L A B M A N U A L

EASWARI ENGINEERING COLLEGE

DEPARTMENT OF INFORMATION TECHNOLOGY

IT6212

PROGRAMMING AND DATA STRUCTURES LABORATORY I

LAB MANUAL

I YEAR IT A & B

JANUARY 2015 TO MAY 2015

PREPARED BY

Mrs. M.SOWMIYA
Mrs.MIJULA NIVAS

APPROVED BY

HOD

IT6212 PROGRAMMING AND DATA STRUCTURES LABORATORY I **L T P C**
0 0 3 2

OBJECTIVES:

- To introduce the concepts of structured Programming language.
- To introduce the concepts of pointers and files
- To introduce the concepts of primitive Data Structures.

1. C Programs using Conditional and Control Statements
2. C Programs using Arrays, Strings and Pointers and Functions
3. Representation of records using Structures in C – Creation of Linked List – Manipulation of records in a Linked List
4. File Handling in C – Sequential access – Random Access
5. Operations on a Stack and Queue – infix to postfix – simple expression evaluation using stacks - Linked Stack Implementation – Linked Queue Implementation
6. Implementation of Sorting algorithms
7. Implementation of Linear search and Binary Search.

TOTAL: 45 PERIODS

LIST OF EXPERIMENTS

1. CONDITIONAL AND CONTROL STATEMENTS

Objectives:

- Learn the basics and syntax of conditional and control statements
- Write programs using the above statements

Description:

Control statements enable us to specify the flow of program control; (i.e), the order in which the instructions in a program must be executed. They make it possible to make decisions, to perform tasks repeatedly or to jump from one section of code to another.
There are four types of control statements in C:

1. Decision making statements
2. Selection statements
3. Iteration statements
4. Jump statements

Decision making and selection statements are called conditional statements.

Decision Making Statements:

If-else Statement

The if-else statement is used to carry out a logical test and then take one of two possible actions depending on the outcome of the test (ie, whether the outcome is true or false).

Syntax:

```
if (condition)
{
  Statements;
}
  else
{
  Statements;
}
```

Nested if and if-else Statements

It is also possible to **embed** or to **nest** if-else statements one within the other. Nesting is useful in situations where one of several different courses of action need to be selected.

Syntax:

```
if(condition1)
{
// statement(s);
}
else if(condition2)
{
//statement(s);
}
```

```
. . .
else
{
//statement(s);
}
```

The above is also called the **if-else ladder**. During the execution of a nested if-else statement, as soon as a condition is encountered which evaluates to true, the statements associated with that particular if-block will be executed and the remainder of the nested if-else statements will be bypassed. If neither of the conditions are true, either the last else-block is executed or if the else-block is absent, the control gets transferred to the next instruction present immediately after the else-if ladder.

Selection Statement:

Switch-case Statement

A switch statement is used for **multiple way selections** that will branch into different code segments based on the value of a variable or expression. This expression or variable must be of integer data type.

Syntax:

```
switch (expression)
{
  case value1:
    code segment1;
    break;
. . .
  case valueN:
    code segmentN;
    break;
  default:
    default code segment;
}
```

The value of this **expression** is either generated during program execution or read in as user input. The case whose value is the same as that of the **expression** is selected and executed. The optional **default** label is used to specify the code segment to be executed when the value of the expression does not match with any of the case values.

The break statement is present at the end of every case. If it were not so, the execution would continue on into the code segment of the next case without even checking the case value.

The switch statement must be used when one needs to make a choice from a given set of choices. The switch case statement is generally used in **menu-based applications**. The most common use of a switch-case statement is in data handling or file processing.

Iteration Statements (Looping)

Iteration statements are used to execute a particular set of instructions repeatedly until a particular condition is met or for a fixed number of iterations.

For loop:

The for loop repeatedly executes the set of instructions that comprise the body of the for loop until a particular condition is satisfied.

Syntax:

```
for(initialization; termination; increment/decrement)
          {
             //statements to be executed
          }
```

The for loop consists of three expressions:

- The **initialization expression**, which initializes the **looping index**. The looping index controls the looping action. The initialization expression is executed only once, when the loop begins.
- The **termination expression**, which represents a condition that must be true for the loop to continue execution.
- The **increment/decrement expression** is executed after every iteration to update the value of the looping index.

While loop:

The while statement executes a block of statements repeatedly while a particular condition is true.

Syntax:

```
while (condition)
{
   //statements to be executed
}
```

The statements are executed repeatedly until the condition is true.

Do-while loop:

The do-while statement evaluates the condition at the end of the loop after executing the block of statements at least once. If the condition is true the loop continues, else it terminates.

Syntax:

```
do
{
   //statements to be executed;
} while(condition);
```

The difference between while and do-while is that the while loop is an **entry-controlled loop** — it tests the condition at the beginning of the loop and will not execute even once if the condition is false, whereas the do-while loop is an **exit-controlled loop** — it tests the condition at the end of the loop after completing the first iteration.

1. a. Check whether the given year is leap year or not

Aim:

Write a C program to check whether the given year is leap year or not.

Algorithm:

```
    Input  : 4-digit integer
    Output : Leap year or not
1. Get the year in four digits
2. If the year is divisible by 100
    a. If the year is divisible by 400
        i. print the year is leap year
    b. Otherwise, if the year is divisible by 4
        i. print this is a leap year
    c. Or, print not a leap year
```

Sample I/O:

```
Input  : 2004
Output: 2004 is a leap year
Input  : 1900
Output: 1900 is not a leap year
```

Result:

Thus, the C program to check whether the given year is leap year or not is written and is verified for output.

1. b. Compute the sum of digits of a given number

Aim:

Write a C program to compute the sum of digits of a given number.

Algorithm:

```
    Input  : Any number (in the range 0 to 32768)
    Output : Sum of the digits of the given number
1. Get the number from the user in variable 'num'.
2. Initialize a variable 'sum' to 0.
3. Repeat until we have no digits left to process
4. Find the rightmost digit (hint: use % operator)
5. Add the rightmost digit to 'sum'.
6. Remove the rightmost digit from 'num'. (hint: use / operator)
7. Print 'sum'.
```

Sample I/O:

```
Input  : 456
Output: Sum of the digits of 456 is: 15
Input  : 7896
Output: Sum of the digits of 7896 is: 30
```

Result:

Thus, the C program to compute the sum of the digits of the given number is written and is verified for output.

1. c. Test the given number for primality

Aim:

Write a C program to check whether the given number is prime or not.

Algorithm:

```
    Input  : Any number (in the range 0 to 32768)
    Output : Prime Number or not
1. Get the number to be checked for primality as 'num'
2. Initialize a variable 'i' to 2
3. Repeat while 'i' is less than or equal to half of 'num' (i.e.)num/2)
        a. If the number is exactly divisible by i
            i. print "Not a Prime number" and exit;
        b. Otherwise increment 'i' by 1
4. Check if 'i' is equal to half of 'num' + 1
        a. Print "Prime number"
```

Sample I/O:

```
Input  : 17
Output: 17 is a prime number
Input  : 256
Output: 256 is not a prime number
```

Result:

Thus, the C program to check whether the given number is prime or not is written and is verified for output.

1. d. Perform the basic arithmetic operations using switch case

Aim:

Write a C program to perform the basic arithmetic operations such as addition, subtraction, multiplication and division using switch case.

Algorithm:

```
    Input  :
            • Any two numbers (in the range 0 to 32768)
            • Any one of the basic arithmetic operators (+,-,*,/)
    Output :
            • Prints sum of the given two  numbers if operator is +
            • Prints difference of the given two  numbers if operator is -
            • Prints product of the given two  numbers if operator is *
            • Prints quotient if the operator is /
    1. Get two numbers n1, n2
    2. Get the operator
        a. If operator is '+', add 'n1' and 'n2'.
        b. If operator is '-' , subtract 'n2' from 'n1'.
        c. If operator is '+', multiply 'n1' and  'n2'.
        d. If operator is '+', divide 'n1' by 'n2'.
```

3. Print the respective result.

Sample I/O:

```
Input  :  15  3  +
Output :  18
Input  :  15  3  /
Output :  5
```

Result:

Thus, the C program to perform the basic arithmetic operations such as addition, subtraction, multiplication and division using switch case is written and is verified for output.

1.e. Find the roots of a quadratic equation

Aim:

Write a C program to find the roots of a quadratic equation.

Algorithm:

Input : Three coefficients a, b, c [Note: a, b, c are coefficients of this quadratic equation $ax^2 + bx + c = 0$]

Output :

Roots of the quadratic equation.

```
1. Read the coefficients a, b and c.
2. Find determinant, d= (b*b)-(4*a*c).
3. If d is greater than 0 then
     • Print the roots are real and distinct.
     • Find the root r1=((-b+ sqrt(d)/2*a).
     • Find the root r2=((-b- sqrt(d)/2*a).
     • Print the roots r1 and r2.
4. Or, if d is equal to 0 then
     • Print the real and equal.
     • Find the roots r1=r2=(-b/(2*a))
     • Print the roots r1 and r2.
5. Otherwise,
     • Assign d=-d
     • Print the roots are imaginary.
     • Find p=(-b/(2*a))
     • Find q=(sqrt(d)/(2*a))
     • Print the roots p + i q and p + i q.
```

Sample I/O:

Input : 1.0 5.0 2.0

Output: Roots are real and distinct

```
          r1=-0.438447237 and r2=-4.561553
```

Result:

Thus, the C program to find the roots of quadratic equation is written and is verified for output.

PRACTCE EXERCISES:

1. Write a C program to accept any single digit number and print it in words.
2. Write a C program to generate numbers between 1 and 100 which are divisible by 2 and not divisible by 3 and 5.
3. Write a C program to find the sum of odd-positioned digits and even-positioned digits of a number separately
4. Write a C program to print all combinations of a 4-digit number
5. Write a C program to accept a number and print mathematical table of the given number.

VIVA QUESTIONS

1. **Differentiate break and continue.**
   The continue statement forces the next iteration of the loop to take place, skipping the remaining unexecuted statements in the loop. On the other hand, break statement forces termination of the loop.

2. **Describe statement, expression, control structure and function.**
   A **statement** is a piece of code that executes. It is a command. It does something.
   Example: print "Hello World"
   An **expression** is a piece of code that evaluates to a value.
   Example: 2 + 2
   A **function** is a block of code that you call with parameters and it returns a value (even if that value is undefined). Example: function(a) { return a * 2 }
   A **control structure** is a statement that is used to direct the flow of execution.
   Examples:
   if (condition) then { branch_1 } else { branch_2 }
   for (i = 0; i < 10; i += 1) { ... }

3. **Compare Else-if and switch case statements**
   The switch statement is faster than the if statement, but the if-else or if-else-if statement is more suitable than switch statement, when you have to use logical operations like (<,>,<=,>=,!=,==). Switch is usually more compact than lots of nested if else and more readable, but only accepts primitive types as key and constants as cases.

4. **Differentiate between while and do-while statements.**
   The do while loop is very similar to the while loop except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before terminating.

5. **What do you mean by branching? Give example.**
   Branching is deciding what actions to take and looping is deciding how many times to take a certain action.
   It takes an expression in parenthesis and an statement or block of statements. if the expression is true then the statement or block of statements gets executed otherwise these statements are skipped.

Example: for, while and do - While and decision making using if and switch.

6. **What is the output of the following program?**

```
#include<stdio.h>
void main()
{       int a=8,b=4;
        int c;
        c=(a<6) || (b=10);
        printf("\nc= %d, a=%d, b=%d",c,a,b); }
```

7. **What is the output of the following program?**

```
void main()
{    int i=10;
   switch(i)
   {
     case 1: printf(" i=1");              break;
     case 10: printf(" i=10");
     case 11: printf(" i=11");              break;
     case 12: printf(" i=12");
    }     }
```

8. **What is the output of the following program?**

```
void main()
{     int i=10;
    if(i ==10)
        printf("I is 10");
    else if ( i <= 10)
        printf(",i is less than or equal to 10");
    else
    {         break;    }     }
```

9. **What is the output of the following program?**

```
void main()
{     int i=1,j=1;
    while (++i < 10)
        printf("%d ",i);        printf("\n");
    while (j++ < 10)
        printf("%d ",j);   }
```

10. **What is the output of the following program?**

```
void main()
{     int i=1;
    do
    {         printf("%d ",i);
        i++;
    } while(i > 5);
    printf(" End");  }
```

2. ARRAYS AND STRINGS

Array

An array is a data structure that is used for the storage of homogeneous data, i.e. data of the same type.

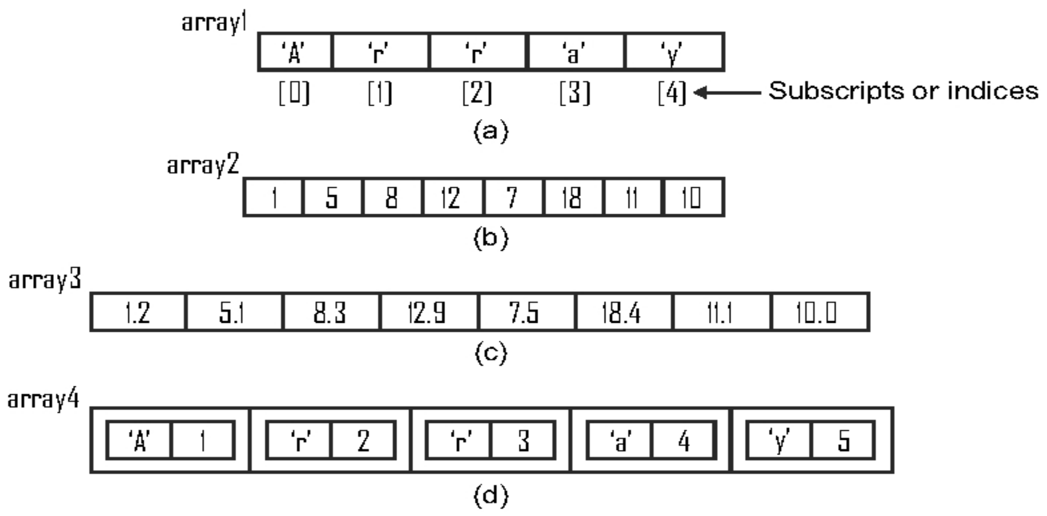


Figure (a) Character array; (b) integer array; (c) float array; (d) array of user-defined type

An array is a collection of elements of the same data type. The elements of an array are stored in contiguous (i.e. continuous) memory locations.

In general, arrays are classified as:

1. Single-dimensional arrays
2. Two dimensional arrays
3. Multi-dimensional arrays

Declare and Initialize a One-Dimensional Array

Syntax:

dataType arrayName [numberOfElements]={initialValues};

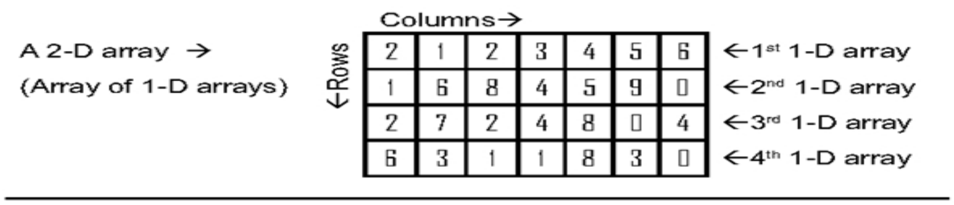
Example

char letters[3] = { 'A', 'B', 'C' } ;

int num[3] = { 0, 0, 0}; or int num[3] = {0};

The elements of a single-dimensional array can be accessed by using a subscript operator (i.e. []) and a subscript.

A **two-dimensional array** has its elements arranged in a rectangular grid of rows and columns. The elements of a two-dimensional array can be accessed by using a row subscript (i.e. row number) and a column subscript (i.e. column number). A two-dimensional array is popularly known as a matrix.



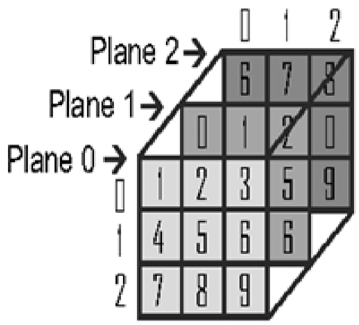
Declaration:

```
dataType arrayName [no._of_ rows] [no._of_columns];
```

Initialization:

```
int a[3][4] = {  { 0, 1, 2, 3 } ,   /*  initializers for row indexed by 0 */
                 { 4, 5, 6, 7 } ,   /*  initializers for row indexed by 1 */
                 { 8, 9, 10, 11 }   /*  initializers for row indexed by 2 */
              };
Or,
    int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

A **three-dimensional array** can be visualized as a cube that has a number of planes. Each plane is a two-dimensional array.



String

A character string literal constant or just a string literal is a sequence of zero or more characters enclosed within double quotes. For example, "GOD Bless!!" is a string literal constant. The prototype of all string handling functions is defined in string.h and ctype.h.

String.h

Common functions

1. To compare two strings
   ```
   int strcmp (const char *string1, const char *string2)
   ```
2. To compare first n characters of two strings
   ```
   int strncmp(const char *string1, char *string2, size_t n)
   ```
3. To determine the length of a string
   ```
   int strlen(const char *string)
   ```
4. To copy string2 to string1
   ```
   char *strcpy(const char *string1,const char *string2)
   ```
5. Copy first n characters of string2 to string1
   ```
   char *strncpy(const char *string1,const char *string2, size_t n)
   ```
6. Append string2 to string1
   ```
   char *strcat(const char *string1, char *string2)
   ```
7. Append n characters from string2 to string1
   ```
   char *strncat(const char *string1, char *string2, size_t n)
   ```

8. Compares the C string s1 to the C string s2
   int strcmp(const char *s1, const char *s2)

9. Case insensitive version of strcmp()
   int strcmpi(const char *s1, const char *s2)

10. Compares up to n characters of the C string s1 to those of the C string s2
    int strncmp(const char *s1, const char *s2, int n).

11. Case insensitive version of strncmp()
    int strncmpi(const char *s1, const char *s2, int n)

String Searching

The library also provides several string searching functions:

1. Find first occurrence of character c in string
   char *strchr(const char *string, int c)

2. Find last occurrence of character c in string
   char *strrchr(const char *string, int c)

3. Locates the first occurrence of the string s2 in string s1
   char *strstr(const char *s1, const char *s2)

4. Returns the number of characters at the begining of s1 that match s2
   size_t strspn(const char *s1, const char *s2)

5. Returns the number of characters at the begining of s1 that do not match s2
   size_t strcspn(const char *s1, const char *s2)

ctype.h

Character testing

True if c is alphanumeric.	int isalnum(int c)
True if c is a letter.	int isalpha(int c)
True if c is ASCII .	int isascii(int c)
True if c is a control	charactint iscntrl(int c)
True if c is a decimal digit	int isdigit(int c)
True if c is a graphical character	int isgraph(int c)
True if c is a lowercase letter	int islower(int c)
True if c is a printable character	int isprint(int c)
True if c is a punctuation character	int ispunct (int c)
True if c is a space character	int isspace(int c)
True if c is an uppercase letter	int isupper(int c)
True if c is a hexadecimal digit	int isxdigit(int c)

Character Conversion

Convert c to ASCII.	int toascii(int c)
Convert c to lowercase.	int tolower(int c)
Convert c to uppercase.	int toupper(int c)

2.a. Find the largest element in a row and in a column of a given matrix

Aim:

Write a C program to find the largest element in a row and largest element in a column of a given matrix.

Algorithm:

Input: Number of rows and columns of the matrix
Elements of the matrix
Row or column number for which the largest element is to be found.

Output: Largest element of the respective row or column

```
1.  Get the number of rows and columns.
2.  Get the elements of the matrix.
3.  Get the row or column number for which the largest element is to be found.
4.  If the choice is row, set a variable r to the read row number
        •  Assign the first element of that row to a variable max, which will hold
           the maximum value.
        •  Loop through the remaining elements starting from second element.
        •  When a value larger than the present max value is found, set that
           value as max
5.  If the choice is column, set a variable r to the read column number
        •  Assign the first element of that column to a variable max, which will
           hold the maximum value.
        •  Loop through the remaining elements starting from second element.
        •  When a value larger than the present max value is found, set that
           value as max.
6.  Print the value of the variable max.
```

Sample I/O:

```
Input: Enter the number of rows and columns: 3 3
                   Enter the elements of the matrix
                          12   7   4
                            5   6   8
                           4  23   9
                   Enter the choice  (r- row or c- column) : r
                 Enter the row number : 2
Output: The Largest element in row 2 is: 23
```

Result: Thus the C program to find the largest element in a row and largest element in a column of a given matrix is written and is verified for output.

2.b Find the inverse of a square matrix

Aim:

Write a c program to find the inverse of a square matrix.

Algorithm:

Input: Number of rows and columns of the matrix
Elements of the matrix

Output: Inverse of the given matrix

1. Get the number of rows and columns.
2. Get the elements of the matrix A.
3. Check whether the determinant of the given matrix is zero or not.
    a. If zero, print inverse matrix does not exist.
    b. Otherwise, proceed to step 4
4. Form the augmented matrix B, It is formed by augmenting the matrix A with an identity matrix of the same dimension of matrix A.
5. Apply elementary row operations on B, so that its first part reduces to identity matrix. Inverse matrix appears in the second part.
6. Undo the matrix augmentation, to retrieve the inverse matrix.
7. Print the inverse of the matrix A.

Sample I/O:

Input:
```
Enter the number of rows and columns: 3 3
Enter the elements of the matrix
        3   0  2
        2   0  -2
        0   1  1
```
Output:
```
The inverse of the given matrix
        0.2   .02    0
        -0.2   0.3   1
        0.2   -0.3   0
```

Result:

Thus the C program to find the inverse of a given matrix is written and is verified for output.

2. c. Perform string comparison without using Library function

Aim:

Write a C program to perform String comparison without using library functions.

Algorithm:

Input: Two strings

Output: Strings are equal or not equal.

1. Get the two strings to be compared as s1 and s2.
2. Set a flag variable to zero
3. Repeat the following until end of string
    a. Compare each character of s1 and s2

```
        b. If equal, repeat the loop
        c. Otherwise, set flag as one and terminate the loop.
    4. Check if flag is zero
        a. If zero, print "strings are equal"
        b. Otherwise print "strings are not equal"
```

Sample I/O:

```
Input:   Enter the two strings to be compared:
               program
               program
Output: Strings are equal
```

Result:

Thus the C program to perform string comparison without using library function is written and is verified for output.

2.d. Count the number of sentences, words and characters

Aim:

Write a C program to count the number of sentences, words and characters.

Algorithm:

Input: Line or paragraph of text

Output: Number of sentences, words and characters

```
1. Get a line or paragraph of text.
2. Initialize variables sc, wc, cc with zero, which will hold the sentence count, word
   count and character count respectively.
3. Repeat the following for each character until end of the paragraph or end of the
   line
    a. If the character read is '.', '?', or '!', increment sc and cc by 1.
    b. If the character read is ' ' or '\t', increment wc and cc by 1.
    c. Otherwise increment cc by 1.
4. Print the values of sc, wc+1 and cc.
```

Sample I/O:

```
Input: Enter the input text
To forgive is the highest, most beautiful form of love. In return, you will receive
untold peace and happiness.
Output: Number of sentences : 2
        Number of words      : 19
        Number of characters : 111
```

Result:

Thus the C program to count the number of sentences, words and characters is written and is verified for output.

PRACTICE EXERCISES

1. Write a ‘C’ program to arrange the digits of a number in ascending order.
2. Write a ‘C’ program to count the number of times a digit is present in a number.
3. Write a C program to count a character that appears in a given text for number of times using while loop.
4. Write a C program to print the abbreviation for a given sentence. (Example: American Standard Code for Information Interchange as ASCII)
5. Write a c program to remove the occurrence of “the” word from entered string.

VIVA QUESTIONS

1. **Why is it necessary to give the size of an array in an array declaration?**
   When an array is declared, the compiler allocates a base address and reserves enough space in the memory for all the elements of the array. Hence, the size must be given in the array declaration.

2. **What is the difference between strings and character arrays?**
   String will have static storage, whereas character array is explicitly specified by using the static keyword.
   Two strings of same value[1] may share same memory area. For example, in the following declarations:
   ```
   char *s1 = “Calvin and Hobbes”;
   char *s2 = “Calvin and Hobbes”;
   ```
   The strings pointed by s1 and s2 may reside in the same memory location. But, it is not true for the following:
   ```
   char ca1[] = “Calvin and Hobbes”;
   char ca2[] = “Calvin and Hobbes”;
   ```

3. **Write the limitations of using getchar() and scanf() functions for reading strings.**
   getchar() reads a single character. A single character will be at least 1 byte in size but may be larger, there is no guarantee of its size, the compiler will determine its size.

   scanf() is a function used for reading input, how that input is interpreted is determined by the format string that is passed to the function. scanf can be used to read the majority of C's primitive types and can also be used for reading types that have no explicit primitive in C - for example scanf can read Octal integers even though no explicit Octal type exists scanf will convert the Octal integer into a Base 10 int.

4. **What value is automatically assigned to array elements that are not explicitly initialized?**
   Compiler will assign 0, if the array not initialized. If an array is partially initialized, elements that are not initialized receive the value 0 of the appropriate type. Initializing a string constant places the null character (\0) at the end of the string if there is room or if the array dimensions are not specified.

5. **Which extra character is added at the end of the string when character array length is specified?**

The amount of memory allocated for the character array may extend past the null character that normally marks the end of the string. But the term allocation size is always used to refer to the total amount of memory allocated for the string, while the term length refers to the number of characters up to (but not including) the terminating null character.

6. **What is the output of the following program, if starting address of the array is 1000?**

```
#include<stdio.h>
void main()
{
    int arr[]={10,20,30,40};
    printf("%u %u %u",arr,&arr[0],&arr);
}
```

7. **What is the output of the following program, if starting address of the array is 1000?**

```
#include<stdio.h>
void main()
{
        int arr[]={10,20,30,40,50};
        printf("%u %u %u",arr,arr+1,&arr+1);
}
```

8. **What is the output of the following program?**

```
#include<stdio.h>
int main()
{
  char s[ ]="man";
  int i;
  for(i=0;s[i];i++)
   printf("%c%c%c%c\t",s[i],*(s+i),*(i+s),i[s]);
}
```

3. FUNCTIONS AND POINTERS

Functions

A C program is made up of functions. Functions interact with each other to accomplish a particular task.Based upon who develops the function, functions are classified as:

1. User-defined functions
2. Library functions

User-defined Functions: User-defined functions are the functions that are defined (i.e. developed) by the user at the time of writing a program. The user develops the functionality by writing the body of the function. These functions are sometimes referred to as programmer-defined functions.
All the functions need to be declared or defined before they are used (i.e. called). The general form of a function declaration is:

[return_type] function_name ([parameter_list or parameter_type_list]);

Function prototypes (i.e. function declarations) are important and their necessity can be seen from two different perspectives:
User perspective: It tells the user how to use a pre-defined or library function. It tells the user the number of parameters along with their types that a function expects and its return type.
Compiler perspective: It allows the compiler to perform type checking. By type checking the compiler ensures that while making a function call, the user provides the correct number and the correct type of arguments.

Function Definition: Function definition, also known as function implementation, means composing a function. Every function definition consists of two parts:

1. Header of the function
2. Body of the function

Header of the function: The general form of header of a function is:

[return_type] function_name([parameter_list])

Body of a Function: The body of a function consists of a set of statements enclosed within braces. The body of a function can have non-executable statements and executable statements.

A **function with no input–output** does not accept any input and does not return any result. Since no input is to be given to the function, the parameter list of such functions is empty. If a function does not return any value, then the return type of the function should be specified as void (means nothing). Functions whose return type is void are known as **void functions**.

The expressions that appear within the parentheses of a function call are known as **actual arguments**, and the variables declared in the parameter list in the function header are known as **formal parameters**.
The **return statement** is used to return the result of the computations performed in the called function and/or to transfer the program control back to the calling function.

Depending upon whether the values or addresses (i.e. pointers) are passed as arguments to a function, the argument passing methods in C language are classified as:

1. Pass by value
2. Pass by address

Pass by value: The method of passing arguments by value is also known as **call by value**. In this method, the values of actual arguments are copied to the formal parameters of the function.

Pass by address: The method of passing arguments by address or reference is also known as **call by address** or **call by reference**. In this method, the addresses of the actual arguments are passed to the formal parameters of the function. If the arguments are passed by reference, the changes made in the values pointed to by the formal parameters in the called function are reflected back to the calling function.

Recursion
In recursive programming, a function calls itself. A function that calls itself is known as a **recursive function**, and the phenomenon is known as recursion. Recursion is classified according to the following criteria:

1. Whether the function calls itself directly (i.e. **direct recursion**) or indirectly (i.e. **indirect recursion**).
2. Whether there is any pending operation on return from a recursive call. If the recursive call is the last operation of a function, the recursion is known as **tail recursion**.
3. Pattern of recursive calls. According to the pattern of recursive calls, recursion is classified as:

a. Linear recursion
b. Binary recursion
c. n-ary recursion

Based upon the number of arguments a function accepts, functions are classified as follows:

1. Fixed argument functions
2. Variable argument functions

Fixed Argument Functions: A function that accepts a fixed number of arguments is called a fixed argument function.

Variable Argument Functions: A function that accepts a variable number of arguments is called a variable argument function. For example, printf is a variable argument function, which can accept one or more arguments.

Example:

```
int func(int, ... )
{
  .
  .
  .
}
```

```
int main()
{
  func(1, 2, 3);
  func(1, 2, 3, 4);
}
```

It should be noted that function func() has last argument as ellipses i.e. three dotes (...) and the one just before the ellipses is always an int which will represent total number variable arguments passed. To use such functionality you need to make use of stdarg.h header file which provides functions and macros to implement the functionality of variable arguments and follow the following steps:

- Define a function with last parameter as ellipses and the one just before the ellipses is always an **int** which will represent number of arguments.
- Create a **va_list** type variable in the function definition. This type is defined in stdarg.h header file.
- Use **int** parameter and **va_start** macro to initialize the **va_list** variable to an argument list. The macro va_start is defined in stdarg.h header file.
- Use **va_arg** macro and **va_list** variable to access each item in argument list.
- Use a macro **va_end** to clean up the memory assigned to **va_list** variable.

Pointers

A pointer is a variable that holds the address of a variable or a function. A pointer variable can be declared as:

A pointer variable declaration consists of a type specifier (i.e. referenced type), punctuator * and an identifier (i.e. name of pointer variable).

The following declarations are valid:

```
int *iptr;      //←iptr is pointer to an integer
float *fptr;    //←fptr is pointer to a float
char *cptr;     //←cptr is pointer to a character
```

In **referencing operation**, a pointer variable is made to refer to an object. The reference to an object can be created with the help of a reference operator.

The object pointed to or referenced by a pointer can be indirectly accessed by **dereferencing the pointer**. A dereferencing operation allows a pointer to be followed to the data object to which it points. A pointer can be dereferenced by using a dereference operator (i.e. *).

A pointer can be assigned or initialized with the address of an object. A pointer variable cannot hold a non-address value and thus can only be assigned or initialized with l-values.

Arithmetic operations can be applied to pointers in a restricted form. When arithmetic operators are applied on pointers, the outcome of the operation is governed by pointer arithmetic.

void pointer: void is one of the basic data types available in C language. void means nothing or not known. It is not possible to create an object of type void. For example, the following declaration statement is not valid and leads to ‘Size of var unknown or zero’ compilation error.

Example of void pointer:

```
void var;
```

Null Pointer: A null pointer is a special pointer that does not point anywhere. It does not hold the address of any object or function. It has numeric value 0. The following declaration statement declares nptr as a null pointer:

Example of Null pointer:

```
int *nptr=0;
```

Dangling pointer:

Pointer assigned some memory and then frees that memory. After freeing when not assigned to NULL it still points to same memory address, such pointer are dangling pointer.

Example of dangling pointer:

```
int* ptr=new int;
<some ops with ptr >
delete ptr; // at this stage, ptr is dangling pointer
```

Wild pointer:

Pointer which are not initialized during its definition holding some junk value(a valid address) are Wild pointer.

Example of wild pointer:

```
int *ptr;
```

3.a Compute sum of array elements

Aim:

Write a C program to compute the sum of all elements stored in an array using pointers.

Algorithm:

```
Input: Number of elements in the array
       Elements of the array
Output: Sum of the elements in the array
1. Get the number of array elements
2. Get the array elements
3. Initialize sum to zero
4. Assign the address of array to a pointer variable
5. Repeat the following for the all the array elements
    a. Get the value at each array position using dereferencing operator(*)
    b. Add the value to sum
6. Print the sum
```

Sample I/O:

Input: Enter the number of array elements: 5

Enter the elements:

```
6  9   3  7  2
```

Output: The sum of the given array is: 27

Result:

Thus the C program to compute the sum of all elements stored in an array using pointers is written and is verified for output.

3.b Find number of words, blank spaces, special symbols, digits, vowels using pointers

Aim:

Write a C program to count the number of words, blank spaces, special symbols, digits, vowels using pointers

Algorithm:

Input: Line or paragraph of text

Output: Number of words, blank spaces, special symbols, digits, vowels

```
1. Declare a character pointer
2. Get a line or paragraph of text using the character pointer.
3. Initialize variables wc, bc, sc, dc and vc with zero, which will hold the word
   count, blank space count, special symbol count, digit count and vowel count
   respectively.
4. Read each character through dereferencing operator and repeat the following
   until end of the paragraph or end of the line
    i. If the character read is ‘ ’ increment wc and bc by 1.
```

```
        ii.  If the character read is a digit, increment dc by 1.
        iii. If the character read is an alphabet and also a vowel, increment vc by 1.
        iv.  If the character read is neither an alphabet nor a digit increment sc by 1.
    5. Print the values of wc+1, bc, sc, dc and vc.
```

Sample I/O:

Input: Enter the input text

```
Rockcliffe Book Fair: Not Just Awesome Books but Awesome Vinyl LPs/12"s/45's as Well!
```

Output:

```
Number of words 13
Number of blank spaces 12
Number of vowels 23
Number of digits 4
Number of special characters 6
```

Result:

Thus the C program to count the number of sentences, words and characters is written and is verified for output.

3.c Conversion of decimal to binary, octal and hexadecimal using functions

Aim:

Write a C program to convert a given decimal number to binary, octal and hexadecimal using functions

Algorithm:

Input: Decimal number (integer)

Output: Binary, Octal and Hexadecimal representation

main():

```
1. Get a decimal number
2. Call the decimal_to_binary function and store the return value in an integer array.
3. Call print_reverse to display the array elements in reverse.
4. Call the decimal_to_octal function and store the return value in an integer array.
5. Call print_reverse to display the array elements in reverse.
6. Call the decimal_to_hexadecimal function and store  the return value in an integer
   array.
7. Call print_reverse to display the array elements in reverse
```

Decimal_to_binary():

Input: Decimal number from main

Output: Returns binary representation to main

```
1. Repeat the following until the decimal number becomes zero
    a. Divide the input by 2, and store the remainder in an array
    b. Increment the array index
    c. Divide by 2 and store the quotient as input for next iteration
2. Return the array to main function
```

Decimal_to_octal():

Input: Decimal number from main

Output: Returns octal representation to main

```
1. Repeat the following until the decimal number becomes zero
    a. Divide the input by 8, and store the remainder in an array
    b. Increment the array index
    c. Divide by 8 and store the quotient as input for next iteration
2. Return the array to main function
```

Decimal_to_hexadecimal():
Input: Decimal number from main
Output: Returns hexadecimal representation to main

```
1. Repeat the following until the decimal number becomes zero
    a. Divide the input by 16, and store the remainder in an array
    b. Increment the array index
    c. Divide by 16 and store the quotient as input for next iteration
2. Return the array to main function
```

print_reverse():
Input: Array and its size from the main function
Output: Reverse of input array

```
1. Repeat the following from array_size to 0.
    a. Print the element.
    b. Decrement the array_size by 1.
```

Sample I/O:

Input: Enter the decimal number: 25

Output:

```
Binary representation of 25: 11001
Octal representation of 25: 31
Hexadecimal representation of 25:19
```

Result:

Thus the C program to convert a given decimal number to binary, octal and hexadecimal using functions is written and is verified for output.

3.d Compute average of N numbers using variable argument functions

Aim:

Write a C program to compute the average of N numbers using variable argument functions.

Algorithm:

main():
Input: Number of elements and the elements (integer)
Output: Average of given numbers

```
1. Call the average function with the number of elements and the elements
2. Print the average returned from the called function
```

average():
Input: Number of the elements and the elements
Output: Returns average to the main function.

```
1. Initialize a variable sum to zero.
```

2. Create a va_list type variable in the function definition. This type is defined in stdarg.h header file.
3. Use int parameter and va_start macro to initialize the va_list variable to an argument list. The macro va_start is defined in stdarg.h header file.
4. Use va_arg macro and va_list variable to access each item in argument list.
5. Add each item in the argument list to sum.
6. Use a macro va_end to clean up the memory assigned to va_list variable.
7. Return average value to main function.

Sample I/O:

Input: 6, 18, 21
Output: Average of 3 numbers: 15
Input: 7, 8, 4, 15, 6
Output: Average of 5 numbers: 8

Result:

Thus the C program to compute the average of N numbers using variable argument functions is written and is verified for output.

3.e. Generate Fibonacci series using recursion

Aim:

Write a C program to generate n fibonacci numbers using recursive function.

Algorithm:
main():
Input: An integer n.
Output: n Fibonacci numbers.

1. Get n, the number of Fibonacci numbers to be generated.
2. Call Fibonacci function with n as argument
3. Print the Fibonacci numbers.

fibonacci(():
Input: Number of fibonacci terms to be generated
Output: Returns the Fibonacci terms to main function

1. Check if n is less than or equal to 1.
   a. If yes, return n to main
   b. Otherwise, return the sum obtained by recursively calling Fibonacci function with n-1 and n-2 as arguments.

Sample I/O:

Input: Enter the value of n: 7
Output: 0 1 1 2 3 5 8

Result:

Thus the C program to generate n fibonacci numbers using recursive function is written and is verified for output.

PRACTICE EXERCISES

1. Write a C program to find ${}^{n}C_{r}$ and ${}^{n}P_{r}$ using recursive function.
2. Write a C program to find length of the string using pointer.
3. Write a C program to find substring of string without using library function
4. Write a C program to computer n^th triangular number using function.

VIVA QUESTIONS

1. **What is function prototyping? Why is it necessary?**
   A function prototype is a function declaration that specifies the data types of its arguments in the parameter list. The compiler uses the information in a function prototype to ensure that the corresponding function definition and all corresponding function declarations and calls within the scope of the prototype contain the correct number of arguments or parameters, and that each argument or parameter is of the correct data type.

2. **What is an argument? Differentiate between formal arguments and actual arguments?**
   An argument is an entity used to pass the data from calling function to the called function. Formal arguments are the arguments available in the function definition. They are preceded by their own data types.
   Actual arguments are available in the function call.

3. **Explain the three elements of a function.**
   Function Declaration: All identifiers in C need to be declared before they are used. The declaration statement needs to be before the first call of the function.
   Function definition: It is actual code for the function to perform the specific task.
   Function call: In order to use function in the program we use function call to access the function.

4. **What is the purpose of return statement?**
   Return statement forces the function to return immediately, possibly before reaching the end of the function body.

5. **What are the different ways of passing arguments to a function?**

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by reference	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

6. **What are the uses of Pointers?**

Pointers are used to return more than one value to the function

- Pointers are more efficient in handling the data in arrays
- Pointers reduce the length and complexity of the program
- They increase the execution speed
- The pointers save data storage space in memory

7. **Differentiate Iteration and Recursion.**

Iteration	Recursion
Recursion Uses selection structure	Iteration uses repetition structure
Infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on some condition.(base case)	An infinite loop occurs with iteration if the loop-condition test never becomes false
Recursion terminates when a base case is recognized	Iteration terminates when the loop-condition fails
Recursion is usually slower then iteration due to overhead of maintaining stack	Iteration does not use stack so it's faster than recursion
Recursion uses more memory than iteration	Iteration consume less memory
Infinite recursion can crash the system	infinite looping uses CPU cycles repeatedly

8. **What is the difference between an array and pointer?**

Array	Pointer
An array is a collection of variables of the same data type stored contiguously under a common name.	A pointer is a variable that holds the address of a variable or a function
They are static in nature. Once memory is allocated , it cannot be resized or freed dynamically.	Pointer is dynamic in nature. The memory allocation can be resized or freed later.
Array can be initialized at definition. Example: int num[] = { 2, 4, 5}	Pointer can’t be initialized at definition.
Size of(array name) gives the number of bytes occupied by the array.	Sizeof(pointer name) returns the number of bytes used to store the pointer variable.

9. **What are * and & operators?**

‘*’ operator is called ‘value at the address’ operator

‘&’ operator is called ‘address of’ operator, gives address of the variable.

4. STRUCTURE, UNION AND FILES

Structure

A structure is a user defined data type. Arrays can be used to represent a group of data items that belong to the same type, such as int or float. Structures are used to represent a collection of data items of different types using a single name.

Syntax:

Structure definition:

```
struct structure_name
 {
 datatype element 1;
datatype element 2;
 …………….
datatype element n;
 };
```

Structure variable declaration:

```
struct structure_name var1,var2,…..,var n;
```

For Example:

```
struct student
      {
      int rollno;
      char name[25];
      float totalmark;
      };
 struct student stud1,stud2;
```

Thus, the stud1 and stud2 are structure variables of type student. The above structure can hold information of 2 students.

Initializing Structure Members:

```
struct stucture_name  var={val1,val2,val3…..};
```

Union

Union is a data type with two or more member similar to structure but in this case all the members share a common memory location. The size of the union corresponds to the length of the largest member. Since the member share a common location they have the same starting address.

The real purpose of unions is to prevent memory fragmentation by arranging for a standard size for data in the memory. By having a standard data size we can guarantee that any hole left when dynamically allocated memory is freed will always be reusable by another instance of the same type of union. A union is declared in the same way as a structure.

The syntax of union declaration is

```
union union_name
        {
         type element 1;
         type element 2;
         …………….
```

```
                type element n;
               };
```
This declares a type template. Variables are then declared as:

```
                    union union_name x,y,z;
```

For example, the following code declares a union data type called Student and a union variable called stud:

```
        union student
                {
                int rollno;
                float totalmark;
                };
        union student stud;
```

File Handling

Files used for storing information, which can be processed by our programs. In order to store information permanently and retrieve it, we need to use files. Files are not only used for data. Our programs are also stored in files.

Creating or Opening Files

Syntax:

```
    FILE * fopen (const char *filename, const char *mode);
```

The fopen () function opens a file whose name is pointed to by ‘filename’ and returns the stream that is associated with it. The type of operation that will be allowed on the file are defined by the vale of mode. The legal values of modes are shown in the following table.

File Type	Meaning
“r”	Open an existing file for reading only
“w”	Open a new file for writing only. If a file with the specified file-name currently exists, it will be destroyed and a new file created in its place.
“a”	Open an existing file for appending (i.e., for adding new information at the end of the file). If a file with the specified file-name currently does not exist, a new file will be created.
“r+”	Open an existing file for both reading and writing.
“w+”	Open a new file for both reading and writing. If a file with the specified file-name currently exists, it will be destroyed and a new file created in its place.
“a+”	Open an existing file for both reading and appending. If a file with the specified file-name currently does not exist, a new file will be created.

If fopen () is successful in opening the specified file, a FILE pointer is returned. If the file cannot be opened, a NULL pointer is required.

The following code uses fopen () to open a file named TEST for output.

```
FILE *fptr;
fptr = fopen("TEST","w");
```

Closing a File

To close a file, use the fclose( ) function. The prototype of this function is:

```
    int fclose( FILE *fp );
```

The fclose( ) function returns zero on success, or EOF if there is an error in closing the file.

Writing to a file

fputc()

Declaration: int fputc(int ch,FILE *stream);

The fputc() function writes the character ch to the specified stream at the current file position and then advance the file position indicator.

fputs()

Declaration: char * fputs(const char *str , FILE *stream);

The fputs() function writes the content of the string pointed to by str to the specified stream.

fscanf()

Declaration: fscanf (FILE *stream, const char *format,…..);

The fscanf function works exactly like the scanf function except that it reads the information from the stream specified by stream instead of standard input device.

Example:

```
fscanf(fptr,"%-15s%d%d%d%d%d%d%f",&name,&regno,&m[0],&m[1],&m[2],&m[3],&tot,&avg);
```

Reading from a file

fgetc()

Declaration: int fgetc(FILE *stream);

The fgetc() function returns the next character from the specified input stream and increments the file position indicator.

fgets()

Declaration: char * fgets(char *str,int num, FILE *stream);

The fgets() function reads up to num-1 character from stream and store them into a character array pointed to by str. Characters are read until either a newline or an EOF is received or until the specified limit is reached. After the character has been read, a null is stored in the array immediately after the last character is read. A newline character will be retained and will be a part of the array pointed to by str.

fprintf()

Declaration: fprintf (FILE *stream, const char *format,…..);

The fprintf() function outputs the values of the arguments that makes up the argument list as specified in the format string to the stream pointed to by stream.

Example:

```
 fprintf(fptr,"%15s%d%d%d%d%d%d%f\n",name,regno,m[0],m[1],m[2],m[3],tot,avg);
```

Functions for Random Access to Files

To randomly access file means only a particular part of a file is being accessed by using the following functions.

- Ftell
- Rewind
- Fseek

ftell() function

Ftell takes a file pointer and returns a number of type long that corresponds to the current position. This function is useful in saving the current position of a file, which can be used later in the program. It is used as follows:

```
N=ftell(fp);
```

N would give the relative offset (in bytes) of the current position. This means that n bytes have already been read.

rewind() function

Rewind takes a file pointer and resets the position to the start of the file.

For eg., the statements

```
rewind(fp);
N=ftell(fp);
```

Would assign 0 to n because has been set to the start of the file by rewind. This function helps us in reading a file more that once, without having to close and open the file.

fseek() function

fseek function is used to move the file position to a desired location within the file. Its syntax is:

```
Fseetk(fileptr, offset, position).
```

The file position indicator is set to offset bytes (characters) from a point in the file determined by the value of position. Offset may be negative.

The position parameter can be one of three values:

SEEK_SET – from the start;
SEEK_CUR – from the current position;
SEEK_END – from the end of the file.

4.a Find the difference between two dates

Aim:

Write a C program to create a structure called date with day, month and year as data member and find the difference between two dates.

Algorithm:

Input: Date of birth and current date

Output: Age in years, months and days

1. Declare a structure date with members day, month and year.
2. Declare three structure variables date_of_birth, today,difference;
3. Get date_of_birth and today's date
4. Compute the difference between the two dates.
    a. If it exceeds 30 convert to months and add quotient to months and store remainders as day of difference.
    b. Compute the difference between the two months.
    c. If it exceeds 12 convert to years and add quotient to years and store remainder as month of difference.
    d. Compute the difference between the two years and store it as year of difference.
5. Display the difference.

Sample I/0:

Input: Enter date of birth: 19 09 2003
Enter today's date:07 01 2014

Output: Your age: 9 Years 3 Months 18 Days

Result:

Thus the C program to create a structure called date with day, month and year as data member and find the difference between two dates is written and verified.

4.b Calculation of subject average of students using union

Aim:

Write a C Program to implement calculation of subject average of students using union.

Algorithm:

Input: Student Name, Register Number, Subject marks.

Output: Student Name, Register Number with average mark

1. Declare a union with the members register_number, subject1_mark, subject2_mark, subject3_mark and avg_marks.
2. Get the number of students in variable n.
3. Repeat the following for n students
    a. Get the student name, register number and three subject marks.
    b. Compute the average of marks
    c. Display Name, Register number and average mark

Sample I/O:

Input: Enter Student Name, Register Number, Subject1 mark, Subject2 mark, Subject3 mark: John, 23435355, 56, 67, 98

Output:

Student Name: John
Register Number: 23435355
Average mark : 73.67

Result:

Thus the C Program to implement calculation of subject average of students using union is written and is verified for output.

4.c Copying contents of one file to another file

Aim:

Write a C program to copy the contents of one file into another using fgetc and fputc function.

Algorithm:

Input: Text file
Output: Copied to another file

1. Open input file with read mode using file pointer
2. Open output file with write mode using another file pointer
3. Repeat the following until reaching end of input file
    a. Read a character from input file using getc function
    b. Write that character to the output file using putc function

Sample I/O:

Input :

```
Fruits.txt ( file in read mode)
        Apple
        Guava
        Pomegranate
        Litchi
Fruits1.txt (file in write mode)
        No contents
```

Output:

```
Fruits1.txt
        Apple
        Guava
        Pomegranate
        Litchi
```

Result:

Thus the C program to copy the contents of one file into another using fgetc and fputc function is written and is verified for output.

4.d Capitalize each word of a file

Aim:

Write a C program to capitalize first letter of each word in a file.

Algorithm:

Input: Text file

Output: Same file with capitalized every word

```
1. Open input file with read and write mode
2. Repeat the following until reaching end of input file
    a. Read a character from input file using getc function
    b. If it is next to the blank space or new line
        i. Capitalize the character using ascii values
        ii. Update the character using capital letter in the same file using putc
            function
    c. Otherwise move to the next character
```

Sample I/O:

Input:

Array.txt (file in both read and write mode)

An array is a collection of elements of the same data type. The elements of an array are stored in contiguous memory locations.

Output:

Array.txt

An Array Is A Collection Of Elements Of The Same Data Type. The Elements Of An Array Are Stored In Contiguous Memory Locations.

Result:

Thus the C program to capitalize first letter of each and every word in a file is written and is verified for output.

PRACTICE EXERCISES

1. Define a structure called cricket that will describe the following information:

   Player Code, Player Name, Team Name and Batting Average

   Using cricket, declare an array player with ten elements and develop a C program to read the information about all the team wise list containing names of players with their batting average.

2. Use an integer called mode. When mode equals one, display your name. If mode is two, then display your department. This mode variable needs to be declared as a member in the union. Write a C program and Test your program and report the results obtained.

3. Write a C Program to replace a specified line in a text file.
4. Write a C Program to merge lines alternatively from 2 files & print result.
5. Write a C Program to reverse the contents of a file and print it.

VIVA QUESTIONS

1. **What is the difference between structure and union?**
   In structures, value assigned to one member will not cause the change in other members. This is not the case with union. In unions, the value assigned to one member may cause the change in value of other member. A main disadvantage is, all the union member variables cannot be initialized or used with different values at a time.

2. **Compare arrays and structures.**

Arrays	Structures
An array is a collection of related data elements of same type.	Structure can have elements of different types
An array is a derived data type	A structure is a programmer-defined data type
Any array behaves like a built-in data types. All we have to do is to declare an array variable and use it.	But in the case of structure, first we have to design and declare a data structure before the variable of that type are declared and used.

3. **What is a bit-field in structure?**
   The variables defined with a predefined width are called bit fields. A bit field can hold more than a single bit for example if you need a variable to store a value from 0 to 7 only then you can define a bit field with a width of 3 bits as follows:

```
struct
{
  unsigned int age : 3;
} Age;
```

   The above structure definition instructs C compiler that the age variable uses only 3 bits to store the value.

4. **What is the used of typedef in structure?**
   typedef defines and names a new type, allowing its use throughout the program. typedefs usually occur just after the #define and #include statements in a file.

```
(e.g)  typedef struct {
          char name[64];
          char course[128];
          int age;
          int year;
          } student;
```

   This defines a new type student; variables of type student can be declared as follows: student st_rec;

5. **Is it possible for a structure to have an instance of its own?**

No. Structure cannot have an instance of its own. For example,

```
struct regression
{
    int int_member;
    struct regression self_member;
   };
```

The following error messages will be dispalyed

```
struct.c: In function `main':
struct.c:8: field `self_member' has incomplete type
```

6. **Differentiate Binary file and Text file.**

Binary File	Text File
A binary file is treated as raw data and read byte-by-byte.	A text file is considered to contain lines of text that are separated by some end-of-line markings.
Newline handling conversions will not takes place	A newline character is converted into carriage return-line feed combination and vice versa, before being return to the disk
Storage of numbers occupy same disk space as memory	Storage of numbers require more disk space than in memory

7. **Describe the use of fseek, ftell and rewind functions.**

The *fseek*() function shall set the file-position indicator for the stream pointed to by *stream*. If a read or write error occurs, the error indicator for the stream shall be set and *fseek*() fails.

The ftell() function obtains the current value of the file position indicator for the stream pointed to by *stream*

The rewind() function sets the file position to the beginning of the file for the stream pointed to by *stream*. It also clears the error and end-of-file indicators for *stream*.

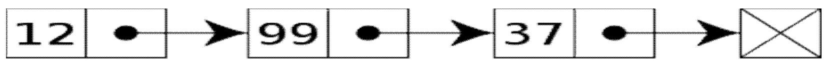
8. **What is the output of the following program?**

```
#include<stdio.h>
#include<string.h>
int main()
{
  char str[50];
  FILE *fp;
  fp=fopen(strcpy(str,"myfile.txt"),"w");
  fclose(fp);
}
```

5. Implementation of list ADT using singly linked list

A linked list is a data structure that consists of a sequence of data records wherein each record contains a field that contains a reference (i.e., a link) to the next record in the sequence apart from its data fields. Each record of a linked list is often called an element or a node. The head of a list is its first node, and the tail is the list minus that node (or a pointer thereto). A linked list is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of a data and a reference (in other words, a link) to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence. Linked lists can be used to implement other common abstract data types, including lists (the abstract data type), stacks, queues, and associative arrays.

The main benefit of a linked list over a conventional array is that the list elements can easily be inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal. A singly linked list contain nodes which have a data field as well as a next field, which points to the next node in the linked list. The diagram below shows a sample singly linked list in which each node contains two fields: a integer values and a link to the next nodes.



Applications:

- Applications where sequential access is required.
- In situations where the number of elements cannot be predicted beforehand

For representing polynomials ad sparse matrices

Aim:

Write a C Program to implement List ADT operations such as insert, delete, deletelist, find, and isempty using singly linked list.

Algorithm:

```
1.   Create a structure NODE with a data element and a pointer of type NODE
2.   Create a header node
3.   Accept the user choice Insert, Delete, DeleteList Find, and Display
4.   If the Choice is Insert
         a. Accept the data
         b. Find the position in the list for the new data
         c. Allocate space for new node and store new data in it.
         d. Insert the new node in the correct position by updating pointers (assume
            the list is sorted)
5.   If the Choice is Delete
     a. Get the data to be deleted from the list.
```

```
          b. Remove the node containing data by updating the pointers
6.        If the Choice is DeleteList
             Delete all the data nodes in the list.
7.        If the choice is Find
          a. Accept the data x
          b. Traverse the list till the data x is found or end of list
          c. If found display "Element found"
8.        If the choice is Display then display all the elements by traversing the list
9.        Stop
```

Sample I/O:

```
Program - Implementation of List ADT using Linked list
1. Insert operation
2. Delete an element
3. Delete the list
4. Find operation
5. Display List elements
6. Quit
 Enter your choice 1
 Enter the element 11

1. Insert operation
2. Delete an element
3. Delete the list
4. Find operation
5. Display List elements
6. Quit
 Enter your choice 1
 Enter the element 5

1. Insert operation
2. Delete an element
3. Delete the list
5. Display List elements
6. Quit
 Enter your choice 4
 Enter the element to be searched 55
Element not found

1. Insert operation
2. Delete an element
3. Delete the list
4. Find operation
5. Display List elements
```

```
6. Quit
 Enter your choice4
 Enter the element to be searched5
Element found

1. Insert operation
2. Delete an element
3. Delete the list
4. Find operation
5. Display List elements
6. Quit
 Enter your choice5
5       11
```

RESULT:

The list ADT is implemented using singly linked list for the operations insert, delete, find, findprevious, deletelist and isempty and is executed and verified for sample input.

VIVA QUESTIONS

1. Why do we need data structures?
2. Define: data structure
3. Name any four data structures.
4. State the advantage of linked lists over arrays.
5. What is a singly linked list?
6. What is the C library function used for dynamic memory allocation?

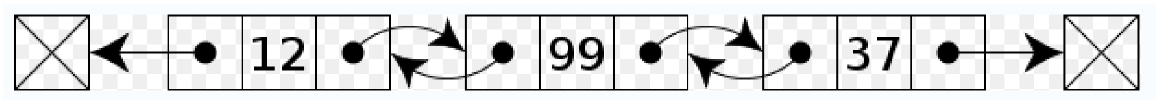
6. Implementation of list ADT using doubly linked list

Aim:

Write a C Program to implement List ADT operations such as insert, delete, deletelist, find, and isempty using doubly linked list.

Description:

In a doubly-linked list, each node contains, besides the next-node link, a second link field pointing to the previous node in the sequence. The two links may be called forward link (flink) and backward link (blink).



A doubly-linked list whose nodes contain three fields: an integer value, the link forward to the next node, and the link backward to the previous node.

Applications:

- Applications where sequential access is required.
- In situations where the number of elements cannot be predicted beforehand
- For representing polynomials ad sparse matrices

Algorithm:

1. Create a structure NODE with a data element ,flink, blink
2. Create a header node
3. Accept the user choice Insert, Delete, DeleteList Find, and Display
4. If the Choice is Insert
    a. Accept the data
    b. Find the position in the list for the new data
    c. Allocate space for new node and store new data in it.
    d. Insert the new node in the correct position by updating pointers (assume the list is sorted)
5. If the Choice is Delete
    a. Get the data to be deleted from the list.
    b. Remove the node containing data by updating the pointers
6. If the Choice is DeleteList
    i. Delete all the data nodes in the list.
7. If the choice is Find
    a. Accept the data x
    b. Traverse the list till the data x is found or end of list
    c. If found display "Element found"
8. If the choice is Display then display all the elements by traversing the list

Sample I/O:

```
Program-Implementation of Doubly Linked List
1.Insert operation
2.Delete an element
3.Delete the list
4.Find operation
5.Display List elements
6.Quit
Enter your choice 5
10      20      30      40      50
1.Insert operation
2.Delete an element
3.Delete the list
4.Find operation
5.Display List elements
6.Quit
Enter your choice 2
 Enter the element to be deleted 30
1.Insert operation
2.Delete an element
3.Delete the list
4.Find operation
5.Display List elements
6.Quit
Enter your choice 5
10      20      40      50
```

Result:

The list ADT is implemented in C using doubly linked list for the operations insert, delete, find, deletelist and isempty and is executed and verified for sample input.

VIVA QUESTIONS

1. What is a doubly linked list?
2. Mention its advantage over singly linked list.
3. Name any one application which involves sequential access of data.
4. State the use of adding header nodes in linked list.
5. What is a circular linked list?

7. Implementation of polynomial ADT using linked list

Aim:

Write a C Program to implement polynomial ADT's addition operation using linked list implementation of list.

Description:

Single-variable polynomials can be implemented using arrays if most of the coefficients are non-zero. But if we have more zero coefficients the running time becomes unacceptable as we spent most time in multiplying zeros ad stepping through nonexistent parts of input polynomials. An alternative way is to use a singly linked list. Each term in the polynomial is contained in one cell and the cells are sorted in decreasing order of exponents.

Algorithm:

1. Define the node structure
2. Create a header node for the first polynomial.
3. Get first polynomial term's coefficients and powers from the user (assume terms/nodes are sorted by exponent) .
4. For each term of the polynomial, create a node and store the coefficient and power.
5. Link the above nodes to the linked list, for first polynomial.
6. Repeat the steps 2 to 5 for the second polynomial.
7. Make 2 pointers to point to first term of both the polynomials.
8. If both the polynomials have terms, do the following
   a. If the exponents of both the terms are equal then add the coefficients.
      i. Create a node of resultant polynomial
      ii. Store the added coefficient and exponent in the node and link it to the resultant linked list.
      iii. Update the pointers to point to next node in each polynomial.
   b. If the exponent of first polynomial is greater than the second
      i. Then copy the node from the first polynomial to the resultant polynomial
      ii. Update the pointer of the first polynomial to point to next node
   c. If the exponent of second polynomial is greater than the first
      i. Then copy the node from the second polynomial to the resultant polynomial
      ii. Update the pointer of the second polynomial to point to next node
9. If only the first polynomial has terms then copy the remaining terms of it to the resultant polynomial.
10. If only end of second polynomial has terms then copy the remaining terms of it to the resultant polynomial.
11. Print the resultant polynomial.

Sample I/O:

```
Polynomial Addition
Create the First polynomial
Enter no. of elements in the polynomial3
Note: Enter nodes sorted by exponent

 Enter the exponent4
 Enter the coefficient11
 Enter the exponent2
 Enter the coefficient10
 Enter the exponent0
 Enter the coefficient7

 Create the Second polynomial
 Enter no. of elements in the polynomial2

 Note: Enter nodes sorted by exponent

 Enter the exponent4
 Enter the coefficient3

 Enter the exponent1
 Enter the coefficient66

11x^4 + 10x^2 + 7x^0
3x^4 + 66x^1
14x^4 + 10x^2 + 66x^1 + 7x^0
```

Result:

The polynomial ADT is implemented in C using Linked list to perform polynomial addition and is executed ad verified.

VIVA QUESTIONS

1. Represent the following polynomial using array and linked list
   $3x^5+7x^3+6x^2+4x^0$
2. Mention the drawback of using arrays for polynomial representation when compared to linked list implementation.
3. Write the steps to perform polynomial multiplication.

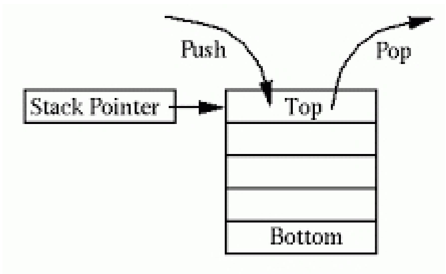
8. Implementation of stack ADT using linked list

Aim:

Write a C program to implement stack ADT to perform operations such as push, pop and top using linked list implementation.

Description:

A stack is a list in which insertions and deletions can be performed only at the end of the list called top. Stack list is also called a last-in-first-out (LIFO) list. The fundamental operations on a stack ADT are push which is inserting an element on the top and pop which is deleting the most recently inserted element. A stack is represented as follows:



A LIFO Stack

A simple singly linked list is sufficient to implement a stack—it only requires that the head node or element can be removed, or popped, and a node can only be inserted by becoming the new head node. The push() operation both initializes an empty stack, and adds a new node to a non-empty one. It works by receiving a data value to push onto the stack, along with a target stack, creating a new node by allocating memory for it, and then inserting it into a linked list as the new head. A pop() operation removes the head from the linked list, and assigns the pointer to the head to the previous second node. It checks whether the list is empty before popping from it.

Algorithm:

1. Create a structure with an element (Element) and a pointer (Next) that points to the next node in the list.
2. Create a head pointer using malloc function and assign the resultant pointer variable to the head pointer S.
3. In the Push operation, create a new node, called TmpCell using the malloc function.
4. Assign the new element X to TmpCell->Element, make TmpCell point to the node that was previously pointed to by the head pointer.
5. Make the head pointer point to TmpCell by the statements: TmpCell->Next=S->Next and S->Next=TmpCell.
6. In the Pop operation, store the node pointed to by the head pointer in a pointer called FirstCell by the statement FirstCell=S->Next.
7. Make the head pointer point to the node next to FirstCell by the statement S->Next=S->Next->Next.
8. Free FirstCell.
9. In the Top operation, return the element pointed to by the head pointers next node by the statement S->Next->Element.

Sample I/O:

```
Implementation of stack using linked list

1. Push operation
2. Pop operation
3. Top operation
4. Display stack elements
5. Quit
Enter your choice: 1
Enter the element:  5
Enter your choice: 1
Enter the element:  6
Enter your choice: 1
Enter the element:  7
Enter your choice: 4
7 6 5
Enter your choice: 2
Enter your choice: 4
6 5
Enter your choice: 3
6
Enter your choice: 2
Enter your choice: 2
Stack is empty
Enter your choice: 2
Empty stack. Cannot delete.
```

Result:

Thus the C program to implement stack ADT to perform operations such as push, pop and top using linked list implementation is written and is verified for output.

VIVA QUESTIONS

1. Define: Stack
2. List the operations that can be performed using stack.
3. List any four applications of stack.
4. Write the steps to do push and pop operations using arrays.
5. How stack is used in recursive programming?

9. Implementation of queue ADT using linked list

Aim:

Write a C program to implement queue ADT to perform operations such as enqueue, dequeue and front using linked list implementation.

Description:

A queue is an ordered list in which all insertions take place at one end, the rear, while all deletions take place at the other end, the front. A queue is also called a FIFO (First In First Out) to demonstrate the way it accesses data. The basic linked list implementation uses a singly-linked list with a tail pointer to keep track of the back of the queue. The basic operations are inserting an element (enqueue) and deleting an element (dequeue). To enqueue an element add it to the back of the item pointed to by the tail pointer. So the previous tail is considered next compared to the item being added and the tail pointer points to the new item. If the list was empty, this doesn't work, since the tail iterator doesn't refer to anything. The front item on the queue is just the one referred to by the linked list's head pointer. To dequeue an element, point the head pointer to the previous from head item. The old head item is the one you removed of the list. If the list is now empty, we have to fix the tail iterator.

Algorithm:

```
1. Create a structure with an element (Element) and a pointer (Next) that points to
   the next node in the list.
2. Create a head pointer using malloc function and assign the resultant pointer
   variable to the head pointer Q.
3. During the Enqueue operation of a new element X:
    a. Create a new node called TmpCell using the malloc function. Assign the
       element to that node by the statement: TmpCell - > Element = X and assign
       TmpCell-> Next to NULL.
    b. If, at the time of insertion, Q->Next is NULL ( i.e. the queue is initially
       empty)assign Q->Next to TmpCell.
    c. Otherwise, slide a pointer to the node, P, to the end of the queue by the
       statement P=P->Next until P->Next!=NULL and assign P->Next= TmpCell.
4. During the Dequeue operation:
    a. Store the value of Q->Next in a temporary pointer FirstCell.
    b. Make Q->Next = Q-> Next->Next.
    c. Free the memory contents of FirstCell.
```

Sample I/O:

```
Implementation of queue using linked list

1. Enqueue operation
2. Dequeue operation
3. Front operation
4. Display queue elements
5. Quit
Enter your choice: 1
Enter the element:  15
Enter your choice: 1
Enter the element:  12
Enter your choice: 1
Enter the element:  7
Enter your choice: 4
15 12 7
Enter your choice: 2
Enter your choice: 4
15 12
Enter your choice: 3
15
Enter your choice: 2
Enter your choice: 2
Queue is empty
Enter your choice: 2
Empty queue. Cannot delete.
```

RESULT: Thus the C program to implement queue ADT to perform operations such as enqueue, dequeue and front using linked list implementation is written and is verified for output.

VIVA QUESTIONS

1. Define: queue
2. List the operations that can be performed using queue.
3. Write the steps to do push and pop operations using arrays.
4. State the conditions to test for an full queue and empty queue.
5. List any four applications of queue.
6. What is a circular queue?
7. Compare dequeue and deque.

10. a. Conversion of infix expression to postfix expression using stack

Aim:

Write a C program to convert an infix expression to postfix expression using Stack ADT.

Description:

Stacks are widely used in the design and implementation of compilers. For example, they are used to convert arithmetic expressions from infix notation to postfix notation. There are three types of expressions namely:

- Infix expression – operator appears between the operands (e.g 2*3). This is how we used to write expressions in standard mathematical notation.
- Prefix expression – operator appears before the operands (e.g *23)
- Postfix expression – operator appears after the operands (e.g 23*). Postfix expressions are also called reverse polish notation.
- The purpose of the stack is to reverse the order of the operators in the expression. It also serves as a storage structure, since no operator can be printed until both of its operands have appeared.

There are no precedence rules to learn, and parentheses are never needed in postfix expressions. Because of this simplicity, some popular hand-held calculators use postfix notation to avoid the complications of the multiple parentheses required in nontrivial infix expressions. Because of this simplicity they are preferred in computer systems.

Applications:

- Converting a decimal number into a binary number
- Towers of Hanoi
- Expression evaluation and syntax parsing
- Conversion of an Infix expression that is fully parenthesized into a Postfix expression

Algorithm:

```
1. Scan the Infix string from left to right.
2. Initialise an empty stack.
3. If the scannned character is an operand, add it to the Postfix string.
4. If the scanned character is an operator and if the stack is empty Push the character
   to stack.
     a. If the scanned character is an Operator and the stack is not empty,
          i. Pop all operators having higher precedence over scanned character
             and add it to the postfix string.
         ii. Push the scanned character to stack.
5. Repeat step 4 & 5 till all the characters are scanned.
          i. If stack is not empty pop all operators in the stack and add it to the
             postfix string.
6. Return the Postfix string.
```

Sample I/O:

```
Enter Infix Expression : A + B + C / (E - F)
Postfix Expression is : A B + C E F - / +
```

Result:

Thus the conversion of an infix expression to postfix expression using stack ADT is implemented in C and is executed and verified.

VIVA QUESTIONS

1. What are the different types of expressions?
2. Why do compilers convert infix expressions to postfix expressions?
3. Convert the following infix expression to postfix expression
   (A*(B+D)/E-F*(G+H/K)
4. Convert the infix expression to its prefix form.
   (a+b*c-d)/(e*f-g)

10. b. Expression evaluation using stack ADT

Aim:

Write a C program to perform postfix expression evaluation using Stack ADT.

Algorithm:

```
1. Get the postfix expression to be evaluated.
2. Initialize an empty stack.
3. Repeat the following till all the characters in the postfix expression are scanned
     a. If the scanned character is an operand, add it to the stack.
     b. If the scanned character is an operator
          i.   Pop the top 2 elements from the stack.
          ii.  Apply the operator to the operands.
          iii. Push the result into the stack.
4. Pop the result form the stack and display it.
```

Sample I/O:

```
Input:  Enter the postfix Expression : 23+52-+
Output:  Postfix Expression is :   8
```

Result:

Thus the postfix expression evaluation using stack ADT is implemented in C, executed and is verified for output.

VIVA QUESTIONS

1. Why postfix expressions are preferred by computer systems?
2. Show the steps in evaluating the postfix expression 4 5 7 2 + - *
3. Write the steps to evaluate a postfix expression.

11.a. Bubble sort

Aim:

Write a C Program to sort an array of elements in ascending order using bubble sort algorithm.

Algorithm:

```
    Input  : An array of elements
    Output : Sorted array
 1. Get the array size in variable n
 2. Get the array elements in an array variable a
 3. Repeat the following  n-1 times (use a variable i to range from 0 to n-1)
     a. Repeat the following n-i times (use a variable j to range from 0 to n-i)
         i. If the jth element is greater than j+1th element ,swap the elements
 4. Repeat the following for n-1  times(use a variable i to range from 0 to n-1)
     a. Display the element at the corresponding array location
```

Sample I/O:

```
Input: Enter the no. of elements:  8
       Enter the array elements: 30 52 29 87 63 27 18 54
Output: Original array: 30 52 29 87 63 27 18 54
       Sorted array (ascending): 18 27 29 30 52 54 63 87
```

Result:

Thus, the C program to sort an array of elements in ascending order using bubble sort algorithm is written and is verified for output.

11.b. Insertion sort

Aim:

Write a C Program to sort an array of elements in ascending order using insertion sort algorithm.

Algorithm:

```
    Input  : An array of elements
    Output : Sorted array
 1. Get the array size in variable n
 2. Get the array elements in an array variable a
 3. Repeat the following  n times (use a variable i to range from 1 to n)
     a. Store the kth element of array in a variable temp
     b. Set a variable j to k-1
     c. Repeat until temp is less than or equal to jth element of array
         i. Store jth element of array as j+1th element
        ii. Decrement j by 1

     d. Store temp as j+1th element of array
 4. Repeat the following for n-1  times(use a variable i to range from 0 to n-1)
     a. Display the element at the corresponding array location
```

Sample I/O:

```
Input:  Enter the no. of elements:  7
        Enter the array elements: 39 9 45 63 18 81 36
Output: Original array: 39 9 45 63 18 81 36
        Sorted array (ascending): 9 18 36 39 45 63 81
```

Result:

Thus, the C program to sort an array of elements in ascending order using insertion sort algorithm is written and is verified for output.

11.c. Selection sort

Aim:

Write a C Program to sort an array of elements in ascending order using selection sort algorithm.

Algorithm:

Input : An array of elements
Output : Sorted array

```
1. Get the array size in variable n
2. Get the array elements in an array variable a
3. Repeat the following n-1 times( use a variable i to range from 0 to n-1)
4. Repeat the following for i+1 to n times ( use a variable j to range from i+1 to n)
   a. If ith element is greater than jth element, swap the elements
5. Repeat the following for n-1 times(use a variable i to range from 0 to n-1)
   a. Display the element at the corresponding array location
```

Sample I/O:

```
Input:  Enter the no. of elements:  8
        Enter the array elements: 81 39 90 45 9 72 27 18
Output: Original array: 81 39 90 45 9 72 27 18
        Sorted array (ascending): 9 18 27 39 45 72 81 90
```

Result:

Thus, the C program to sort an array of elements in ascending order using selection sort algorithm is written and is verified for output.

VIVA QUESTIONS

1. What is sorting?
2. Why bubble sort algorithm is called so?
3. State the logic of insertion sort.
4. Is selection sort faster than insertion sort? Justify.
5. How does insertion sort works?

12.a. Shell sort

Aim:

Write a C Program to sort an array of elements in ascending order using shell sort algorithm.

Algorithm:

Input : An array of elements
Output : Sorted array

```
1. Get the array size in variable n
2. Get the array elements in an array variable a
3. Set a variable flag to 1 and another variable gap_size to n
4. Repeat the following  while flag is equal to 1 or gap_size is greater than 1
   a. Set flag to zero
   b. Set gap_size  as ( gap_size+1) divided by 2
   c. Repeat the following until a variable i is less than n-gap_size (start i with 0)
         i. If the array element at  i+gap_size  position is greater than ith
            element, Swap the elements
5. Repeat the following for n-1  times(use a variable i to range from 0 to n-1)
   a. Display the element at the corresponding array location
```

Sample I/O:

```
Input: Enter the no. of elements:  6
       Enter the array elements: 22 17 43 27 9 32
Output: Original array: 22 17 43 27 9 32
       Sorted array (ascending): 9 17 22 27 32 43
```

Result:

Thus, the C program to sort an array of elements in ascending order using shell sort algorithm is written and is verified for output.

12.b. Quick sort

Aim:

Write a C Program to sort an array of elements in ascending order using quick sort algorithm.

Algorithm:

Input : An array of elements
Output : Sorted array

1. Get the array size in variable n
2. Get the array elements in an array variable a
3. Pick the middle element from the list and call it the pivot.
4. Reorder the list such that all elements which are less than the pivot are on the left of pivot and all elements that are higher than the pivot are on the right of the pivot. After partitioning the list, the pivot is in its position.
5. Apply steps 1 and 2 recursively for the two sublists
6. Repeat the following for n-1 times(use a variable i to range from 0 to n-1)
    a. Display the element at the corresponding array location

Sample I/O:

```
Input:  Enter the no. of elements:  7
        Enter the array elements: 63 19 7 90 82 36 54
Output: Original array: 63 19 7 90 82 36 54
        Sorted array (ascending): 7 19 36 54 63 82 90
```

Result:

Thus, the C program to sort an array of elements in ascending order using quick sort algorithm is written and is verified for output.

VIVA QUESTIONS

1. State the core idea of shell sort.
2. Why shell sort is called diminishing increment sort?
3. What should be the increment sequence of shell sort?
4. What is divide-and-conquer strategy?
5. What is a pivot element?
6. What are the different ways to choose pivot?
7. Why quick sort is better than other sorting algorithms?

13. a. Merge Sort

Aim:

Write a C Program to sort an array of elements in ascending order using merge sort algorithm.

Algorithm:

Input : An array of elements

Output : Sorted array

1. Get the array size in variable n
2. Get the array elements in an array variable a
3. Divide the array into two sub-arrays
4. Sort the left sub-array
5. Sort the right sub-array
6. Merge the arrays obtained in step 4 and 5
7. Repeat the following for n-1 times(use a variable i to range from 0 to n-1)
   a. Display the element at the corresponding merged array location

Merge():

- Merge sorted arrays B and C into array A as follows:
- Repeat the following until no elements remain in one of the arrays:
  - compare the first elements in the remaining unprocessed portions of the arrays
  - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
- Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Sample I/O:

```
Input:  Enter the no. of elements:  9
        Enter the array elements:  25 63 19 34 7 93 87 36 54
Output: Original array: 25 63 19 34 7 93 87 36 54
        Sorted array (ascending): 7 19 25 34 36 54 63 87 93
```

Result:

Thus, the C program to sort an array of elements in ascending order using merge sort algorithm is written and is verified for output.

13.b. Radix Sort

Aim:

Write a C Program to sort an array of elements in ascending order using radix sort algorithm.

Algorithm:

Input : An array of elements
Output : Sorted array

```
1. Get the array size in variable n
2. Get the array elements in an array variable a
3. Find the largest number in the array a as large
4. Set a variable nop(number_of_pass) equal to the number of digits in large
5. Set pass to zero
6. Repeat the following while pass is less than or equal to nop-1
    a. Set i to zero (initialize buckets)
    b. Repeat the following while i is less than n-1
        i. Store the digit at pass^th place of the i^th element of array in a
           variable digit
        ii. Add the i^th element of the array to the bucket numbered digit
        iii. Increment the bucket counter of the bucket numbered digit
    c. Collect the number in the bucket into an array
7. Repeat the following for n-1  times(use a variable i to range from 0 to n-1)
    a. Display the element at the corresponding merged array location
```

Sample I/O:

```
Input: Enter the no. of elements:  9
       Enter the array elements:  345 654 924 123 567 472 555 808 911
Output: Original array: 345 654 924 123 567 472 555 808 911
       Sorted array (ascending): 123 345 472 555 567 654 808 911 924
```

Result:

Thus, the C program to sort an array of elements in ascending order using radix sort algorithm is written and is verified for output.

VIVA QUESTIONS

1. What are the basic steps involved in merge sort?
2. Why is merge sort preferred over quick sort for sorting linked lists?
3. What is the output of merge sort after the 3rd pass given the following sequence of numbers: 25 57 48 37 12 92 86 33.
4. State the logic of radix sort.
5. How will you determine the no. of passes required to sort a set of numbers?
6. Why radix sort is not used often?
7. List the applications of radix sort.

14. a. Linear Search

Aim:

Write a C Program to search an element in an array using linear search algorithm.

Algorithm:

Input : An array of elements

Output : Element found in Position or element not found

```
1. Get the array size in variable n
2. Get the array elements in an array variable a
3. Get the element to be searched in variable key
4. Set a variable flag to zero
5. Repeat the following for n-1 times (use a variable i to range from 0 to n-1)
    a. Check if the element at ith position of array is equal to key
        i. If yes, set flag to 1 and exit the loop
6. Check if flag is equal to zero
    a. If yes, display the key value is not present in the array
    b. Otherwise, display the key value is at ith position
```

Sample I/O:

```
Input: 21,13,9,11,17,25,56
Enter the element to be searched
17

Output: The element 11 is at position 3
Input: 21,13,9,11,17,25,56
Enter the element to be searched
33
Output: The element 33 is not present in the array
```

Result:

Thus, the C program to search an element in an array using linear search algorithm is written and is verified for output.

14.b. Binary Search

Aim:

Write a C Program to search an element in an array using binary search algorithm.

Algorithm:

```
1.   Get the element(search key) to be searched
2.   Compare the search key with the element at the middle
     2.1   If the value matches, return the value
     2.2   If the search key < the middle element, search the elements between
           the first element to the element before the middle element (MIDDLE -
           1)
     2.3   If the search key > the middle element, search the elements between
           the MIDDLE+ 1 elements to the last element
3    Search is repeated until the searched key is found or the last element in the
     subset is traversed
```

Sample I/O:

```
Input: 1,3,9,11,17,25,56
Enter the element to be searched
11
Output: The element 11 is at position 3
Input: 1,3,9,11,17,25,56
Enter the element to be searched
51
Output: The element 51 is not present in the array
```

RESULT:

Thus, the C program to search an element in an array using binary search algorithm is written and is verified for output.

VIVA QUESTIONS

1. Why linear search is called so?
2. State the prerequisite of binary search.
3. Binary search uses divide-and-conquer strategy. Justify.
4. What is the maximum number of key comparisons made when searching a list L of length n for an item using a binary search?

15. SIMPLE GRAPHICS PROGRAMS

15.a. Draw a triangle using built-in function

Aim:

Write a C Program to draw a triangle using built-in graphical functions.

Algorithm:

Input: Coordinates of the triangle

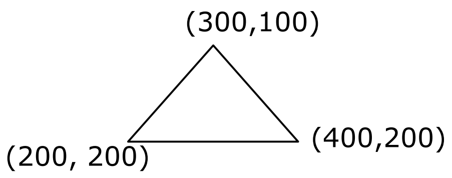
Output: Triangle image

1. Include the header file graphics.h
2. Detect the graphics driver and mode using DETECT macro
3. Initialized the graphics system using initgraph function
4. Use the built-in function line with starting point (x1,y1) and end point(x2,y2).
5. Exit from the graphics mode using closegraph function.

Sample I/O:

Input: Enter the coordinate of line1: 300 100 200 200
Enter the coordinate of line2: 300 100 400 200
Enter the coordinate of line2: 200 200 400 200

Output:



Result:

Thus the C program to draw a triangle using the built-in graphical function line is written and is executed for output.

15.b. Draw a house using built-in functions

Aim:

Write a C Program to draw house using built-in graphical functions.

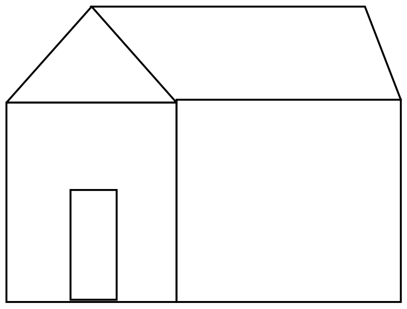
Algorithm:

Input: Coordinates of the house

Output: House image

1. Include the header file graphics.h
2. Detect the graphics driver and mode using DETECT macro
3. Initialized the graphics system using initgraph function
4. Use the built-in function line with starting point (x1,y1) and end point(x2,y2) and the rectangle function with 2 coordinates (left, top) and (right, bottom)
5. Exit from the graphics mode using closegraph function.

Sample Output:



Result:

Thus the C program to draw a house using the built-in graphical function line and rectangle is written and is executed for output.

16. HASHING TECHNIQUES

HASH FUNCTION

A hash function is any well-defined procedure or mathematical function which converts a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an index into an array. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

Hash functions are mostly used to speed up table lookup or data comparison tasks — such as finding items in a database, detecting duplicated or similar records in a large file, finding similar stretches in DNA sequences, and so on.

A hash function may map two or more keys to the same hash value. In many applications, it is desirable to minimize the occurrence of such collisions; which means that the hash function must map the keys to the hash values as evenly as possible

HASH TABLES

Hash functions are primarily used in hash tables, to quickly locate a data record (for example, a dictionary definition) given its search key (the headword). Specifically, the hash function is used to map the search key to the hash. The index gives the place where the corresponding record should be stored. Hash tables, in turn, are used to implement associative arrays and dynamic sets.

Hashing with Separate Chaining

Hash table is a popular and a very efficient way to implement dictionaries for keys with an available hash function.

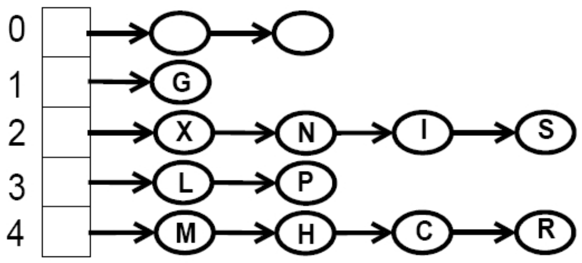
Hash Functions:

- Hash(X)=X mod TableSize
- Hash function is used to insert elements in an array. Insert element with key x into position Hash(x).
- When elements hash to same position collision occurs.
- Hashing with seperate chaining overcomes the collision problem by using buckets in each cell (lists of possible contents).

Example:

x: A S E R C H I N G X M P L

h(x): 0 2 0 4 4 4 2 2 1 2 4 3 3



Applications:

- In network security for message authentication
- Error checking
- Dynamic storage of databases

16. a. Implement hash table using separate chaining technique

Aim:

Write a C Program to implement hash table using separate chaining technique.

Algorithm:

Input: Table size and elements
Output: Hash table

1. Define the node (listnode)with a data element and pointer to the node.
2. Define a structure (hash table) with tablesize and pointer to the pointer of the listnode.
3. Allocate space for the hash table.
4. Create a header node for the lists in the hash table.
5. Insert new element into the hash table by applying hash function to the data and locate the corresponding list.
6. If collision occurs the data goes to the same list at the end.
7. To find a data element, apply the hash function and locate the list in the hash table, then locate the element in the identified list.

Sample I/O:

Input:

```
Collision handling by separate chaining
Table size is 10
enter the element to be inserted 6
enter the element to be inserted 12
enter the element to be inserted 25
enter the element to be inserted 15
enter the element to be inserted 11
enter the element to be inserted 17
enter the element to be inserted 9
enter the element to be inserted 22
enter the element to be inserted 7
enter the element to be inserted 36
```

Output:

```
Bucket 0:
Bucket 1:  11
Bucket 2:  22  12
Bucket 3:
Bucket 4:
Bucket 5:  15  25
Bucket 6:  36  6
Bucket 7:  7  17
Bucket 8:
Bucket 9:  9
```

Result:

Thus the C Program to implement hash table using separate chaining technique is written and is verified for output.

16. b. Implement hash table using linear probing technique

Aim:

Write a C Program to implement hash table using linear probing technique.

Algorithm:

Input: Table size and elements
Output: Hash table

```
1. Allocate an array(table) of size n (n depends on the number of elements).
2. Use the hash function Hash(x)+F(i)) mod tablesize to find the slot (key) for insertion
        i. If the slot is free insert the element.
```

ii. If not, search for free slots sequentially till end of the array is reached and insert the element in the free slot found.

iii. If no slot is free, then wrap around and search to find free slots till key-1.

   a. If a slot is found insert the element

   b. Otherwise display the table is full.

Sample I/O:

Input:

```
Collision handling by linear probing
Table size is 10
 Enter the element to be inserted: 5
 Enter the element to be inserted: 10
 Enter the element to be inserted: 11
 Enter the element to be inserted: 4
 Enter the element to be inserted: 15
 Enter the element to be inserted: 6
 Enter the element to be inserted: 12
 Enter the element to be inserted: 22
 Enter the element to be inserted: 8
 Enter the element to be inserted: 9
 Enter the element to be inserted: 2
Hash table is full
```

Output:

```
HASH TABLE
Slot   Element

0      10
1      11
2      12
3      22
4      4
5      5
6      15
7      6
8      8
9      9
```

RESULT

Thus the C program to implement hash table using linear probing method is implemented, executed and verified.