CS1002

Programming Fundamentals

Lab 10

Recursion &

Structures

Instructors:

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

LAB 10

Learning Objectives

This lab will cover the following topics:

- Recursion
- Structures

Recursion

When a function invokes itself, the call is known as a recursive call. Recursion (the ability of a function to call itself) is an alternative control structure to repetition (looping). Rather than use a looping statement to execute a program segment, the program uses a selection statement to determine whether to repeat the code by calling the function again or to stop the process.

Flowchart for recursion:

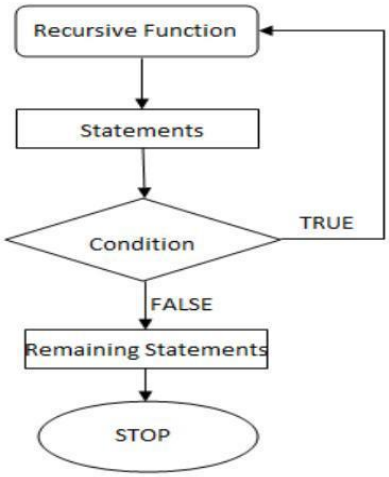


Fig: Flowchart showing recursion

Each recursive solution has at least two cases: the base case and the general case.

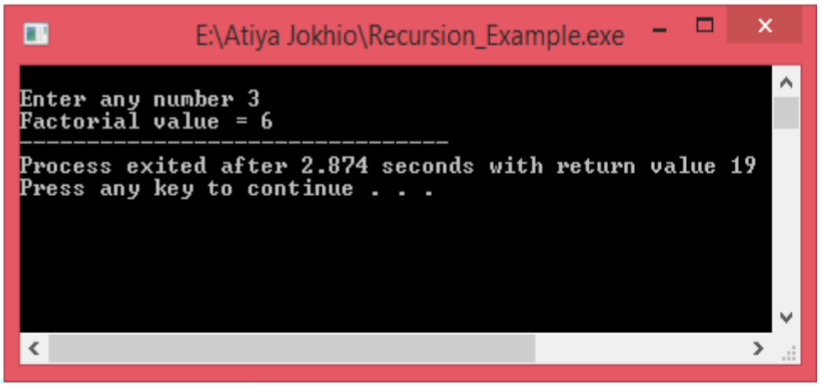
The **base case** is the one to which we have an answer; the **general case** expresses the solution in terms of a call to itself with a smaller version of the problem. Because the general case solves a smaller and smaller version of the original problem, eventually the program reaches the base case, where an answer is known, and the recursion stops.

For example, a classic recursive problem is the factorial. The factorial of a number is defined as the number times the product of all the numbers between itself and 0: N! = N * (N-1)! The factorial of 0 is 1. We have a base case, Factorial (0) is 1, and we have a general case, Factorial (N) is N * Factorial (N-1). An if statement can evaluate N to see if it is 0 (the base case) or greater than 0 (the general case). Because N is clearly getting smaller with each call,

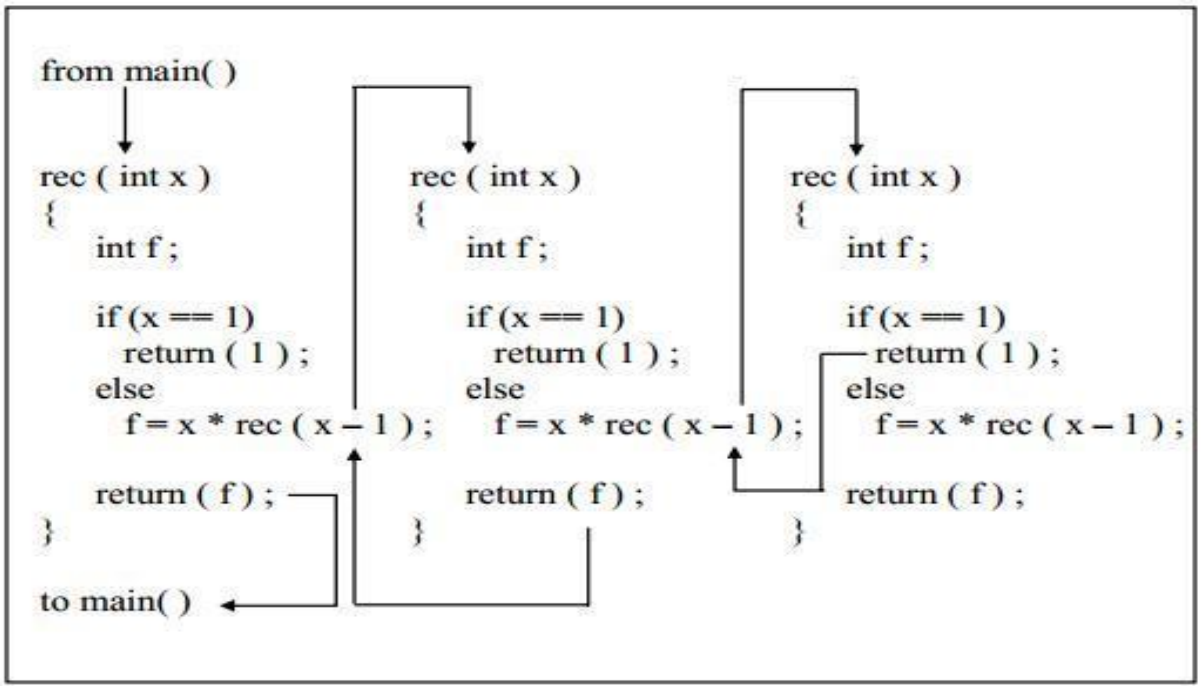
The base case is reached.

Following is the recursive version of the function to calculate the factorial value.

```
#include <stdio.h>
int main( )
{
    int a, fact ;
    printf ( "\nEnter any number " ) ;
    scanf ( "%d", &a ) ;
    fact = rec ( a ) ;
    printf ( "Factorial value = %d", fact ) ;
    }
    rec ( int x )
    {
    int f ;
    if ( x == 1 )
    return ( 1 ) ;
    else
    f = x * rec ( x - 1 ) ;
    return ( f ) ;
    }
```



Assume that the number entered through scanf( ) is 3. The figure below explains what exactly happens when the recursive function rec( ) gets called.



Disadvantages of recursion

- Recursive programs are generally slower than non-recursive programs. This is because,recursive function needs to store the previous function call addresses for the correct
  Program jump to take place.
- Requires more memory to hold intermediate states. It is because, recursive programrequires the allocation of a new stack frame and each state needs to be placed into the stack frame, unlike non-recursive (iterative) programs.

Structures:

We studied earlier that array is a data structure whose element are all of the same data type. Now we are going towards structure, which is a data structure whose individual elements can differ in type. Thus a single structure might contain integer elements, floating– point elements and character elements. Pointers, arrays and other structures can also be included as elements within a structure. The individual structure elements are referred to as members. This lesson is concerned with the use of structure within a 'c' program. We will see how structures are defined, and how their individual members are accessed and processed within a program. The relationship between structures and pointers, arrays and functions will also be examined. Closely associated with the structure is the union, which also contains multiple members.

In general terms, the composition of a structure may be defined as

```
struct tag
{  member  1;
member 2;
------------
-------------
member m; }
```

In this declaration, struct is a required key-word; tag is a name that identifies structures of this type. The individual members can be ordinary variables, pointers, arrays or other structures. The member names within a particular structure must be distinct from one another, though a member name can be same as the name of a variable defined outside of the structure.

A storage class, however, cannot be assigned to an individual member, and individual members cannot be initialized within a structure-type declaration. For example:

```
struct student
```

```
{
char name [80];
introll_no;
float marks;
};
```

We can now declare the structure variable s1 and s2 as follows:

```
struct student s1, s2;
```

s1 and s2 are structure type variables whose composition is identified by the tag student.

It is possible to combine the declaration of the structure composition with that of the structure variable as shown below.

```
storage- class struct tag
{
member 1;
member 2;
i.  ——
ii.  —-
iii. member m;
} variable 1, variable 2 --------- variable n;
```

The tag is optional in this situation.

```
struct student {
char name [80];
introll_no;
float marks;
} s1, s2;
```

The s1, s2, are structure variables of type student. Since the variable declarations are now combined with the declaration of the structure type, the tag need not be included. As a result, the above declaration can also be written as

```
struct {
char name [80];
introll_no;
float marks ;

} s1, s2;
```

A structure may be defined as a member of another structure. In such situations, the declaration of the embedded structure must appear before the declaration of the outer structure. The members of a structure variable can be assigned initial values in much the same manner as the elements of an array. The initial values must appear in the order in which they will be assigned to their corresponding structure members, enclosed in braces and separated by commas. The general form is

storage-class struct tag variable = { value1, value 2,-------,value m};

A structure variable, like an array can be initialized only if its storage class is either external or static. e.g. suppose there are one more structure other than student.

```
structdob
{
        Intmonth;
int day; int year;
};

struct student
{ char name [80];
introll_no; float marks;
structdob d1; };
```

struct student st = { “ali”, 2, 99.9, 17, 11, 01};

PROCESSING A STRUCTURE

The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by writing

variable.member name

This period (.) is an operator, it is a member of the highest precedence group, and its associativity is left-to-right.

E.g. if we want to print the detail of a member of a structure then we can write as printf(“%s”,st.name); or printf(“%d”, st.roll_no) and so on. Morecomplex expressions involving the repeated use of the period operator may also be written. For example, if a structure member is itself a structure, then a member of the embedded structure can be accessed by writing:

variable.member.submember

Thus in the case of student and dob structure, to access the month of date of birth of a student, we would write

st.d1.month

Structure Passing to a Function

```c
#include <stdio.h>
struct student {
   char name[50];
   int age;
};

// function prototype
void display(struct student s);

int main() {
   struct student s1;

   printf("Enter name: ");

   // read string input from the user until \n is entered
   // \n is discarded
   scanf("%s", s1.name);

   printf("Enter age: ");
   scanf("%d", &s1.age);

   display(s1); // passing struct as an argument

   return 0;
}

void display(struct student s) {
   printf("\nDisplaying information\n");
   printf("Name: %s", s.name);
   printf("\nAge: %d", s.age);
}
```

Return struct from a function

```c
#include <stdio.h>
struct student
{
    char name[50];
    int age;
};

// function prototype
struct student getInformation();

int main()
{
    struct student s;

    s = getInformation();

    printf("\nDisplaying information\n");
    printf("Name: %s", s.name);
    printf("\nRoll: %d", s.age);

    return 0;
}
struct student getInformation()
{
  struct student s1;

  printf("Enter name: ");
  scanf ("%s", s1.name);

  printf("Enter age: ");
  scanf("%d", &s1.age);

  return s1;
}
```

Arrays of Structure

```c
#include<stdio.h>
struct student{
    char name[50];
};

main(){
    struct student stu[2];
    printf("\n Enter the name of the student 1");
    scanf("%s", &stu[0].name);
    printf("\n Enter the name of the student 2");
    scanf("%s",&stu[1].name);
    printf("\n the name of the student 1 is %s", stu[0].name );
    printf("\n the name of the student 2 is %s", stu[1].name );
}
```

Question 1:

Write a recursive function that checks whether a number is a palindrome.

Question 2: *write a program to find the lcm of two numbers using recursion.*

Question 3:

A phone number, such as (212) 767-8900, can be thought of as having three parts: e.g., the area code (212), the exchange (767), and the number (8900). Write a program that uses a structure to store these three parts of a phone number separately. Call the structure phone.

Create two structure variables of type phone. Initialize one, and have the user input a number for the other one. Then display both numbers.

The interchange might look like this:
Enter area code: 415
Enter exchange: 555
Enter number: 1212
Then display like below:
My number is (212) 767-8900
Your number is (415) 555-1212

Question 04:

Write a C Program to Store Information of N Students Using Structure, where N is provided by the user.
Student information should contain Student_id, stu_age, stu_name; street no; state; city; country;

Question 05:

Write a C program that uses functions to perform the following operations:
i) Reading a complex number
ii) Writing a complex number
iii) Addition of two complex numbers
iv) Multiplication of two complex numbers
(Note: represent complex numbers using a structure.)