BIRZEIT UNIVERSITY

Comp.133

Programming with C-Language

LABORATORY WORK BOOK

Compiled and Prepared by:

Mr. Iyad Jaber

Mr. Wahbeh Musa

Dr. Yousef Hasouneh

Miss. Maram Khatib

Approved By:

Computer Science Department

2014

Acknowledgement

The material included in this manual has been entirely adopted from the following references:

1. Laboratory work book, Department of Electronic Engineering, N.E.D. University of Engineering & Technology, Karachi – Pakistan.
2. Problem Solving and program Design in C, by Jeri R. Hanly and Elliot B. Koffman, fourth Edition.
3. Laboratory Manual CSCI 2170, Dept. of Computer Science, Middle Tennessee State University, by Thomas Cheatham, Judith Hankins and Brenda Parker, 2003 Edition.

Introduction

This work book is especially designed to help the students to generate their own logic for the accomplishment of the assigned tasks. Every lab is provided with the syntax of the statements or commands which will be used to make the programs for exercises. In order to facilitate for the students some program segments are also provided explaining the use of commands. For a wide scope of usage of the commands several examples are also given so that the students can understand how to use the commands. The conditions, limitations and memory allocation are also mentioned where necessary.

This work book starts the programming of C Language from scratch and covers most of the programming structures of C-Language. For some commands or structures more than one lab is designed so that the students can thoroughly understand their use.

Contents

Lab No.01

Objective

Design computer algorithms

Theory

An algorithm (pronounced AL-go-rith-um) is a procedure or formula for solving a problem. The word derives from the name of the mathematician, Mohammed ibn-Musa al-Khwarizmi, who was part of the royal court in Baghdad and who lived from about 780 to 850. Al-Khwarizmi's work is the likely source for the word algebra as well.

To make a computer do anything, you have to write a computer program. To write a computer program, you have to tell the computer, step by step, exactly what you want it to do. The computer then "executes" the program, following each step mechanically, to accomplish the end goal.

When you are telling the computer what to do, you also get to choose how it's going to do it. That's where computer algorithms come in. The algorithm is the basic technique used to get the job done. Let's follow an example to help get an understanding of the algorithm concept.

Let's say that you have a friend arriving at the airport, and your friend needs to get from the airport to your house. Here are four different algorithms that you might give your friend for getting to your home:

The taxi algorithm:

1. Go to the taxi stand.
2. Get in a taxi.
3. Give the driver my address.

The call-me algorithm:

1. When your plane arrives, call my cell phone.
2. Meet me outside baggage claim.

The rent-a-car algorithm:

1. Take the shuttle to the rental car place.
2. Rent a car.
3. Follow the directions to get to my house.

The bus algorithm:

1. Outside baggage claim, catch bus number 70.
2. Transfer to bus 14 on Rukab Street.
3. Get off on Jerusalem street.
4. Walk two blocks north to my house

All four of these algorithms accomplish exactly the same goal, but each algorithm does it in completely different way. Each algorithm also has a different cost and a different travel time. Taking a taxi, for example, is probably the fastest way, but also the most expensive. Taking the bus is definitely less expensive, but a whole lot slower. You choose the algorithm based on the circumstances.

Pseudocode

Pseudocode is a kind of structured English for describing algorithms. It allows the designer to focus on the logic of the algorithm without being distracted by details of language syntax. At the same time, the pseudocode needs to be complete. It describes the entire logic of the algorithm so that implementation becomes a rote mechanical task of translating line by line into source code.

Example:
Calculate the class average for 10 students:

Pseudocode:

```
Set total to zero

Set counter to one

While counter is less than or equal to ten

        Input the next grade

        Add the grade into the total

ENDWhile

Set the average to the total divided by ten

Print the class average
```

EXERCISE

1. Write an algorithm (pseudo code) that asks the user to enter any three different integer numbers (n1, n2, and n3) and then will display the value of the number (n1, n2, or n3) that is the middle value. (e.g. if the numbers entered were n1= 10, n2 = 30, and n3 = 18 then your algorithm should print the value of n3 (i.e. 18) to the screen).

2. Write an algorithm that asks the user to enter an integer number of any size and then counts and prints the number of digits in that number.

3. Write an algorithm that asks the user to enter an integer number of any size and then finds and displays which is larger the sum of the even digits in the number or the sum of the odd digits.

4. Write an algorithm that keeps asking the user to enter one letter (a-z or A-Z) at a time and counts and displays the number of vowels and consonants entered. Your algorithm should stop when the letter x or X are entered by the user.

5. Write an algorithm to decide and print whether a given number is perfect or not. A perfect number is that which equals the sum of all its divisors excluding itself (e.g. 6 is a perfect number since it equals 1+2+3).

6. Modify the algorithm in question 5 such that it will print the first three positive perfect numbers.

Lab No.02

Objective

C Building Blocks

Theory

In any language there are certain building blocks:

• Constants

• Variables

• Operators

• Methods to get input from user (scanf(), getch() etc.)

• Methods to display output (Format Specifiers, Escape Sequences etc.) etc.

Format Specifiers

Format Specifiers tell the **printf** statement where to put the text and how to display the text.

The various format specifiers are:

%d => integer

%c => character

%f => float

…

Variables and Constants

If the value of an item is to be changed in the program then it is a variable. If it will not change then that item is a constant. The various variable types (also called data type) in C are: **int, float, char, long ,double, …** they are also of the type **signed or unsigned**.

Escape Sequences

Escape Sequence causes the program to escape from the normal interpretation of a string, so that the next character is recognized as having a special meaning. The back slash "\" character is called the **Escape Character**". The escape sequence includes the following:

\n => new line

\t => tab

\b => back space

\r => carriage return

\" => double quotations

\\ => back slash etc.

Taking Input from the User

The input from the user can be taken by the following techniques: **scanf( ), getch( ),**

Operators

There are various types of operators that may be placed in three categories:

Basic: + ,-,*, /, %

Assignment: =, += ,-= ,*=, /=, %=

Relational: <, > ,<=, >=, == ,!=

Logical: && , || , !

EXERCISE

1. Write a program which shows the function of each escape sequence character.

   eg:

   printf("the new line is inserted like \n this");

   printf("the tab is inserted like \t this");

2. Write down C statements to perform the following operations:

   i. $x = a^2+2ab+b^2$

   ii. $$Z = \frac{4.2(x+y)/z-0.25x/(y+z)}{(x+y)^2}$$

3. What will be the output of the mix mode use of integers and float.

```
int a,b;
double k,m;
a=9/6;
b=9/6.0;
k=9/6;
m=9/6.0;
printf(" a=%d \n b=%d \n k=%f \n m= %f",a,b,k,m);
```

4. What will be the output of the following statements

```
double  n=12.4587;
printf("%4.2f",n);
```

5. find the sum of three-digit number

   for example : if user insert 387 the program should calculate 3+8+7 and print 18

6. print a reverse of three-digit number

   for example : if user insert 387 the program should print 783

Lab No.03

Objective

Functions in C-Language programming

Theory

Functions are used normally in those programs where some specific work is required to be done repeatedly and looping fails to do the same.

Three things are necessary while using a function.

i. Declaring a function or prototype:

The general structure of a function declaration is as follows:

```
return_type function_name(arguments);
```

Before defining a function, it is required to declare the function i.e. to specify the function prototype. A function declaration is followed by a semicolon ';'. Unlike the function definition only data type need to be mentioned for arguments in the function declaration.

ii. Calling a function:

The function call is made as follows:

```
variable = function_name(arguments);
```

ii. Defining a function:

All the statements or the operations to be performed by a function are given in the function definition which is normally given at the end of the program outside the main.

Function is defined as follows

```
return_type function_name(arguments)
{
    Statements;
}
```

There are certain functions that you have already used e.g:

getche( ), clrscr( ), printf( ), scanf( ) etc.

There are four types of functions depending on the return type and arguments:

- Functions that take nothing as arguments and return nothing.
- Functions that take arguments but return nothing.
- Functions that do not take arguments but return something.
- Functions that take arguments and return something.

A function that returns nothing must have the return type "void**". If nothing is specified then the return type is considered as "int".**

EXERCISE

1. Write a program which takes two integers from the user and prints their sum. Use a function to find the sum.

2. Write a program which takes two points from a file and prints the distance between them to another file. Use a function to find the distance.

$$d = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

Note: use the math library to find the square root -> sqrt(double)

3. Write a program to find

a. Surface area (**A= 2 r π**)

b. volume(**v=r² π h**)

of a cylinder using functions .One for the area and the other for the volume. The program should take the height and the radius from the user then call functions to calculate the values and print the results in the main scope

Note: use math library to find **r²** -> pow(double, double)

4. Write a program which takes a decimal number with fraction and rounds it to two decimal places

eg: 25.287 would round to 25.29

Note : round the number and print the result in the same function

5. Write a program to take a length of a square (in inches) as input data; compute and display the area of the square in cm and inches. The relevant formulas are

Area = length * length

Cm =2.54 * inch

Include two functions in your program. Function **area** should compute and return the area of square measured in inches . Function **inch_cm** should convert the area to cm.

Lab No.04

Objective

Decision making the if and if-else structure, the Switch case and conditional operator.

Theory

Normally, your program flows along line by line in the order in which it appears in your source code. But, it is sometimes required to execute a particular portion of code only if certain condition is true; or false i.e. you have to make decision in your program. There are three major decision making structures. **The 'if' statement, the if-else statement, and the switch statement.** Another less commonly used structure is the **conditional operator**.

The if statement

The if statement enables you to test for a condition (such as whether two variables are equal) and branch to different parts of your code, depending on the result or the conditions with relational and logical operators are also included..

The simplest form of an if statement is:

```
if (expression)
        statement;
```

The if-else statement

Often your program will want to take one branch if your condition is true, another if it is false. If only one statement is to be followed by the if or else condition then there is no need of parenthesis. The keyword else can be used to perform this functionality:

```
if (expression)
{
        statement/s;
}
else
```

```
{
        statement/s;
}
```

The switch Statement

Unlike if, which evaluates one value, switch statements allow you to branch on any of a number of different values. There must be break at the end of the statements of each case otherwise all the preceding cases will be executed including the default condition. The general

form of the switch statement is:

```
switch (identifier variable)
{
case  identifier One :     statement; . . .
                                    break;
case identifier Two   :    statement; . . .
                                    break;
....
case identifier N       :  statement; . . .
                                    break;
default              :     statement; . . .
}
```

Conditional Operator

The conditional operator **(? :)** is C's only ternary operator; that is, it is the only operator to take three terms.

The conditional operator takes three expressions and returns a value:

(expression1) ? (expression2) : (expression3)

It replaces the following statements of if else structure

```
If(a>b)
        c=a;
```

else

```
c=b;
```

can be replaced by

```
c=(a>b)? a : b
```

This line is read as "If expression1 is true, return the value of expression2; otherwise, return the value of expression3." Typically, this value would be assigned to a variable.

EXERCISE

1. Write a program that inputs a grade of a student and determines if it is a pass (60 or more) or a fail.

2. Write a program that inputs an integer and determines if it is divisible by 3 or not.

3. Write a program which takes a character as input and checks whether it is a consonant or a vowel.

4. Write a program which takes three sides a, b and c of a triangle as input and calculates its area if these conditions are satisfied a+b>c, b+c>a and a+c>b

   (Help area = $\sqrt{s(s-a)(s-b)(s-c)}$, where s=(a+b+c)/2) use a function to calculate the area

5. Write a program to make a simple calculator which should be able to do +,-,*,/,% operations. Use switch statement.

6. Write a program which takes **5** integers as input and prints the smallest one.

7. What will be the output of the following statements supposing a = 5. try each statement separately.

```
printf("%d",++a);
printf("%d",a++);
```

Lab No.05

Objective

Looping constructs in C-Language, and Nested looping

Theory

Types of Loops

There are three types of Loops:

1) for Loop

    i. simple for loop

    ii. nested for loop

2) while Loop

    i. simple while loop

    ii. nested while loop

3) do - while Loop

    i. simple do while loop

    ii. nested do while loop

Nesting may extend these loops.

The for Loop

```
for (initialize(optional) ; condition(compulsory) ; increment(optional))
{
    Body of the Loop;
}
```

This loop runs as long as the condition in the center is true. Note that there is no semicolon after the "for" statement. If there is only one statement in the "for" loop then the braces may be removed. If we put a semicolon after the for loop instruction then that loop will not work for any statements.

The Nested for Loop

```
for (initialize ; condition ; increment)
{
        Body of the loop;
        for (initialize ; condition ; increment)
        {
                Body of the loop;
        }
        Body of the loop;
}
```

The inner loop runs as many times as there is the limit of the condition of the external loop. This loop runs as long as the condition in the parenthesis is true. We can nest many loops inside as needed.

The while Loop

```
while ( condition is true )
{
        Body of the Loop;
}
```

This loop runs as long as the condition in the parenthesis is true. Note that there is no semicolon after the "while" statement. If there is only one statement in the "while" loop then the braces may be removed.

The nested while Loop

```
while (condition is true)
{
```

```
        while (condition is true)
        {
                Body of the loop;
        }
        Body of the loop;
}
```

The inner loop runs as many times as there is the limit of the condition of the external loop. This loop runs as long as the condition in the parenthesis is true. We can nest many loops inside as needed.

The do-while Loop

```
do
{
        Body of the Loop;
} while ( condition is true);
```

This loop runs as long as the condition in the parenthesis is true. Note that there is a semicolon after the “while” statement. The difference between the “while” and the “do-while” statements is that in the “while” loop the test condition is evaluated before the loop is executed, while in the “do” loop the test condition is evaluated after the loop is executed. This implies that statements in a “do” loop are executed at least once. However, the statements in the “while” loop are not executed if the condition is not satisfied.

The Nested do-while Loop

```
do
{
        do
        {
                body of the loop;
        } while (condition is true);
        body of the loop;
} while (condition is true);
```

This loop runs as long as the condition in the parenthesis is true. Note that there is a semicolon after the “while” statement. The difference between the “while” and the “do-while” statements is that in the “while” loop the test condition is evaluated before the loop is executed, while in the “do” loop the test condition is evaluated after the loop is executed. This implies that statements in a “do” loop are executed at least once. The inner loop runs as many times as there is the limit of the condition of the external loop. This loop runs as long as the condition in the parenthesis is true. We can nest many loops inside as needed.

EXERCISE

1. Write down the output of the following program statement

```
for (i=1; i<=10;i++)
        printf(“%d \n”,i);
```

2. Write a program that prints the first positive 50 numbers that are divisible by 4.

3. Write a program which will read pairs of integers from the user in a loop. Each time it reads a new pair, it checks if the first integer divides the second (second integer % first integer = 0). When the program reads such a pair it will exit the loop.

4. Write down the output of the following program statements

```
for (int a=1,j=1; j<=5;j++)
        for (i=1; i<=5;i++)
        {   printf(“%d\n”,a);
                a++;
        }
```

5. Write a program which takes an integer as input and checks whether it's prime or not.

6. Write a program to print a series of the first positive 50 prime numbers.

7. Write a program which prints the following pattern up to 10 lines

```
0
111
22222
3333333
```

8. Write a program to print the following pattern up to z only

```
a  b  c  d
e  f  g  h
i  j  k  l
m  n  o  p
q  r  s  t
u  v  w  x
y  z
```

Lab No.06

Objective

Modular Programming and pointers

Theory

In Lab No. 04 you learned how to write the separate functions of a program. The functions correspond to the individual steps in a problem solution. You also learned how to provide inputs to a function and how to return a single output. In This lab you complete your study of functions, learning how to connect the functions to create a program system- an arrangement of parts that makes your program operate like a stereo system as it passes information from one function to another.

1. Parameters enable a programmer to pass data to functions and to return multiple results from functions. The parameter list provides a highly visible communication path between a function and its calling program. Using parameters enables a function to process different data each time it executes thereby making it easier to reuse the function in other programs.

2. Parameters may be used for input to a function, for output or sending back results, and for both input and output. An input parameter is used only for passing data into a function. The parameter's declared type is the same as the type of the data. Output and input/output parameters must be able to access variables in the calling function's data area so they are declared as pointers to the result data TYPES. The actual argument corresponding to an input parameter may be an expression or a constant; the actual argument corresponding to an output or input/output parameter must be the address of a variable.

3. The scope of an identifier dictates where it can be referenced. A parameter or local variable can be referenced anywhere in the function that declares it. A function name is visible from its declaration (the function prototype) to the end of the source file except within functions that have local variables of the same name.

EXERCISE

1. Determine the output of the following program:

   Note : %p used to print the address in hexadecimal

```
int  main()
{
        int q=2;
        int *p;
        p=&q;
        *p=100;
         printf("%d\n",q);
         printf("%p\n",p);
         printf("%d\n",*p);
        printf("%p\n",&q);
        printf("%p\n",&p);
        return 0;
}
```

2. Determine the output of the following program:

```
int main()
{
    int x=3,y=4,z=6;
    int *p1,*p2,*p3;
    p1=&x;
    p2=&y;
    p3=&z;
    *p1=*p2+*p3;
```

```
    (*p1)++;

    (*p2)--;

    *p1=(*p2)*(*p3);

    *p2=(*p2)*(*p1);

    x=y+z;

    printf("%d\n",x);

    printf("%d\n",y);

    printf("%d\n",z);

    printf("%d\n",*p1);

    printf("%d\n",*p2);

    printf("%d\n",*p3);

    return 0;

}
```

3. Write a program which takes two integers and print their sum, subtraction, multiplication, and division. Use a function with four output parameters.

4. Write a program to dispense change. The user enters the amount paid and the amount due. The program determines number of bills and coins you need to return from the following categories: 200, 100, 50 and 20 bills and 10, 5, 2 and 1 coins. Write a function with eight output parameters that determines the quantity of each kind of coins and bills.

Lab No.07

Objective

Arrays in C (one dimensional)

Theory

An array is a collection of data storage locations, each of which holds the same type of data. Each storage location is called an element of the array. You declare an array by writing the type, followed by the array name and the subscript. The subscript is the number of elements in the array, surrounded by square brackets. For example,

```
long LongArray[25];
```

declares an array of 25 long integers, named LongArray. When the compiler sees this declaration, it sets aside enough memory to hold all 25 elements. Because each long integer requires 4 bytes, this declaration sets aside 100 contiguous bytes of memory

EXERCISE

1. Write a program that declares two arrays of integers. Fill the first one from the user and the second is static, and exchanges their values.

2. Write a program that takes 10 integers as input and prints the smallest integer and its location in the array.

3. Write a program which takes a string as input and then counts the total number of vowels in that string.

4. Write a function which take two arrays and return an array contains the sum of them.

5. Write a program to pass an integer array of 10 elements to a function which returns the same array after sorting it in descending order. Print the array.

6. The electric company charges according to the following rate schedule:

   9 cents per kilowatt-hour (kwh) for the first 300 kwh

   8 cents per kwh for the next 300 kwh (up to 600 kwh)

   6 cents per kwh for the next 400 kwh (up to 1,000 kwh)

   5 cents per kwh for all electricity used over 1,000 kwh

   Write a function to compute the total charge for each customer. Write a main function to call the charge calculation function using the following data:

Customer Number	Kilowatt-hours Used
123	725
205	115
464	600
596	327
…	…

   The program should print a three column chart listing the customer number, the kilowatt hours used, and the charge for each customer. The program should also compute and print the number of customers, the total kilowatt hours used, and the total charges.

Lab No.08

Objective

Arrays in C (Multidimensional) and string functions.

Theory

A Multidimensional array is a collection of data storage locations, each of which holds the same type of data. Each storage location is called an element of the array. You declare an array by writing the type, followed by the array name and the subscript. The subscript is the number of elements in the array, surrounded by square brackets. For example,

```
int a[5][10]
```

declares an array of 50 integers, named a. Its declaration shows that array a comprises of 5 one dimensional arrays and each one dimensional array contains 10 elements. When the compiler sees this declaration, it sets aside enough memory to hold all 50 elements. Because each integer requires 2 bytes, this declaration sets aside 100 contiguous bytes of memory.

EXERCISE

1. Write a program that adds up two 4x4 arrays and stores the sum in third array.

2. Write a program which takes names of five countries as input and prints them in alphabetical order.

3. Write and test a function hydroxide that returns a 1 for true if its string argument ends in the substring OH.

   Try the function hydroxide on the following data:

   KOH H2O2 NaCL NaOH C9H8O4 MgOH

4. Write a program that takes data one line at a time and reverses the words of the line. For example,

   Input : birds and bees

   Output : bees and birds

Lab No. 09

Objective:

To become familiar with the concept of recursion

To learn basic guidelines in writing recursive functions

To learn how recursion is implemented and how recursion can be traced.

To compare recursion and iteration.

Theory:

What is recursion:-

A function is said to be recursive if the function calls itself. Therefore, **recursion** is a problem solving technique which allows a function to call itself and this technique can be applied when the solution to a problem can be stated in terms of itself. In this lab we will examine some problems which can be stated in terms of themselves and show how these functions can be written recursively.

The classic example of a problem where the solution to the problem can be stated in terms of itself is the calculation of n factorial for n >= 1. By definition:

```
n! = 1 * 2 * 3 * .. * n
```

We easily see that n! = n * (n - 1)!

Notice that to calculate n factorial recursively, we calculate n - 1 factorial and multiply by n. To calculate n-1 factorial we calculate n - 2 factorial and multiply by n - 1, etc. For example,

```
4! = 4 * 3!
   = 4 * 3 * 2!
   = 4 * 3 * 2 * 1!
   = 4 * 3 * 2 * 1
```

To calculate factorial we need a definition. The following is a recursive definition of factorial:

```
n! = 1,             if n = 0
n! = n * (n - 1)!,  if n > 0
```

A recursive definition consists of a *base case* step that defines the beginning elements in the set and a *recursive step* that expresses the relationship between elements in the set.

Here is a recursive C function to calculate n! Notice how closely it follows the recursive definitions of factorial as shown above.

```
// Recursive factorial function.  Assume n >= 0.
int fact(int n)
{
   if (n == 0)
      return (1);
   else
      return (n * fact(n - 1));
}
```

The value of fact(4) is calculated as follows:

```
fact(4) = 4 * fact(3)
   fact(3) = 3 * fact(2)
      fact(2) = 2 * fact(1)
         fact(1) = 1 * fact(0)
            fact(0)  is 1 (base case)
```

When the computer reaches the base case fact(0), there are three suspended computations. After calculating the value returned for the stopping case, it resumes the most recently suspended computations to determine the value of fact(1). After that, the computer completes each of the suspended computations, using each value computed as a value to plug into another suspended computation, until it reaches and completes the computation for the original call fact(4). The details of the final computation are illustrated below:

Since fact(0) = 1, then fact(1) = 1 * 1 = 1
Since fact(1) = 1, then fact(2) = 2 * 1 = 2
Since fact(2) = 2, then fact(3) = 3 * 2 = 6
Since fact(3) = 6, then fact(4) = 4 * 6 = 24

Recursion Guidelines:

Writing recursive functions is usually not an easy accomplishment for the beginning programmer. The most common mistake that beginning programmers make is writing recursive functions which continue to make recursive calls which means the program will never terminate. Therefore, caution needs to be taken to assure that the function will eventually stop calling itself. Let's observe some simple guidelines to aid in writing recursive functions.

1. A recursive function must call itself; i.e. the function ***fact*** calls the function ***fact***.
2. At each successive call to the recursive function, the "size" of the problem is diminished. The next call to **fact** solves a problem that is identical in nature to the original problem but smaller in size.

3. There is one instance of the problem that is treated differently from all of the others. This special case is called the **base case** or the **stopping condition** and the solution to the problem is known in that particular case. The base case allows recursion to stop.
4. The manner in which the size of the problem is reduced must cause the base case to be reached.

Another classic example on recursion is the Fibonacci sequence which was defined by a mathematician named Fibonacci. He used it to explain how rabbits multiply (which is quite rapidly). Suppose we agree that a rabbit takes one month to grow to maturity. A mature pair of rabbits produces a pair of rabbits each month (one male, one female). How many rabbits exist at the beginning of month m, if we begin with a pair of baby rabbits(one male, one female)?

Beginning of Month	Number of Pairs	Explanation
1	1	pair of baby rabbits
2	1	pair of grown rabbits
3	2	original grown; new baby pair
4	3	original grown; new grown; new pair of babies of original
5	5	original grown; another grown; new grown; 2 baby pairs
6	8	5 grown from previous plus 3 baby pairs from 3 grown
7	13	…….
8	21	…….
9	34	…….

There is a pattern here. The number of rabbits at the beginning of month m is the sum of two previous months (since we have all those grown the last month plus babies from all those from the previous month). These numbers **1, 1, 2, 3, 5, 8,...** form the Fibonacci sequence. They appear frequently in nature, mathematics, and computing. For example, the number of petals at different levels of certain flowers appears in a Fibonacci sequence.

How we can write the above recursive function??? (Hint what is the base case or the stop condition).

How recursion is implemented and can be traced:

recursive function call, like any function call, causes certain data to be placed on the system stack. A ***stack*** is a data structure such that the first item referenced or removed is always the last item entered into the stack. It is like a stack of plates in a cafeteria. When a plate is added to the stack, it is added to the top. When a plate is removed from the stack, it is removed from the top of the stack and is the plate that was added most recently.

A recursive call causes the return address to be added to the system stack as well as the current value of all local variables because when the return occurs these variables must be set back to their value before the call. For example, given the following recursive function

```
// An example of recursion.
int func_a(int n)
{
   int x;
   x = 5 + n;
   if (n == 1)
       return x;
   else
       return (n*func_a(n - 1));
}
```

If a main function were to call func_a(2), then space would be set aside in memory for n (which is initialized to 2) and x, and the instruction pointer (which tells the address of the next instruction to be executed) would be set to the address of the first instruction in func_a. func_a would then begin to execute. The first statement would assign 7 to x. Since n is 2, the else portion would be executed which would call func_a(1). Before func_a(1) can be executed, the value for n (2), the value for x (7), and the current address stored in the instruction pointer would have to be stored on the stack. Then func_a(1) can be executed. When func_a(1) is executed, n is set to 1, then x is set to 6. Since n is 1, the if portion is executed and 6 is returned. The return statement causes control to pass back to func_a(2). Before func_a(2) can continue execution its values for n, x, and the instruction pointer must be retrieved from the stack. This will cause n to be 2, and func_a(2) will finish executing the return statement (i.e., it will return 12 to main).

Recursion and iterations:

Some languages like LISP (also quite old) use recursion almost exclusively. The only way to loop in some older versions of LISP was by using recursion. You should understand that recursion often replaces a loop. It is rare to find a recursive call inside a loop (in simple programs). Such a design is a common error of those who are not adept at using recursion.

We will dedicate some part of questions below to compare the efficiency between normal iterations and recursion.

EXERCISE:

1- Trace the following functions and identify the purpose of each one

A:
```
int f1(int n)
{
        if (n==1)
                return 4;
        else
                return (4*f(n-1));
}
```

B:
```
int f2(int n)
{
        if (n==1)
                return 1;
        else
                return (n + f(n-1));
}
```

C:
```
void f3(int a)
{
        int d;
        d = a % 2;
        if(a != 0)
        {
        f3(a / 2);
        printf("%d", d);
        }
}
```

2- **Write the multiplication and division of two integers recursively.**

3- **Write the power function recursively power(m,n)** $\rightarrow m^n$.

4- **Write Fibonacci series function once using recursion and another without recursion and see the difference for fibo(60).**

5- **Write a c program that reads a set of integers into array and do this operation recursively**

A. Print the set
B. Print the set in reverse order
C. Search for an element using linear search
D. Search for an element using Binary search

6- **Write a c program that reads a string and check whether its palindrome or not recursively. (A palindrome string is a string which reads the same backward or forward).**

Lab No.10

Objective

Structures

Theory

If we want a group of same data type we use an array. If we want a group of elements of different data types we use structures. For Example: To store the names, prices and number of pages of a book you can declare three variables. To store this information for more than one book three separate arrays may be declared. Another option is to make a structure. No memory is allocated when a structure is declared. It simply defines the “form” of the structure. When a variable is made then memory is allocated. This is equivalent to saying that there is no memory for “int”, but when we declare an integer i.e. int var; only then memory is allocated.

The structure for the above mentioned case will look like

```
Struct books
{
        char bookname[20];
        float price;
        int pages;
}
struct book[50];
```

the above structure can hold information of 50 books.

EXERCISE

1. Write a program to maintain the library record for 100 books with book name, author's name, and edition, year of publishing and price of the book.
2. Write a program to make a tabulation sheet for a class of 50 students with their names, seat no's, marks, percentages and grades.
3. Define a structure to represent a complex number in rectangular format.
4. Consider the following structure definition:

```
        struct Student
        {
```

```
        char  name[10];
        int section;
        float grade;
};
```

Write a program that declare a list (struct Student) with size > 0. Your Program will do the following:

(a) Raise all the grades in the class 5% to a maximum of 100.

(b) Give everyone in the class whose name starts with “Al” a grade of 100.

(c) Sort the list by grades, highest to lowest.

Lab No.11

Objective

Filing in C-Language

Theory

Data files

Many applications require that information be “written & read” from an auxiliary storage device.

This information is written in the form of **Data Files**.

Data files allow us to store information permanently and to access and alter that information whenever necessary.

Types of Data files

Standard data files .(stream I/O)

System data files. (low level I/O)

Standard I/O

Easy to work with, & have different ways to handle data.

Four ways of reading & writing data:

1. Character I/O.
2. String I/O.
3. Formatted I/O.
4. Record I/O.

File Protocol

1. fopen
2. fclose

fopen:-

It is used to open a file.

Syntax:

fopen (file name , access-mode).

“r” open a file for reading only.

“w” open a file for writing.

“a” open a file for appending .

“r+” open an existing file for reading & writing.

“w+” open a new file for reading &writing.

“a+” open a file for reading & appending & create a new file if it does exist.

Example:-

```
#include <stdio.h>
main
{
        FILE *fpt;
        fpt = fopen (“first.txt”,”w”);
        fclose(fpt);
}
```

Establish buffer area, where the information is temporarily stored before being transferred b/w the computer memory & the data file.

File is a special structure type that establishes the buffer area.

fpt is a pointer variable that indicates the beginning of buffer area.

fpt is called stream pointer.

fopen stands for File Open.

A data file must be opened before it can be created or processed.

Standard I/O:

Four ways of reading and writing data:

Character I/O.

String I/O.

Formatted I/O.

Record I/O.

Character I/O:

In normal C program we used to use getch, getchar and getche etc. In filling we use putc and getc.

putc( );

It is used to write each character to the data file.

Putc requires specification of the stream pointer *fpt as an argument.

Syntax:

putc(c, fp);

c = character to be written in the file.

fp = file pointer.

putch or putchar writes the I/P character to the consol while putc writes to the file.

Example (1):

Reading the Data File:

```
#include < stdio.h >
main( )
{
        FILE*fpt;
        char c;
        fpt = fopen ( " star.dat","r");
        if( fpt = = NULL)
                printf( "Error-cant open");
        else
        do
        {
                putchar(c= getc(fpt) );
        } while(c!= '\n' );
        fclose( fpt );
}
```

Exercise:

1. What will be the output of the given program

```
#include < stdio.h >
main( )
{
   FILE*fpt;
   char c;
   fpt = fopen ( " star.dat","w");        // a new file is made

   do
   {
            putc((c= getchar( ) ), fpt );
   }while(c!= '\n' );                // or '\r'
   fclose( fpt );
}
```

2. Write a program to store strings in a file.

3. Write a program segment that writes an array to a file.