V V COLLEGE OF ENGINEERING

VVNagar,Arasoor,Tisaiyanvilai

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

OCS752 – Introduction to C Programming

Unit 1 to 5

Handwritten Notes

Prepared by
Mrs.K.Vishnu Priya
AP / CSE
VVCOE

OCS752 **Introduction to C Programming** **L T P C**

3 0 0 3

Objectives :

- To develop C Programs using basic programming constructs
- To develop C programs using arrays and strings
- To develop applications in C using functions and structures

UNIT I INTRODUCTION **9**

Structure of C program – Basics: Data Types – Constants –Variables - Keywords – Operators: Precedence and Associativity - Expressions - Input/Output statements, Assignment statements – Decision-making statements - Switch statement - Looping statements – Pre-processor directives - Compilation process – Exercise Programs: Check whether the required amount can be withdrawn based on the available amount – Menu-driven program to find the area of different shapes – Find the sum of even numbers

Text Book: Reema Thareja (Chapters 2,3)

UNIT II ARRAYS

Introduction to Arrays – One dimensional arrays: Declaration – Initialization - Accessing elements – Operations: Traversal, Insertion, Deletion, Searching - Two dimensional arrays: Declaration – Initialization - Accessing elements – Operations: Read – Print – Sum – Transpose – Exercise Programs: Print the number of positive and negative values present in the array – Sort the numbers using bubble sort - Find whether the given is matrix is diagonal or not.

Text Book: Reema Thareja (Chapters 5)

UNIT III STRINGS

Introduction to Strings - Reading and writing a string - String operations (without using built-in string functions): Length – Compare – Concatenate – Copy – Reverse – Substring – Insertion – Indexing – Deletion – Replacement – Array of strings – Introduction to Pointers – Pointer operators – Pointer arithmetic - Exercise programs: To find the frequency of a character in a string - To find the number of vowels, consonants and white spaces in a given text - Sorting the names.

Text Book: Reema Thareja (Chapters 6 & 7)

UNIT IV FUNCTIONS

Introduction to Functions – Types: User-defined and built-in functions - Function prototype - Function definition - Function call - Parameter passing: Pass by value - Pass by reference - Built-in functions (string functions) – Recursive functions – Exercise programs: Calculate the total amount of power consumed by ‘n’ devices (passing an array to a function) – Menu-driven program to count the numbers which are divisible by 3, 5 and by both (passing an array to a function) – Replace the punctuations from a given sentence by the space character (passing an array to a function)

T**ext Book: Reema Thareja (Chapters 4)**

UNIT V STRUCTURES

Introduction to structures – Declaration – Initialization – Accessing the members – Nested Structures – Array of Structures – Structures and functions – Passing an entire structure – Exercise programs: Compute the age of a person using structure and functions (passing a structure to a function) – Compute the number of days an employee came late to the office by considering his arrival time for 30 days (Use array of structures and functions)

Text Book: Reema Thareja (Chapters 8)

Outcomes:

- Develop algorithmic solutions to simple computational problems.
- Read, Write, execute by hand simple C programs.
- Structure with simple C Programs for solving problems using statements.
- Represent data using arrays and strings operations.
- Decompose a C program into functions and pointers
- Represent and write program using structure and union.

Unit - I

Introduction

Structure of C Program.

→ A C Program is divided into different sections.

→ There are six main sections to a basic program.

Documentation
Link Section
Definition Section
Global Declaration Section
Main Function
Sub Program Section.

(i) Documentation Section:

The documentation is the part of the program where the programmer gives the name of the program, details of the author and description of the program.

(ii) Link Section:

→ This part of the code is used to declare all the header files.

(iii) Definition Section :

→ This section is used to declare different constants.

```
#define PI = 3.14
```

(iv) Global Declaration Section :

→ All the global variables used are declared in this part.

(v) Main Function Section :

→ Every C- Programs has the main function. Each main function contains 2 parts.

* Declaration part
* Execution part.

→ Declaration part is the part where all the variables are declared.

(vi) Sub Program Section :

→ All the user-defined functions are defined in this section of the program.

Example :

```
int sum(int x, int y)
{ return x+y;
}
```

Sample Program:

Aim: A 'c' Program to print the area of square.

```
#include <stdio.h>
#include <conio.h>
void main()
{
  int side, area;
  printf("Enter the value of side");
  scanf("%d", &side);
  area = side * side;
  printf("The area of square is %d", area);
  getch();
}
```

Basic (i) Data Types:

The datatypes are used to define the type of data for particular variables.

Example:

Char, unsigned, signed char, int, unsigned int, signed int, short int, unsigned short int, signed short int, long int, signed long int, float, double, long double.

(ii) Constants :

→ Constants are identifiers whose value does not change.

→ Types :

* Integer Type
* Floating Point Type
* Character Type
* String Type.

Integer Type Constant :

→ A constant of integer type consists of a sequence of digits.

Eg : 1, 34, 586, 2802, etc.,

Floating Point Type Constant :

→ A constant of floating point consists of an integer part, a decimal part and an exponent field containing an e or E.

Eg : 0.02, −0.23, 123.345, ± 0.34 etc.

Character Constant :

→ Character constant consists of a single character enclosed in single quotes. Eg : 'a'.

String Constant :

→ It is a sequence of characters enclosed in double Quotes. Eg : "apple".

Declaring Constant.

```
#define PI 3.14159
```

Rules for declaring Constant.

Rule 1 : Constant names are usually written in Capital Letters.

Rule 2 : No blank spaces are permitted in b/w the # symbol and define keyword.

Rule 3 : Blank space must be used between #define and constant name and constant value.

Rule 4 : #define must not end with semi-colon.

(iii) Variables :

→ A variable is defined as a meaningful name given to the data storage location.

* Numeric Variables :

→ Numeric variable can be used to store either integer value or floating point values.

* Character Variables :

→ Character variable can include any letter from the alphabet or from the ASCII Chart and numbers 0-9 that are given within Single Quotes.

(iv) Keywords :

→ Keywords are special reserved words associated with some meaning.

Operators :

(i) Arithmetic Operators :

→ Addition, Subtraction, Multiplication, Division and Modulus (+, −, *, /, %).

(ii) Unary Operators :

→ Unary Plus Operator (+), Unary minus Operator (−), Increment Operator (++), Decrement Operator (--).

(iii) Relational Operators :

→ Relational operators are used to test condition and results true or false.

→ ==, !=, >, <, >=, <=

(iv) Logical Operators :

→ Logical Operators are used to combine more than one condition.

→ &&, ||, ! (AND, OR, NOT).

(v) Assignment Operator.

* Simple Assignment:

=, assigns right hand side value to left hand side variable.

* Compound Assignment:

+=, -=, /=, %=, &=, |=, ^=,

>>=, <<= assigns right hand side value after the computation to left hand side variable.

(vi) Bitwise Operators:

* Bitwise AND (&)
* Bitwise OR (|)
* Bitwise EXOR (^)
* Bitwise NOT (~ unary operator)
* Shift Left (<<)
* Shift Right (>>)

Expressions:

An expression is a formula in which operands are linked to each other by the use of operators to compute a value. An operand

can be a function reference, a variable, an array element or a constant.

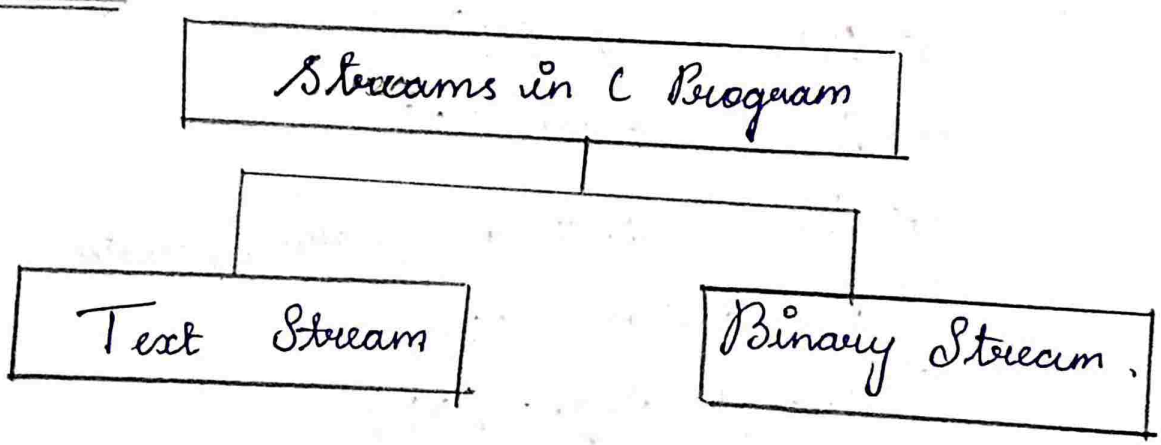
Example: x = 9/2 + a - b;

Input / Output Statements:

Input means to provide the program with some data to be used in the program.

Output means to display data on screen or write the data to a pointer or a file.

Streams.



Sample Program:

```
#include <stdio.h>
#include <conio.h>
void main()
{
  float i;
  printf("Enter the value");
```

```
    scanf ("%f", &i);
    printf ("The value is %f =", i);
    getch();
}
```

Putchar() & getchar() functions:

The getchar() reads a character from the terminal and returns it as an integer. This function reads only single character at a time.

The putchar() function displays the character passed to it on the screen.

Assignment Statements:

→ An assignment statement sets the value stored in the storage location denoted by a variable_name.

Syntax:

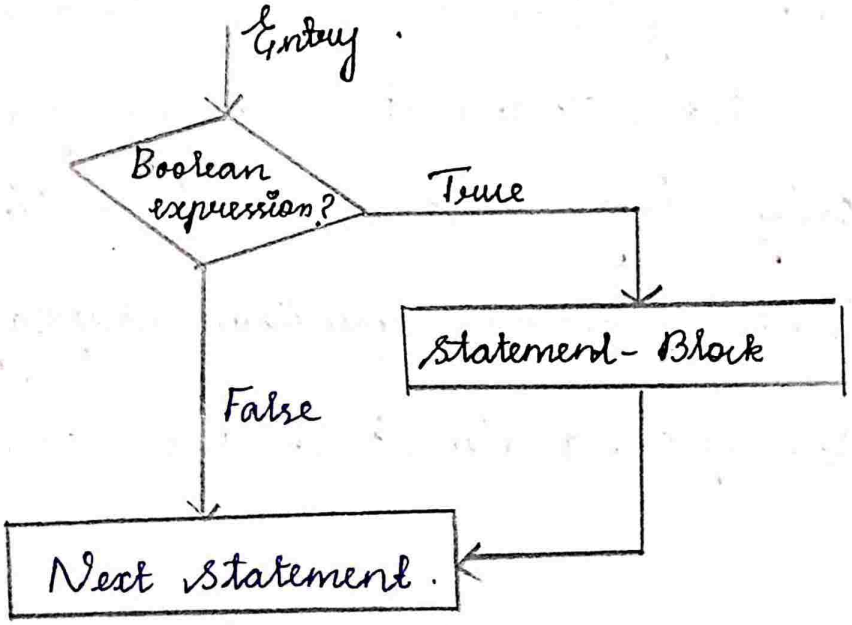
```
Variable = expression;
```

Decision Making Statements :

→ Decision making statements are mainly three types.

* if
* if .... else
* if .... else ... if .

Simple if .



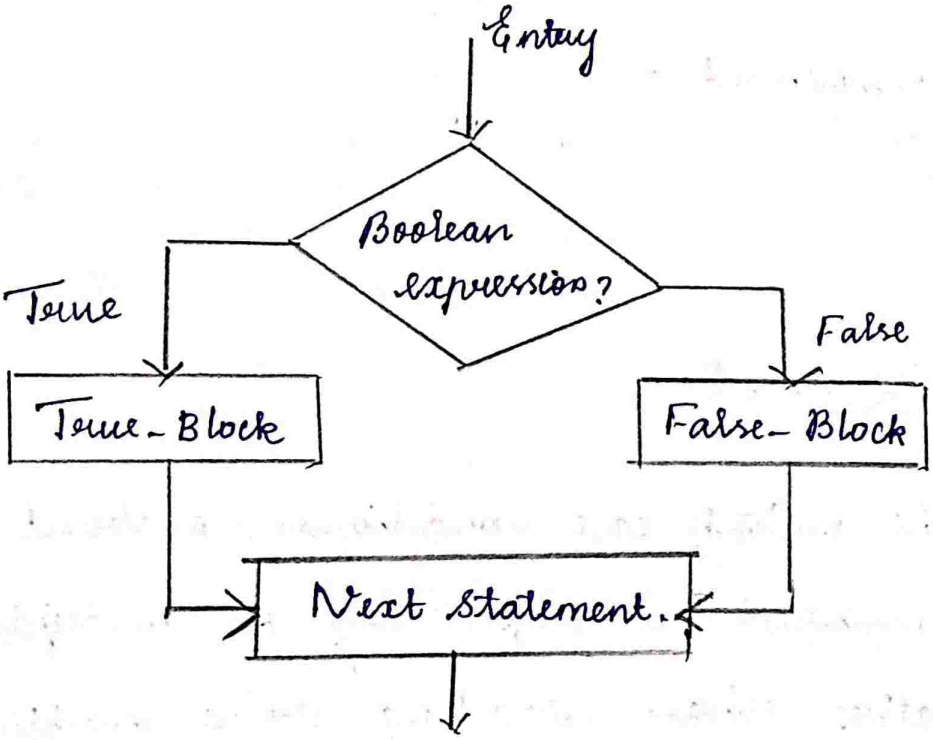
Syntax .

```
if( Boolean expression )
{
    Statement_block ;
}
Next statement ;
```

if ... else statement :

Syntax :

```
if (boolean expression)
{
    True-block statements);
}
else.
{
    False-block statements;
}
Next statement;
```



Cascading if.... else.

Syntax.

```
if (condition 1)
{
    statement 1;
}
.....
else if (condition)
{
    statement-n;
}
else
{
    default statement;
}
next statement;
```

Switch Statement.

→ The switch case conditional construct is a more structured way of testing for multiple conditions rather than resorting to a multiple if statement.

Syntax :

```
switch (expression)
{
   case 1 : Case 1 block ;
   break ;
   case 2 : case 2 block ;
   break ;
   default : default block ;
   break ;
}
   statement ;
```

Looping Statements :

→ A loop executes the sequence of statements many times until the stated condition becomes false.

* for
* while
* do .... while .

(i) for Loop.

→ The for loop initialize the value before the first step. Then checking the condition against the current value of variable and execute the loop statement and then perform the step taken for each execution of loop body.

Syntax:

```
for (initialization ; Condition ; increment/decrement)
{
    Body of the loop
}
```

(ii) While Loop :

It is a entry controlled loop, the condition in the while loop is evaluated, and if the condition is true, the code within the block is executed. This repeats until the condition becomes false.

Syntax :

```
while (condition)
{
    Body of the loop.
}
```

(iii) do... while Loop

It is a exitby exit controlled loop, the body of the loop gets executed first followed by checking the condition.

Continues with the body if the condition is true, else loops gets terminated.

Syntax.

```
do
{
   body of the loop
}
while (Boolean expression);
```

Preprocessor Directives:

→ This preprocessor is a macro processor that is used automatically by the C compiler to transform the program before actual compilation.

→ It is also called macro processor because it allows you to define macros, which

are brief abbreviations of longer constructs. A

→ A macro is a segment of code which is replaced by the value of macro.

→ Macro is defined by #define directive.

→ Preprocessing directives are lines in the program that starts with #

→ The # is followed by an identifier that is the directive name.

Some of the preprocessor directives are:

#include, #define, #undef, #ifdef, #if, #else, #elif, #endif, #error, #pragma.

(i) #include

It is used to paste code of given file into current file. It is used do

(ii) #define.

A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive

Syntax: `#define token value`

(iii) #undef

To undefine a macro is nothing but to cancel its definition.

Syntax:

```
#undef token
```

Example:

```
#include <stdio.h>
#define PI 3.1415
#undef PI
main()
{
  printf("%f", PI);
}
```

(iv) #ifdef

→ It checks if macro is defined by #define. If yes, it executes the code.

Syntax:

```
#ifdef MACRO
// code
#endif
```

(v) #if

→ The #if preprocessor directive evaluates the expression or condition.

→ If condition is true, it executes the code.

Syntax :

```
#if expression
// code
#endif
```

(vi) #else

→ The #else preprocessor directives evaluates the expression or condition, if the condition of #if is false.

→ It can be used with #if, #elif, #ifdef, and #ifndef directives.

Syntax :

```
#if
// code
#else
// else code
#endif.
```

(vii) #error

→ It indicates error.

→ The compiler gives fatal error if #error directive is found and skips further compilation process.

```
#include <stdio.h>
#ifndef _MATH_
# error First include then compile.
#else
Void main()
{
   int a;
   a= sqrt(9);
   printf("%f",a);
}
#endif.
```

(viii) #pragma

→ The #pragma preprocessor directive is used to provide additional information to the compiler.

→ It is used by the compiler to offer machine or Operating system feature.

Syntax: `#pragma token`

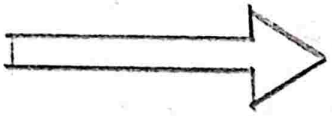
Compilation Process :

→ The compilation is the process of converting the source code into object code.

→ It is done with the help of compiler.

→ The compiler checks the source for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.

```
#include <stdio.h>
void main()
{
  printf("Hello c");
}
```



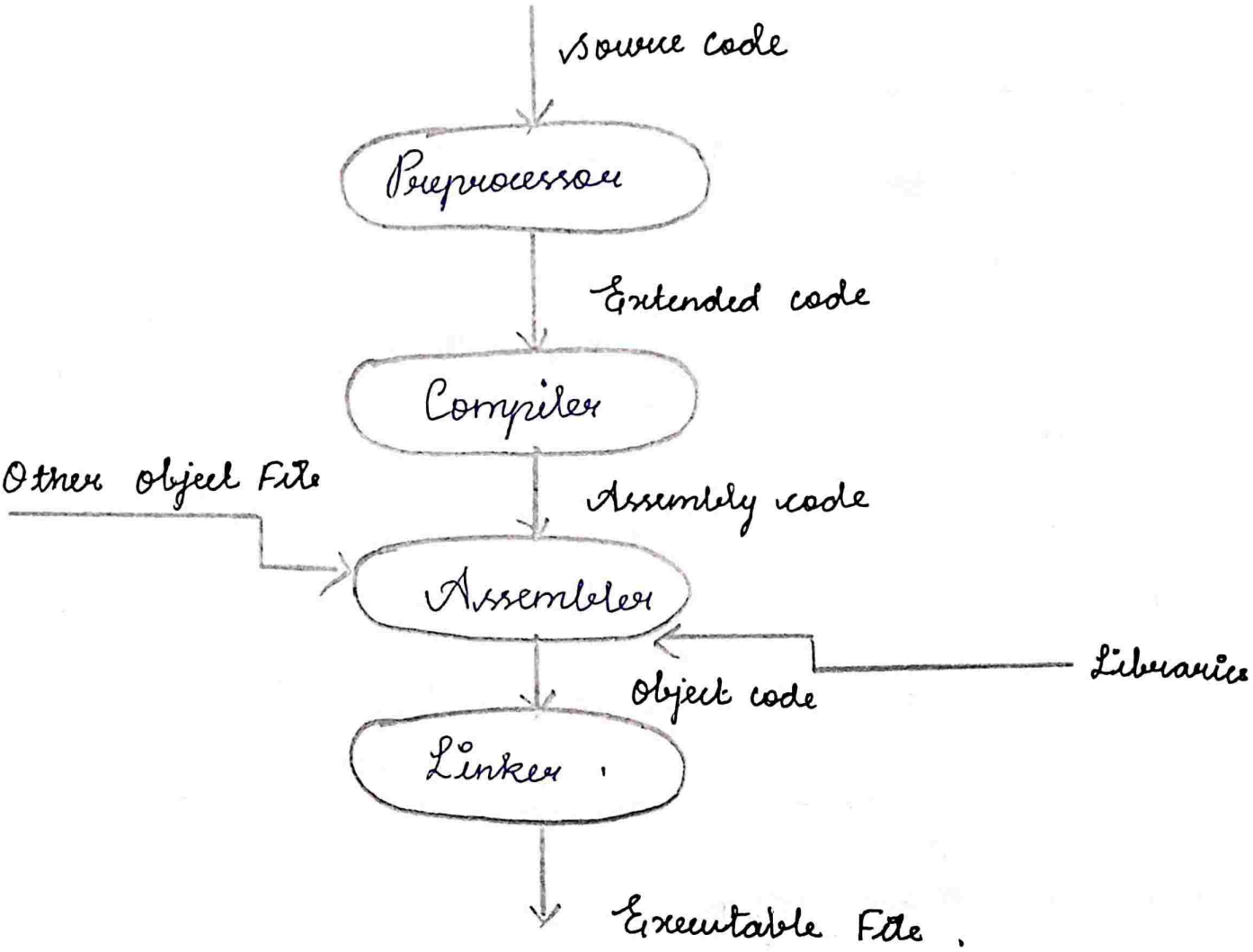
```
00000000 11010111 11
00110101 10010101
10101010 10001111

00100011 01100110

10101010 10101011 11
```

→ The 'c' compilation process converts the source code taken as input into the object code or machine code.

⇒ The compilation process can be divided into four steps (i.e) Pre-processing, Compiling, Assembling, and Linking.



(i) Preprocessor.

→ The source code is a code which is written in a text editor with ".c" as file extension.

→ It is first passed to the preprocessor, and then the preprocessor expands the code.

→ The expanded code is then passed to the compiler.

(ii) Compiler :

→ The compiler converts the expanded code into assembly code.

(iii) Assembler :

→ The assembly code is converted into object code by using an assembler.

→ The name of the object file generated by the assembler is the same as the source file.

→ The extension of the object file in DOS is '.obj', and in UNIX, the extension is 'o'.

→ If the name of the source file is 'welcome.c', then the name of the object file would be 'hello.obj'.

(iv) Linker :

→ The job of the linker is to link the object code of the program with the object code of the library files and other files.

→ The output of the linker is the executable file.

→ The name of the executable file is the same as the source file but differs only in their extensions.

→ In DOS, the extension of the executable file is '.exe', and in UNIX, the executable file can be named as 'a.out'.

Exercise Programs:

1. Check whether the required amount can be withdrawn based on the available amount.

```
#include<stdio.h>
Unsigned long amount = 1000, deposit, withdraw;
int ch, pin, i;
Char transaction = 'y';
Void main()
{
    while (pin != 9090)
    {
        printf ("Enter your secret Pin Number:");
        scanf ("%d", &pin);
        if (pin != 9090)
        printf ("Please enter valid password\n");
    }
    do
    {
        printf ("*** Welcome to ATM service ***");
        printf ("1. Check Balance\n");
        printf ("2. Withdraw Cash\n");
        printf ("3. Deposit Cash\n");
        printf ("4. Quit\n");
        printf ("**********\n\n");
        printf ("Enter your choice:");
```

```
vscanf("%d", &ch);
vswitch (ch)
{
    case 1 : vprintf ("\n YOUR BALANCE IN Rs: %lu", amount);
    break;
    case 2 :
        printf ("\n Enter the amount to withdraw :");
        vscanf ("%lu", & withdraw);
        if (withdraw %100 != 0)
        {
            printf ("\n Please Enter the Amount in
                        Multiples of 100");
        }
        else if (withdraw > (amount - 500))
        {
            printf ("\n Insufficient Balance");
        }
        else
        {
            amount = amount - withdraw;
            printf ("\n\n Please collect cash");
            printf ("\n Your Current Balance is %lu",
                                    amount);
        }
        break;
```

```
Case 3:
    printf("\n Enter the amount to Deposit");
    scanf("%.lu", & Deposit);
    amount = amount + deposit;
    printf("Your balance is %.lu", amount);
    break;
Case 4:
    printf("\n Thank You Using ATM");
    break;
default:
    printf("\n Invalid Choice");
}
printf("\n\n\n Do you want to Continue? (y/n):\n");
fflush(stdin);
scanf("%.c", & transaction);
if (transaction == 'n' || transaction == 'N')
l=1;
}
while(!l);
printf("\n\n Thanks for using out ATM service);
}
```

2. Menu - Driven Program to find the area of different shape.

```
#include<stdio.h>
void main()
{
    int ch, rad, length, width, breadth, height;
    float area;
    printf("Input 1 for area of circle \n");
    printf("Input 2 for area of rectangle \n");
    printf("Input 3 for area of triangle \n");
    printf("Input your choice:");
    scanf("%d", &ch);
    switch(ch);
    {
      case 1:
          printf("Input radius of the circle:");
          scanf("%d", &rad);
          area = 3.14 * rad * rad;
          break;
      case 2:
          printf("Input length and width of the
                  rectangle:");
          scanf("%d %d", &length, &width);
```

```
        area = length * width ;
        break ;
    case 3 :
        printf ("Input the base and height of the triangle:");
        scanf ("%d %d", &breadth, &height);
        area = 0.5 * breadth * height;
        break ;
    }
    printf ("The area is : %f\n", area);
}
```

3. Find the sum of seven numbers :

```
#include <stdio.h>
void main()
{
    int i, x, sum=0;
    /* Input upper limit from user */
    printf ("Enter upper limit : ");
    scanf ("%d", &x);
    for(i=2, i<=x; i+=2)
    {
        /* Add current seven numbers to sum */
        sum = sum+i ;
    }
    printf ("Sum of all seven number between 1 to %d
                                    = %d", x, sum);
}
```

Unit-II

Introduction to Arrays.

Array Introduction:

→ An Array is a collection of similar data elements.

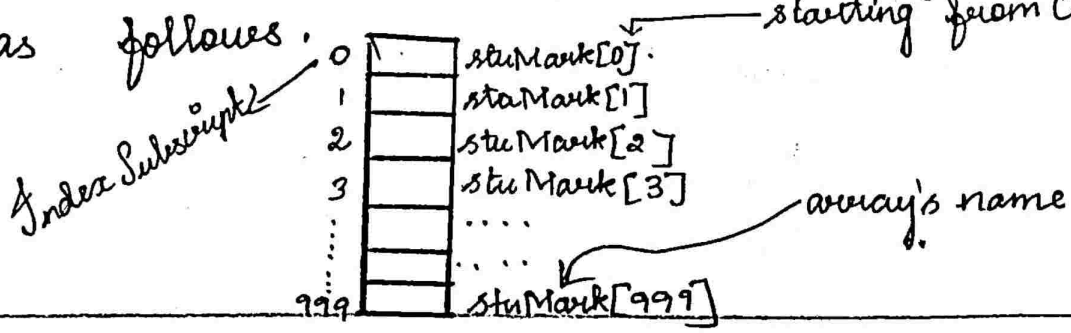
→ These data elements have the same data type.

→ The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as subscript).

→ If one subscript, then we call as one Dimensional Array.

Memory Representation in an Array:

→ The Array elements are stored in contiguous memory locations. For the array, int stuMark[ ] = {43, 70, 56}; the memory representation shown as follows:



→ By using an array, we just declare this,

```
int studMark[1000];
```

→ This will reserve 1000 contiguous memory locations for storing the students marks.

→ Compared to the basic data type (int, float, char and double), it is an aggregate or derived data type.

→ All the elements of an array occupy a set of contiguous memory locations.

—x—

One Dimensional Array.

Dimensions refers to the array's size, which is how big the array is,

Declaration of One Dimensional Array:

```
type name[size];
```

Example:

```
Char cName[30];
```

Initialization of an array:

An array may be initialized at the time of declaration

For example:

```
int idNum[7] = {1,2,3,4,5,6,7};
float fFloatNum[5] = {5.6,5.7,5.8,5.9,5.10};
```

Store Values in the array.

→ 3 possible ways.

1) Initialize the elements.
2) Inputting values for the elements.
3) Assigning values to the elements.

Accessing Elements.

To access all the elements of the array, you must use a loop.

```
int i, marks[10];
for(i=0; i<10; i++)
    marks[i] = -1;
```

Calculating the address of array elements.

Address of the data element,

$$A[K] = BA(A) + w(K - lower\_bound)$$

Here,

A is the array.

K is the index of the element.

BA is the base address of the array A.

w is the word size of one element in memory.

Calculating the length of the array:

$$Length = Upper\ bound - lower\text{-}bound + 1$$

where,

upper bound is the index of the last element

lower bound is the index of the first element in the array.

Example Program:

Write a program to read and display 'n' numbers using an array.

```
#include <stdio.h>
#include<conio.h>
int main()
{
    int i=0, n, arr[20];
    printf("\n Enter the number of elements");
    scanf("%d", &n);
    for(i=0; i<n; i++)
    {
        printf("\n arr[%d] =", i);
        iscanf("%d", & arr[i]);
    }
    printf("\n The array elements are");
    for(i=0; i<n; i++)
        printf("arr[%d] = %d\t", i, arr[i]);
    return 0;
}
```

→ Arrays allow programmers to group

related items of the same data type in one variable.

———— × ————

Operations :

→ Operations includes

1) Traversal
2) Selection
3) Insertion
4) Deletion
5) Searching.

1) Traversal :

→ Traversal is an operation in which each element of a list, stored in an array, is visited.

Algorithm :

Step 1 : Get the elements.

Step 2 : Visit all the elements from 0^{th} element to the last element.

Step 3 : Chk for element is <0, $=0$ and >0, if so do count of each criteria

Step 4 : Count of negative, zero and positive in which travel proceeds from 0^{th} to last.

Step 5 : print the count for each criteria.

2) Selection :

→ An array allows selection of an element for given index

→ Array is called as random access data structures.

Algorithm :

Step 1 : Enter size of the list.

Step 2 : Enter the merit list one by one.

Step 3 : Get into menu of two choice 1- Query and 2. Quit.

Step 4 : Get the pos value and find the value in that pos value.

Step 5 : Print that value.

3) Insertion :

→ Insertion is the operation that inserts an element at a given location of the list.

→ To insert an element at i^{th} location of the list, then all elements from the right

of $i+1^{th}$ location have to be shifted one step towards right.

Algorithm:

Step 1: Set upper_bound = upper bound + 1

Step 2: Set A[upper_bound] = VAL

Step 3: Exit.

4) Deletion:

→ Deletion is the operation that removes an element from a given location of the list.

→ To delete an element from the i^{th} location of the list, then all elements from the right of $i+1^{th}$ location have to be shifted one step towards left to preserve contiguous locations in the array.

Algorithm:

Step 1: Set upper_bound = upper_bound - 1.

Step 2: Exit.

5) Searching :

→ Search is an operation in which a given list is searched for a particular value.

→ A list can be searched sequentially wherein the search for the data item starts from the beginning and continues till the end of the list.

→ This method is called LINEAR SEARCH.

Linear Search.

```
#include<stdio.h>
void main()
{
    int numlist[20];
    int n, pos, val, i;
    printf("\n Enter the size of the list");
    scanf("%d", &n);
    printf("\n Enter the elements one by one");
    for(i=0; i<n; i++)
    {
        scanf("%d", &numlist[i]);
    }
```

```
printf("\n Enter the values to be searched");
scanf("%d", &val);
for(i=0; i<n; i++)
{
    if (val==numlist[i])
    {
        if printf("%d is present at location %d\n",
                                    val, i+1);
        break;
    }
    if (i==n)
        printf("%d isn't present in the array\n",
                                    val);
    }
}
```

Binary Search:

Binary Search in C language to find an element in a sorted array. If the array is'nt sorted we have to sort it using a sorting technique such as bubble sort, insertion, selection sort.

→ If the element to search is present in the list, then we print its location.

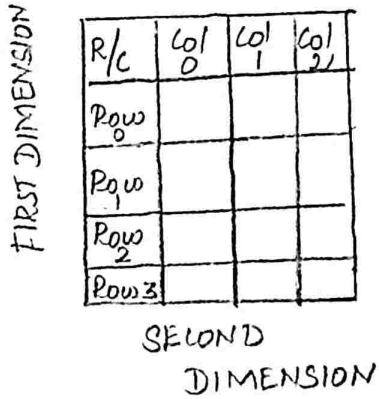
Two Dimensional Arrays -

→ Multi Dimensional Array.

→ The 2-D array be visualized as a rectangular grid of rows and columns.

Declaration of 2D Arrays:

```
datatype array-name [row size] [column size]
```



Memory Representation of 2D Array.

→ Two ways of storing Array.

(i) Row major order of storage.

(ii) Column major order of storage.

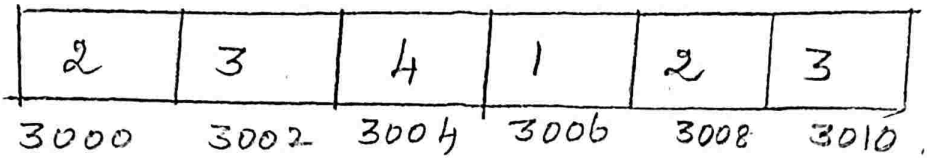
(i) Row major Order.

→ Elements are stored row wise.

For example,

```
int a[2][3] = {{2,3,4}, {1,2,3}};
```

Memory representation is



a[0][0] a[0][1] a[0][2] a[1][0] a[1][1] a[1][2]

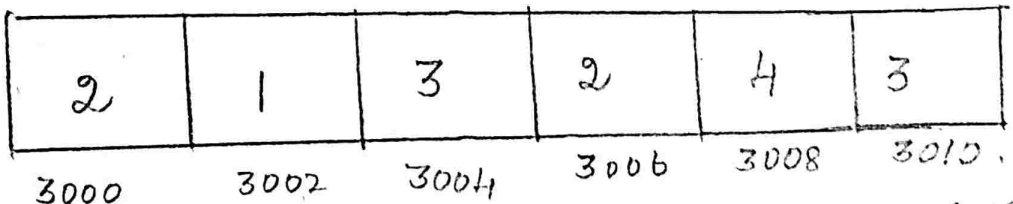
(ii) Column Major Order:

→ Elements are stored column wise.

For example,

```
int a[3][2] = {{2,1}, {3,2}, {4,3}};
```

Memory Representation is,



a[0][0] a[0][1] a[0][2] a[1][0] a[1][1] a[1][2]

Initialization of 2D Array.

```
array-size[row-size][col-size] = {{row 0 element}, {row 1 element},
                                  ..... {row n element}};
```

Accessing of 2D Array.

```
for(i=0; i<row; i++)
{
    for(j=0; j<col; j++)
        scanf("%d", &arr[i][j]);
}
```

Advantages and Limitations :

Advantages :

→ Arrays support direct indexing

Limitations :

→ Arrays are static : size of array cannot be expanded or squeezed at run time.

Applications areas of an Array.

→ An array is an example of static storage structure. It is used when a list of similar data needs to be stored and the number of items is known. It is frequently used in various datatypes.

Operations of 2D Array.

* Read
* Print
* Sum of matrix.
* Transpose.
* Sorting

(i) Internal Sorting.

→ If all the data that is to be sorted can be accomodated at a time in memory is called internal sorting.

(ii) External sorting:

→ It is applied to huge amount of data that cannot be accomodated in memory all at a time.

Types of Internal Sorting:

* Bubble sort
* Insertion sort
* Selection sort
* Quick sort
* Merge sort.

Exercise Programs :

Traverse on the list and print the number of positive and negative values present in the array as <0, =0, >0.

```
#include <stdio.h>
Void main()
{
    int list[10];
    int n;
    int i, neg=0, zero=0, pos=0;
    printf("\n Enter the size of the list \n");
    scanf ("%d", &n);
    printf (" Enter the elements one by one");
    for(i=0; i<n; i++)
    {
        printf ("\n Enter number %d number", i);
        scanf(" %d", & list[i]);
    }
    for(i=0; i<n ; i++)
    {
        printf ("\n Enter number %d number", i);
```

```
scanf(
if (list[i]<0)
    neg = neg+1;
else
    if (list[i] == 0)
        zero = zero+1;
    else
        pos = pos+1;
}
printf("No of Negative numbers in given list
        are %d", neg);
printf("No of zeros in given list are %d",
                                zero);
printf("No of Positive numbers in given list
        are %d", pos);
}
```

Output:

```
Enter the size of the list
3
Enter the elements one by one
Enter number 0 number
1
```

Enter number 1 number

-2

Enter number 2 number

0

No. of negative numbers in the given list are 1

No. of Zeros in given list are 1

No. of positive numbers in given list are 1.

—— x ——

Exercise Program 2:

Sort the numbers using bubble sort.

```
#include<stdio.h>
int main()
{
    int count, num[50], i;
    printf("How many elements to be sorted:");
    scanf("%d", &count);
    printf("Enter the elements:\n");
    for(i=0; i<count; i++)
    {
        printf("num[%d]:", i);
        scanf("%d", &num[i]);
    }
```

```
printf("\n Array Before Souting :\n");
for (i=0; i< count ; i++)
printf (" %.5d", num[i]);
int pass, current, temp ;
for(pass =1;(pass< count); pass++)
{
    for(current =1 ; current <= count - pass ;
                                current++ )
    {
        if(num[current-1] > num[current])
        {
            temp = num[current-1];
            num[current-1] = num[current];
            num[current] = temp;
        }
    }
}
printf ("\n Array after Souting :\n");
for(i=0; i< count ; i++)
printf ("%.5d", num[i]);
}
```

Comparison: Bubble Sort - Insertion Sort - Selection Sort.

Bubble Sort:

→ Very primitive algorithm like linear search, and least efficient.

→ No. of swapping are more compare with other sorting techniques.

→ It is not capable of minimizing the travel through the array like Insertion Sort.

Insertion Sort:

→ Sorted by considering one item at a time

→ efficient to use on small sets of data.

→ twice as fast as the bubble sort.

→ 40% faster than the selection sort.

→ no swapping is required.

→ It is said to be online sorting because it continues the sorting a list as and when it receives.

→ It does not change the relative order of elements with equal keys.

→ Reduces unnecessary travel through the array.

→ Requires low and constant amount of extra memory space.

→ less efficient for larger lists.

Selection Sort:

→ No. of swapping will be minimized (i-e). One swap and one pass.

→ Generally used for sorting files with large objects and small keys.

→ It is 60% more efficient than bubble sort and 40% less efficient than insertion sort.

→ It is preferred over bubble sort for jumbled array as it requires less item to be exchanged.

→ Uses internal sorting that requires more memory space.

Exercise Program 3 :

Find whether the given matrix is diagonal or not.

/* Matrix Diagonal - Program to check whether a given matrix is diagonal matrix */

/* A diagonal matrix is that square matrix whose diagonal elements from upper left to lower right are non-zero and all other elements are zero. For example 2 0 0 */
0 4 0
0 0 6

```
#include <stdio.h>
Void main()
{
    int a[10][10], nr, nc, r, c, flag;
    printf("Enter the number of rows and columns");
    scanf("%d%d", &nr, &nc);
    if (nr==nc)
    {
        /* Checking for square matrix */
```

```
printf("Enter the elements of the matrix:\n");
for(r=0; r<nr; r++)
    for(c=0; c<nc; c++)
        if(r==c) /* true for diagonal
                    elements */
        {
            if(x[r][c]==0)
            {
                flag=0;
            }
            else
            {
                if(x[r][c]!=0)
                flag=10;
            }
        if(flag==0)
            printf("The matrix is diagonal");
        else
            printf("The matrix is not diagonal");
    }
else
    printf("The matrix is not a square matrix");
}
```

Output :

```
Enter the number of rows and columns:

2

2

Enter the elements of the matrix:

1

0

0

1

The matrix is diagonal.
```

Multi Dimensional Array.

A Multi dimensional array is an array of arrays.

There are n indices in a n-dimensional array or multi dimensional array.

Example:

$$A[I_1][I_2][I_3] \cdots [I_n]$$

Program to read and display a 2x2x2 Array.

```
#include <stdio.h>
int main()
{
  int array1[3][3][3], i, j, k;
  printf("\n Enter the elements of the matrix");
  for(i=0; i<2; i++)
  {
    for(j=0; j<2; j++)
    {
      for(k=0; k<2; k++)
      {
        printf("\n array[%d][%d][%d] = ", i, j, k);
        scanf("%d", &array1[i][j][k]);
      }
    }
  }
  printf("\n The matrix is :");
  for(i=0; i<2; i++)
  {
    for(j=0; j<2; j++)
    {
      for(k=0; k<2; k++)
      {
        printf("\t array[%d][%d][%d] = %d",
                i, j, k, array1[i][j][k]);
```

```
    }
  }
}
```

Output :

```
Enter the elements of the matrix
array[0][0][0] = 1
array[0][0][1] = 2
array[0][1][0] = 1
array[0][1][1] = 2
array[1][0][0] = 1
array[1][0][1] = 2
array[1][1][0] = 1
array[1][1][1] = 2
The matrix is
array[0][0][0] = 1
array[0][0][1] = 2
array[0][1][0] = 1
array[0][1][1] = 2
array[1][0][0] = 1
array[1][0][1] = 2
array[1][1][0] = 1
array[1][1][1] = 2
```

Unit - III

Strings.

Introduction to Strings :

A string is a null-terminated character array.

For example,

```
Char c[] = "C String";
```

C		S	t	r	i	n	g	\0

Declaration of string :

```
Char str[size];
Char s[5];
```

→ A string can be declared with a character array or with a string pointer.

Initialization of strings :

```
Char c[] = "abcd";
Char c[5] = {'a', 'b', 'c', 'd', 'e'\0};
Char *c = "abcd";
```

Assigning Values to Strings:

Array and strings are second-class citizen in C; they do not support the assignment operator once it is declared.

For example,

```
char c[100];
```

Note: Use the strcpy() function to copy the string instead.

Example:

```
#include <stdio.h>
#include <conio.h>
int main()
{
    char str[10]={'H','E','L','L','O','\0'};
    printf("Greeting message %s", str);
}
```

Output:

```
Greeting message HELLO
```

Reading and Writing a String :

Reading Strings :

The string can be read by the users by using three ways

* use scanf function
* using gets() function
* using getchar(), getch() or getche() function repeatedly.

Writing Strings :

* use printf() function
* using puts() function
* using putchar() function repeatedly.

Example :

```
int main()
{
    char name[30];
    printf("Enter name:");
    gets(name);
```

```
    printf("Name:");
    puts(Name);
}
```

Output:

Enter name: Hello World

Name: Hello World.

String Operations (Using built-in String functions).

→ C provides string manipulating functions in the "string.h" library.

Function	Purpose	Example
strlen	Returns the number of characters in a string	strlen("Hi") returns 2.
strlwr	Converts string to all lowercase	strlwr("Hi") returns hi.
strupr	Converts s to all uppercase	strupr("Hi");
strrev	Reverses all characters in s1 (except for the terminating null)	strrev(s1, "more");
strtok	Breaks a string into tokens by delimiters.	strtok("Hi, Chao", " ,");
strcpy	Makes a copy of a string	strcpy(s1, "Hi");
Strncpy	Copy the specified number of characters	strncpy(s1, "SVN",2);
strcat	Appends a string to the end of another string	strcat(s1, "more");
Strncat	Appends a string to the end of another string up to n	strncat(s1, "more",2);

Function	Purpose	Example
strcmp	Compare two strings alphabetically	strcmp(s1, "Hu");
Strncmp	Compare two string upto given n character	strncmp("mo", "more",2);
Stricmp	Compare two strings alphabetically without case sensitivity.	stricmp("hu", "Hu");
strchr()	Find first occurrence of a given character in the string	strchr(str1,c); Where c is the character variable
strrchr()	Find the last occurrence of a given character in the string	strrchr(str1,c)
strstr()	Finds the first occurrence of a given string in another string	strstr(str1,str2); Where str2 is the string to be searched in str1
strset()	sets all characters of a string to a given character	strset(str1,c);
strnset()	Sets first character of a string to a given character	Strnset(str1,c,n)

(i) Length - reverse - uppercase and lower case

Length of the string:

→ The number of characters in the string constitutes the length of the string.

Syntax:

```
Var = strlen(string_name);
```

Reversing a String:

strrev() function reverses a given string in 'C' language.

Syntax:

```
char *strrev(char *string);
```

Uppercase of a string:

→ Strupr() converts a given string into uppercase.

Syntax:

```
char *strupr(char *string);
```

Lowercase of a string:

→ strlwr() converts a given string to lowercase.

Syntax:

```
char *strlwr(char *string);
```

Example:

```
#include <stdio.h>
#include <string.h>
Void main()
{
    Char str1[10] = "DENNIS", str2[10] = "SHREE";
    int len;
    len = strlen(str1);
    strupr(str1);
    strlwr(str2);
    printf("\n Length of string %d", len);
    printf("\n Upper case is %s", str1);
    printf("\n lower case is %s", str2);
    strrev(str1);
    printf("\n Reverse of string is %s", str1);
}
```

Output:

```
Length of string is 6
Upper case is DENNIS
Lower case is shree
Reverse of string is SINNED
```

(ii) Concatenate - Copy & Append.

Concatenating two strings to form a new string.

strcat() function in C language concatenates two given strings.

Syntax:

```
char *strcat(char *destination,
             char *source.
```

Copy one string to another.

strcpy() function copies contents of one string into another string.

Example.

```
strcpy (str1, str2).
```

Example Program:

```
#include <stdio.h>
#include <string.h>
Void main()
{
  char str1[10] = "mic", str2[10] = "mouse";
```

```
char str3[10] = "donald", str4[10] = "duck";
char str5[10] = "denny", str6[10];
strcat (str1, str2);
printf ("\n After Concatenating strings : %s", str1);
strncat (str3, str4, 2); // appends first two
                         // char of str4 to str3
printf ("\n %s", str3);
strcpy (str6, str5);
printf ("\n Copied string is %s", str6);
}
```

Output :

```
After Concatenating Strings : micmouse
donalddu
Copied string is denny.
```

(iii) Comparing two strings :

→ strcmp() function is used to compare two strings.

Write a program to compare two strings.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{
        char str1[50], str2[50];
        int i=0, len1=0, len2=0, same=0; clrscr();
        printf("\n Enter the first string : ");
        gets(str1);
        printf("\n Enter the second string : ");
        gets(str2);
        len1 = strlen(str1);
        len2 = strlen(str2);
        if(len1 == len2)
        {
                while(i<len1)
                {
                        if(str1[i] == str2[i])
                                i++;
                        else
                                break;
                }
                if(i==len1)
                {
                        same=1;
                        printf("\n The two strings are equal");
                }
        }
        if(len1!=len2)
                printf("\n The two strings are not equal");
                if(same == 0)
                {
                        if(str1[i]>str2[i])
                                printf("\n String 1 is greater than string 2");
                        else if(str1[i]<str2[i])
                                printf("\n String 2 is greater than string 1");
                }
        getch();
        return 0;
}
```

OUTPUT:

```
Enter the first string : Hello
Enter the second string : Hello
The two strings are equal
```

(iv) Substring / find out occurence?

→ A substring is itself a string that is part of longer string. For

→ Functions used to find the substring are given below.

* strchr()
* strrchr()
* strstr().

→ strchr() returns pointer to the first occurrence of the character in a given string.

→ strrchr() returns pointer to the last occurrence of the character in a given string.

→ strstr() function returns pointer to the first occurrence of the string in a given string.

Example:

```
#include <stdio.h>
#include <string.h>
```

```
Void main()
{
    char str1[15] = "Rose Lotus";
    printf("\n Using strchr : %s", strchr(str1,'i'));
    printf("\n Using strchr : %s", strchr(str1,'i'));
    printf("\n Using strstr : %s", strstr(str1,'ky'));
}
```

Output:

```
Using strchr : ose Lotus
Using strchr : ose
Using strstr : ky Lotus
```

V) Indexing & Replacement:

→ strset() function sets all the characters in a string to given character.

Syntax:

```
char *strset(char *string, int c);
```

String Operations (Without using string built in/ lib functions.

→ Operations that can be performed on strings without using built-in functions which includes,

* length
* compare
* concatenate
* copy
* Reverse
* substring
* Insertion
* Indexing
* Deletion
* Replacement.

NOTE: Even blank spaces are counted as characters in the string - index is from 0 and position is from 1.

index = position − 1 or index + 1 = position.

APPENDING A STRING TO ANOTHER.

Write a program to append a string to another string.

```
#include <stdio.h>
#include <conio.h>
int main()
{
char Dest_Str[100], Source_Str[50];
int i=0, j=0;
printf("\n Enter the source string : ");
gets(Source_Str);
printf("\n Enter the destination string : ");
gets(Dest_Str);
while(Dest_Str[i] != '\0')
i++;
while(Source_Str[j] != '\0')
{
Dest_Str[i] = Source_Str[j];
i++;
j++;
}
Dest_Str[i] = '\0';
printf("\n After appending, the destination string is : ");
puts(Dest_Str);
return 0;
}
```

OUTPUT

```
Enter the source string : How are you?
Enter the destination string : Hello,
After appending, the destination string is : Hello,How are you?
```

Copying the contens of one string to another string.

```
#include <stdio.h>
int main()
{
int i;
char str1[100], str2[100];
printf("\n Enter the string1 : ");
gets(str1);
for (i=0;str1[i]!='\0';i++)
str2[i]=str1[i];
}
str2[i]='\0';
printf("The copied string is %s",str2);
return 0;
```

Reversing a String :

Write a program to reverse a given string.

```
#include <stdio.h>
int main()
{
int i,j=0,len=0;
char str1[100], rev[100];
printf("\nEnter the string1 : ");
gets(str1);
for (i=0;str1[i]!='\0';i++)
len=len+1;
for (i=len-1;i>=0;i--)
{
rev[j]=str1[i];
j=j+1;
}
rev[j]='\0';
printf("The reversed string is %s",rev);
return 0;
}
```

Substring :

→ To extract substring from a given string, we need the following three parameters.

* main string,
* position of the first character of the substring.
* the maximum number of characters | length of the substring.

Write a program to extract a substring from the middle of a given string.

```
#include <stdio.h>
#include <conio.h>
int main()
{
char str[100], substr[100];
int i=0, j=0, n, m;
printf("\n Enter the main string : ");
gets(str);
printf("\n Enter the position from which to start the substring: ");
scanf("%d", &m);
printf("\n Enter the length of the substring: ");
scanf("%d", &n);
i=m;
while(str[i] != '\0' && n>0)
{
substr[j] = str[i];
i++; j++; n--;
}
substr[j] = '\0';
printf("\n The substring is : "); puts(substr);
return 0;
}
```

OUTPUT

```
Enter the main string : Hi there
Enter the position from which to start the substring: 1
Enter the length of the substring: 4
The substring is : i th
```

Insertion :

The insertion operation inserts a string S in the main text T at the K^{th} position.

Syntax :

INSERT(text, position, string).

Write a program to insert a string in the main text.

```
#include <stdio.h>
int main() {
char text[100], str[20], ins_text[100]; int i=0, j=0, k=0,pos;
printf("\n Enter the main text : ");
gets(text);
printf("\n Enter the string to be inserted : ");
gets(str);
printf("\n Enter the place at which the string has to be inserted: ");
scanf("%d", &pos);
while(text[i] != '\0'){
if(i==pos)
{
while(str[k] != '\0')
{
ins_text[j] = str[k];
j++;
k++;
}
}
else
{
ins_text[j] = text[i];
j++;
}
i++;}
ins_text[j] = '\0';
printf("\n The new string is : ");
puts(ins_text);
return 0;
}
```

OUTPUT

```
Enter the main text : newsman
Enter the string to be inserted : paper
Enter the place at which the string has to be inserted: 4
The new string is: newspaperman
```

Deletion :

The deletion operation deletes a substring from a given text. We can write it as,

DELETE(text, position, length).

Algorithm :

Step 1 : [INITIALIZE] SET I = 0 and J = 0

Step 2 : Repeat steps 3 to 6 while TEXT[I] != NULL

Step 3 : If I = M

Repeat while N > 0

Set I = I+1

Set N = N-1

[End of Inner Loop]

[End of IF]

Step 4 : set NEW_STR[J] = TEXT[I]

Step 5 : set J = J+1

Step 6 : set I = I+1

[End of Outer Loop]

Step 7 : Set New_str[J] = NULL

Step 8 : Exit.

Write a program to delete a substring from a text.

```
#include <stdio.h>
int main() {
char text[200], str[20], new_text[200];
int i=0, j=0, found=0, k, n=0, copy_loop=0;
printf("\n Enter the main text : ");
gets(text);
printf("\n Enter the string to be deleted : ");
gets(str);
while(text[i]!='\0'){
j=0, found=0, k=i;
while(text[k]==str[j] && str[j]!='\0')
{
k++;
j++;
}
if(str[j]=='\0')
copy_loop=k;
new_text[n] = text[copy_loop];
i++;
copy_loop++;
n++;
}
new_text[n]='\0';
printf("\n The new string is : ");
puts(new_text);
getch();
return 0;
}
```

Replacement :

→ The replacement operation is used to replace the pattern P1 by another pattern P2. This is done by writing,

REPLACE (text, pattern, pattern)

For Example,

("AAABBBCCC", BBB", "X") = AAAXCCC.

Write a program to replace a pattern with another pattern in the text.

```
#include <stdio.h>
int main() {
char str[200], pat[20], new_str[200], rep_pat[100]; int i=0, j=0, k, n=0,
copy_loop=0, rep_index=0;
printf("\n Enter the string : ");
gets(str);
printf("\n Enter the pattern to be replaced: ");
gets(pat);
printf("\n Enter the replacing pattern: ");
gets(rep_pat);
while(str[i]!='\0'){
j=0,k=i;
while(str[k]==pat[j] && pat[j]!='\0'){
k++;
j++;}
if(pat[j]=='\0'){
copy_loop=k;
while(rep_pat[rep_index] !='\0'){
new_str[n] = rep_pat[rep_index];
rep_index++;
n++;}}
new_str[n] = str[copy_loop];
i++;
copy_loop++;
n++;}
new_str[n]='\0';
printf("\n The new string is : ");
puts(new_str);
return 0;
}
```

Arrays of Strings :

→ String is an array of characters.

→ An array of a string is an array of arrays of characters.

→ Each string is terminated with a null '\0' character.

WRITE A PROGRAM TO READ AND PRINT THE NAMES OF N STUDENTS OF A CLASS

```
#include<stdio.h>
#include<conio.h>
main()
{
char names[5][10];
int i, n;
clrscr();
printf("\n Enter the number of students : ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n Enter the name of %dth student : ", i+1);
gets(names[i]);
}
printf("\n Names of the students are : \n");
for(i=0;i<n;i++)
puts(names[i]);
getch();
return 0;
}
```

→ A string can be defined as a 1D array of characters, so an array of string is two-dimensional array of characters.

Memory Representation:

Char name[5][10] = {"Ram", "Krithvik", "Sree", "Hari", "Rosi"};

name[0]	R	a	m	'\0'						
name[1]	K	r	i	t	h	v	i	k	'\0'	
name[2]	S	r	e	e	'\0'					
name[3]	H	a	r	i	'\0'					
name[4]	R	o	s	e	'\0'					

Introduction to Pointers:

→ A pointer is a variable that contains the memory location of another variable.

Syntax:

```
data_type *pointer_name;
```

Declaration of Pointers:

```
data_type *pointer_name;
```

Example:

```
int *a;
char *a;
float *a;
```

Accessing Variable through Pointer.

→ If a pointer is declared and assigned to a variable, then the variable can be accessed through the pointer.

Example:

```
int *ptr;
int x=10;
ptr = &x;
```

Example Program:

```
#include <stdio.h>
int main()
{
    int num, *a;
    a = &num;
    printf("\n Enter the number :");
    scanf("%d", &num);
    printf("\n The number is:%d", *a);
}
```

Output:

```
Enter the number :10
The number is : 10
```

Pointer Operators / Expressions.

Pointer variables can be used in expressions.

For example,

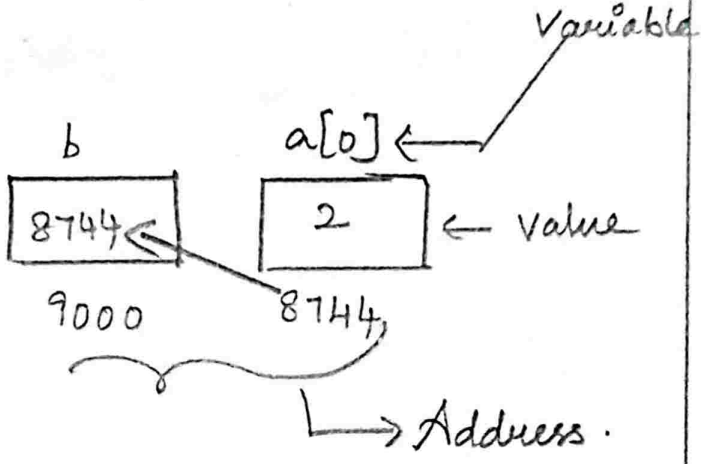
```
#include <stdio.h>
int main()
{
   int a=2, b=3;
   int *c, *d;
   c=&a;
   d=&b;
int sum=*c+*d;
 int   mul=sum**c;
       d+=1;
   int div=9+*a/*b-30;
   printf("%d %d %d", sum, mul, div);
}
```

Pointers and Arrays:

→ The elements of the array can also be accessed through a pointer.

Example:

```
int a[3] = {2,3,7};
int *b;
b = a;
```



Example Program:

```
#include <stdio.h>
Void main()
{
    int a[3] = {2,3,7};
    int *b;
    b = a;
    printf("\n The value of a[0] = %d", a[0]);
    printf("\n The Address of a[0] = %u", &a[0]);
    printf("\n The Value of b = %d", b);
}
```

Output:

```
The value of a[0] = 2
The address of a[0] = 8744
The value of b = 8744.
```

Exercise Programs:

Exercise program1 : To find the frequency of a character in a string

```
#include <stdio.h>
#include <string.h>
int main()
{
char string[100];
int c = 0, count[26] = {0}, x;
printf("Enter a string\n");
gets(string);
while (string[c] != '\0') {
/** Considering characters from 'a' to 'z' only and ignoring others. */
if (string[c] >= 'a' && string[c] <= 'z') {
x = string[c] -'a';
count[x]++;
}
c++;
}
for (c = 0; c < 26; c++)
printf("%c occurs %d times in the string.\n", c +'a', count[c]);
return 0;
}
```

Exercise program 2:
To find the number of vowels, consonants and white spaces in a given text

```
#include <stdio.h>
int main() {
char line[150];
int vowels, consonant, digit, space;
vowels = consonant = digit = space = 0;
printf("Enter a line of string: ");
gets(line);
//fgets(line, sizeof(line), stdin);
for (int i = 0; line[i] != '\0'; ++i) {
if (line[i] == 'a' || line[i] == 'e' || line[i] == 'i' ||
line[i] == 'o' || line[i] == 'u' || line[i] == 'A' ||
line[i] == 'E' || line[i] == 'I' || line[i] == 'O' ||
line[i] == 'U') {
++vowels;
} else if ((line[i] >= 'a' && line[i] <= 'z') || (line[i] >= 'A' && line[i] <= 'Z')) {
++consonant;
} else if (line[i] >= '0' && line[i] <= '9') {
++digit;
} else if (line[i] == ' ') {
++space;
}
```

```
}
printf("Vowels: %d", vowels);
printf("\nConsonants: %d", consonant);
printf("\nDigits: %d", digit);
printf("\nWhite spaces: %d", space);
return 0;
}
```

Exercise program3: Sorting the names

```
/* C Program to Sort Names in Alphabetical Order */
#include <stdio.h>
#include <string.h>
void main()
{
char name[10][8], tname[10][8], temp[8];
int i, j, n;
}
printf("Enter the value of n \n");
scanf("%d", &n);
printf("Enter %d names: \n" , n);
for (i = 0; i < n; i++)
{
scanf("%s", name[i]);
strcpy(tname[i], name[i]);
}
for (i = 0; i < n - 1 ; i++)
{
for (j = i + 1; j < n; j++)
{
if (strcmp(name[i], name[j]) > 0)
{
strcpy(temp, name[i]);
strcpy(name[i], name[j]);
strcpy(name[j], temp);
}
}
}
printf("\n----------------------------------------\n");
printf("Input NamestSorted names\n");
printf("----------------------------------------\n");
for (i = 0; i < n; i++)
{
printf("%s\t\t%s\n", tname[i], name[i]);
}
printf("--------------------\n");
}
```

Unit IV

Functions.

Introduction to Functions:

→ A function is a group of statements that together perform a task.

→ A function declaration tells the compiler about a function's name, return type, and parameters.

→ A function definition provides the actual body of the function.

→ A function can also be referred as a method or a sub-routine or a procedure, etc.

Why we need functions in C:

(a) To improve the readability of code.

(b) Improves the reusability of the code.

(c) Debugging of the code would be easier if you use functions, as errors are easy to be traced.

(d) Reduces the size of the code.

Scope of Functions:

→ A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it can be accessed.

* Local Variables
* Global Variables
* Formal Parameters.

(i) Local Variables:

→ Variables that are declared inside a function or block are called local variables.

→ They can be used only by statements that are inside that function or block of code.

Example:

```
#include<stdio.h>
int main()
/* local variable declaration */
int a,b;
int c;
/* actual initialization */
 a=20;
 b=10;
  c=a+b;
printf("Value of 'a' = %d, b=%d and c=%d\n",
                                a,b,c);
}
```

(ii) Global Variables:

→ Global variables are defined outside a function, usually on top of the program.

→ Global variables hold their values throughout the lifetime of the program.

→ A global variable can be accessed by any function.

Example :

```
#include <stdio.h>
int x; /* global variable declaration */
int main()
{
   int a, b; /* local variable declaration */
   a = 20;
   b = 10;   } → actual initialization.
   x = a+b;
   printf("Value of a = %d, b = %d and x = %d\n",
                                   a, b, x);
}
```

A PROGRAM CAN HAVE SAME NAME FOR LOCAL AND GLOBAL VARIABLES BUT THE VALUE OF LOCAL VARIABLE INSIDE A FUNCTION WILL TAKE PREFERENCE.

Types of Functions:

→ There are 2 types of functions

* Standard Library functions / pre defined function / built in functions.
* User defined functions.

(i) Standard Library Functions:

→ These are built-in functions in C programming.

→ printf() is a standard library function to send formatted output to the screen. scanf() is used to get the input from the user.

Advantages of Using C library Functions:

(i) The functions are optimized for performance.

(ii) It saves considerable development time.

(iii) The functions are portable.

(ii) User defined function:

→ The function that we create in a program is known as user defined functions.

(or)

→ The functions created by users is known as user defined functions.

Syntax:

```
returntype function_name(argument list)
{
    set of statements;
    ......;
    ......;
}
```

Example:

```
#include <stdio.h>
int add(int a, int b)
{
    int c;
    c = a+b;
    return c;
}
```

```
int main()
{
    int x, y;
    printf("Enter the value for x and y");
    scanf("%d %d", &x, &y);
    int z = add(x, y)
    printf("%d", z);
}
```

Output :

```
Enter the value for x and y
10
5
15
```

Advantages of User-defined Functions :

→ The program will be easier to understand, maintain and debug.

→ Reusable codes that can be used in other programs.

→ A large program can be divided into smaller modules. Hence, a large project can be

divided among many programmers.

Function Prototype:

The function prototype are used to tell the compiler about the number of arguments and about the required datatypes of a function parameter, it also tells about the return type of the function.

→ By this information, the compiler cross checks the function signatures before calling it.

Example:

```
#include <stdio.h>
main()
{
   function(50);
}
Void function (int x)
{
    printf(" The value of x is : %d", x);
}
```

Function Definition :

→ A Function definition in C Programming consists of a function header and a function body.

* Return Type:

→ A function may return a value. The return_type is the data type of the value the function returns.

* Function Name:

→ This is the actual name of the function. The function name and parameter list together constitute the function signature.

* Parameters:

→ A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument.

Function Body:

→ The function body contains a collection of statements that define what the function does.

Syntax:

```
return_type function_name (data_type
                Var1, datatype Var2, .....)
{
    .....
    return(Var);
}
```

Function call:

→ The function call statement invokes the function. When the function is invoked the compiler jumps to the called function to execute the statements that are a part of that function.

→ Once the called function is executed,

the program control passes back to the calling function.

Syntax:

```
function_name(var1, var2, ... );
```

Parameter Passing:

Function Arguments:

→ Basically there are two types of arguments.

* Actual Arguments
* Formal Arguments.

Example:

```
#include<stdio.h>
void add(int a, int b)
{
 ....
}
int main()
{ int x,y;
  add(x,y);
}
```

→ formal arguments.

→ actual arguments.

Categories of Functions:

1. Function with no argument and no return value.
2. Function with argument and no return value
3. Function with no argument & return value
4. Function with argument and return value.

(i) Function with no argument and no return value:

→ When a function has no argument, it does not receive any data from the calling function.

→ When it does not return a value, the calling (called) function does no receive any data from the called function.

Example:

```
#include <stdio.h>
int main()
{
   add();
}
```

```
void add()
{
   int a,b,c;
   printf("Enter the value of 'a and b");
   scanf("%d,%d", &a,&b);

    c = a+b;
   printf("%d", c);
}
```

(ii) Function with arguments and no return values.

In this category, the main() function passes argument to the user defined function and the user defined function does not return any value to the main function.

Example:

```
#include <stdio.h>
void main()
{
   int a,b;
   printf("Enter the value of a, b");
```

```
    scanf("%d %d", &a, &b);
    add(a, b);
}
void add(int x, int y)
{
    int z = x + y;
    printf("The sum is %d", z);
}
```

(iii) Function with no argument and return value.

In this category, the main() function does not pass any arguments to the user-defined function. Here, the user-defined function returns a value to the main() function.

Example:

```
#include<stdio.h>
void main()
{
    int c;
    c = add();
    printf("%d", c);
}
```

```
int add()
{
   int x,y;
   printf("\n Enter the value for x & y");
   scanf("%d %d", &x,&y);
   int z = x+y;
   return z;
}
```

(iv) Function with argument and return value.

→ In this category, it receive data from the calling function through arguments and send back value.

Example:

```
#include<stdio.h>
void main()
{
   int x,y;
   printf("Enter the value for x & y");
   scanf("%d %d", &x,&y);
    int z;
    z= add(x,y);
    printf("%d", z);
}
```

```
int add(int a, int b);
{
    int c;
    c = a+b;
    return c;
}
```

Passing Parameters to the function :

→ When a function is called, the calling function may have to pass some value to the called function.

→ There are two types of arguments.

They are,

* Actual Arguments
* Formal Arguments.

→ The variables declared in the function prototype or definition are known as Formal arguments.

→ The values that are passed to the called function from the main function are known as Actual arguments.

→ Basically, arguments or parameters can be passed to the calling Function. They include,

* Call by Value
* Call by reference.

Pass by Value:

The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function.

Example:

```
#include <stdio.h>
void swap(int x, int y);
int main()
{
    int a =100;
    int b = 200;
    printf (" Before swap, value of a : %d", a);
    printf (" Before swap, value of b: %d", b);
    swap(a,b);
}
```

```
void swap (int x, int y)
{
   printf ("After swapping %d", y);
   printf ("After swapping, of y %d", x);
}
```

Output:

Before swap value of a: 100

Before swap value of b: 200

After swapping : 200

After swapping of y : 100.

Pass by reference:

→ The pass by reference method of passing arguments to a function copies the address of an argument into the formal parameter.

→ Inside the function, the address is used to access the actual argument used to call.

Built-In Functions : [STRING FUNCTION]

→ All the string functions are given below.

String functions	Description
strcat ( )	Concatenates str2 at the end of str1
strncat ( )	Appends a portion of string to another
strcpy ( )	Copies str2 into str1
strncpy ( )	Copies given number of characters of one string to another
strlen ( )	Gives the length of str1
strcmp ( )	Returns 0 if str1 is same as str2. Returns <0 if strl < str2. Returns >0 if str1 > str2
strcmpi ( )	Same as strcmp() function. But, this function negotiates case. "A" and "a" are treated as same.
strchr ( )	Returns pointer to first occurrence of char in str1
strrchr ( )	last occurrence of given character in a string is found
strstr ( )	Returns pointer to first occurrence of str2 in str1
strrstr ( )	Returns pointer to last occurrence of str2 in str1
strdup ( )	Duplicates the string
strlwr ( )	Converts string to lowercase
strupr ( )	Converts string to uppercase
strrev ( )	Reverses the given string
strset ( )	Sets all character in a string to given character
strnset ( )	It sets the portion of characters in a string to given character
strtok ( )	Tokenizing given string using delimiter

Recursive Function :

```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

recursive call

→ A function that calls itself again and again is known as a recursive function.

Example:

Write a c Program to find factorial of a number using Recursive Function.

```
#include <stdio.h>
int factorial(int i)
{
    if (i<=1)
    {
      return 1;
    }
    return i*factorial(i-1);
}
int main()
{
   int i=5;
   printf("Factorial of %d is %d\n", i, factorial(i));
}
```

Output:

Factorial of 5 is 120.

Exercise Programs:

Calculate the total amount of power consumed by 'n' devices (Passing an array to a function):

```
#include <stdio.h>
#include <stdlib.h>
int calc_Electricity();//function prototype
int devices(int n[],int size);
int main()
{
        int size,i,n[50],s;
        printf("enter the size of an array");
        scanf("%d",&size);
        s=devices(n ,size);
        printf("the value is",s);
        return 0;
}
int calc_Electricity(int unit)
{
        printf("Enter total units consumed\n");
        scanf("%d",&unit);
        double amount;
        if((unit>=1)&&(unit<=50))//between 1 - 50 units
        {
                amount=unit*1.50;
        }
        else if((unit>50)&&(unit<=150))//between 50 150 units
        {
                amount=((50*1.5)+(unit-50)*2.00);
        }
        else if((unit>150)&&(unit<=250))
        {  //between 150 - 250 units
                amount=(50*1.5)+((150-50)*2.00)+(unit-150)*3.00;
        }
        else if(unit>250)
        {       //above 250 units
                amount=(50*1.5)+((150-50)*2.00)+((250-150)*3.00)+(unit-250)*4;
        }
        else
        {
                printf("No usage ");
                amount=0;
        }
        return amount;
}
int devices(int n[],int size)
{
        int total=0;
        int i;
        int unit=0;
        int p;
        for (i=0;i<size;i++)
        {
```

```
            p=calc_Electricity(unit);
            printf("the amount of one bill %d is",p);
            n[i]=p;
            total=total+n[i];
            printf("the total amount of n devices is %d",total);
        }
}
```

Output:

```
Enter the number of devices : 3
Enter total unit consumed : 200
The total amount of 1 device is 425, Total amount of n device is 425
Enter total unit consumed : 250
The total amount of 1 device is 575, Total amount of n device is 1000
Enter total unit consumed : 200
The total amount of 1 device is 425, Total amount of n device is 1425
```

Exercise:

Menu-driven program to count the numbers which are divisible by 3, 5 and both (passing an array to a function)

```
#include<stdio.h>
int menudriven(int a[]);
int main()
{
        int k,s,i;
        int a[3]={0,1,2};
        menudriven(a);
}
int menudriven(int a[])
{
        int i,n,s,p,j;
        printf("Enter the value of n");
        scanf("%d",&n);
        for(j=0;j<3;j++)
        {
                switch(a[j])
                {
                        case 0:
                                for (i=1; i<=n; i++)
                                {
                                        if(i%5==0)
                                        {
                                                p=p+1;
                                        }
                                }
                                printf("\nthe total numbers divisible by 5 is %d",p);
                        case 1:
                                p=0;
                                for (i=1; i<=n; i++)
                                {
                                        if (i%3==0)
                                        {
                                                p=p+1;
```

```
                        }
                }
                printf("\nthe total numbers divisible by 3 is %d",p);
        case 2:
                p=0;
                for (i=1; i<=n; i++)
                {
                        if (i%3==0&&i%5==0)
                        {
                                p=p+1 ;
                        }
                }
                printf("\nthe total numbers divisible by both are %d",p);
        }
    }
}
```

Output:

```
Enter the value of n : 100
The total numbers divisible by 5 is : 20
The total numbers divisible by 3 is : 33
The total numbers divisible by both is : 6
```

Unit - V

Introduction to Structures.

Introduction to structures :

C Data types

* Primary data types
* Derived Data types
* User-defined data types.

Derived Types.

* Function type
* Array type
* Pointer type
* Structure type
* Union Type.

Structure :

→ Collection of one or more related variables of different data types, grouped under a single name.

Need of Structures.

→ In a library, each book is an object, and its characteristics like title, no of pages, price are grouped and represented by one record.

→ The characteristics are different types and grouped under a aggregate variable of different types.

→ A record is group of fields and each field represents one characteristic. In C, a record is implemented with a derived data type called structure. The characteristics of record are called the members of the structures.

* A structure is defined to be collection of different data items, that are stored under a common name.

Declaration of Structures :

```
struct struct-name
{
  datatype var-name;
};
```

Example:

```
struct student
{
  int s_no;
  char name[20];
  char course[20];
  float fees;
};
```

→ The structure definition does not allocate any memory. It just gives a template that conveys to the C compiler.

Type def declaration:

→ The typedef keyword enables the programmer to create a new data type name by using an existing data type.

→ By using typedef, no new data is created, rather than an alternate name is given to a known data type.

Syntax:

```
typedef existing-data-type new-data-type;
```

Example:

typedef int INTEGER;

→ Then INTEGER is the new name of data type int.

INTEGER num = 5;

Accessing the members of a structure:

→ Each member of a structure can be used just like a normal variable, but its name will be a bit longer.

→ A structure member variable is generally accessed using a '.' (dot operator).

Syntax:

```
struct_var.member_name
Stud1.R_no
```

membership Operator.

→ The dot operator is used to select a particular member of the structure.

→ To input values for data members of the structures variable stud1, we may write,

```
scanf("%d", &stud1.r_no);
scanf("%s", stud1.name);
```

→ To print,

```
printf("%s", stud1.course);
```

→ Memory is allocated only when we declare the variables of the structure.

Initialization of Structures :

Syntax :

```
struct struct_name
{
    data_type member_name1;
    data_type member_name2;
    .....
} struct_var = {constant1, ..... };
```

[OR]

```
struct struct_name
{
    data_type member_name1;
    data_type member_name2;
    .....
};
struct struct_name struct_var = {constant1...
                                ...};
```

Example:

```
Struct student
{
  int r_no;
  Char name[20];
  Char course[20];
  float fees;
} stud1 = {01, "Rahul", "BCA", 45000};
```

[Or]

```
struct student stud1 = {01, "Rahul", "BCA", 45000"};
```

Example Program: Write a program using structures to read and display the information about a student.

```
#include<stdio.h>
#include <conio.h>
#include <string.h>
struct employee
{
    int empid;
    char name[35];
    int age;
    float salary;
};
int main()
{
  struct employee emp1;
  printf (" Enter the details of employee 1:");
  scanf (" %d %s %d %f", & emp1. empid,
          emp1. name, & emp1. age, & emp1. salary);
  printf (" Emp ID : %d \n Name: %s \n Age :
      %d \n Salary : %f", emp1. empid,
```

```
empl.name, empl.age, empl.salary);
}
```

Output :

```
Enter the details of Employee 1:
1001
Keithvik
25
5,00,000
EmpID : 1001
Name : Keithvik
Age : 25
Salary : 5,00,000
```

—— x ——

Write a program using structures to read 3 marks of student and display the total and average of the student.

```
#include <stdio.h>
#include <conio.h>
```

```
struct stud
{
    int regno;
    char name[20];
    int m1;
    int m2;
    int m3;
};
struct stud s;
void main()
{
    float tot, avg;
    printf("\n Enter the student regno, name,
            m1, m2, m3:");
    scanf("%d %s %d %d %d", &s.regno,
        &s.name, &s.m1, &s.m2, &s.m3);
    tot = s.m1 + s.m2 + s.m3;
    avg = tot/3;
    printf("\n The student Details are:");
    printf("\n %d \t %s \t %f \t %f", s.regno,
                s.name, tot, avg);
}
```

Output :

```
Enter the student regno, name, m1, m2, m3
100
aaa
87
98
78
The student details are:
100    aaa    263.00    87.666
```

———— × ————

Copying and Comparing Structures :

→ We can assign a structure to another structure of the same type.

→ For example,

```
struct student stud1 = {01, "Rahul",
                        "BCA", 45000};
struct student stud2;
```

To assign,

```
Stud2 = stud1;
```

Nested Structures:

→ A structure can be placed within another structure.

→ That is, a structure may contain another structure as its member. Such a structure that contains another structure as its member is called a nested structure.

Example:

```
typedef struct
{
    char fname[20];
    char mname[20];
    char lname[20];
} NAME;
typedef struct
{
    int dd;
    int mm;
    int yy;
} DATE;
```

```
typedef struct
{
    int r_no;
    NAME name;
    Char course[20];
    DATE DOB;
    float fees;
} student;
```

→ In this example, we see that the structure student contains two other structures, NAME and DATE. Both these structures have their own fields.

To assign values to the structure fields, we will write,

```
struct student stud1;
stud1.name.fname = "aaa";
stud1.name.mname = "bbb";
stud1.name.lname = "ccc";
stud1.course = "BE";
stud1.DOB.dd = 15;
stud1.DOB.mm = 09;
stud1.DOB.yy = 1990;
```

stud1. fees = 45000 ;

→ In case of nested structures, we use the dot operator in conjunction with the structure variables to access the members of the innermost as well as outermost structures.

Example: Write a C Program to read and display information of a student using structure within a structure.

```
#include <stdio.h>
int main()
{
   struct DOB
   {
      int day;
      int month;
      int year;
   };
   struct student
   {
      int roll_no;
      Char name[100];
```

```
    float fees;
    struct DOB date;
};
struct student stud1;
printf("\n Enter the roll number:");
scanf("%d", &stud1.roll_no);
printf("\n Enter the name:");
scanf("%s", stud1.name);
printf("\n Enter the fees:");
scanf("%f", &stud1.fees);
printf("\n Enter the DOB:");
scanf("%d%d%d", &stud1.date.day, &stud1.date.
    month, &stud1.date.year);
printf("\n ***** STUDENT'S DETAILS *****");
printf("\n ROLL No. = %d", &stud1.roll_no);
printf("\n NAME. = %s", stud1.name);
printf("\n FEES. = %f", stud1.fees);
printf("\n DOB = %d-%d-%d", stud1.date.
        day, stud1.date.month, stud1.date.year);
}
```

Output :

```
Enter the roll number : 1
Enter the name: aaa
Enter the fees : 50000
Enter the DOB : 07 10 2000

***** STUDENT'S DETAILS *****
ROLL NO. = 1
NAME. = aaa
FEES. = 50000
DOB. = 07 10 2000
```

———— × ————

Arrays Of Structures :

Syntax :

```
struct struct_name struct_var[index];
```

Consider the given structure definition,

```
struct student
{
    int r_no;
    char name[20];
    char course[20];
    float fees;
};
```

A student array can be declared by writing,

```
struct student stud[30];
```

Example: Write a program to read and display information of all the students in the class. (using array of structure).

```
#include <stdio.h>
int main()
{
  struct student
  {
    int roll_no;
```

```
    char name[80];
    float fees;
    char DOB[80];
};
struct student stud[50];
int n, i;
printf("\n Enter the number of students:");
scanf("%d", &n);
for(i=0; i<n; i++)
{
    printf("Enter the roll number:");
    scanf("%d", &stud[i].roll_no);
    printf("Enter the name:");
    scanf("%s", stud[i].name);
    printf("Enter the fees:");
    scanf("%f", stud[i].fees);
    printf("Enter the DOB:");
    scanf("%s", stud[i].DOB);
}
for(i=0; i<n; i++)
{
    printf("\n* DETAILS OF %d th STUDENT*", i+1);
```

```
        printf ("\n ROLL No. = %d", stud[i].roll_no);
        printf ("\n Name. = %s", stud[i].name);
        printf ("\n ROLL No. = %f", stud[i].fees);
        printf ("\n DOB. = %d", stud[i].DOB);
    }
}
```

Output :

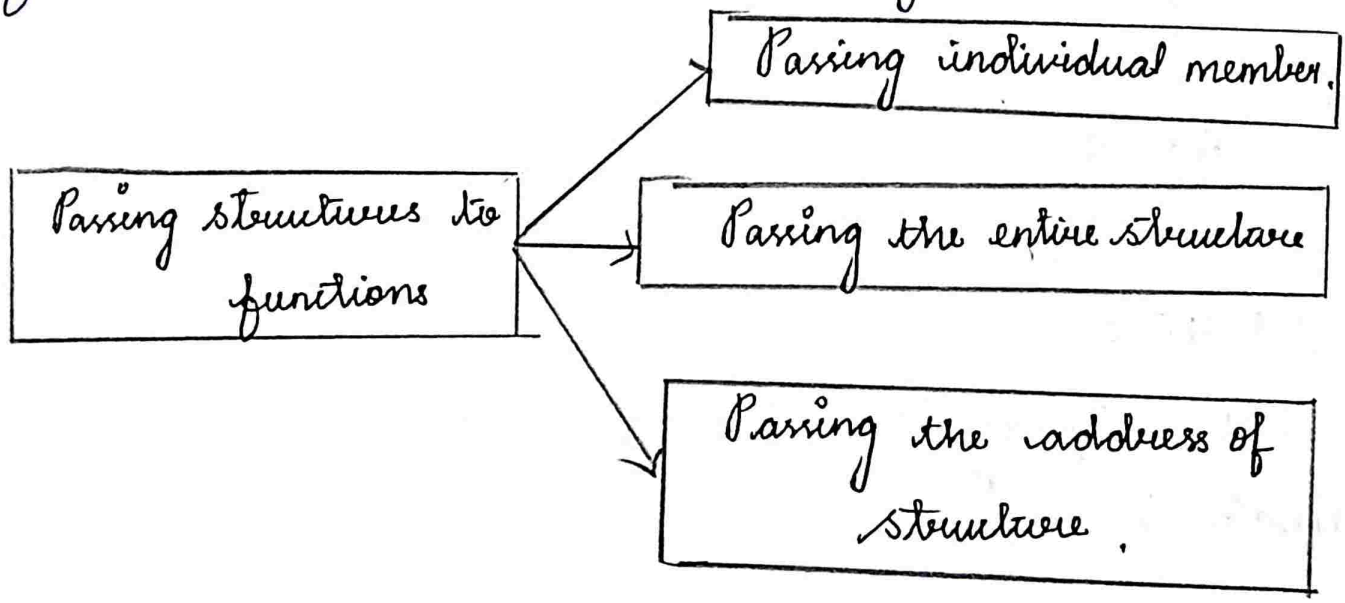
```
Enter the number of students : 2
Enter the roll number : 1
Enter the name: aaa
Enter the fees : 3500
Enter the DOB : 12-12-2000
Enter the roll number : 2
Enter the name : bbb
Enter the fees : 4500
Enter the DOB : 13-12-2000
* Details of the student *
Roll No. = 1
Name. = aaa
fees. = 3500
```

```
DOB. = 12-12-2000
ROLL NO. = 2
NAME : = bbb
fees. = 4500
DOB. = 13-12-2000.
```

Structures and Functions :

→ For structures to be fully useful, we must have a mechanism to pass them to functions and return them.

→ A function may access the members of a structure in three ways.



1) Passing Individual Structure Members to a function.

→ To pass any individual member of the structure to a function we must use the direct selection operator to refer to the individual members for the actual parameters.

→ The called program does not know if the two variables are ordinary variables or structure members.

Sample Program:

```
#include<stdio.h>
typedef struct
{
    int x;
    int y;
} POINT;
void display(int, int);
main()
{
    POINT p1 = {2,3};
    display(p1.x, p1.y);
```

```
    return 0;
}
void display(int a, int b)
{
    printf(" The coordinates of the points are :
        %d %d", a, b);
}
```

Output :

The coordinates of the points are : 2 3

(ii) Passing a Structure to a Function.

→ When a structure is passed as an argument, it is passed using call by value method.

Syntax :

```
func_name (struct struct_name struct_var);
```

Example :

```
#include <stdio.h>
typedef struct
```

```
{
    int x;
    int y;
} POINT;
void display (POINT);
main()
{
    POINT p1 = {2,3};
    display (p1);
    return 0;
}
void display (POINT p)
{
    printf ("The coordinates of the point are:
    %d %d", p.x, p.y);
}
```

Output :

The coordinates of the point are : 2 3

Passing Structures through Pointers :

→ C allows to create a pointer to a structure.

Syntax :

```
struct struct_name
{
    data_type member_name1;
    data_type member_name2;
    .........
} *ptr;

[OR]

struct struct_name *ptr;
```

Example: Write a program using pointer to structure to initialize the members in the structure.

```
#include <stdio.h>
#include <string.h>
struct student
{
```

```
    int r_no;
    char name[20];
    char course[20];
    float fees;
};
main()
{
    struct student stud1, *ptr_stud1;
    ptr_stud1 = &stud1;
    ptr_stud1 -> r_no = 01;
    strcpy (ptr_stud1 -> name, "Rahul");
    strcpy (ptr_stud1 -> course, "BE");
    ptr_stud1 -> fees = 45000;
    printf ("\n DETAILS OF STUDENT");
    printf ("\n ----------");
    printf ("\n ROLL NUMBER = %d", ptr_stud1 -> r_no);
    printf ("\n NAME =", puts(ptr_stud1 -> name));
    printf ("\n COURSE =", puts(ptr_stud1 -> fees));
}
```

Output :

DETAILS OF STUDENT

ROLL NUMBER = 1
NAME = Rahul

COURSE = BE

FEES = 45000.

Self Referential Structures.

→ A self referential structure is one that includes at least one member which is a pointer to the same structure type.

→ With self referential structures, we can create very useful data structures such as linked-lists, trees and graphs.

→ Self referential structures are those structures that contain a reference to data of its same type.

Example:

```
struct node
{
    int val;
    struct node *next;
};
```

→ Here, the structure node will contain two types of data - an integer 'val' and 'next', that is a pointer to a node.

Exercise Programs:

Compute the age of a person using structures and functions (passing a structure to a function)

Age Calculator: This program will read your date of birth and print the current age. The logic is behind to implement this program. Program will compare given date with the current date and print how old are you?

```
/* Age Calculator */
#include <stdio.h>
int date_diff(struct date dt1, struct date dt2);
struct date
{
    int day, month, year;
};
```

```
int main()
{
    struct date dt1 = {05, 10, 2020};
    struct date dt2 = {17, 05, 2004};
    int cmp = date_diff(dt1, dt2);
    return cmp;
}
int date_diff(struct date dt1, struct date dt2)
{
    int years, months, days;
    if (dt2.year > dt1.year)
    {
        years=0;
        months=0;
        days=0;
        printf("\n I cant travel in time");
    }
    else if(dt2.year == dt1.year) { years=0;
    if (dt2.month >= dt1.month)
    {
        months=0; days=0; printf("\n I cant travel
        if (dt2.day > dt1.day)    in time"); }
else if (dt2.month == dt1.month { months=0;
```

```
if (dt2.day > dt1.day)
{
    days = 0;
    printf("\n Jj can't travel in time");
}
else if (dt2.day == dt1.day)
{
    days = 0;
    printf("\n Welcome to Earth");
}
else
    days = dt1.day - dt2.day;
}
else
{
    months = dt1.month - dt2.month;
    if (dt2.day > dt1.day)
    {
        months--;
        days = 30 - dt2.day + dt1.day;
    }
    else
        days = dt1.day - dt2.day;
}
```

```
else
{
    years = dt1.year - dt2.year;
    if (dt2.month > dt1.month)
    {
        years--;
        months = 12 - dt2.month + dt1.month;
        days = 30 - dt2.day + dt1.day;
    }
  else
    {
        months = dt1.month - dt2.month;
        if (dt2.day > dt1.day)
        {
            months--;
            days = 30 - dt2.day + dt1.day;
        }
        else
            days = dt1.day - dt2.day;
    }
}
printf ("\n Your age is %d years, %d months,
            %d days", years, months, days);
}
```

Output :

Your age is 16 years, 4 months, 18 days.