i<n; ++i)
   {
      printf("%6d", array[i]);
      if ((i+1) % 10 == 0) printf("\n");
   }
   printf("\nThe minimum is %d\n", term_min(n-1));
   printf("\nPress a key!");
   return 0;
}
```

2.2.3. The recursive and non-recursive version of the n^{th} Fibonacci term computation.

Definition of Fibonacci string:

$$Fib(0)=0;\ \ Fib(1)=1;$$
$$Fib(n)=Fib(n-1)+Fib(n-2) \qquad \text{for } n \geq 2$$

```
/* Program L7Ex3.c */

#include <stdio.h>
#include <conio.h>
/* Fibonacci string*/
int fib1(int n)
```

```
/* RECURSIVE VERSION */
{
    if (n==0) return 0;
   else if (n==1) return 1;
       else return (fib1(n-1)+fib1(n-2));
}
int fib2(int n)
/* NON-RECURSIVE VERSION */
{
   int i, x, y, z;
   if (n==0) return 0;
   else if (n==1) return 1;
       else
       {
           x=1;y=0;
           for(i=2; i<=n; ++i)
           {
               z=x;
               x=x+y;
               y=z;
           }
           return x;
       }
}
int main(void)
{
   int n;
   char ch;
   ch='Y';
   while ((ch=='y')|| (ch=='Y'))
  {
       printf("\nInput n=");
       scanf("%d",&n);
       printf("\nRECURSIVE COMPUTATION:  fib(%d)=%d\n",n,fib1(n));
       printf("\nNON-RECURSIVE COMPUTATION: fib(%d)=%d\n",n,fib2(n));
       printf("\nContinue ? [Yes=Y/y] ");
       ch=getchar();
  }
   return 0;
}
```

The recursive call rapidly grows the stack, and thus the non-recursive implementation is preferable.

3. Lab Tasks

3.1. Analyze and execute the examples provided above. Trace the stack status for a concrete case; watch its size during the self-calling cycle, and how it decreases upon return from the recursive call.

3.2. Write a recursive and a non-recursive function to calculate the values of the Hermite polynomials H(x), defined as follows:

$$\begin{cases} H_0(x) = 1; \\ H_1(x) = 2x \\ H_n(x) = 2nH_{n-1}(x) - 2(n-1)H_{n-2}(x) \quad \text{for } n \geq 2. \end{cases}$$

3.3. The towers of Hanoi. Consider three vertical pegs ***A***, ***B***, ***C***, and ***n*** disks of different diameters. Initially, all the disks are placed on peg ***A***, in descending order of their diameters (with the biggest at the bottom and the smallest at the top). The problem is to move all the disks from peg ***A*** to peg ***C***, using peg ***B*** as intermediary, under the following conditions:

a) at each step you can move only one disk: the disk located at top of one of the three sticks;
b) it is forbidden to put a disk of a greater diameter on a disk having a smaller diameter;
c) in the end, the disks must all be on peg ***C*** in the same order they were on peg ***A*** at the beginning.

3.4. Write a recursive program that reads ***n*** words and displays them in reversed order.

3.5. Write a recursive program to generate the Cartesian product (product set, direct product, cross product) of ***n*** sets.

3.6. Write a recursive program to generate the subsets of ***k*** elements of a set ***A*** with a total number of ***n*** elements (i.e. combinations of ***k*** elements taken from a total of ***n*** elements).

3.7. Write a program to solve the problem of the eight queens: i.e. find their placement on the chess board so that they have do not have the possibility to attack each other.

3.8. Generate recursively the permutations of the set ***A*** having ***n*** elements.

3.9. You have a rod ***R*** of length ***m***, and ***n*** component rods of lengths $l_1, l_2, \ldots, l_n$ that must be cut from this rod. The aim is to get at least one component of each size, with minimum loss.

3.10. Ackermann's function is defined recursively as follows:

$$\begin{cases} Ack(0,n) = n+1 & \text{for } n \in N \\ Ack(m,0) = Ack(m-1,1) & \text{for } m \in N^* \\ Ack(m,n) = Ack(m-1, Ack(m,n-1)) & \text{for } m,n \in N^* \end{cases}$$

Write a recursive program to compute Ackermann's function:

3.12. Write a recursive program to generate the partitions of a natural number ***n***. Example: the partitions of the number 4 are: {1, 1, 1, 1}, {1, 1, 2}, {1, 2, 1},{1, 3}, {2, 1, 1}, {2, 2}, {3, 1}, {4}. Two partitions may differ either in the value of their elements, or in the order the elements are listed.

3.13. A labyrinth is coded using a ***m*** x ***n*** matrix. The passages are represented by elements equal to 1 placed on successive positions on a row or on a column. The requirement is to display all the traces that lead to the exit of the labyrinth, starting from an initial position (***i, j***). It is forbidden to pass more than once through the same point.

3.14. A knight is placed in the upper left square on a chess board of size ***n*** x ***n***. Display all the possible paths of the knight on the chess board, such a way that every square is reached only once.

3.15. Consider a ***n*** x ***m*** matrix, with elements decimal digits (natural numbers between 1 and 9), representing colors. A ***connected set*** associated to an element is the set of elements that may be reached from this element, by successive moves on a same row or column preserving the same color. It is to determine the size and the color of the biggest connected set. In case of multiple solutions, display them all.

3.16. There are ***n*** cubes, each of them characterized by its color and the length of the edge. Generate, with some of these cubes, a maximum height tower, under the following conditions:

a) a cube is placed on another only if its edge length is less than the edge length of the cube placed bellow it.

b) two neighboring cubes must have different colors.