CHAPTER 1

INTRODUCTION TO C

Introduction to C

Software is a collection of program and a program is a collection on instructions given to the computer. Development of software is a stepwise process. Before developing a software, number of processes are done. The first step is to understand the user requirements. Problem analysis arises during the requirement phase of software development. Problem analysis is done for obtaining the user requirements and determine the input and output of the program.

For solving the program, an “algorithm” is implemented. Algorithm is a sequence of steps that gives method of solving a problem. This “algorithm” creates the logic of program. On the basis of this “algorithm”, program code is written. The steps before writing program code are as:-

$$\begin{tikzcd}
\text{User requirements} \arrow[d] \\
\text{Problem analysis} \arrow[d] \\
\text{Input and Output} \arrow[d] \\
\text{Designing algorithm} \arrow[d] \\
\text{Program coding}
\end{tikzcd}$$

Process of program development

Programming Languages

Before learning any language, it is important to know about the various types of languages and their features. It is interesting to know what the basic requirement of the programmer were and what difficulties they faced with the exiting languages. The programming languages can be classified into two types:-

1) Low Level Language
2) High Level Language

1) Low Level Language

There are two types of low level language Machine level language and Assembly level language.

1.1 Machine Level Language

Computer can understand only digital signals, which are in binary digits i.e. 0 and 1. So the instruction given to the computer can be only binary codes. The machine language consists of instructions that are in binary 0 and 1. Computers can understand only machine level language.

Writing a program in machine level language is a difficult task because it is not easy for programmer to write instruction in binary code. It is not portable.

1.2 Assembly Level Language

The difficulties faced in machine level language were reduced to some extent by using a modified form of machine level language called assembly level language. In assembly level language instructions are given in English like word, such as MOV, ADD, SUB, etc. So it is easier to write and understand assembly level programs. Since a computer can understand only machine level language, hence assembly language program must be translated into machine language. The translator that is use for translating is called "assembler".

2) High Level Language

High level languages are designed keeping in mind the features of portability i.e. these languages are machine independent. These are English like language, so it is easy to write and understand the programs of high level language. For translating a high level language program into machine level language, complier or interpreter is used.

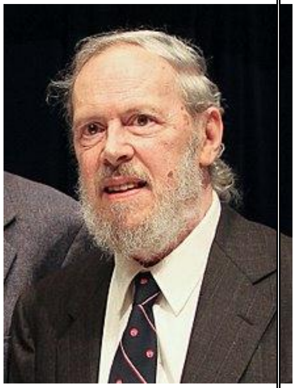
Translator

Translator are used to translate the high level language and low level language into machine level language. There are three type of translator are as follow:-

a) Assembler- It is used for converting the code of low level language into machine level language.
b) Complier- It is used to convert the code of high level language into machine level language.
c) Interpreter- It is used to convert the code of high level language into machine level language.

History of C

In earlier days, every language was designed for some specific purpose. For example FORTRAN (Formula Translator) was used for scientific and mathematical application, COBOL (Common Business Oriented Language) was used for business applications. So need of such a language was felt which could withstand most of the purposes. "Necessity is the mother of invention". From here the first step towards C was put forward by Dennis Ritchie.



The C language was developed in 1970's at Bell laboratories by Dennis Ritchie. Initially it was designed for programming in the operating system called UNIX. After the origin of C, the whole UNIX operating system was rewritten using it. Now almost the entire UNIX operating system and the tools supplied with it including the C compiler itself are written in C.

The C language is derived from B language, which was written by Ken Thompson at AT&T Bell laboratories. The B language was adopted from a language called BCPL (Basic Combined Programming Language), which was developed by Martin Richards at Cambridge University.

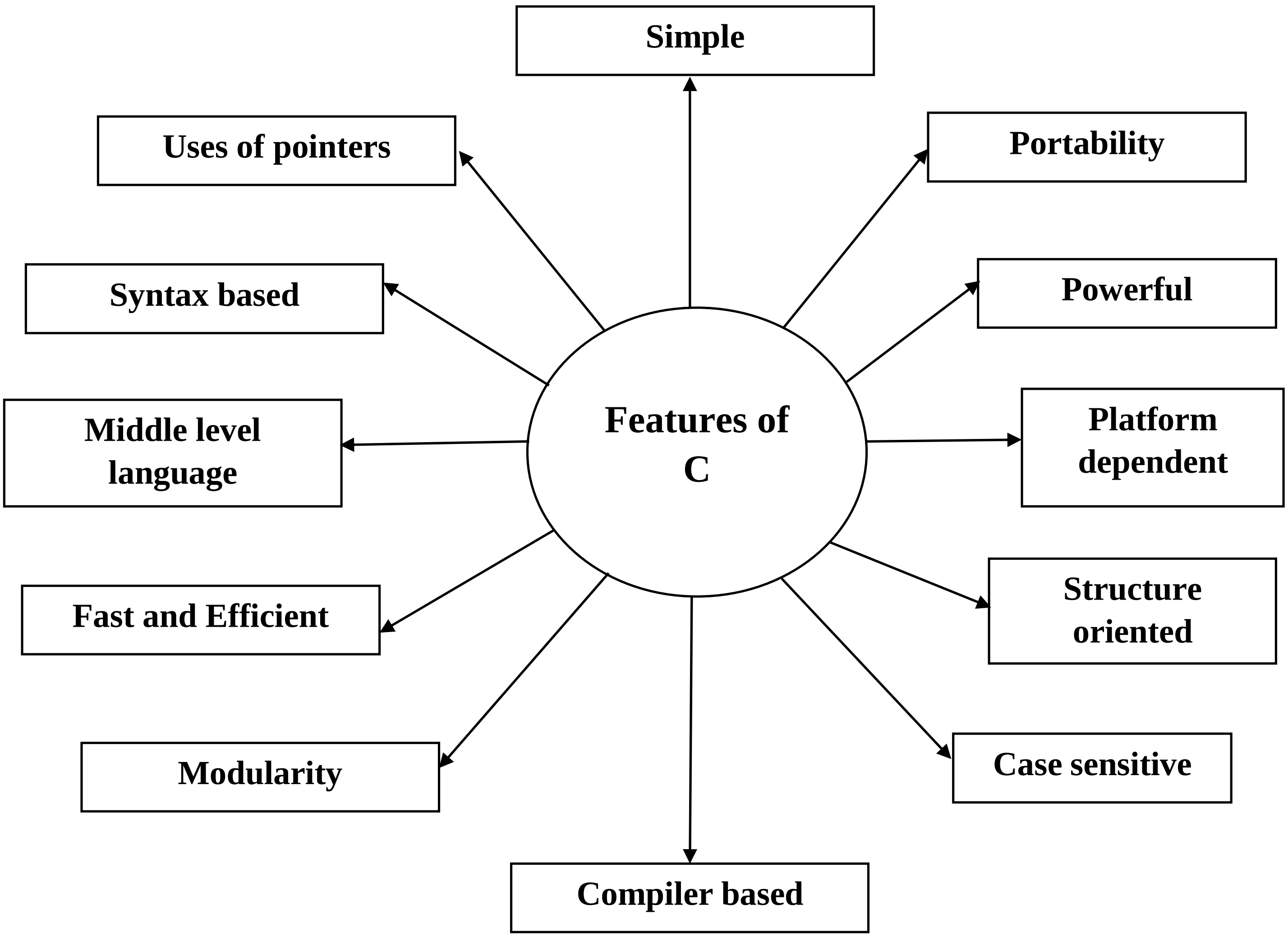
In 1982 a committee was formed by ANSI (American National Standard Institute) to standardize the C language. Finally in 1989, the standard for C language was introduced known as ANSI C.

Features of C

It is a very simple and easy language, C language is mainly used for develop desktop based application. All other programming languages were derived directly or indirectly from C programming concepts. This language have following features

- Simple
- Portability
- Powerful
- Platform dependent
- Structure oriented
- Case sensitive
- Compiler based

- Modularity
- Middle level language
- Syntax based language
- Use of Pointers



Simple

Every c program can be written in simple English language so that it is very easy to understand and developed by programmer.

Platform dependent

A language is said to be platform dependent whenever the program is execute in the same operating system where that was developed and compiled but not run and execute on other operating system. C is platform dependent programming language.

Note: .obj file of C program is platform dependent.

Portability

It is the concept of carrying the instruction from one system to another system. In C Language **.C**file contain source code, we can edit also this code. **.exe** file contain application, only we can execute this file. When we write and compile any C program on window operating system that program easily run on other window based system.

When we can copy .exe file to any other computer which contain window operating system then it works properly, because the native code of application an operating system is same. But this exe file is not execute on other operation system.

Powerful

C is a very powerful programming language, it have a wide verity of data types, functions, control statements, decision making statements, etc.

Structure oriented

C is a Structure oriented programming language.Structure oriented programming language aimed on clarity of program, reduce the complexity of code, using this approach code is divided into sub-program/subroutines. These programming have rich control structure.

Modularity

It is concept of designing an application in subprogram that is procedure oriented approach. In c programming we can break our code in subprogram.

For example we can write a calculator programs in C language with divide our code in subprograms.

Case sensitive

It is a case sensitive programming language. In C programming 'break and BREAK' both are different.

If any language treats lower case latter separately and upper case latter separately than they can be called as **case sensitive** programming language [Example c, c++, java, .net are sensitive programming languages.] other wise it is called as **case insensitive** programming language [Example HTML, SQL is case insensitive programming languages].

Middle level language

C programming language can supports two level programming instructions with the combination of low level and high level language that's why it is called middle level programming language.

Compiler based

C is a compiler based programming language that means without compilation no C program can be executed. First we need compiler to compile our program and then execute.

Syntax based language

C is a strongly tight syntax based programming language. If any language follow rules and regulation very strictly known as strongly tight syntax based language. Example C, C++, Java, .net etc. If any language not follow rules and regulation very strictly known as loosely tight syntax based language.
Example HTML.

Efficient use of pointers

Pointers is a variable which hold the address of another variable, pointer directly direct access to memory address of any variable due to this performance of application is improve. In C language also concept of pointer are available.

Rules of C Programming

1) Extension of C programming is .C
2) All statement should written in small case letters.
3) In C program every statement must end with ; (semicolon).It act as a terminator.
4) In C programming no space are allowed.

5) To use printf() or scanf() function it is necessary to use #include<stdio.h> at beginning of program. #include is the preprocessor directive.

6) #include<stdio.h> stands for standard input output, #include<conio.h> stands for control input output. These are most commonly used for Linking.
7) Format specifier or format string are like %f, %d, %c, %l ,%s used for input and output.
8) \n in the statement is for print output in next line.
9) \t in the statement is for tab/space.
10)\b is used to moves the cursor to the previous position of the current line.
11)\r is used to moves the cursor to beginning of the current line.

12){ } is used for open and close the main function and program.

13)() is used in the expression.

14): is used for label.

Shortcuts in C programming

Sr No.	**Shortcuts keys**	**Action**
1	F1	For Help
2.	F2	Save
3.	F3	Open
4.	F4	Go to cursor
5.	F5	Zoom
6.	F6	Next
7.	F7	Trace into
8.	F8	Step over
9.	F9	Make
10.	F10	Menu
11.	Alt + X	Quit
12.	Alt + Bksp	Undo
13.	Shift + Alt + Bksp	Redo
14.	Shift+Del	Cut
15.	Ctrl + Ins	Copy
16.	Shift + Ins	Paste
17.	Ctrl + Del	Clear
18.	Ctrl + L	Search again
19.	Alt + F7	Previous error
20.	Alt + F8	Next error
21.	Ctrl + F9 'or' Alt + R + Enter	Run
22.	Ctrl + F2	Program reset
23.	Alt + F9	Compile
24.	Alt + F4	Inspect
25.	Ctrl + F4	Evaluate/Modify
26.	Ctrl + F3	Call stack

27.	Ctrl + F8	Toggle breakpoint
28.	Ctrl + F5	Size/Move
29.	Alt + F3	Close
30.	Alt + F5	User screen
31.	Alt + 0	List all
32.	Shift + F1	Index
33.	Ctrl + F1	Topic search
34.	Alt + F1	Previous topic
35.	Ctrl + F7	Add watch
36.	Alt + Enter	Toggle screen mode(Full Screen / Window)*

Datatypes

Data types specify how we enter data into our programs and what type of data we enter. C language has some predefined set of data types to handle various kinds of data that we can use in our program. These datatypes have different storage capacities.

C language supports 2 different type of data types:

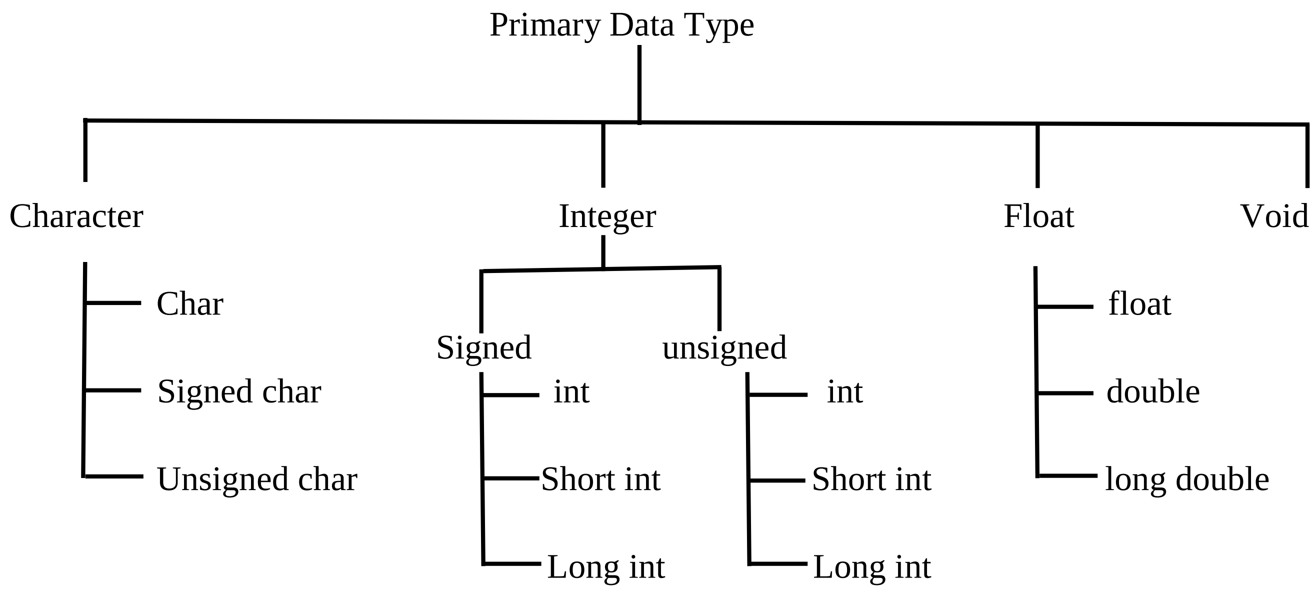
1. **Primary data types:**

   These are fundamental data types in C namely integer (int), floating point (float), character (char) and void.

2. **Derived data types:**

   Derived data types are nothing but primary datatypes but a little twisted or grouped together like **array**, **stucture**, **union** and **pointer**. These are discussed in details later.

Data type determines the type of data a variable will hold. If a variable x is declared as int. it means x can hold only integer values. Every variable which is used in the program must be declared as what data-type it is.



Integer type

Integers are used to store whole numbers.

Type	Size(bytes)	Range
int or signed int	2	-32,768 to 32767
unsigned int	2	0 to 65535
short int or signed short int	1	-128 to 127
unsigned short int	1	0 to 255
long int or signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

Floating point type

Floating types are used to store real numbers.

Type	Size(bytes)	Range
Float	4	3.4E-38 to 3.4E+38
Double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

Character type

Character types are used to store characters value.

Type	Size(bytes)	Range
char or signed char	1	-128 to 127
unsigned char	1	0 to 255

void type

void type means no value. This is usually used to specify the type of functions which returns nothing. We will get acquainted to this datatype as we start learning more advanced topics in C language, like functions, pointers etc.

keywords in C

There are 32 keywords in C, which are as follows:-

Auto	double	int	struct
Break	else	long	switch
Case	enum	register	typedef
Char	extern	return	union
Const	float	short	unsigned
Continue	for	signed	void
Default	goto	sizeof	volatile
Do	If	static	while

Identifiers

All the words that we'll use in C programs will be either keywords or identifiers. Keywords are predefined and can't be changed by the user, while identifier are user defined words and are used to give names to entities like variable, arrays, function, structures etc. Rules for naming identifiers are given below:-

1) The name should consists of only alphabets (both upper and lower case), digits and underscore sign(_).
2) First character should be an alphabet or underscore.
3) The name should not be a keyword.
4) Since C is case sensitive, the uppercase and lowercase letters are considered different. For example code , Code and CODE are three different identifiers.
5) An identifiers name may be arbitrary long. Some implementation of C recognize only the first eight character, through most implementations recognize 31 characters. ANSI standard compliers recognize 31 character.

   The identifiers are generally given meaningful names. Some example of valid identifier names:-

   Value a net_pay rec2 _data MARKS

   Some example of invalid identifier names are:-

5ab	first character should be alphabet or underscore
int	int is a keyword
rec#	# is a special character
avg no	blank space not permitted

Some ASCII values are

1) A-Z ASCII value (65-90)
2) a-z ASCII value (97-122)
3) 0-9 ASCII value (48-57)
4) ; ASCII value (59)

Units

8 bits=1 byte=1 character

1024 bytes=1 kilobyte

1024*1024 bytes=1 Megabyte

1024*1024*1024= 1 Gigabyte

1024*1024*1024*1024 bytes=1 Terabyte

Scientific values

Real (%f) float 4 bytes

4 bytes=32 bits

1	1.00E+00
10	1.00E+01
100	1.00E+02
1000	1.00E+03
10000	1.00E+04

How Data Types are generated values

Integer :- integer(%d) int 2 bytes

2 bytes= 16 bits

2power 16=65536

=65536/2

=32768

Header Files

1) Conio.h
2) Stdio.h
3) Stdlib.h
4) String.h
5) Math.h

1) Conio.h

conio.h header used in C programming contains functions for console input/output. Functions of conio.h can be used to clear screen, change color of text and background, move text, check whether a key is pressed or not and many more.

Member functions

i) **Kbhit()** Determines if a keyboard key was pressed.
ii) **Gcgets()** Reads a string directly from the console.
iii) **cscanf ()** Reads formatted values directly from the console.
iv) **Putch()** Writes a character directly to the console.
v) **Cputs()** Writes a string directly to the console.
vi) **Cprintf()** Formats values and writes them directly to the console.
vii) **Clrscr()** Clears the screen.
viii) **Getch()** Get char entry from the console

2) Stdio.h

The first thing you will notice is the first line of the file, the #include "stdio.h" line. This is very much like the #define the preprocessor, except that instead of a simple substitution, an entire file is read in at this point. The system will find the file named "stdio.h" and read its entire contents in, replacing this statement.

Member functions

i) **printf()** This function is used to print the character, string, float, integer, octal and hexadecimal values onto the output screen
ii) **scanf()** This function is used to read a character, string, numeric data from keyboard.
iii) **getc()** It reads character from file
iv) **gets()** It reads line from keyboard
v) **getchar()** It reads character from keyboard
vi) **puts()** It writes line to o/p screen
vii) **putchar()** It writes a character to screen
viii) **clearerr()** This function clears the error indicators
ix) **f open()** All file handling functions are defined in stdio.h header file
x) **f close()** closes an opened file
xi) **getw()** reads an integer from file
xii) **putw()** writes an integer to file
xiii) **f getc()** reads a character from file
xiv) **putc()** writes a character to file
xv) **f putc()** writes a character to file
xvi) **f gets()** reads string from a file, one line at a time
xvii) **f puts()** writes string to a file
xviii) **f eof()** finds end of file
xix) **f getchar** reads a character from keyboard
xx) **f getc()** reads a character from file
xxi) **f printf()** writes formatted data to a file
xxii) **f scanf()** reads formatted data from a file
xxiii) **f getchar** reads a character from keyboard
xxiv) **f putchar** writes a character from keyboard
xxv) **f seek()** moves file pointer position to given location
xxvi) **SEEK_SET** moves file pointer position to the beginning of the file
xxvii) **SEEK_CUR** moves file pointer position to given location
xxviii) **SEEK_END** moves file pointer position to the end of file.
xxix) **f tell()** gives current position of file pointer
xxx) **rewind()** moves file pointer position to the beginning of the file
xxxi) **putc()** writes a character to file
xxxii) **sprint()** writes formatted output to string
xxxiii) **sscanf()** Reads formatted input from a string
xxxiv) **remove()** deletes a file
xxxv) **fflush()** flushes a file

3) Stdlib.h

stdlib.h is the header of the general purpose standard library of C programming language which includes functions involving memory allocation, process control, conversions and others. It is compatible with C++ and is known as cstdlib in C++. The name "stdlib" stands for "standard library".

4) String.h

String is an array of characters. In this guide, we learn how to declare strings, how to work with strings in C programming and how to use the pre-defined string handling functions. We will see how to compare two strings, concatenate strings, copy one string to another & perform various string manipulation operations. We can perform such operations using the pre-defined functions of “string.h” header file. In order to use these string functions you must include string.h file in your C program.

5) Math.h

“math.h” header file supports all the mathematical related functions in C language like sqrt(),pow10().

ALGORITHM

Algorithm is defined as a finite set of instructions which, if following accomplish a particular task.

In addition every algorithm must satify the following criteria-

1) Input: there are zero or more quantities which are externally supplied.
2) Output: at least one quantity is produced.
3) Definiteness: each instruction must be clear and unambiguous.
4) Effectiveness: every instruction must be sufficient basic that it can in principle to carry out by a person using only pencil and paper.

History of Algorithm

The term and method algorithm was invented by a mathematician AL-Khwarizmi (780AD-850AD) (his full name is Abu Jafar Muhammad ibn Musa al-Khwarizmi & bon in Baghdad). He is known as Inventor or father of Algebra.

There are various definition of algorithm. Few of them are given below-

- A specific sets of rules or instructions for problem solving, that includes precision, input, output, determinism, termination, and generality.
- A well-defined sequence of steps, clearly enough explained that even a computer can do them.
- A detailed sequence of action to perform to accomplish some tasks.
- A set of finite, ordered steps for solving mathematical problems.
- A limited set of well-defined instructors to solve a task, which leads reliably from given starting point to a corresponding identifiable end point.
- A systematic procedure for carrying out a calculation or solving a problem in a limited number of stages.

Advantages of Algorithm

1) It is a step-by-step representation of solution to a given problem.
2) It is very easy to understand.
3) It has got a defined procedure.
4) It is easy to develop algorithm first and then convert to flowchart, then covert to flowchart, then to pseudo code and into a computer program.
5) It is independent of programming language.
6) It is easy to debug as every step is got its own logical sequence.

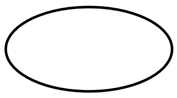
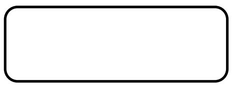
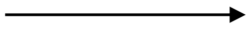
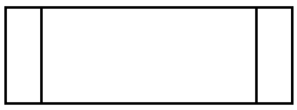
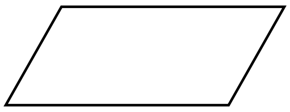
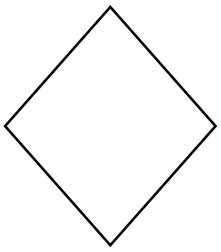
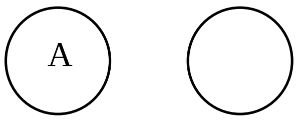
FLOWCHART

Flowchart is a pictorial or graphical or diagrammatic tool for describing computer algorithm. Flowchart are used in designing and documenting complex processes or programs. Flowcharting is a technique that results in a series of symbol are performed. Each symbol contains either natural language or pseudo-code statement. Each symbol in flowchart has a unique purpose.

There are four general types of flowchart

1) Document flowcharts:- shows control over flow of documents through a system.
2) Data flowchart:- shows control over a flow of data in a system.
3) System flowcharts:- shows control at a physical or resource level.
4) Program flowchart:- shows control in a program within a system

5) Other flowchart types that are used in business and government are
    - Logical flowchart
    - Decision flowchart
    - Product flowchart
    - Process flowchart
6) In a program logic development, we have to learn program flowchart. Symbols used in program flowchart and their meaning is tabular below-

Symbol and its Name	**Purpose**
ovals  Rounded Rectangles 	Start/End or Terminator
Arrows 	Flow of Control
Rectangle with double-struck vertical edges 	Subroutine/Predefined process
Parallelogram 	Input/ Output
Diamond 	Condition/Decision
Circle 	Connector

Advantages of Flowchart

1) It is graphical representation of program logic.
2) It is a standardized tool.
3) It is programming language independent.
4) Use of the connector symbol that allow to expand or change a flow easily.
5) It is two dimensional.
6) It is easy to convert flowchart into pseudo-code.

Prog 1.1: WAP to calculate addition of two static number.

```
#include<stdio.h>                 -header file
#include<conio.h>                 -header file
main()
{
int a=5,b=7,sum;                  -int- datatype, a, b, sum- variable
clrscr();                         -clear screen
sum=a+b;                          -calculation part
printf("sum=%d",sum);             -printf is used for print the ans
getch();
}
```

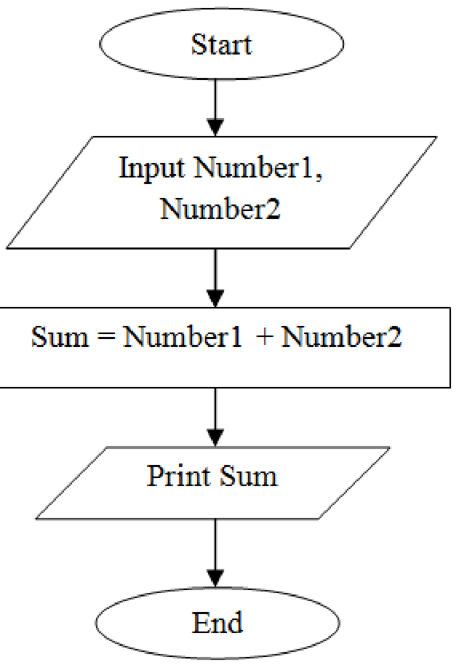
Output:

```
sum=12
```

Algorithm:

```
Step 1: Read two numbers say a and b
Step 2: Calculate sum as addition of a and b
Step 3: Write sum
Step 4: Stop
```

Flowchart:



Prog 1.2: WAP to calculate addition of two numbers.

```
#include<stdio.h>
#include<conio.h>
main()
{
        float a,b,sum;
        clrscr();
        printf("enter the value of a and b");
        scanf("%f/n%f",&a,&b);
        sum=a+b;
        printf("sum=%f",sum);
        getch();
}
```

Output:

```
Enter the value of a and b 3.2 6.1
Sum=9.3
```

Prog 1.3 : WAP to calculate arithmetic operation between two numbers.

```
#include<stdio.h>
#include<conio.h>
main()
{
        int a ,b, sum ,sub, mul, div, mod;
        clrscr();
        printf("enter value of a and b");
        scanf("%d %d",&a, &b);
        sum=a+b;
        sub=a-b;
        div=a/b;
        mul=a*b;
        mod=a%b;
        printf("sum=%d\nsub=%d\ndiv=%d\n mul=%d\n mod=%d",sum,sub,div,mul,mod);
        getch();
}
```

Prog 1.4 : WAP to create salary sheet.

```
#include<stdio.h>
#include<conio.h>
main()
{
        float basic,hra,ta,cl,gpf,pf,dedcu,grsal,netsal;
        clrscr();
        printf("enter the basic value");
        scanf("%f",&basic);
        hra=basic*0.25;
        ta=basic*0.20;
        cl=basic*0.15;
        gpf=basic*0.10;
        pf=basic*0.05;
        grsal=basic+hra+ta+cl+gpf+pf;
        dedcu=gpf+pf;
        netsal=grsal-dedcu;
        printf("hra=%f\n  ta=%f\n  cl=%f\n  gpf=%f\n  pf=%f\n  grsal=%f\n  dedcu=%f\n
netsal=%f\n",hra,ta,cl,gpf,pf,grsal,dedcu,netsal);
        getch();
}
```

CHAPTER 2

OPERATORS AND EXPRESSION

C includes a large number of operators that fall under several different categories, which are as follows:-

1) Arithmetic Operator
2) Assignment Operator
3) Increment and Decrement Operator
4) Relational Operator
5) Logical Operator
6) Conditional Operator
7) Comma Operator
8) Sizeof Operator
9) Bitwise Operator
10) Other Operator

1. Arithmetic Operator

The Arithmetic operators are some of the C Programming Operator, which are used to perform arithmetic operations includes operators like Addition, Subtraction, Multiplication, Division and Modulus. All these operators are binary operators which means they operate on two operands. Below table shows all the Arithmetic Operators with examples.

Arithmetic Operators	**Operation**	**Example**
+	Addition	10 + 2 = 12
–	Subtraction	10 – 2 = 8
*	Multiplication	10 * 2 = 20
/	Division	10 / 2 = 5
%	Modulus – It returns the remainder after the division	10 % 2 = 0 (Here remainder is zero). If it is 10 % 3 then it will be 1.

2.1 WAP to understand the floating point arithmetic operation.

```
#include<stdio.h>
#include<conio.h>
main()
{
    float a=12.4,b=3.8;
    clrscr();
    printf("sum=%f\n",a+b);
    printf("difference=%f\n",a-b);
    printf("product=%f\n",a*b);
    printf("a/b=%f\n",a/b);
    getch();
}
```

Output:

```
Sum=16.20
Difference=8.60
Product=47.12
a/b=3.26
```

2. Assignment Operator

A value can be stored in a variable with the used of assignment operator. This assignment operator “=” is used in assignment expressions and assignment statements.

Eg: x=8

Y=x

Example:

```
  int main()
{
    int value;
    value = 55;
    return (0);
}
```

3. Increment and Decrement Operator

These are unary operators because they operate on a single operand. The increment operator(++) increase the value by 1 and decrement operator(--) decrease by value by 1.

There are two types of operator:-

1) Prefix increment/decrement (eg: ++x or –x)
2) Postfix increment/decrement (eg: x++ or x--)

2.2 WAP to understand the use of prefix increment/decrement.

```
#include<stdio.h>
#include<conio.h>
main()
{
    int x=8;
    clrscr();
    printf("x=%d\t",x);
    printf("x=%d\t",++x);       //prefix increment//
    printf("x=%d\t",x);
    printf("x=%d\t",--x);       //prefix decrement//
    printf("x=%d\t",x);
    getch();
}
```

Output:

```
x=8    x=9    x=9    x=8    x=8
```

2.3 WAP to understand the use of postfix increment/decrement

```
#include<stdio.h>
#include<conio.h>
main()
{
    int x=8;
    clrscr();
    printf("x=%d\t",x);
    printf("x=%d\t",x++);      //postfix increment//
    printf("x=%d\t",x);
    printf("x=%d\t",x--);      //postfix decrement//
    printf("x=%d\t",x);
    getch();
}
```

Output:

```
x=8     x=8    x=9    x=9    x=8
```

4. Relational Operator

Relational operators are used to find the relation between two variables. i.e. to compare the values of two variables in a C program.

Operators	Example/Description
>	x > y (x is greater than y)
<	x < y (x is less than y)
>=	x >= y (x is greater than or equal to y)
<=	x <= y (x is less than or equal to y)
==	x == y (x is equal to y)
!=	x != y (x is not equal to y)

2.4 WAP to understand the use of relational operators.

```
#include<stdio.h>
#include<conio.h>
main()
{
    int a,b;
    clrscr();
    printf("enter the value for a and b");
    scanf("%d%d",&a,&b);
    if(a<b)
    printf("%d is less than %d\n",a,b);
    if(a<=b)
    printf("%d is less than or equal to %d\n",a,b);
    if(a==b)
    printf("%d is equal to %d\n",a,b);
    if(a!=b)
    printf("%d is not equal to %d\n",a,b);
    if(a>b)
    printf("%d is greater than %d\n",a,b);
    if(a>=b)
    printf("%d is greater than or equal to %d\n",a,b);
    getch();
}
```

Output:

```
Enter values for a and b: 12 7
12 is not equal to 7
12 is greater than 7
12 is greater than or equal to 7
```

5. Logical Operator

These operators are used to perform logical operations on the given expressions.

There are 3 logical operators in C language. They are, logical AND (&&), logical OR (||) and logical NOT (!).

Operators	Meaning
&&	AND
||	OR
!	NOT

AND(&&) Operator

Condition 1	Condition 2	Result
False	False	False
False	True	False
True	False	False
True	True	True

OR(||) Operator

Condition 1	Condition 2	Result
False	False	False
False	True	True
True	False	True
True	True	True

NOT(!) Operator

Condition	Result
True	False
False	True

2.5 WAP to understand the use of conditional operator.

```
  #include <stdio.h>

int main()
{
    int m=40,n=20;
    nt o=20,p=30;
    if (m>n && m !=0)
    {
            printf("&& Operator : Both conditions are true\n");
     }
     if (o>p || p!=20)
    {
             printf("|| Operator : Only one condition is true\n");
    }
     if (!(m>n && m !=0))
    {
            printf("! Operator : Both conditions are true\n");
     }
     else
    {
    printf("! Operator : Both conditions are true. " \n"But, status is inverted as false\n");
    }
 }
```

Output:

```
&& Operator : Both conditions are true
|| Operator : Only one condition is true
! Operator : Both conditions are true. But, status is inverted as false
```

6. Conditional Operator

Conditional operators return one value if condition is true and returns another value is condition is false. This operator is also called as ternary operator.

```
Syntax     :       (Condition? true_value: false_value);
Example :          (A > 100  ?  0  :  1);
```

In above example, if A is greater than 100, 0 is returned else 1 is returned. This is equal to if else conditional statements.

2.6 WAP to print the larger of two numbers using conditional operator.

```
#include<stdio.h>
#include<conio.h>
main()
{
            int a,b,max;
            clrscr();
            printf("enter the values for a and b:");
            scanf("%d%d",&a,&b);
            max=a<b?a:b;                    //ternary operator//
            printf("largest of %d and %d is %d\n",a,b,max);
            getch();
 }
```

Output:

```
Enter values for a and b
Larger of 12 and 7 is 12
```

7. Comma Operator

The comma operator is used to permit different expression to appear in situation where only one expression would be used. The expression is separated by comma operator.

Ex: a=9, b=2, c=1;

Sum=(a=3, b=5, a+b);

2.7 WAP to understand the use of comma operator

```
#include<stdio.h>
#include<conio.h>
main()
{
    int a,b,c,sum;
    clrscr();
    sum=(a=8,b=7,c=9,a+b+c);
    printf("sum=%d\n",sum);
    getch();
}
```

Output:

```
 Sum=24
```

2.8 WAP to interchange the value of two variables using comma operator

```
#include<stdio.h>
#include<conio.h>
main()
{
    int a=8,b=7,temp;
    clrscr();
    printf("a=%d,b=%d\n",a,b);
    temp=a,
    a=b,
    b=temp;
    printf("a=%d,b=%d\n",a,b);
    getch();
}
```

Output:

```
  a=8, b=7
a=7, b=8
```

8. Sizeof Operator

Sizeof operator is used to calculate the size of data type or variables.

2.9 WAP to understand the sizeof operator

```
#include<stdio.h>

#include<conio.h>

main()

{

    int var;

    clrscr();

    printf("size of int=%d\n",sizeof(int));

    printf("size of float=%d\n",sizeof(float));

    printf("size of var=%d\n",sizeof(var));

    printf("size of an integer constant=%d\n",sizeof (45));

    getch();

}
```

Output:

```
Size of int=2

Size of float=4

Size of var=2

Size of an integer constant=2
```

9. Bitwise Operator

Bitwise operators are used for operations on individual bits. Bitwise operators operate on integer only. The bitwise operators are as follows:-

Bitwise operator	**Meaning**
&	Bitwise AND
|	Bitwise OR
~	One's complement
<<	Left shift
>>	Right shift
^	Bitwise XOR

2.10 WAP to understand the bitwise operator

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf("Output = %d", a&b);
    return 0;
}
```

Output:

```
8
```

Let us suppose the bitwise AND operation of two integers 12 and 25.

```
12 = 00001100 (In Binary)

25 = 00011001 (In Binary)



Bit Operation of 12 and 25

  00001100

& 00011001

  ________

  00001000  = 8 (In decimal)
```

2.11 WAP to understand the use of XOR bitwise operator

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf("Output = %d", a^b);
    return 0;
}
```

Output

```
21
```

The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite. It is denoted by ^.

```
12 = 00001100 (In Binary)

25 = 00011001 (In Binary)



Bitwise XOR Operation of 12 and 25

  00001100

| 00011001

  ________

  00010101  = 21 (In decimal)
```

Shift Operators in C programming

There are two shift operators in C programming:

- Right shift operator
- Left shift operator.

Right Shift Operator

Right shift operator shifts all bits towards right by certain number of specified bits. It is denoted by >>.

```
212 = 11010100 (In binary)

212>>2 = 00110101 (In binary) [Right shift by two bits]

212>>7 = 00000001 (In binary)
```

212>>8 = 00000000

212>>0 = 11010100 (No Shift)

Left Shift Operator

Left shift operator shifts all bits towards left by certain number of specified bits. It is denoted by <<.

212 = 11010100 (In binary)

212<<1 = 110101000 (In binary) [Left shift by one bit]

212<<0 =11010100 (Shift by 0)

212<<4 = 110101000000 (In binary) =3392(In decimal)

2.12 WAP to understand the use of shift operators

```
#include <stdio.h>
int main()
{
    int num=212, i;
    for (i=0; i<=2; ++i)
        printf("Right shift by %d: %d\n", i, num>>i);

     printf("\n");

     for (i=0; i<=2; ++i)
        printf("Left shift by %d: %d\n", i, num<<i);

     return 0;
}
```

Output:

```
Right Shift by 0: 212
Right Shift by 1: 106

Right Shift by 2: 53



Left Shift by 0: 212

Left Shift by 1: 424

Left Shift by 2: 848
```

Types of Operators	**Description**
Arithmetic operators	These are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus
Assignment operators	These are used to assign the values for the variables in C programs.
Relational operators	These operators are used to compare the value of two variables.
Logical operators	These operators are used to perform logical operations on the given two variables.
Bit wise operators	These operators are used to perform bit operations on given two variables.

Conditional (ternary) operators	Conditional operators return one value if condition is true and returns another value is condition is false.
Increment/decrement operators	These operators are used to either increase or decrease the value of the variable by one.
Special operators	&, *, sizeof( ) and ternary operators

CHAPTER 3

CONTROL STATEMENT AND LOOPING STATEMENT

In C program, statement are executed sequentially in the order in which they appear in the program. But sometimes we may want to use a conditional for executing only a part of program. Also many situation arise where we may want to execute some statements several times. Control statements enable us to specify the order in which the various instruction in the program are to be executed. This determines the flow of control. Control statements define how the control is transferred to other parts of the program. C language support four types of control statements, which are as follow:-

1) If…..else
2) Goto
3) Switch
4) Loop
    a) While
    b) Do……while
    c) For

1) If…else

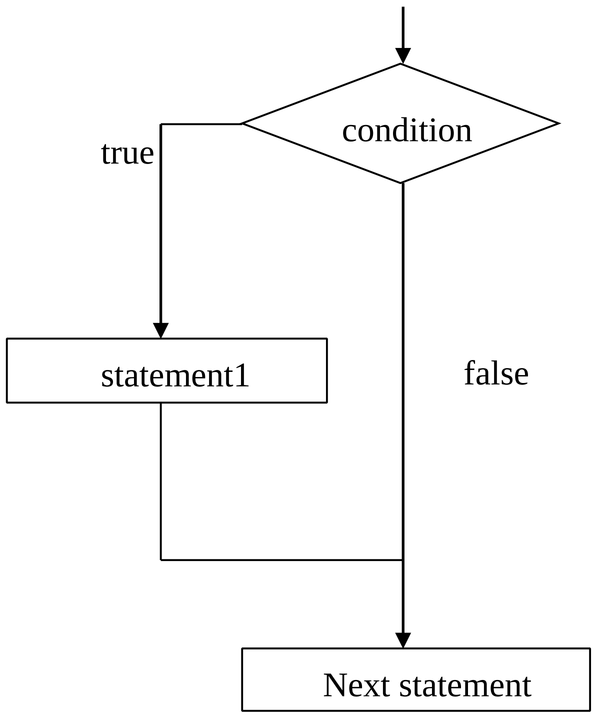
i) If statement

This is a bidirectional conditional control statement. This statement is used to test a condition and take one of the two possible action.

Syntax:

```
If(condition)
{
        Statement;
}
```

There can be a statement or a block of statement after the if part.



3.1 WAP to print a message if negative number is entered

```
#include<stdio.h>
#include<conio.h>
main()
{
        int num;
        clrscr();
        printf("Enter a number");
        scanf("%d",&num);

        if(num<0)
        printf("number entered is negative\n");
        printf("value of num is:%d\n",num);
        getch();
    }
```

Output:

```
 Enter a number: -6
 Number entered is negative
 Value of num is -6
```

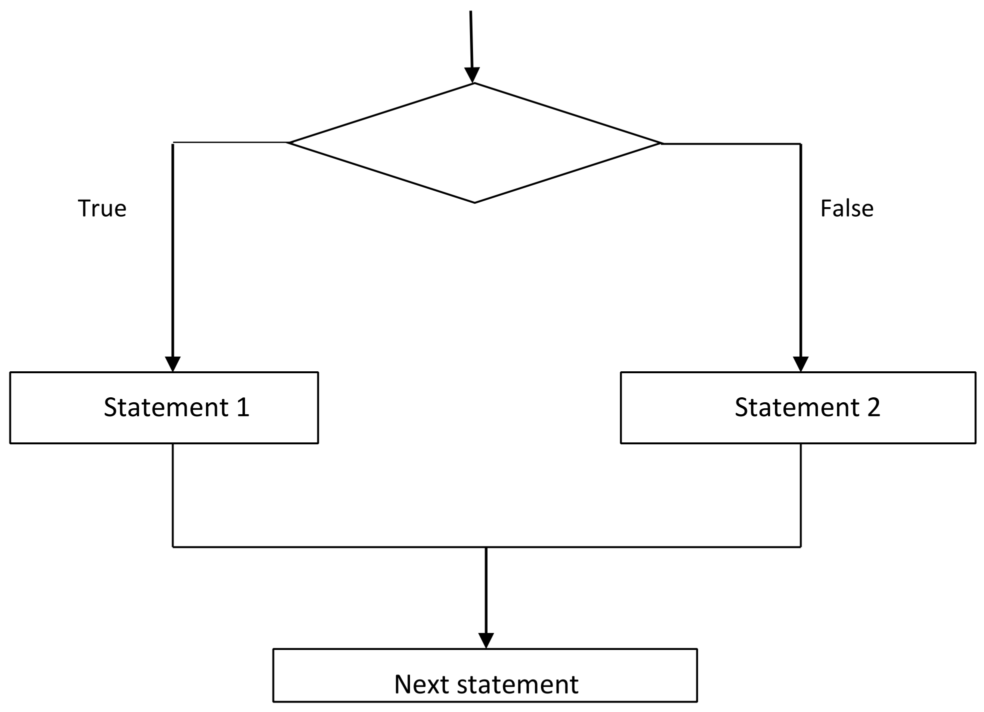
ii) If……..else

This is a bi-directional conditional control statement. This statement is used to test a condition and take one of the possible actions. If the condition is true then a single statement or a block of statement is executed (one part of the program), otherwise another single statement or a block of statements is executed (other part of the program). Recall that in C, any non-zero value is regarded as true while zero is regarded as false.

```
if(condition)
        statement1;
else
        statement2;
```

OR

```
   if(condition)
{
        statement;
 }
else
 {
        Statement;
  }
```



Flowchart of if……else control statement

3.2 Program to print the larger and smaller of the two numbers

```
#include<stdio.h>
#include<conio.h>
 main()
{
int a,b; clrscr();
printf("enter a first number:");
scanf("%d",&a);
printf("enter the second number:");
scanf("%d",&b);
if(a>b)
printf("larger number=%d and smaller number=%d\n",a,b);
else
printf("larger number=%d and smaller number=%d\n",b,a);
getch();
}
```

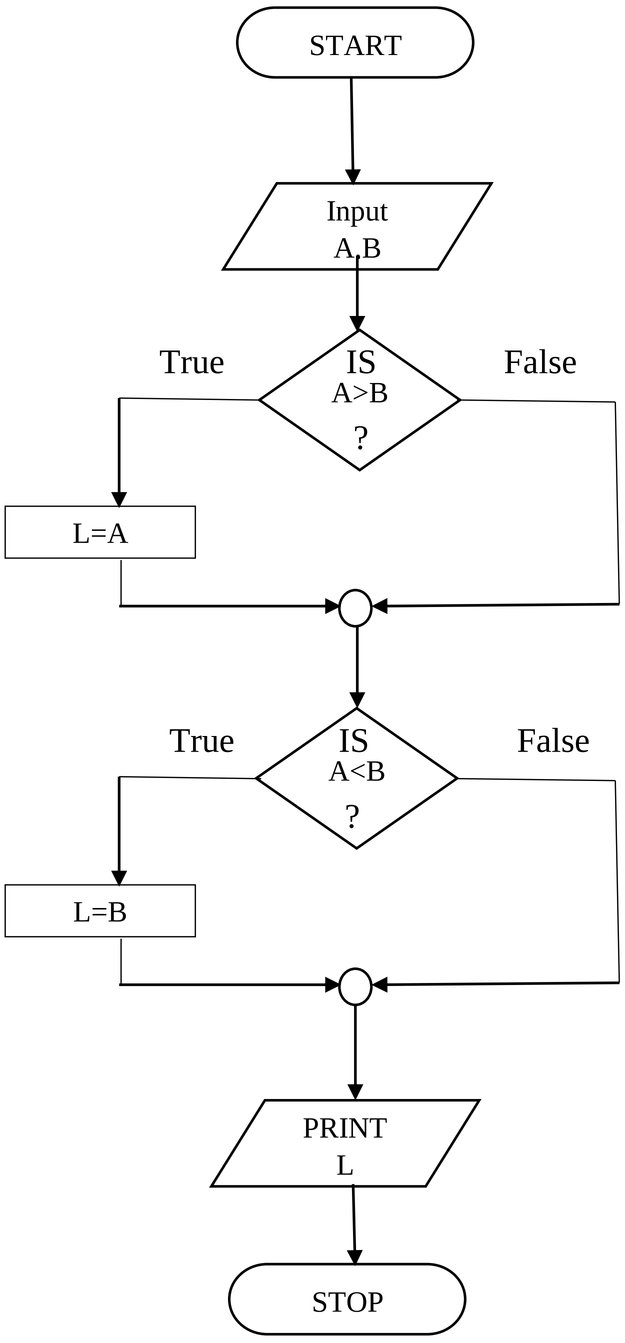
Output:

```
enter the first number:9 enter the second number:11
larger number =11 and smaller number=9
```

Algorithm:

```
Step 1: Read two numbers say A and B
Step 2: Compare A and B
    a) If A is larger than B then L=A
    b) If B is larger than A then L=B
Step 3: Write L
Step 4: Stop
```

Flowchart:


START
Input A,B
True
IS A>B ?
False
L=A
True
IS A<B ?
False
L=B
PRINT L
STOP

3.3 Program to print whether the number is even or odd

```
#include<stdio.h>
#include<conio.h>
main()
{
        int num; clrscr();
        printf("enter a number:");
        scanf("%d",&num);
        if(num%2==0)
        printf("number is even\n");
        else
        printf("number is odd\n");
        getch();
}
```

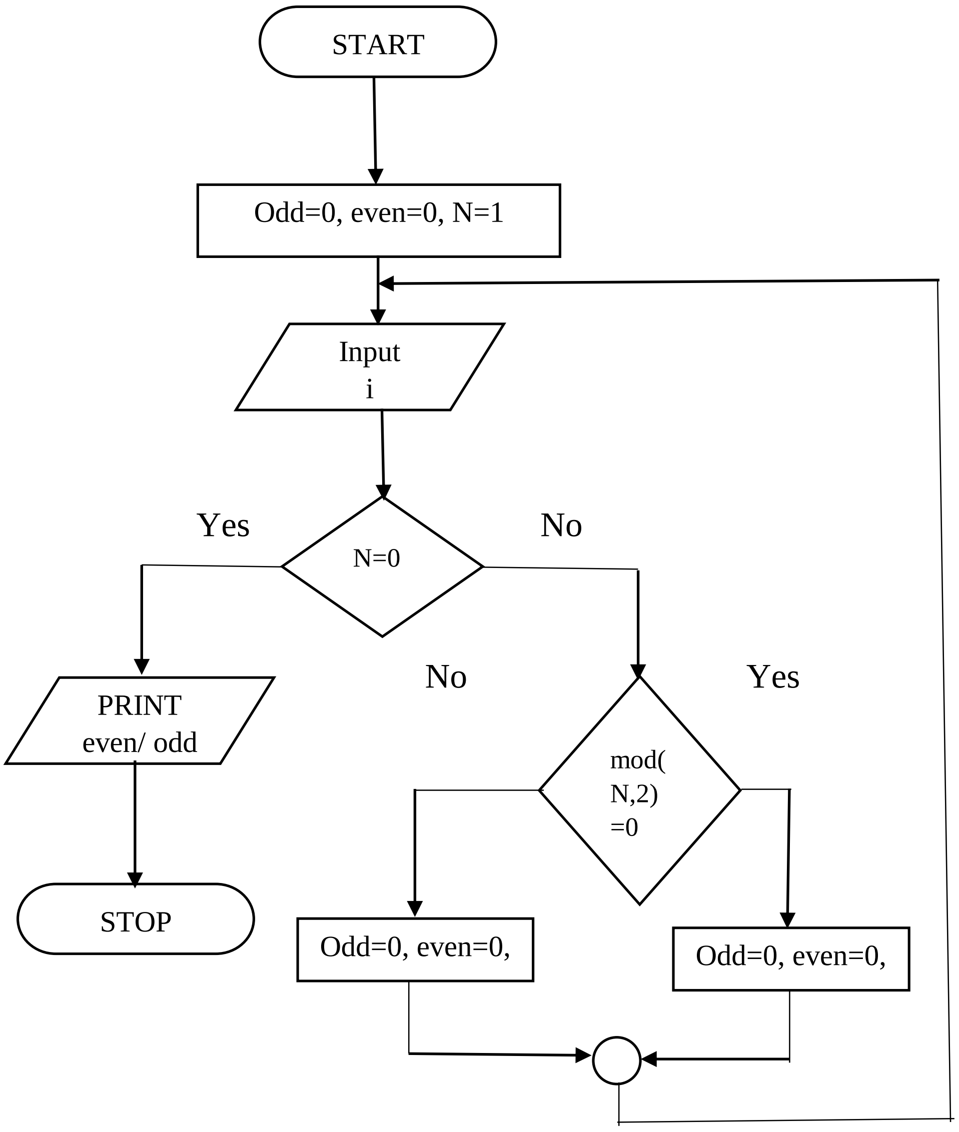
Output:

```
enter the number:15
number is odd
```

Algorithm:

```
Step 1: n=1, even=0, odd=0
Step 2: Repeat steps (a) and (b) until n is 0
    a) Check n for even/odd
    b) If n is even then increment even by 1 otherwise increment odd by 1
Step 3: Write even, odd
Step 4: Stop
```

Flowchart:



iii) Nesting of if…..else

if(condition 1) We can have another if…..else statement in the if block or the else block. This is called nesting of if….else statements. Here is an example of nesting where we have if….else inside both if block and else block.

```
 {
 if(condition 2)
         statement a1;
 else
         statement a2;
}
else
```

```
{       if(condition 3)
        statement b1;
else
        statement b2;
}
```

3.4 WAP to find largest number from 3 numbers

```
#include<stdio.h>
#include<conio.h>
main()
{
int a,b,c,large; clrscr();
printf("enter a number:");
scanf("%d%d%d",&a,&b,&c);
if(a>b)
{
        if(a>c) large=a;
        else
        large=c;
}
Else
{
        if(b>c) large=b;
        else
        large=c;
 }
 printf("largest number is %d\n",large);
 getch();
 }
```

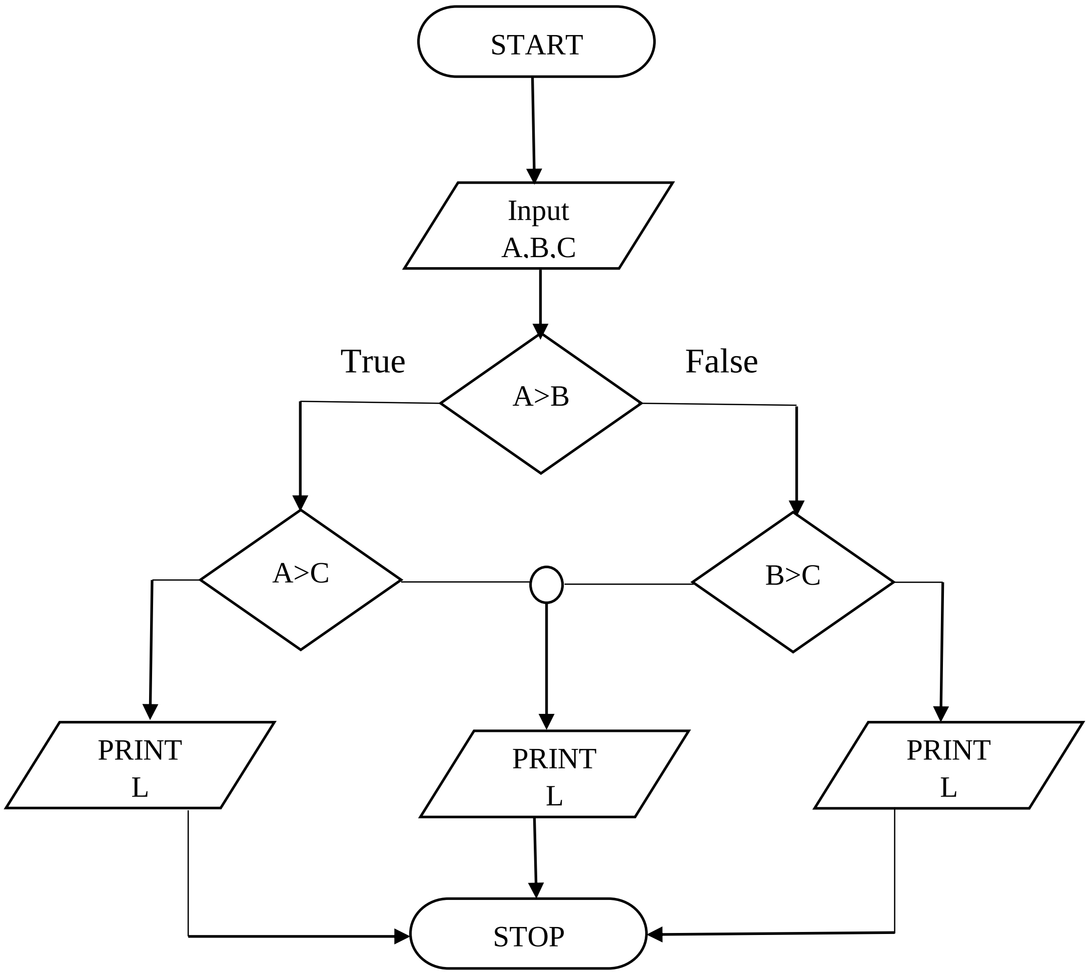
Output:

```
enter a number
5 4 8
largest number is 8
```

Algorithm:

```
Step 1: Read two numbers say A ,B and C
Step 2: Compare A and B
    c) If A is larger then
            L=Larger (A,C)
    d) If B is larger then
            L=Larger (B,C)
Step 3: Write L
Step 4: Stop
```

Flowchart:



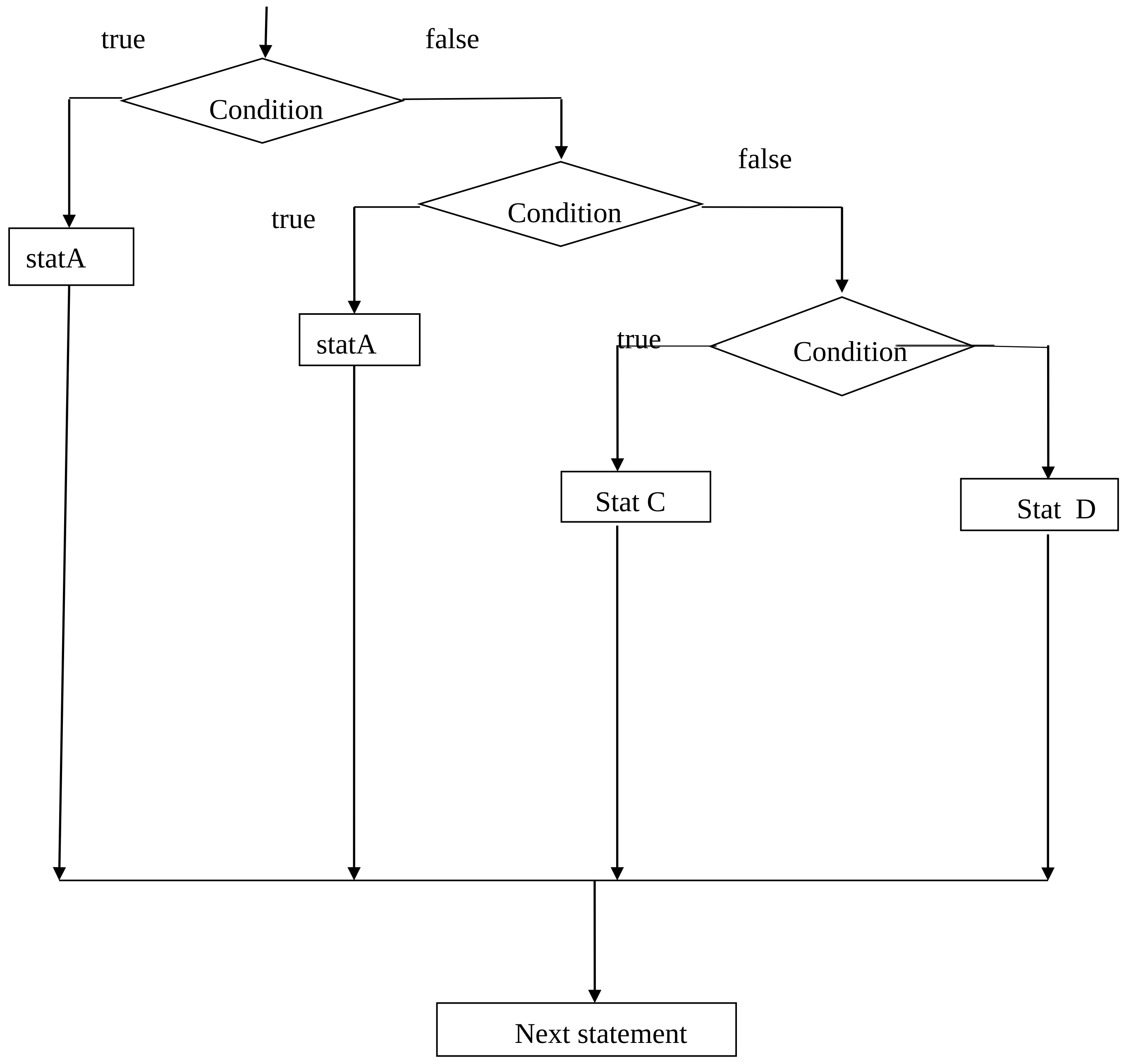
The next program finds whether a given year or not. A centennial(divisible by 100)year is leap if it is divisible by 400,and a non centennial year is leap if it is divisible by 4

3.5 program to find whether a year is leap or not

```
#include<stdio.h>
#include<conio.h>
main()
{
int  year;
clrscr();
printf("enter  year :");
scanf("%d",&year);

if(year%100==0)
{
        if(year%400==0)
        printf("leap  year\n");
        else
        printf("Not  leap\n ");
}
Else
{
        if(year%4==0)
        printf(leap year\n)
        else
        printf("Not  leap");
}
getch();
}
```

iv) else if ladder

```
if(condition 1)
        statement A;
else
        if(condition 2)
        statement B;
  else
        if(condition 3)
        statement C;
     else
           statement D;
```



Flowchart of else…..if ladder

3.6 WAP to find out the grade of a student when the marks of 4 subjects are given. The method of assigning grade is as-

Per>=85	grade=A
Per<85 and per>=55	grade=B
Per<70 and per>=55	grade=C
Per<55 and per>=40	grade=D Per<40
grade=E	

Here per is percentage.

```
#include<stdio.h>
#include<conio.h>
main()
{
float m1,m2,m3,m4,total,per; char grade;
clrscr();
printf("enter marks of 4 subjects:");
scanf("%f%f%f%f",&m1,&m2,&m3,&m4);
total=m1+m2+m3+m4;
per=total/4; printf("total=%f\n",total);
printf("percentage =%f\n",per);
if(per>=85)
grade='a';
else if(per>=70)
grade='b';
else if(per>=55)
grade='c';
else if(per>=40)
grade='d';
else
grade='e';
printf(" percentage is %f\n grade is %c\n",per,grade);
getch();
}
```

If we don’t use the else if ladder, the equivalent code for this problem would be- If(per>=85)

Grade=’A’;

If(per<85&&per>=70)

Grade=’B’;

If(per<70&&per>=55)

Grade=’C’;

If(per<55&&per>=40)

Grade=’D’; If(per<40)

Grade=’E’;

Loops:

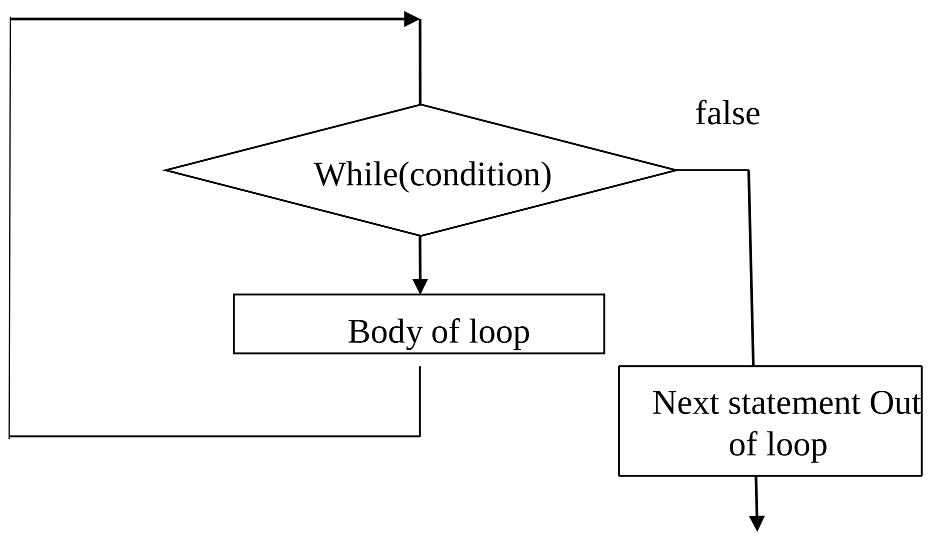
Loops are used when we want to execute of the program or a block of statements several times. For example, suppose we want to print “C is the best” 10 times. One way to get the desired output is – we write 10 printf statements, which is not preferable. Other way out is – use loop. Using loop we can write one loop statement and only one printf statement, and this approach is definitely better than the first one. With the help of loop we can execute a part of the program repeatedly till some condition is true. There are three loop statements in C-

There are 3 loop statement in C-

e) while
f) do while
g) for

a) while loop

```
 while(condition)
{          statement;
statement;
 }
```



Flowchart of while loop

3.7 Program to print the numbers from 1 to 10 using while loop

```
#include<stdio.h>
#include<conio.h>
main()
{
int i=1;
clrscr();
 while(i<=10)
{
        printf("%d\t",i);
        i=i+1;
}
printf("\n");
getch();
}
```

Output:

```
1  2  3  4  5  6  7  8  9  10
```

3.8 Program to print numbers in reverse order with a difference of 2

```
#include<stdio.h>
#include<conio.h>
main()
{
int k=10;
clrscr();
while(k>=2)
{
        printf("%d\t",k);
        k=k-2;
}
printf("\n");
getch();
}
```

Output:

```
10    8     6    4    2
```

3.9 Program to print the sum of digits of any number

```
#include<stdio.h>
#include<conio.h>
main()
{
int n, sum=0,rem;
clrscr();
printf("enter the number:");
scanf("%d",&n);
while(n>0)
{
```

```
            rem=n%10;
            sum+=rem;
             n/=10;
 }
 printf("product of digits=%d\n",sum);
 getch();
 }
```

Output:

```
enter the number:1452
sum of digits=12
```

3.10 WAP to find the product of digits of any number

```
#include<stdio.h>
#include<conio.h>
main()
{
int n,prod=1,rem;
clrscr();
printf("enter the number:");
scanf("%d",&n);
while(n>0)
{
            rem=n%10;
            prod*=rem;
            n/=10;
}
printf("prod of digits=%d\n",prod);
getch();
}
```

Output:

```
enter the number 234
product of digits=24
```

3.11 Program to find the factorial of any number

```
#include<stdio.h>
#include<conio.h>
main()
{
int n,num;
long fact=1;
clrscr();
printf("enter the number:");
scanf("%d",&n);
num=n;
 if(n<0)
printf("no factorial of negative number\n");
else
{
while(n>1)
{
        fact*=n; n--; }
        printf("factorial of %d=%ld",num,fact);
}
getch();
}
```

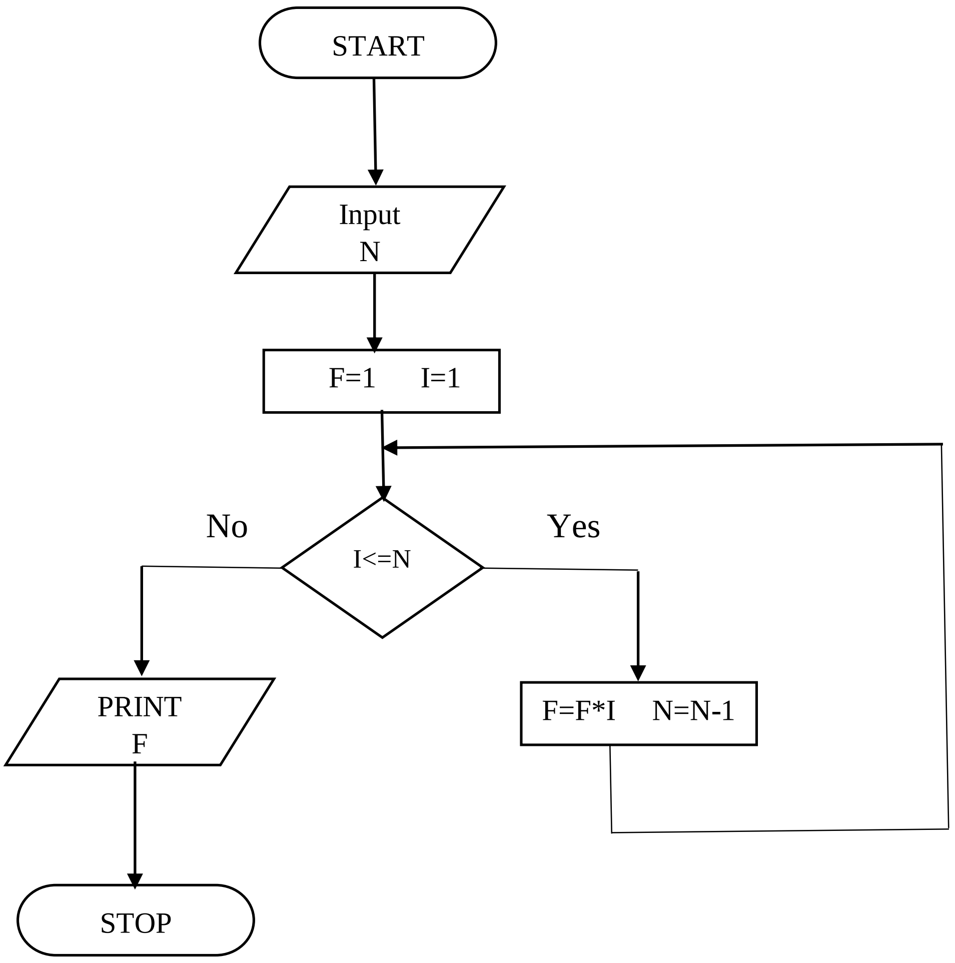
Output:

```
enter the number:4

factorial of 4: 24
```

Algorithm:

```
Step 1: Read number N
Step 2: F=1, I=1
Step 3: Repeat while I<=N
    a) F=F*I
    b) I=I+1
Step 4: Write F
Step 5: Stop
```

Flowchart:

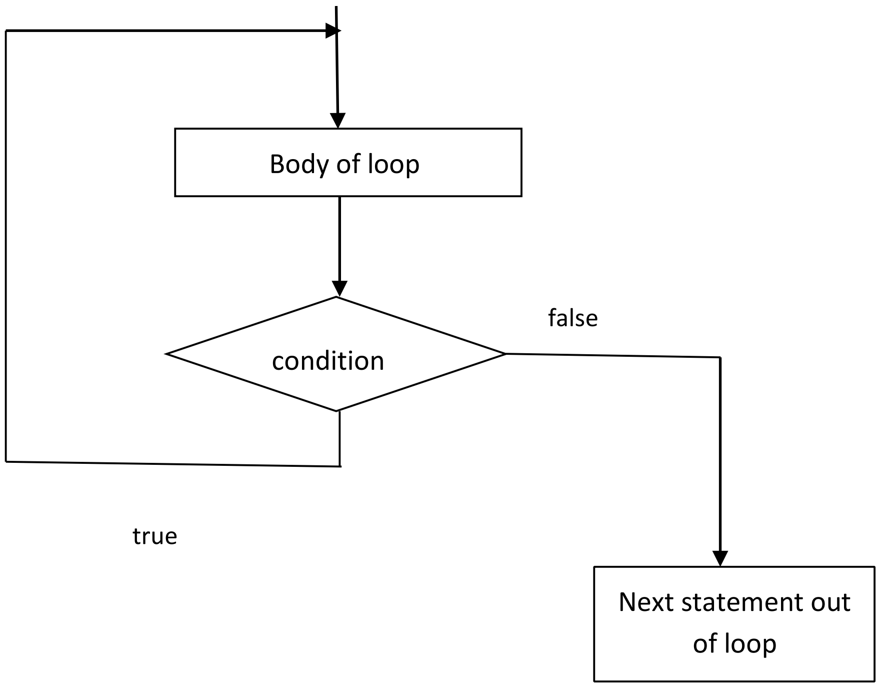


Here firstly the statement inside loop body are executed and then the condition is evaluated. If the condition is true, the again the loop body is executed and this process continues until the condition becomes false. Not that unlike while loop, here a semicolon is placed after the condition.

In a 'while' loop, first the condition is evaluated and then the statements are executed whereas in a do while loop, first the statements are executed and then the condition is evaluated. So if initially the condition is false the while loop will not execute ate all, whereas the do while loop will always execute at least once.

b) do……while loop

```
do
{
statement;
………
} while(condition);
```



Flowchart of do…while loop

3.12 WAP to print the numbers from 1 to 10 using do…while loop

```
#include<stdio.h>
#include<conio.h>
main()
{
int i=1;
clrscr();
do
{
        printf("%d\t",i);
        i=i++;
 }
while(i<=10);
 printf("\n");
getch();
}
```

Output:

```
1    2   3    4    5   6    7    8    9    10
```

3.13 WAP to count the digits in any number

```
#include<stdio.h>
#include<conio.h>
main()
{
int n,count=0,rem;
printf("enter the number ");
scanf("%d",&n);
do
{
n/=10; count++;
}
while(n>0);
printf("munber of digits=%d\n",count);
getch();
}
```

Output:

```
enter the number: 56 number of digits=2
```

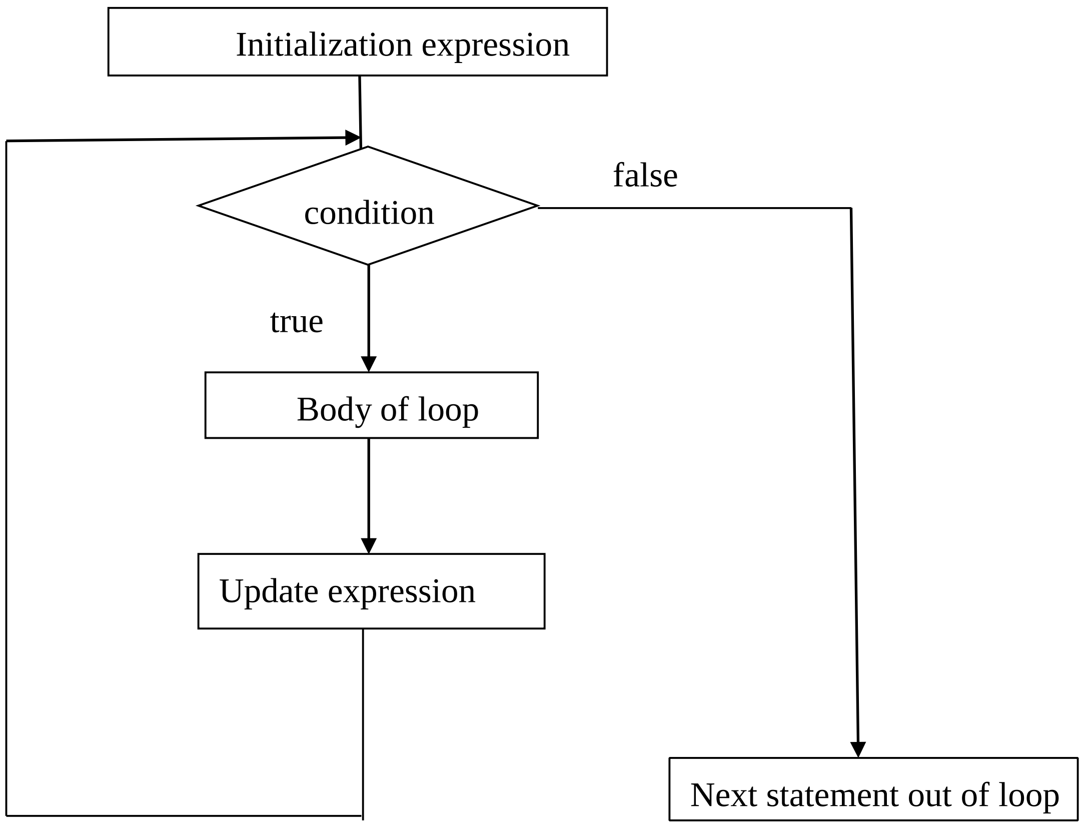
c) For loop:

The ‘for’ statement is very useful while programming in C. it has three expressions and semicolons are used for separating these expressions. The ‘for’ statement can be written as-

```
for(expression1; expression2; expression3) statement;
for(expression1; expression2; expression3)
{
Statement;
Statement;
………
}
```

The loop body can be a single statement or block of statements.

Expression 1is an initialization expression, expression2 is a test expression or condition and expression3 is an update expression. Expression1 is executed only once when the loop starts and is used to initialize the loop variables. This expression is generally an assignment expression. Expression2 is a condition and is tested before each iteration of the loop. This condition generally uses relational and logical operators. Expression3 is an update expression and is executed each time after the body of the loop is executed.



Flowchart of for loop

3.14 WAP to print the numbers in reverse order using for loop

```
#include<stdio.h>
#include<conio.h>
main()
{
int k;
clrscr();
for(k=10;k>=2;k-=2)
printf("%d\t",k);
printf("\n");
getch();
}
```

Output:

```
10    8    6    4   2
```

3.15 Find the sum of this series upto n terms 1+2+4+7+11+16+……….

```
#include<stdio.h>
#include<conio.h>
main()
{
int i,n,sum=0,term=1;
clrscr();
printf("enter number of terms");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
sum+=term; term=term+i;
}
printf("the sum of series upto %d terms is %d\n",n,sum); getch();
}
```

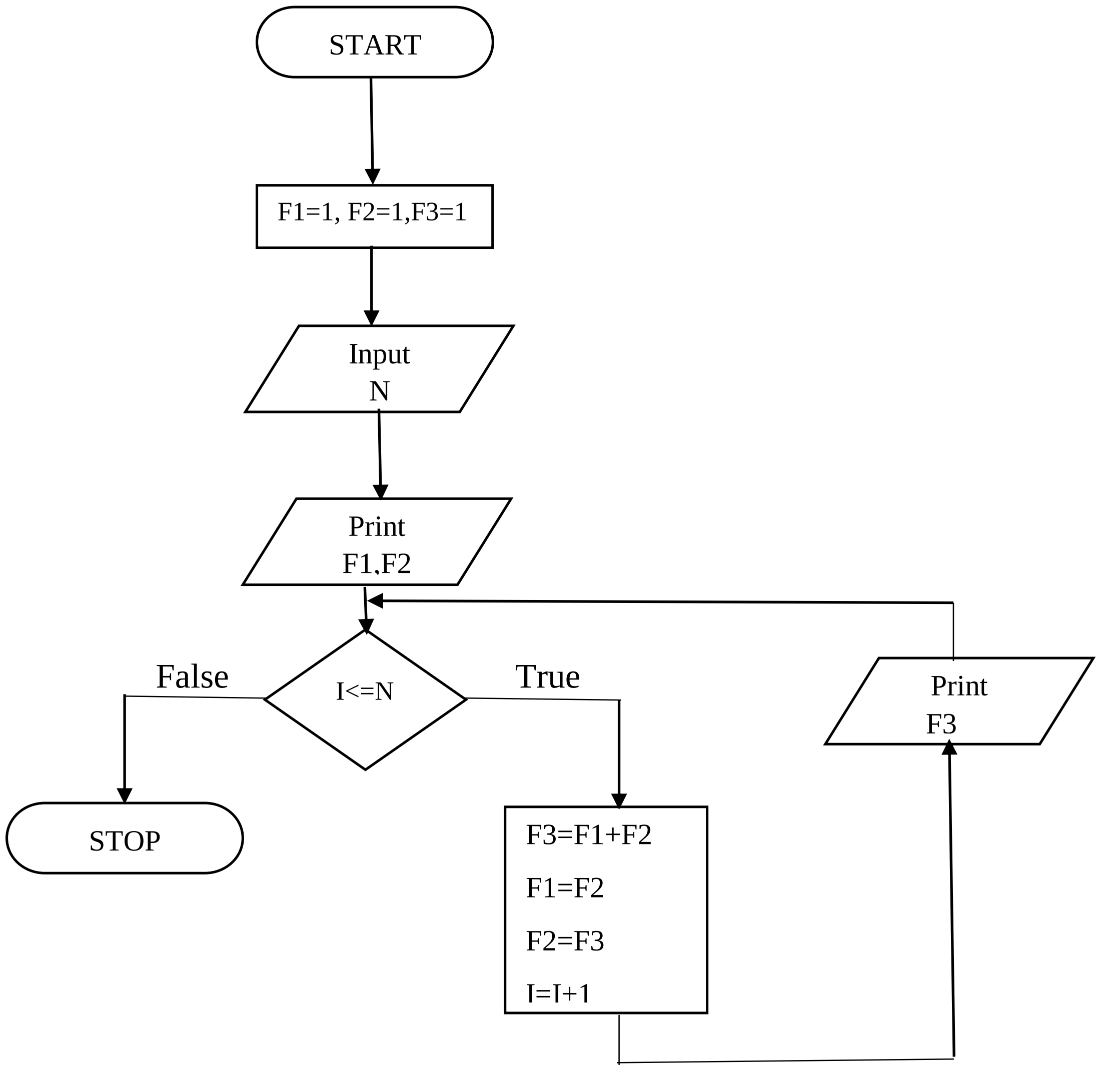
3.16 WAP to generate Fibonacci series

```
#include<stdio.h>
#include<conio.h>
main()
{
long x,y,z;
int i,n;
 x=0;
 y=1;
clrscr();
printf("enter the number of terms");
scanf("%d",&n);
printf("%ld\t",y);
 for(i=1;i<n;i++)
{
        z=x+y;
        printf("%ld\t",z);
        x=y;
        y=z;
}
printf("\n");
getch();
}
```

Algorithm:

```
Step 1: Read number N
Step 2: F1=0, F2=1
Step 3: Write F1, F2
Step 4: TERM=3
Step 5: Repeat while TERM<=N
    a) F3=F1+F2
    b) Write F3
    c) F1=F2, F2=F3
    d) TERM=TERM+1
Step 6: Stop
```

Flowchart:

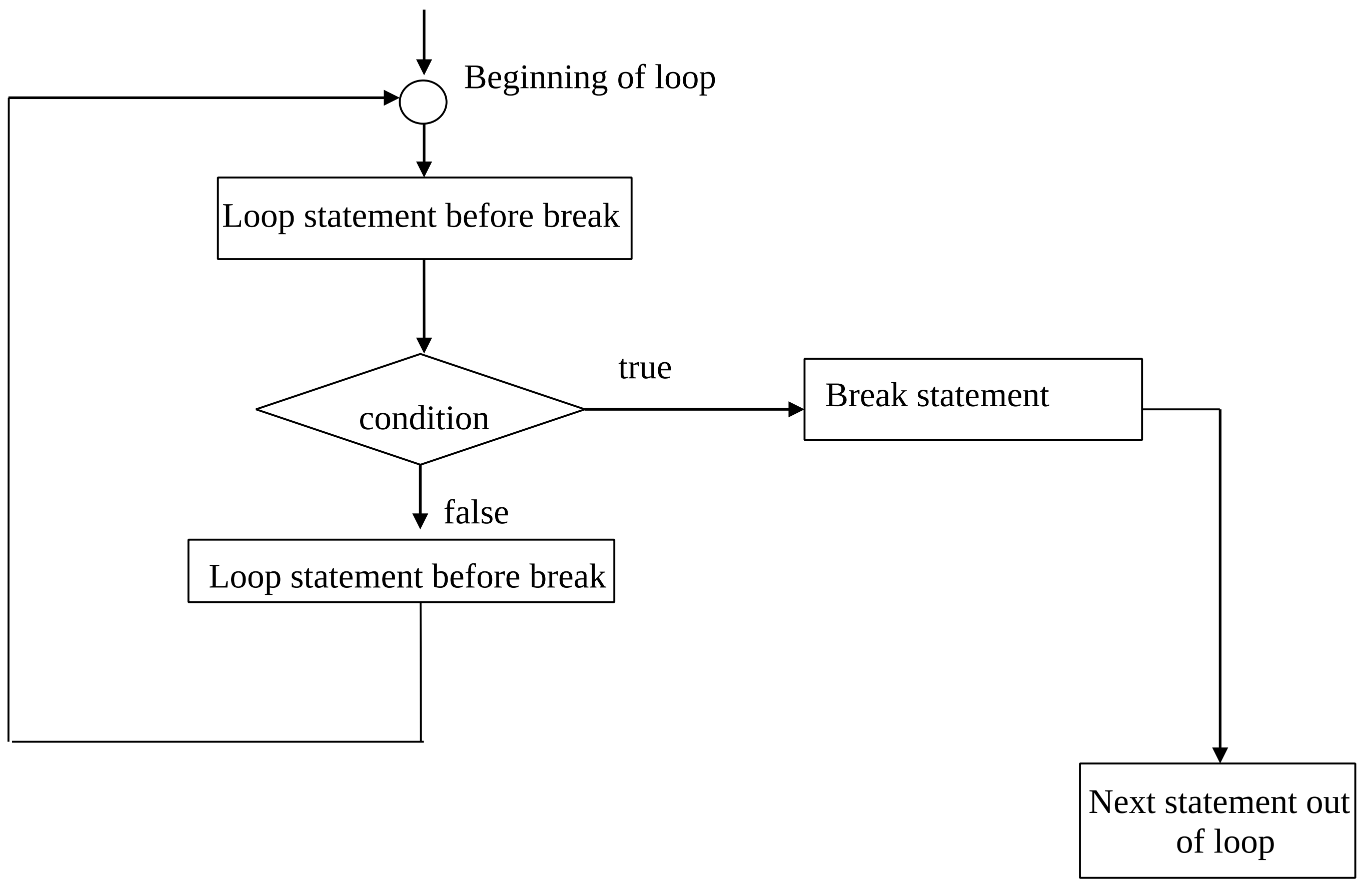


3.17 WAP to print armstrong numbers from 100 to 999

```
#include<stdio.h>
#include<conio.h>
main()
{
int num,n,cube,d,sum;
clrscr();
printf("armstrong numbers are \n");
for(num=100;num<=500;num++)
{
n=num; sum=0;
while(n>0)
{
d=n%10; n/=10; cube=d*d*d;
sum=sum+cube;
}
if(num==sum)
 printf("%d\n",num);
 }
  getch();
 }
```

Break statement

break statement: Break statement is used inside loops and switch statement. Sometimes it becomes necessary to come out of the loop even before the loop condition becomes false. In such a situation, break statement is used to terminate the loop. This statement causes an immediate exit from that loop in which this statement appears. It can be written as- break; when break statement is encountered, loop is terminated and the control is transferred to the statement immediately after the loop. The break statement is generally written along with a condition. If break is written inside a nested loop structure then it causes exit from the innermost loop.


Beginning of loop
Loop statement before break
true
condition
Break statement
false
Loop statement before break
Next statement out of loop

3.18 WAP to understand the used of break statement

```
#include<conio.h>
main()
{
int n;
clrscr();
for(n=1;n<=5;n++)
{
 if(n==3)
 {
printf("i understand the use of break\n"); break;
}
printf("number =%d\n",n);
}
printf("out of for loop\n");
getch();
}
```

Output:

```
number=1 number=2
i understand the used of break out of for loop
```

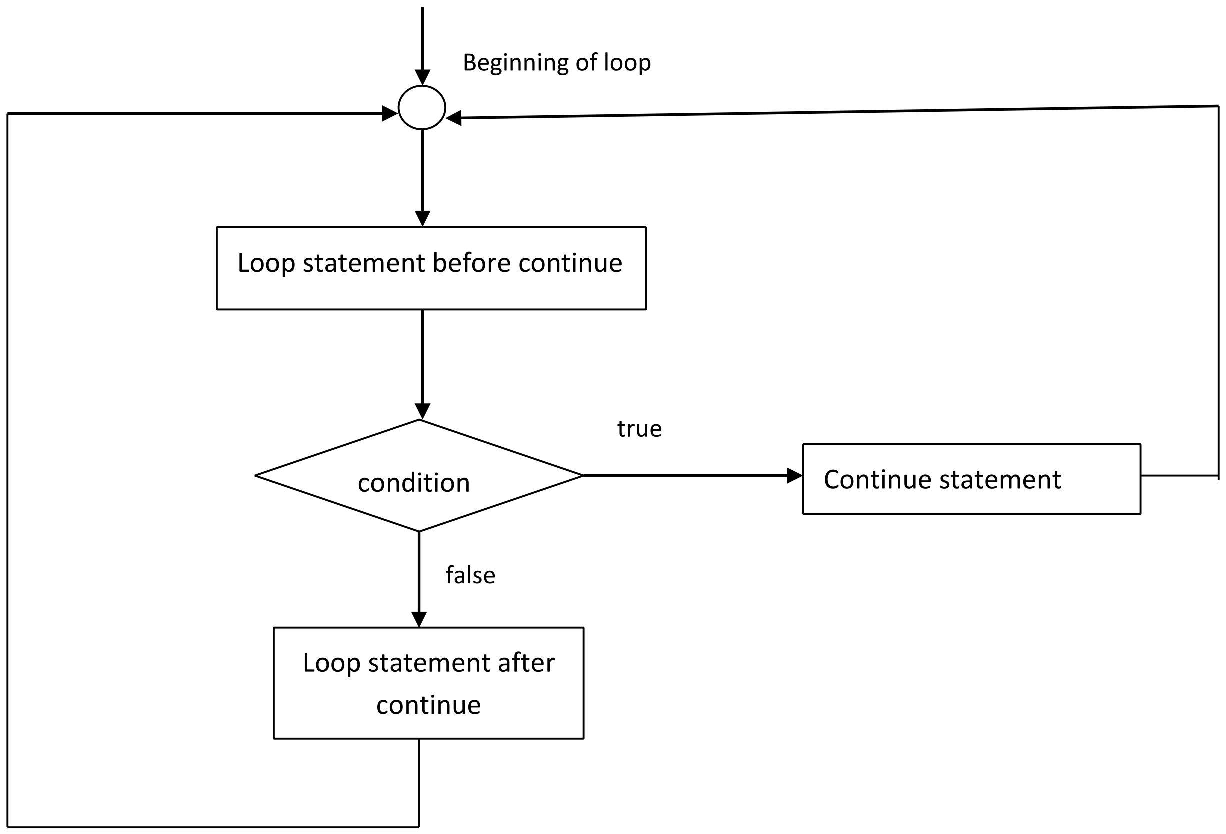
Continue Statement:

The continue statement is used when we want to go to the next iteration of the loop after skipping some statements of the loop. This continue statement can be written simply as- Continue;

It is generally used with a condition. When continue statement is encountered all the remaining statements (statements after continue) in the current iteration are not executed and the loop continues with the next iteration.

The difference between break and continue is that when break is encountered the loop terminates and the control is transferred to the next statement following the loop, but when a continue statement is encountered the loop is not terminated and the control is transferred to the beginning of the loop.

In while and do-while loops, after continue statement the control is transferred to the test condition and then the loop continues, whereas in for loop after continue statement the control is transferred to update expression and then the condition is tested.



Continue (control statement)

3.19 WAP to understand the used of continue statement

```
#include<conio.h>
#include<math.h>
main()
{
int n;
clrscr();
for(n=1;n<=5;n++)
{
if(n==3)
{
printf("i under stand the use of continue\n ");
continue;
}
printf("number=%d\n",n);
}
printf("outof for loop\n");
getch();
}
```

Output:

```
number=1 number=2
i understand the used of continue
number=4 number=5
out of for loop
```

3.20 WAP to find the sum and average of 10 positive integers

```
#include<stdio.h>
#include<conio.h>
main()
{
int i=1,n,sum=0;
float avg;
clrscr();
printf("enter 10 positive num\n");
while(i<=10)
{
        printf("enter number %d",i);
        scanf("%d",&n);
         if(n<0)
         {
printf("enter only positive numbers\n"); continue;
}
sum=sum+n;
i++;
}
avg=sum/10.0;
printf("sum=%d avg=%f\n",sum,avg);
getch();
}
```

goto statement

This is an unconditional control statement that transfers the flow of control to another part of the program.

This control statement can be used as- goto label1;

```
………..
………….. label1: statement;
………
…………..
```

3.21 WAP to print whether the number is even or odd

```
#include<stdio.h>
 #include<conio.h>
 main()
{
int n; clrscr();
printf("enter the number");
 scanf("%d",&n);
if(n%2==0)
goto even;
else
 goto odd;
even: printf("number is even");
 goto end;
 odd: printf("number is odd");
goto end;
end: printf("\n"); getch();
}
```

Output:

```
enter the number:14 number is even
```

Switch-case

```
switch(expression)
{ case constant1; statement
……..
case constant2; statement
……….
……….
………
case constant; statement
…….. default: statement
……….
}
```

This is a multi-directional conditional control statement. Sometimes there is a need in program to make choice among number of alternatives. For making this choice, we use switch statement. This can be written as-

Here switch, case and default are keywords. The “expression” following the switch keyword can be any C expression that yields an integer value. It can be value of any integer or character variable, or a function call returning an integer, or an arithmetic, logical, relational, bitwise expression yielding an integer. It can be any integer or character constant also. Since characters are converted to their ASCII values, so we can also use characters in this expression. Data types long int and short int are also allowed.

The constants following the case keywords should be of integer or character type. They can be either constants or constant expressions. These constants must be different from one another.

We can’t use be floating point or string constants. multiple statements in a single case are not allowed; each case should be followed by only one constant.

3.22 WAP to understand the switch control statement

```
#include<stdio.h>
#include<conio.h>
main()
{
int choice;
 clrscr();
printf("enter your choice");
scanf("%d",&choice);
switch(choice)
{
case 1:
printf("first\n");
 case 2:
printf("second\n");
case 3:
printf("third\n");
 default:
printf("wrong choice\n");
}
getch();
}
```

3.23 WAP to understand the switch with break statement

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

```
main()
{
int choice;
clrscr();
printf("enter your choice");
scanf("%d",&choice);
 switch(choice)
{
case 1:
printf("first\n");
break;
case 2:
printf("second\n");
break;
case 3:
printf("third\n");
 break;
default:
printf("wrong choice\n");
}
getch();
}
```

3.24 WAP to perform arithmetic calculations on integers

```
#include<stdio.h>
#include<conio.h>
 main()
{ char op;
int a,b;
clrscr();
printf("enter number operator and another number");
scanf("%d%c%d",&a,&op,&b);
switch(op)
{
 case '+':
           printf("result=%d\n",a+b);
           break;
case '-':
           printf("result=%d\n",a-b);
           break;
case '*':
           printf("result=%d\n",a*b);
           break;
case '/':
           printf("result=%d\n",a/b);
            break;
case '%':
           printf("result=%d\n",a%b);
           default:
printf("wrong choice\n");
}
getch();
}
```

3.25 WAP to find whether the alphabet is a vowel or consonant

```
#include<stdio.h>
#include<conio.h>
main()
{
char ch;
 clrscr();
printf("enter an alphabet");
scanf("%c",&ch);
switch(ch)
{
 case 'a':
 case 'e':
case 'i':
case 'o':
 case 'u':
printf("alphabet is a vowel\n");
 break;
 default:
printf("alphabet is a consonant\n");
}
getch();
}
```

CHAPTER 4

FUNCTIONS

Function

A function is a self-contained subprogram that is meant to do some specific, well defined task. A C-program consists of one or more functions. If a program has only one function then it must be the main() function.

A large C program is divided into basic building blocks called C function. C function contains set of instructions enclosed by "{ }" which performs specific operation in a C program. Actually, Collection of these functions creates a C program.

Uses of functions

- C functions are used to avoid rewriting same logic/code again and again in a program.
- There is no limit in calling C functions to make use of same functionality wherever required.
- We can call functions any number of times in a program and from any place in a program.
- A large C program can easily be tracked when it is divided into functions.
- The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large C programs.

Advantages of using functions

1. Generally a difficult program is divided into sub problems and then solved. This divide and conquer technique is implemented in C through functions. A program can be divided into functions, each of which performs some specific task. So the use of C function modularizes and divides the work of a program.

2. When some specific code is to be used more than once and at different places in the program, the use of functions avoids repetition of that code.

3. The program becomes easily understandable, modifiable and easy to debug and test. It becomes simple to write the program and understand what work is done by each part of the program.

4. Function can be stored in a library and reusability can be achieved.

C program have two types of function:

1. Library function
2. User-defined function

1. Library functions

C has the facility to provide library functions for performing some operations. These function are present in the C library and they are predefine. For example sqrt() is a mathematical library function which is used for finding out the square root of any number. The functions scanf() and printf() are input output library functions. Similarly we have like strlen(), strcmp() for string manipulations.

To use a library function we have to include corresponding header file using the preprocessor directive #include. For example to use input output function like printf(), scanf() we have to include stdio.h, for ma thematically library functions we have to include math.h for string library string.h should be include. The following program illustrates the use of library function squr().

4.1 Program to find the square root of any number

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
main()
{
double n,s;
clrscr();
printf("enter the number :");
scanf("%lf",&n);
s=sqrt(n);
printf("the square root of %.2lf is :%.2lf\n", n,s);
getch();
}
```

Output:

```
enter a number:16
the square root of 16.00 is:4.00
```

2. User defined functions

User can create their own functions for performing any specific task of the program . This types of functions are called user-defined functions. To create and use these function, we should know about these 3 things-

1. Function definition
2. Function declaration
3. Function call

Before discussing these three points we have written two simple programs that will be used for reference the first program draws a line and the second one add two numbers.

4.2 Program to draw a line*

```
#include<stdio.h>
#include<conio.h>
void drawline(void);
main()
{
clrscr();
drawline();
getch();
}

void drawline(void)
{
        int i;
        for(i=1;i<=8;i++)
        printf("-");
}
```

Output:

```
********
```

4.3 Program to find the multiplication of three numbers

```
#include<stdio.h>
#include<conio.h>
int mult(int x,int y,int z);
main()
{
int a,b,c,s;
clrscr();
printf("enter the values of a,b and c");
scanf("%d%d%d",&a,&b,&c);
s=mult(a,b,c);
printf("product of %d %d and %d is %d\n",a,b,c,s);
getch();
}


int mult(int x,int y, int z)
{
        int s;
        s=x*y*z; return s;
}
```

Function definition

The function definition consists of the whole description and code of a function. It tells what the function is doing and what are its input and outputs. A function definition consists of two parts – a function header and a function body. The general syntax of a function definition is

```
return_type  func_name(type1 arg1, type2 arg2,………)
 {
      local variables declarations;
          statement;
          ……………..
     return(expression);
}
```

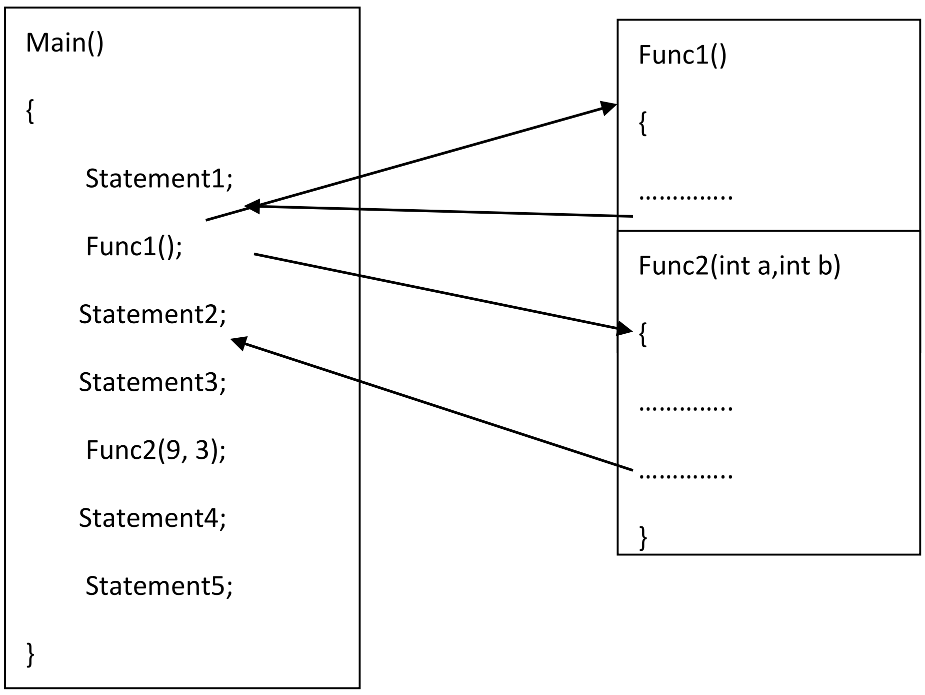
- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** − This is the actual name of the function. The function name and the 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. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

Function call

The function definition describes what a function can do, but to actually use it in the program the function should be called somewhere. A function is called by simply writing its name followed by the argument list inside the parentheses.

Func _ name(arg1,arg2,arg3…)

These argument arg1, arg2, …are called actual arguments.



Transfer of control when function is called

Function Declaration

The calling need information about the called function. If definition of the called function is placed before the calling function, then declaration is not needed. For example if in program p4.3, we write the definition of sum() before main(), then declaration is not needed

4.4 Program to find the sum of two numbers

```
#include<stdio.h>
#include<conio.h>
int sum(int x,int y)
{
int s;
s=x+y;
return s;
}
```

```
main()
{
int a,b,s;
clrscr();
printf("enter the value for a and b");
scanf("%d%d",&a,&b);
s=sum(a,b);
printf("sum of %d and %d is %d\n",a,b,s);
getch();
}
```

Here the definition of sum() is written before main(), so main() knows everything about the function sum(). But generally the function main() is placed at the top and all function are placed after it. In this case, function declaration is needed. The function declaration is also known as the function prototype, and it inform the complier about the following three things:-

1) Name of the function.
2) Number and type of arguments received by the function.
3) Type of value returned by the function.

Function declaration tells the compiler that a function with these features will be define and used later the program. The general syntax of a function declaration is-

Return_type func_name(type1 arg1,type2 arg2,……..);

This looks just the header of function definition, except that there is a semicolon at the end. The names of the arguments while declaring a function are optional. These are only used for descriptive purpose. So we can write the declaration in this way also-

Return_type func_name (type1, type2,……);

4.5 Program that finds whether a number is even or odd

```
#include<stdio.h>
#include<conio.h>
void find(int n);
main()
{
int num;
clrscr();
printf("enter a number");
scanf("%d",&num); find(num);
getch();
}

void find(int n)
{
if(n%2==0)
        printf("%d is even\n",n);
else
        printf("%d is odd\n",n);
}
```

4.6 Program that finds the larger of two numbers

```
#include<stdio.h>
#include<conio.h>
int max(int x,int y);
main()
{
int a,b;
clrscr();
printf("enter two numbers");
scanf("%d%d",&a,&b);
printf("maximum of %d and %d is: %d\n",a,b,max(a,b));
getch();
}

max(int x,int y)
{
        if(x>y) return x; else return y;
}
```

4.7 Program to understand the use of return statement

```
#include<stdio.h>
#include<conio.h>
void funct(int age,float ht);
main()
{
int age;
float ht;
clrscr();
printf("enter age and height:");
scanf("%d %f",&age,&ht);
funct(age,ht);
getch();
}

void funct(int age,float ht)
{
        if(age>25)
        {
        printf("age should be less than 25\n");
        return;
        }
If(ht<5)
{
        printf(“height should be more than 5\n”);
        return;
}
        printf("selected\n");
}
```

4.8 Program to find out the factorial of a number

```
#include<stdio.h>
#include<conio.h>
long int factorial(int n);
main()
{
int num;
clrscr();
printf("enter a number");
scanf("%d",&num);
if(num<0)
printf("no factoruial of negative number\n");
else
printf("factorial of %d is %ld\n",num,factorial(num));
getch();
}

long int factorial(int n)
{
int i;
long int fact=1;
if(n==0)
        return 1;
for(i=n;i>1;i--)
        fact*=i;
        return fact;
}
```

C functions aspects	**Syntax**
function definition	Return_type function_name (arguments list) { Body of function; }
function call	function_name (arguments list);
function declaration	return_type function_name (argument list);

Function Arguments

The calling function sends some values to the called function for communication; these values are called arguments or parameters.

Actual Arguments

The arguments which are mentioned in the function call are known as actual arguments, since these are the values which are actually set to the called function. Actual arguments can be written in the form of variables, constants or expressions or any function call that returns a value. For example-
Fun(x)

Func(a*b, c*d+k)

Func(22, 43)

Func(1, 2, sum(a, b))

Formal Argument

The name of the arguments, which are mentioned in the function definition are called formal or dummy arguments since they are used just to hold the values that are sent by calling function.

These formal arguments are simply like other local variables of the function which are created when the function call starts and are destroyed when the function ends. However there are two differences. First is that formal arguments are declared inside parentheses which other local variables are declared at the beginning of the function block. The second difference is that formal arguments are automatically initialized with the values of the actual arguments passed, while other local variables are assigned values through the statements written inside the function body.

The order, number and type of actual arguments in the function call should match with the order, number and type of formal arguments in the function definition.

4.9 Program to understand formal and actual arguments

```
#include<stdio.h>
#include<conio.h>
func(int a,int b);
main()
{
int a=6,b=3;
clrscr();

func(a,b);
func(15,4);
func(a+b,a-b);
getch(); }

func(int a,int b)
{
        printf("a=%d b=%d\n",a,b);
}
```

Output:

```
a=6   b=3
a=15 b=4
a=9 b=3
```

4.10 Program to understand formal and actual arguments

```
#include<stdio.h>
#include<conio.h>
multiply(int x,int y);
sum(int x, int y);
main()
{
int m=4,n=2;
clrscr();
printf("%d\t",multiply(m,n));
printf("%d\t",multiply(15,4));
printf("%d\t",multiply(m+n,m-n));
printf("%d\n",multiply(6,sum(m,n)));
getch();
}

multiply(int x,int y)
{
        int p;
        p=x*y;
        return p;
}

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

Output:

```
18 60 27 54
```

Types of User Defined function

A function in C can be called either with arguments or without arguments. These function may or may not return values to the calling functions. All C functions can be called either with arguments or without arguments in a C program. Also, they may or may not return any values.

1. Function with no arguments and no return value
2. Function with no arguments and a return value
3. Function with arguments and no return value
4. Function with arguments and a return value

1. Function with no arguments and no return value

```
void func(void);
main()
{
………..
func();
}
void func(void)
{
………………
statement;
……………
}
```

4.11 Program that uses a function with no arguments and no return values

```
#include<stdio.h>
#include<conio.h>
void dispmenu(void);
main()
{
int choice;
clrscr();
dispmenu();
printf("enter your choice");
scanf("%d",&choice);
getch();
}

void dispmenu(void)
{
printf("1.create database\n");
printf("2.insert new record\n");
printf("3.modify a record\n");
printf("4.delete a record\n");
printf("5.display all records\n");
printf("6.exit\n");
}
```

2.function with no arguments and a return value

```
int func(void);
main()
{
    int r;
    ………………..
    r=func();
     ………………..
}
int func(void)
{
   …………….
   …………….
   return(expression);
}
```

4.12 Program that returns the sum of squares of all odd numbers from 1 to 25

```
#include<stdio.h>
#include<conio.h>
int func(void);
main()
{
clrscr();
printf("%d\n",func());
getch();
}
int func(void)
{
int num,s=0;
for(num=1;num<=25;num++)
{
if(num%2!=0);
s+=num*num;
}
return s;
getch();
}
```

Output:

```
2925
```

3.function with no arguments but no return value

```
void func(int,int);
main()
{
  ………………
  func(a,b);
  …………….
}
void func(int c,int d)
{
   ……………………
   statements
  …………………….
}
```

4.13 Program to find the types and area of a triangle

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void type(float a,float b,float c);
void area(float a,float b,float c);
main()
{
float a,b,c;
printf("enter the sides of triangle");
scanf("%f%f%f",&a,&b,&c);
if(a<b+c && b<c+a && c<a+b)
{
        type(a,b,c);
area(a,b,c);
}
else
        Printf("no triangle possible with these sides\n");
}
void type(float a, float b, float c)
{
        if((a*a)+(b*b)==(c*c)||(b*b)+(c*c)==(a*a)||(c*c)+(a*a)==(b*b))
```

```
                printf("the triangle is right angled triangle\n");
        if(a==b && b==c)
                printf("the triangle is equilateral\n");
        else if(a==b || b==c || c==a)
                printf("the triangle is isosceles\n");
        else
                printf("the triangle is scalene\n");
}

void area(float a, float b, float c)
{
float  s, area;
s=(a+b+c)/2;
area=sqrt(s*(s-a)*(s-b)*(s-c));
printf("the area of triangle=%f\n",area);
}
getch();
}
```

4.function with arguments and a return value

```
int func(int,int);
main()
{
    int r;
    ………….
   r=func(a,b);
   …………..
    func(c,d);
}
int func(int a,int b)
{
  ………………….
```

```
   …………………
  return (expression);
}
```

4.14 Program to find the sum of digits of any number

```
#include<stdio.h>
#include<conio.h>
int sum(int n);
main()
{
int num;
clrscr();
printf("enter the number");
scanf("%d",&num);
printf("sum of digits of %d is %d\n",num,sum(num));
getch();
}

int sum(int n) { int i,sum=0,rem; while(n>0)
{
rem=n%10; sum+=rem;
n/=10; }
return(sum);
}
```

CHAPTER 5

ARRAY

The variables that we have used till now are capable of storing only one value at time. Consider a situation when we want to store and display the age of 100 employees. For this we have to do the following-

1. Declare 100 different variables to store the age employees.
2. Assign a value to each variable.
3. Display the value of each variable.

Although we can perform our task by the above three steps but just imagine how difficult it would be to handle so many variable in the program and the program would become very lengthy. The concept of array is useful in these types of situations where we can group similar type of data items.

An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the elements may be any valid data type like char, int or float. The elements of array share the same variable name but each element has a different index number known as subscript.

For the above problem we can take an array variable age[100] of type int. the size of this array variable is 100 so it is capable of storing 100 integer values. The individual elements of this array are- age[0], age[1], age[2], age[3], age[4],……age[98], age[99] In C the subscripts start from zero, so age[0] is the first element, age[1] is the second element of array and so on.

Arrays can be single dimensional or multidimensional. The number of subscripts the dimension of array. A one-dimensional array has one subscript, two dimensional array has two subscripts and so on. The one-dimensional arrays are known as vectors and two-dimensional arrays are known as matrices. In this chapter first we will study about single dimensional arrays and move on to multi-dimensional arrays. The one-dimensional array are known as vectors and two dimensional arrays are known as matrices.

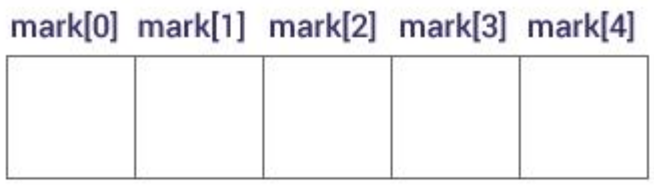
Elements of an Array and How to access them?

For example: float mark[5];

Here, we declared an array, mark, of floating-point type and size 5. Meaning, it can hold 5 floating-point values.

You can access elements of an array by indices.

Suppose you declared an array mark as above. The first element is mark[0], second element is mark[1] and so on.



Few key notes:

- Arrays have 0 as the first index not 1. In this example, mark[0].
- If the size of an array is n, to access the last element, (n-1) index is used. In this example, mark[4].
- Suppose the starting address of mark[0] is 2120d. Then, the next address, a[1], will be 2124d, address of a[2] will be 2128d and so on. It's because the size of a float is 4 bytes.

How to initialize an array in C programming?

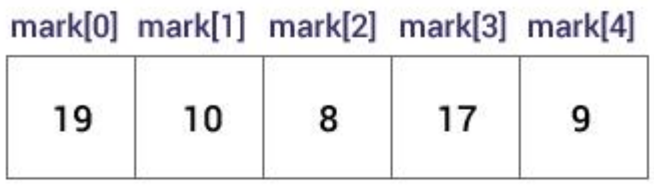
It's possible to initialize an array during declaration.

For example,

```
int mark[5] = {19, 10, 8, 17, 9};
```

Another method to initialize array during declaration:

```
int mark[] = {19, 10, 8, 17, 9};
```



Here,

Here mark[0] is equal to 19
Here mark[1] is equal to 10
Here mark[2] is equal to 8
Here mark[3] is equal to 17

Here mark[4] is equal to 9

One-Dimensional Array

Declaration of 1-D Array

Like other simple variables, arrays should also be declared before they are used in the program. The syntax for declaration of an array is-

data_type array_name[size];

Here array_name denotes the name of the array and it can be any valid C identifier, data_type is the data type of the elements of array. The size of the array specifies the number of elements that can be stored in the array. It may be a positive integer constant or constant integer expression. Here are some examples of array declarations-

```
int age[100];
float sal[15];
char grade[20];
```

Here age is an integer type array, which can store 100 elements of integer type. The array sal is floating array of size 15, can hold float values and third one is a character type array of size 20, can hold character. The individual elements of the above array are-

age[0], age[1], age[2]………………………………..age[99]

sal[0], sal[1], sal[2]………………………sal[14]

grade[0], grade[1], grade[2]…………………………grade[19]

When the array is declared, the complier allocates space in memory sufficient to hold all the elements of the array, so the complier should know the size of array at the compile time. Hence we can't use variable for specifying the size of array in the declaration. The symbolic constant can be used to specify the size of array.

For example:

```
    define SIZE 10
    main()

 {
        Int size=5;
        Float sal[SIZE];
        It marks[SIZE];
        ………………..
        ………………...
    }
```

He use of symbolic constant to specify he size of array makes it convenient to modify the program if the size of array is to be changed later, because the size has to be changed only at one place i.e. in the #define directive.

5.1 Program to input values into an array and display them

```
#include<stdio.h>
#include<conio.h>
main()
{
int arr[3],i;
clrscr();

for(i=0;i<5;i++)
{
printf("enter the values of arr[%d]:",i);
scanf("%d",&arr[i]);
}
printf("the array elements are :\n");
for(i=0;i<5;i++)
printf("%d\t",arr[i]);
printf("\n"); getch();
}
```

Output:

```
enter the value for arr[0]:12
enter the value for arr[1]:45
enter the value for arr[2]:59
enter the value for arr[3]:98
enter the value for arr[4]:21
the array elements are:
12    45    59    98    21
```

5.2 Program to find the maximum and minimum number in an array

```
#include<stdio.h>
#include<conio.h>
main()
{
int i,j,arr[10]={2,5,4,1,8,9,11,6,3,7};
int min,max;
min=max=arr[0];
for(i=1;i<10;i++)
{
if(arr[i]<min)
        min=arr[i];
if(arr[i]>max)
        max=arr[i];
}
printf("minimum=%d, maximum=%d\n",min,max);
}
```

Output:

```
minimum=1,maximum=11
```

5.3 Program to reverse the elements of an array

```
#include<stdio.h>
#include<conio.h>
main()
{
int i,j,temp,arr[10]={1,2,3,4,5,6,7,8,9,10};
clrscr();
for(i=0,j=9;i<j;i++,j--)
{
temp=arr[i]; arr[i]=arr[j];
arr[j]=temp;
}
printf("after reversing the array is ");
for(i=0;i<10;i++)
printf("%d  ",arr[i]);
printf("\n");
getch();
}
```

Output:

```
after reversing the array is 10   9   8   7   6   5    4   3   2   1
```

5.4 Program to search for an item in the array

```
#include<stdio.h>
#include<conio.h>
#define size 7
main()
{
int i,arr[size]={44,75,26,47,85,39,510};
int item;
clrscr();
printf("enter the item to be search");
scanf("%d",&item);
for(i=0;i<size;i++)
{
if(item==arr[i])
{
        printf("%d found at positive %d\n",item,i+1);
        break;
}
}
if(i==size)
        printf ("item %d not found in array\n",item);
        getch();
}
```

Two dimensional array

The syntax of declaration of 2-D array is similar to that of 1-D arrays, but here we have two subscript.

data_type array_name[rowsize][columnsize];

Here rowsize specifies the number of rows and columnsize represents the number of columns in the array. The total number of elements in the array are rowsize * columnsize. For example- Int arr[4][5];

Here arr is a 2-D array with 4 rows and 5 columns. The individual elements of this array can be accessed by applying two subscript, where the first subscript denotes the row number and the second subscript denotes the column number. The starting element of this array is arr[0][0] and the last element is arr[3][4]. The total number of elements in this array is 4*5 = 20.

	Col 0	**Col 1**	**Col 2**	**Col 3**	**Col4**
Row 0	arr[0][0]	arr[0][1]	arr[0][2]	arr[0][3]	arr[0][4]
Row 1	arr[1][0]	arr[1][1]	arr[1][2]	arr[1][3]	arr[1][4]
Row 2	arr[2][0]	arr[2][1]	arr[2][2]	arr[2][3]	arr[2][4]
Row 3	arr[3][0]	arr[3][1]	arr[3][2]	arr[3][3]	arr[3][4]

How to initialize a 2-dimensional array?

There is more than one way to initialize a multidimensional array.

Initialization of a two dimensional array.

Different ways to initialize two dimensional array

```
int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[2][3] = {1, 3, 0, -1, 5, 9};
```

Above code are three different ways to initialize a two dimensional arrays.

5.5 Program to input and display a matrix.

```
#define row 3
#define col 4
#include<stdio.h>
#include<conio.h>
main()
{
int mat[row][col],i,j;
clrscr();
printf("enter the elements of matrix(%dx%d) rowwise :\n",row,col);
for(i=0;i<row;i++) for(j=0;j<col;j++)
scanf("%d",&mat[i][j]);
printf("the matrix that you have entered is :\n");
for(i=0;i<row;i++)
{
        for(j=0;j<col;j++)
        printf("%5d",mat[i][j]);
printf("\n");
```

```
 }
printf("\n");
getch();
}
```

Output:

```
enter the elements of matrix(3*4)row-wise-
2   3   4   7
8   5   1   9
1    8   2   5
the matrix that you have entered is-
2    3    4   7
8    5    1   9
1    8    2   5
```

5.6 Program for addition of two matrices

```
#define row 3
#define col 4
#include<stdio.h>
#include<conio.h>
main()
{
int i,j,mat1[row][col],mat2[row][col],mat3[row][col];
clrscr();
printf("enter matrix mat1(%dx%d) rowwise :\n",row,col);

for(i=0;i<row;i++)
for(j=0;j<col;j++)
scanf("%d",&mat1[i][j]);
printf("enter matrix mat2(%dx%d)rowwise:\n",row,col);
for(i=0;i<row;i++)
for(j=0;j<col;j++)
scanf("%d",&mat2[i][j]);

for(i=0;i<row;i++)
for(j=0;j<col;j++)
mat3[i][j]=mat1[i][j]+mat2[i][j];
printf("the resultant matrix mat3 is \n");

for(i=0;i<row;i++)
{
for(j=0;j<col;j++)
printf("%5d",mat3[i][j]);
printf("\n");
}
getch();
}
```

Output:

```
enter elements of first matrix mat1(3*4)row-wise-
1    2    8    4
5    6    7   8
3    2    1   4
enter elements of first matrix mat2(3*4)row-wise-
2    5    4    2
1    5    2    6
9     4    7    2

the resultant matrix 3 –
3     7     12    6
6    11     9    14
12    6      8    6
```

5.7 Program for multiplication of two matrices

```
#include<stdio.h>
#include<conio.h>
#define row1 3
#define col1 4
#define row2 col1
#define col2 2
main()
{
int i,j,k,mat1[row1][col1],mat2[row2][col2],mat3[row1][col2];
clrscr();
printf("enter matrix mat1(%dx%d) rowwise :\n",row1,col1);

for(i=0;i<row1;i++)
for(j=0;j<col1;j++)
scanf("%d",&mat1[i][j]);
printf("enter matrix mat2(%dx%d)rowwise:\n",row2,col2); for(i=0;i<row2;i++)


for(j=0;j<col2;j++)
scanf("%d",&mat2[i][j]);
for(i=0;i<row1;i++)
for(j=0;j<col2;j++)
{
        mat3[i][j]=0;
        for(k=0;k<col1;k++)
        mat3[i][j]+=mat1[i][k]*mat2[k][j];
}
 printf("the resultant matrix mat3 is :\n");
 for(i=0;i<row1;i++)
{
```

```
for(j=0;j<col2;j++)
printf("%5d",mat3[i][j]);
printf("\n");
}
getch();
}
```

5.8 Program to find the transpose of matrix.

```
#include<stdio.h>
#include<conio.h>
#define ROW 3
#define COL 4
main()
{
int mat1[ROW][COL],mat2[COL][ROW],i,j;
clrscr();
printf(“enter matrix mat1(%dx%d) row-wise : \n”,ROW,COL);

for(i=0;i<ROW;i++)
for(j=0;j<COL;j++)
scanf(“%d”,&mat1[i][j]);

for(i=0;i<COL;i++)
for(j=0;j<ROW;j++)
mat2[i][j]=mat1[j][i];
printf(“transpose of matrix is :\n”);

for(i=0;i<COL;i++)
{
        for(j=0;j<ROW;j++)
        printf("%5d",mat2[i][j]);
        printf("\n");
}
getch();
}
```

Output:

```
Enter matrix mat(3*4) row-wise
3  2  1  5
6  5  8  2
9  3  4  1
Transpose of matrix is
3       6       9
2       5       3
1       8       4
5       2       1
```

BINARY SEARCH

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

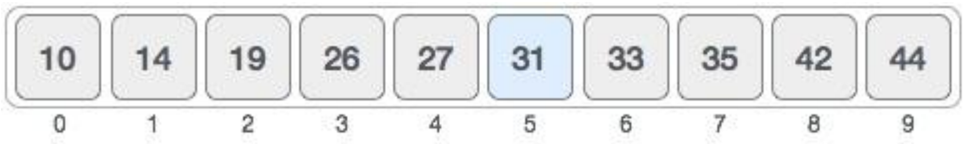
Time Complexity

The time complexity of Binary Search can be written as

$T(n) = T(n/2) + c$

How Binary Search Works

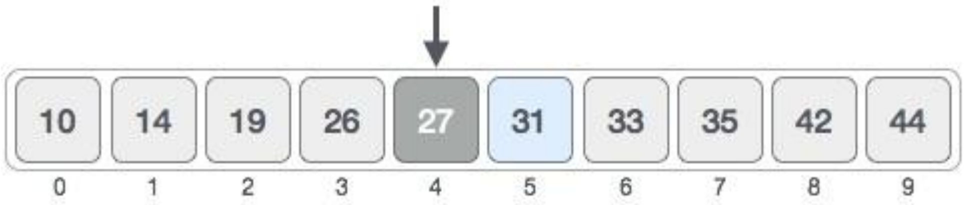
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



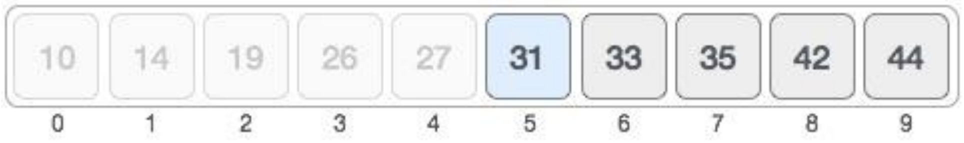
First, we shall determine half of the array by using this formula –

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

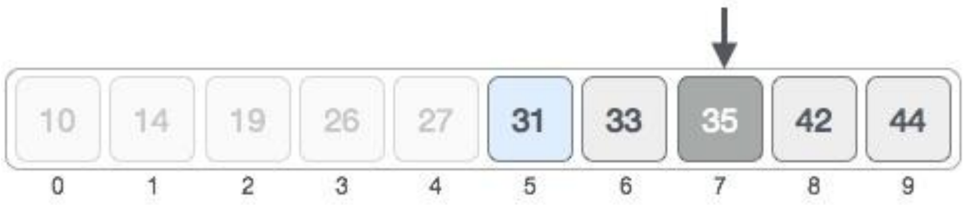


We change our low to mid + 1 and find the new mid value again.

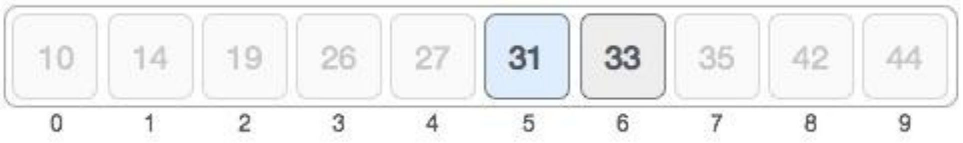
low = mid + 1

mid = low + (high - low) / 2

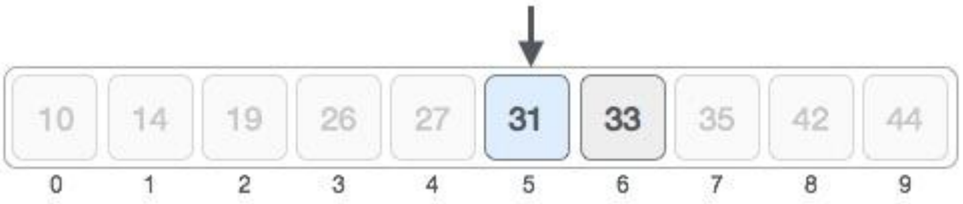
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



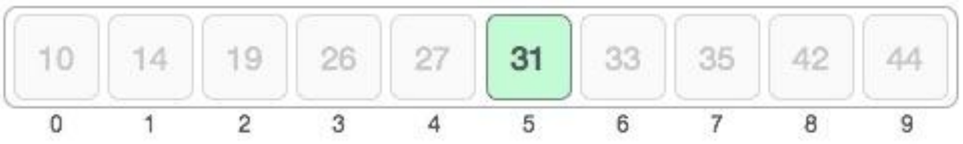
The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

5.9 WAP to understand the use of binary search

```
#include<stdio.h>
#include<conio.h>
#define SIZE 10
main()
{
 int arr[SIZE];
 int low,up,mid,i,item;
 clrscr();

 printf("Enter elements of the array(in sorted order): \n");
for(i=0;i<SIZE;i++)
scanf("%d",&arr[i]);

printf("Enter the item to be searched:");
scanf("%d",&item);

low=0;
up=SIZE-1;
while(low<=up&&item!=arr[mid])
{
        mid=(low+up)/2;
        if(item>arr[mid])
        low=mid+1;
        if(item<arr[mid])
        up=mid-1;
        if(item==arr[mid])
        printf("%d found at position %d\n",item,mid+1);
        if(low>up)
        printf("%d not found in array\n",item);
        }
        getch();
}
```

SELECTION SORT

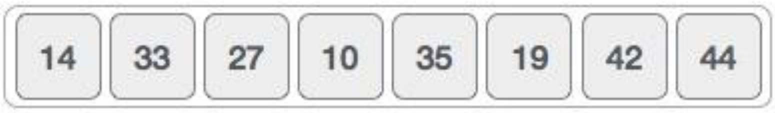
Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

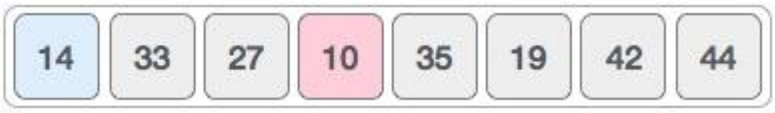
This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

How Selection Sort Works

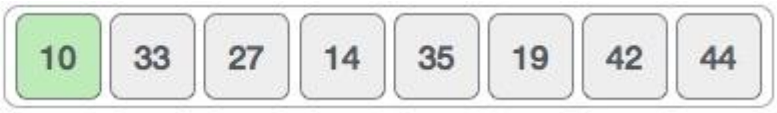
Consider the following depicted array as an example.



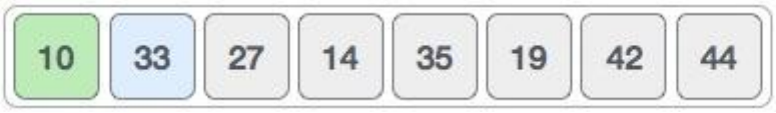
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



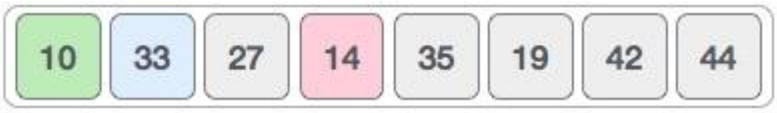
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



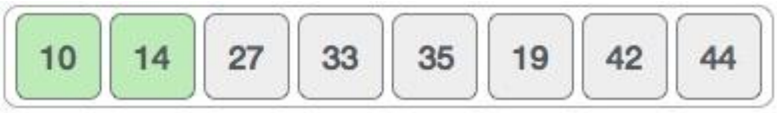
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

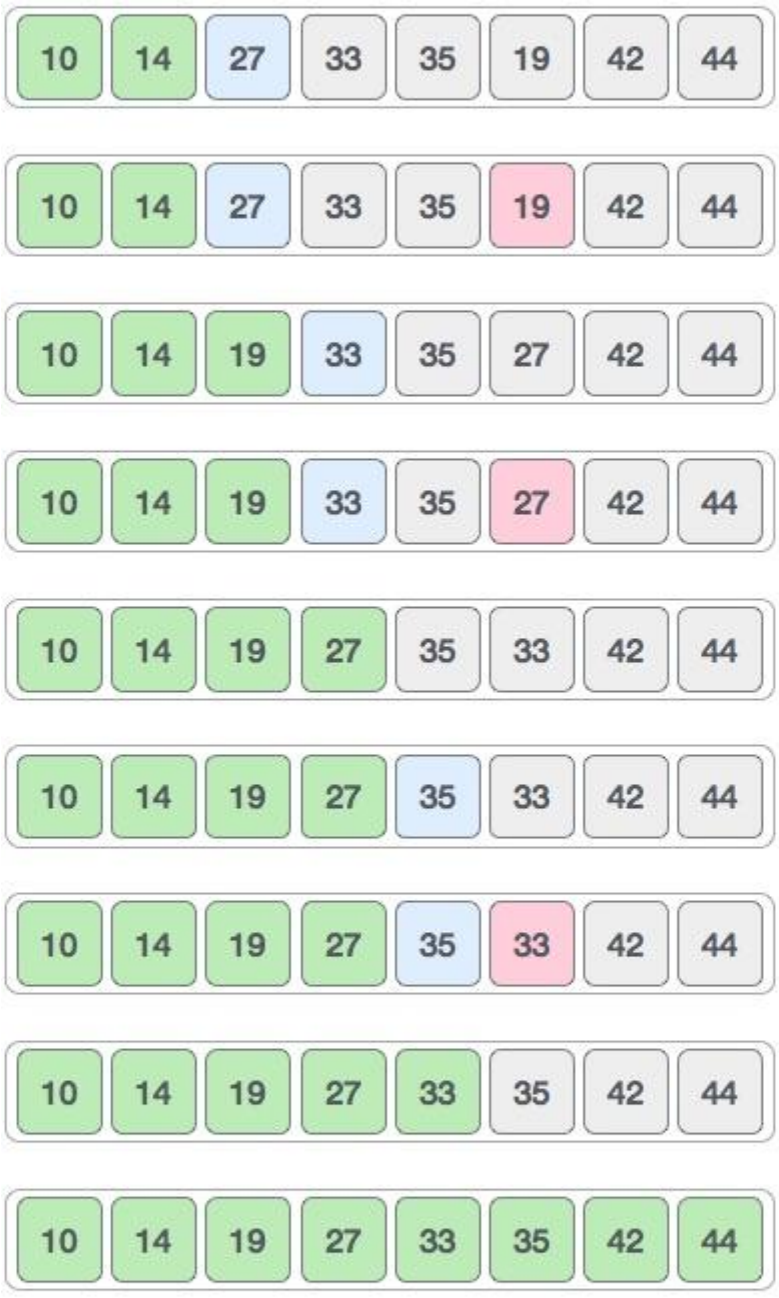


After two iterations, two least values are positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −



5.10 WAP to understand the use of selection sort

```
#include<stdio.h>
#include<conio.h>
#define SIZE 10
main()
{
```

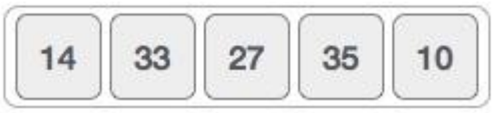
```
int arr[SIZE];
int i,j,temp;
clrscr();
printf("enter elements of the array :\n");

for(i=0;i<SIZE;i++)
scanf("%d",&arr[i]);
for(i=0;i<SIZE-1;i++)
for(j=i+1;j<SIZE;j++)
{
if(arr[i]>arr[j])
    {
    temp=arr[i];
    arr[i]=arr[j];
    arr[j]=temp;
    }
}
printf("sorted array is:\n");
for(i=0;i<SIZE;i++)
printf("%d ",arr[i]);
printf("\n");
getch();
}
```

BUBBLE SORT

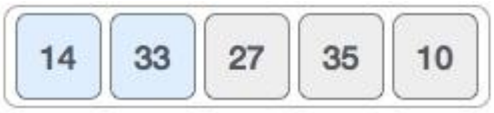
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

How Bubble Sort Works

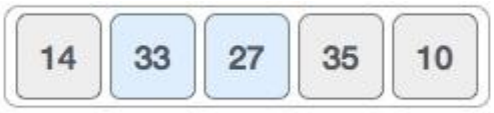
We take an unsorted array for our example. Bubble sort takes O(n^2) time so we're keeping it short and precise.



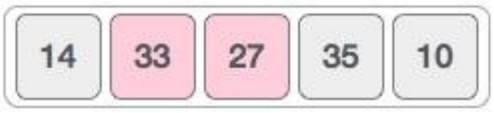
Bubble sort starts with very first two elements, comparing them to check which one is greater.



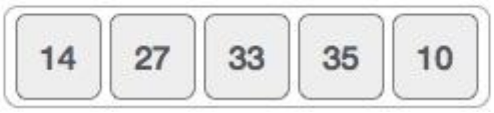
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



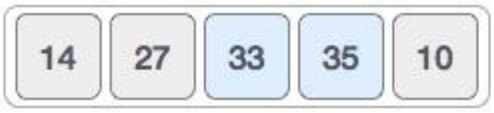
We find that 27 is smaller than 33 and these two values must be swapped.



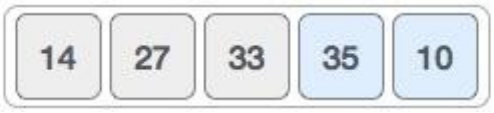
The new array should look like this −



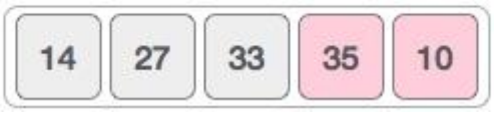
Next we compare 33 and 35. We find that both are in already sorted positions.



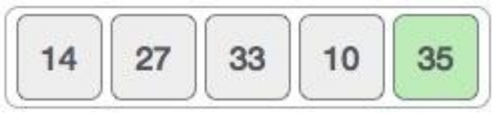
Then we move to the next two values, 35 and 10.



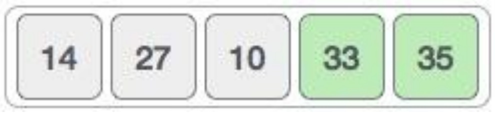
We know then that 10 is smaller 35. Hence they are not sorted.



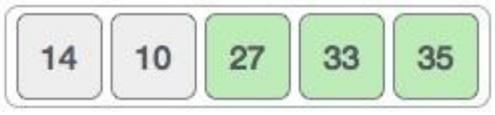
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −



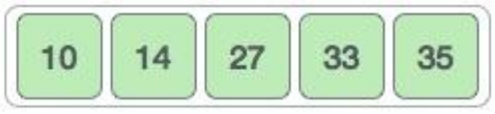
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



5.11 WAP to understand the use of bubble sort

```
#include<stdio.h>
#include<conio.h>
#define SIZE 6
main()
{
        int arr[SIZE];
        int i,j,temp;
        clrscr();
        printf("enter elements of the array :\n");

        for(i=0;i<SIZE;i++)
        scanf("%d",&arr[i]);

        for(i=0;i<SIZE-1;i++)
        for(j=0;j<SIZE-1-i;j++)
        {
```

```
        if(arr[j]>arr[j+1])
            {
            temp=arr[j];
            arr[j]=arr[j+1];
            arr[j+1]=temp;
            }
        }
        printf("sorted array is:\n");
        for(i=0;i<SIZE;i++)
        printf("%d ",arr[i]);
        printf("\n");
        getch();
        }
```

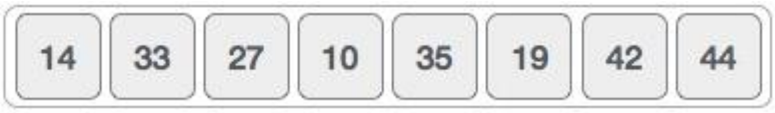
INSERTION SORT

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

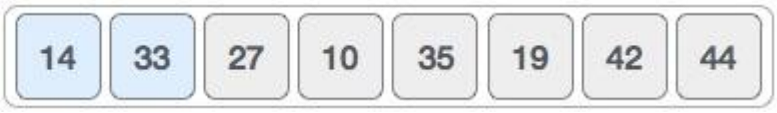
The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n^2), where **n** is the number of items.

How Insertion Sort Works

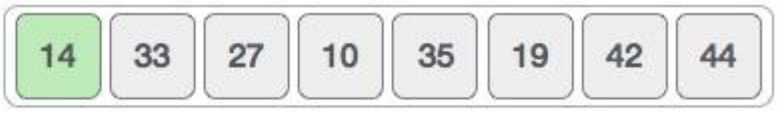
We take an unsorted array for our example.



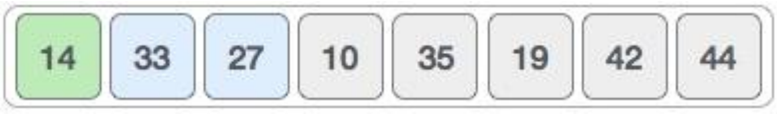
Insertion sort compares the first two elements.



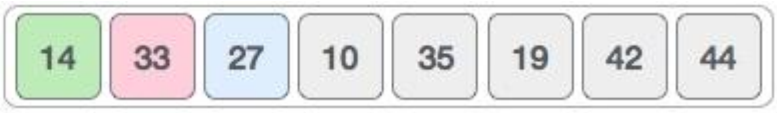
It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



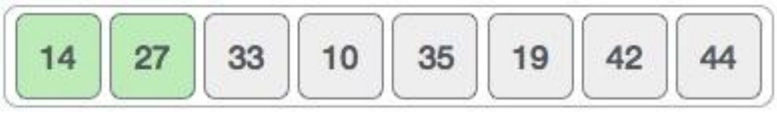
Insertion sort moves ahead and compares 33 with 27.



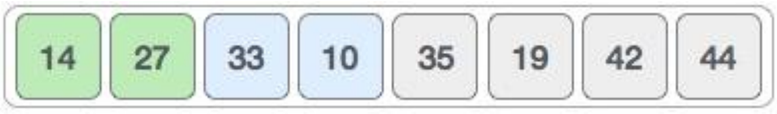
And finds that 33 is not in the correct position.



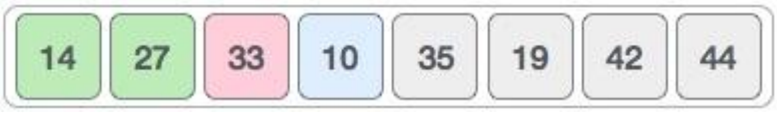
It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



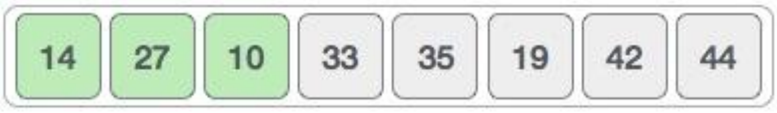
By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



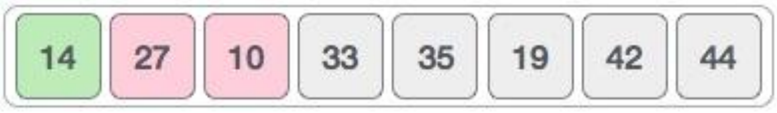
These values are not in a sorted order.



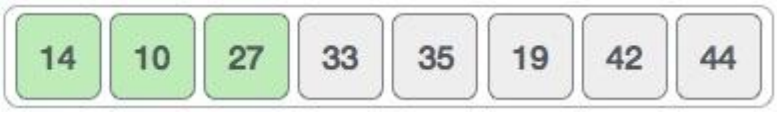
So we swap them.



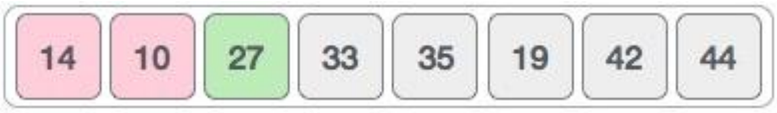
However, swapping makes 27 and 10 unsorted.



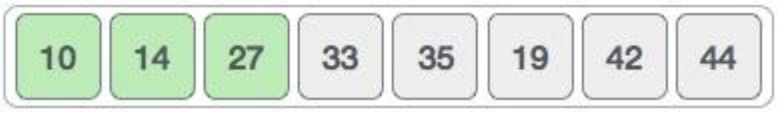
Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

5.12 WAP to understand the use of insertion sort

```
#include<stdio.h>
#include<conio.h>
#define SIZE 6
main()
{
        int arr[SIZE];
        int i,k,item;
        clrscr();
        printf("enter elements of the array :\n");

        for(i=0;i<SIZE;i++)
        scanf("%d",&arr[i]);

        for(k=1;k<SIZE;k++)
        {
        item=arr[k];
        for(i=k-1;item<arr[i]&&i>=0;i--)

           arr[i+1]=arr[i];
           arr[i+1]=item;

        }
printf("sorted array is:\n");
for(i=0;i<SIZE;i++)
printf("%d ",arr[i]);
printf("\n");
getch();
}
```

CHAPTER 6

POINTERS

C is a very powerful language and the real power of C lies in pointers. The concept of pointers is interesting as well as challenging. It is very simple to use pointers provided the basic are understood thoroughly. So it is necessary to visualize every aspect of pointers instead of just having a superficial knowledge about their syntax and usage. The use of pointers makes the code more efficient and compact.

A pointer is essentially a simple integer variable which holds a memory address that points to a value, instead of holding the actual value itself.

The computer's memory is a sequential store of data, and a pointer points to a specific part of the memory. Our program can use pointers in such a way that the pointers point to a large amount of memory - depending on how much we decide to read from that point on.

Address-of Operator (&)

An address-of operator is a mechanism within C that returns the memory address of a variable. These addresses returned by the address-of operator are known as pointers, because they "point" to the variable in memory.

The address-of operator is a unary operator represented by an ampersand (&). It is also known as an address operator.

Address operators commonly serve two purposes:

1. To conduct parameter passing by reference, such as by name
2. To establish pointer values. Address-of operators point to the location in the memory because the value of the pointer is the memory address/location where the data item resides in memory.

For example, if the user is trying to locate age 26 within the data, the integer variable would be named age and it would look like this: int age = 26. Then the address operator is used to determine the location, or the address, of the data using "&age".

From there, the Hex value of the address can be printed out using "cout << &age". Integer values need to be output to a long data type. Here the address location would read "cout << long (&age)".

The address-of operator can only be applied to variables with fundamental, structure, class, or union types that are declared at the file-scope level, or to subscripted array references. In these expressions, a constant expression that does not include the address-of operator can be added to or subtracted from the address-of expression.

Dereferencing

Dereferencing is the act of referring to where the pointer points at, instead of the memory address. We are already using dereferencing in arrays - but we just didn't know it yet. The brackets operator - `[0]` for example, accesses the first item of the array. And since arrays are actually pointers, accessing the first item in the array is the same as dereferencing a pointer. Dereferencing a pointer is done using the asterisk operator `*`.

If we want to create an array that will point to a different variable in our stack, we can write the following code:

```
/* define a local variable a */
int a = 1;

/* define a pointer variable, and point it to a using the & operator */
int * pointer_to_a = &a;

printf("The value a is %d\n", a);
printf("The value of a is also %d\n", *pointer_to_a);
```

Execute Code

Notice that we used the `&` operator to point at the variable `a`, which we have just created.

We then referred to it using the dereferencing operator. We can also change the contents of the dereferenced variable:

```
int a = 1;
int * pointer_to_a = &a;

/* let's change the variable a */
a += 1;

/* we just changed the variable again! */
*pointer_to_a += 1;

/* will print out 3 */
printf("The value of a is now %d\n", a);
```

Some of the uses of pointers are

1. Accessing array elements.
2. Returning more than one value from a function
3. Accessing dynamically allocated memory.
4. Implementing data structures like linked lists,trees,and graphs

Pointers Variables

A pointer is a variable that stores memory address. Like all other variables it also has a name, has to be declared and occupies some space in memory. It is called pointer because it points to a particular location in memory by storing the address of that location.

Declaration of Pointer variable

General syntax of pointer declaration is,

datatype *pointer_name;

Data type of a pointer must be same as the data type of the variable to which the pointer variable is pointing. void type pointer works with all data types, but is not often used.

Here are a few examples:

```
int *ip     // pointer to integer variable
float *fp;      // pointer to float variable
double *dp;     // pointer to double variable
char *cp;       // pointer to char variable
```

Initialization of Pointer variable

Pointer Initialization is the process of assigning address of a variable to a pointer variable. Pointer variable can only contain address of a variable of the same data type. In C language address operator & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
#include<stdio.h>

void main()
{
   int a = 10;
   int *ptr;       //pointer declaration
   ptr = &a;       //pointer initialization
}
```

Pointer variablea always point to variables of same datatype. Let's have an example to showcase this:

```
#include<stdio.h>

void main()
{
   float a;
   int *ptr;
   ptr = &a;       // ERROR, type mismatch
}
```

If you are not sure about which variable's address to assign to a pointer variable while declaration, it is recommended to assign a NULL value to your pointer variable. A pointer which is assigned a NULLvalue is called a NULL pointer.

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

```
int main()
{
   int *ptr = NULL;
   return 0;
}
```

6.1 WAP to print address of variables using address of operators

```
#include<stdio.h>
#include<conio.h>
main()
{
int age=30;
float sal=1500.50;
clrscr();
printf("value of age=%d\t,address of age=%u\n",age,&age);
printf("value of sal=%f\t,address of sal=%u\n",sal,&sal);
getch();
}
```

Output:

```
value of age=30,address of age=65524 value of
sal=1500.500000,address of sal=65520
```

6.2 Program to dereference pointer variable

```
#include<stdio.h>
#include<conio.h>
main()
{
```

```
int a=87;
float b=4.5;
int *p1=&a;
float *p2=&b;
clrscr();
printf("value of p1=address of a=%u\n",p1);
printf("value of p2=address of b=%u\n",p2);
printf("address of p1=%u\n",&p1);
printf("address of p2=%u\n",&p2);
printf("value of a=%d %d %d\n",a,*p1,*(&a));
printf("value of b= %f %f %f \n",b,*p2,*(&b));
getch();
}
```

Output:

```
value of p1=address of a=65524
value of p2=address of  b=65520
address of p1=65518
address of p2=65516
value of a=87  87  87
value of b=4.500000    4.500000    4.500000
```

6.3 Program to print the size of pointer variable and size of value dereferenced by that pointer

```
#include<stdio.h>
#include<conio.h>
main()
{
char a='x',*p1=&a;
int b=12,*p2=&b;
float c=12.4,*p3=&c;
double d=18.3,*p4=&d;
clrscr();
printf("sizeof(p1)=%d,sizeof(*p1)=%d\n",sizeof(p1),sizeof(*p1));
printf("sizeof(p2)=%d,sizeof(*p2)=%d\n",sizeof(p2),sizeof(*p2));
printf("sizeof(p3)=%d,sizeof(*p3)=%d\n",sizeof(p3),sizeof(*p3));
printf("sizeof(p4)=%d,sizeof(*p4)=%d\n",sizeof(p4),sizeof(*p4));
getch();
}
```

Output:

```
sizeof(p1)=2,sizeof(*p1)=1
sizeof(p2)=2,sizeof(*p1)=2
sizeof(p3)=2,sizeof(*p1)=4
sizeof(p4)=2,sizeof(*p1)=8
```

6.4 Program to show pointer arithmetic

```
#include<stdio.h>
#include<conio.h>
main()
{
int a=5,*pi=&a;
char b='x',*pc=&b;
float c=5.5,*pf=&c;
clrscr();
printf("value of pi=address of a=%u\n",pi);
printf("value of pc=address of b=%u\n",pc);
printf("value of pf=address of c=%u\n",pf);
pi++;
pc++;
pf++;
printf("now value of pi=%u\n",pi);
printf("now value of pc=%u\n",pc);
printf("now value of pf=%u\n",pf);
getch();
}
```

Output:

```
value of pi=address of a=1000
value of pc=address of b=4000
value of pf=address of c=8000
now value of pi=1002
now value of pc=4001
now value of pf=8004
```

6.5 program to understand the postfix/prefix increment/decrement in a pointer variable of base type int

```
#include<stdio.h>
#include<conio.h>
main()
{
int a=5;
int *p;
p=&a;
clrscr();
printf("value of p = address of a = %u\n",p);
printf("value of p=%u\n",++p);
printf("value of p=%u\n",p++);
printf("value of p=%u\n",--p);
printf("value of p=%u\n",p--);
printf("value 0f p=%u\n",p);
getch();
}
```

Output:

```
value of p=address of a=1000
value of p=1002
value of p=1002
value of p=1002
value of p=1002
value of p=1000
```

6.6 program to print the value and address of the elements of an array

```
#include<stdio.h>
#include<conio.h>
main() {
int arr[5]={5,10,15,20,25};
int i;
clrscr(); for(i=0;i<=5;i++)
{
printf("value of arr[%d]=%d\t",i,arr[i]);
printf("address of arr[%d]=%u\n",i,&arr[i]);
}
getch();
}
```

Output:

```
value of arr[0]=5         address of arr[0]=2000
value of arr[1]=10        address of arr[1]=2002
value of arr[2]=15        address of arr[2]=2004
value of arr[3]=20        address of arr[3]=2006
value of arr[4]=25        address of arr[4]=2008
```

The arguments to the functions can be passed in two ways

Pointers And Functions

The arguments to the functions can be passed in two ways-

1. call by value
2. call by reference

In call by value, only the values of arguments are sent to the function while in call by reference, addresses of arguments are sent to the function. In call by value method, any changes made to the formal arguments do not change the actual argument. In call by reference method, any changes made to the formal arguments change the actual arguments also. C uses only call by value when passing arguments to a function, but we can simulate call by reference by using pointers.

All the functions that we had written so far used call by value method. Here is another simple program that uses call by value-

Comparison between Call by Value and Call by Reference in Programming

	Call by Value	**Call by Reference**
Description	A function to pass data or value to other functions	A function to pass data or value to other functions
Languages used	C based programming languages	C based programming languages such as C++, Java, but not C itself
Purpose	To pass arguments to another function	To pass arguments to another function
Arguments	A copy of actual arguments is passed to respective formal arguments	Reference the location or address of the actual arguments is passed to the formal arguments

Changes	Changes are made in the own personal copy. Changes made inside the function are not reflected on other functions	Any changes made in the formal arguments will also reflect in the actual arguments. Changes made inside the function are reflected outside the function as well.
Value modification	Original value is not modified.	Original value is modified.
Memory Location	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location
Safety	Actual arguments remain safe, they cannot be modified accidentally.	Actual arguments are not safe. They can be accidentally modified. Hence care is required when handling arguments.
Default	Default in most programming languages such as C++, PHP, Visual Basic .NET, C# and REALbasic	Supported by most programming languages, but not as default.

6.7 Program to explain call by value

```
#include<stdio.h>
#include<conio.h>
value(int x, int y);
main()
{
int a=5,b=8;
clrscr();
printf("before calling the function, a and b are are %d,%d\n",a,b)
value(a,b);
printf("after calling the function,a and b are %d,%d\n",a,b);
getch();
}

value(int x,int y)
{
        x++;
        y++;
        printf("in function changes are %d,%d\n",x,y);
}
```

Output:

```
before calling the function,a and b are 5,8 in function
changes are 6, 9
after calling the function, a and b are 5, 8
```

6.8 Program to explain call by reference

```
#include<stdio.h>
#include<conio.h>
ref(int *p,int *q);
main()
{
int a=5,b=8;
clrscr();
printf("before calling the function, a and b are are %d,%d\n",a,b);
ref(&a,&b);
printf("after calling the function, a and b are %d,%d\n",a,b);
getch();
}

ref(int *p,int *q)
{
        (*p)++;
        (*q)++;
printf("in function changes are %d,%d\n",*p,*q);
}
```

Output:

```
Before calling the function, a and b are 5,8
In function changes are 6,9
After calling the function, a and b are 6,9
```

Dynamic Memory Allocation

- It is a process of allocating or de-allocating the memory at run time it is called as dynamically memory allocation.
- When we are working with array or string static memory allocation will be take place that is compile time memory management.
- When we ate allocating the memory at compile we cannot extend the memory at run time, if it is not sufficient.
- By using compile time memory management we cannot utilize the memory properly
- In implementation when we need to utilize the memory more efficiently then go for dynamic memory allocation.
- By using dynamic memory allocation whenever we want which type we want or how much we type that time and size and that we much create dynamically.

Dynamic memory allocation related all predefined functions are declared in following header files.

- <alloc.h>
- <malloc.h>
- <mem.h>
- <stdlib.h>

Dynamic memory allocation related functions

Malloc()

By using malloc() we can create the memory dynamically at initial stage. Malloc() required one argument of type size type that is data type size malloc() will creates the memory in bytes format. Malloc() through created memory initial value is garbage.

Syntax

Void*malloc(size type);

Dynamic memory allocation related function can be applied for any data type that's why dynamic memory allocation related functions return void*.

When we are working with dynamic memory allocation type specification will be available at the time of execution that's why we required to use type casting process.

Syntax

```
int *ptr;
ptr=(int*)malloc(sizeof (int));      //2 byte
long double*ldptr;
ldptr=(long double*)malloc(sizeof(long double))        // 2 byte
char*cptr;
cptr=(char*)malloc(sizeof(char));          //1 byte
int*arr;
arr=(int*)malloc(sizeof int()*10);//20 byte
cahr*str;
str=(char*)malloc(sizeof(char)*50);       //50 byte
```

6.9 Program to explain malloc() function

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

```
#include<alloc.h>
#include<stdlib.h>
 main()
{
        int *p,n,i;
        clrscr();
        printf("enter the number of integers to be entered :");
        scanf("%d",&n);
        p=(int *)malloc(n*sizeof(int));
        if(p==NULL)
        {
                printf("memory not available\n");
                exit(1);
        }
        for(i=0;i<n;i++)
        {
                printf("enter an integer :");
                scanf("%d",p+i);
        }
        for(i=0;i<n;i++)
        printf("%d\t",*(p+i));
        getch();
}
```

Calloc()

By using calloc() we can create the memory dynamically at initial stage. It required 2 arguments of type count, size-type. Count will provide number of elements; size-type is data type size. It will creates the memory in blocks format. First argument specifies the number of blocks and second one specifies the size of each block.

Ptr= (int *) calloc(5, sizeof (int));

This allocates 5 blocks of memory, each block contains 2 bytes and starting address is stored in the pointer variable ptr, which is of type int.

6.10 Program to explain calloc() function

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
int i, n;
int *a;
printf("Number of elements to be entered:");
scanf("%d",&n);

a = (int*)calloc(n, sizeof(int));
printf("Enter %d numbers:\n",n);
for( i=0 ; i < n ; i++ )
{
scanf("%d",&a[i]);
}
printf("The numbers entered are: ");
for( i=0 ; i < n ; i++ )
{
printf("%d ",a[i]);
}
free( a );
return(0);
}
```

Syntax

```
int*arr;
arr=(int*)calloc(10, sizeof(int));   // 20 byte
cahr*str;
str=(char*)calloc(50, siceof(char));        // 50 byte
```

Realloc()

We may want to increase or decrease the memory allocated by malloc() or calloc(). The function realloc() is used to change the size of the memory block. It alters the size of the memory block without losing the old data. This is known as reallocation of memory.

This function takes 2 arguments, first is a pointer to the block of memory that was previously allocated by malloc() and calloc() and second one is the new size for that block.

For example-   ptr=(int *) malloc (size);

This statement allocates the memory specified size and the starting address of this memory block is stored in the pointer variable ptr. If we want to change the size of this memory block, then we can use realloc() as-

```
Ptr=(int *) realloc (ptr, newsize);
```

This statement allocates the memory space of newsize bytes, and the starting address of this memory block is stored I the pointer variable ptr.

Syntax

```
 void*realloc(void*, size-type);
  int *arr;
  arr=(int*)calloc(5, sizeof(int));
  .....
  ........
  ....
  arr=(int*)realloc(arr,sizeof(int)*10);
```

6.11 Program to explain realloc() function

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
#include<stdlib.h>
 main()
{
        int *ptr,i;
        clrscr();
        ptr=(int *)malloc(5*sizeof(int));
        if(ptr==NULL)
        {
                printf("memory not available\n");
                exit(1);
        }
        printf("enter 5 integers:");
        for(i=0;i<5;i++)
                scanf("%d",ptr+i);
                ptr=(int *)realloc(ptr,9*sizeof(int));

        if(ptr==NULL)
        {
        printf("memory not available\n");
        exit(1);
        }
        printf("enter 4 more integer:");
        for(i=5;i< 9;i++)
        scanf("%d",ptr+i);
        for(i=0;i< 9;i++)
        printf("%d",*(ptr+i));
        getch();

}
```

Free()

- When we are working with dynamic memory allocation memory will created in heap area of data segment.
- When we are working with dynamic memory allocation related memory it is a permanent memory if we are not de-allocated that's why when we are working with dynamic memory allocation related program, then always recommended to deleted the memory at the end of the program.
- By using free(0 dynamic allocation memory can be de-allocated.
- free() requires one arguments of type void*.

Syntax

```
void free(voie*);
int *arr;
arr=(int*)calloc(10,sizeof(int));
...
......
free(arr);
```

- By using malloc(), calloc(), realloc() we can create maximum of 64kb data only.

- In implementation when we need to create more than 64kb data then go for formalloc(), forcalloc() and forrealloc().
- By using free() we can de-allocate 64kb data only, if we need to de-allocate more than 64kb data then go for

Syntax

forfree()
formalloc()
voidfor*formalloc(size-type);

Difference between malloc() and calloac() function

Basis of comparison	Malloc()	Calloc()
No of blocks	Allocates an only single block of requested memory.	Allocates multiple blocks of the requested memory.
Syntax	void *malloc(size_t size);	void *calloc(size_t num, size_t size);
Initialization	malloc() doesn't clear and initialize the allocated memory.	calloc() initializes the allocated memory to zero.
Manner of Allocation	malloc() function allocates memory of size 'size' from the heap.	calloc() function allocates memory the size of which is equal to num *size.
Speed	Fast	Comparatively slow.

CHAPTER 7

STRINGS

String: String is a collection of characters.

There is no separate data type for strings in C. they are treated as arrays type char. A character array is a string if it ends with a null character (‘\0’). This null character is an

Escape sequence with ASCII value 0. Strings are generally used to store and manipulate data in text form like words or sentences.

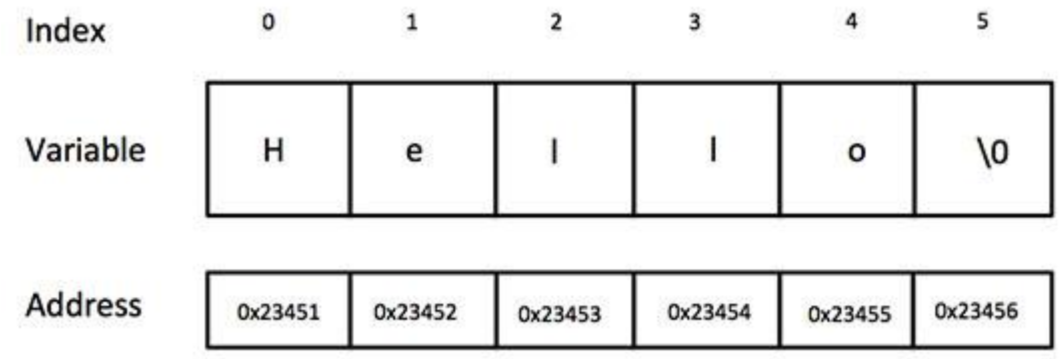
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows –

```
char greeting[] = "Hello";
```

Following is the memory presentation of the above defined string in C −



Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array.

String library functions

1) strlen()
2) strcpy
3) strcat
4) strcmp

7.1 program to show that identical string constants are stored separately

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
printf(“%u\n”,”good”);
printf(“%u\n”,”good”);
if(“bad” == “bad”);
 printf(“same\n”);
else
printf(“not same\n”);
}
```

Output

```
175
183
Not same
```

7.2 program to print characters of a string and address of each character

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
char str[ ]=”India”;
int i;
```

```
for(i=0; str[i]!='\0';i++)
{
Printf("character=%c\t",str[i]);
Printf("address=%u\n",&str[i]);
}
```

Output

```
Character=I        Address=1000
Character=n        Address=1001
Character=d        Address=1002
Character=i        Address=1003
```

7.3 program to print the address and characters of the string using pointer

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
char str[  ] = "India";
char *p;
p=str;
while(*p!='\0')
{ printf("character =%c\t",*p);
printf("address=%u\n",p);
p++;
}
}
```

7.4 program to input and output a string variable using scanf ()

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()

{
        char name[20];
        printf(“enter name);
        scanf(“%s”,name);
        printf(“%s “,name);
        printf(“%s\n “,”Srivastava”);
}
```

Output

1st run:

```
Enter a name: Deepali
Deepali srivastava
```

2nd run:

```
Enter a name: suresh kumar
Suresh srivastava
```

7.5 program to understand the use of gets( ) puts( )

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
```

```
   {
         char name[20];
         printf(“enter name :”);
         gets(name);
         printf(“entered name is : “);
         puts(name);
   }
```

Output

```
enter name : suresh kumar
entered name is : suresh kumar
```

Strlen()

This function returns the length of the string i.e. the number of characters in the string excluding the terminating null character. It accepts a single argument, which is pointer to the first character of the string. For example strlen(“suresh”) returns the returns the value 6. Similarly if s1 is an array that contains the name “deepali” then strlen(s1) returns the value 7

7.6 Program to understand the work of strlen() function

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
      char str[20];
```

```
        int length;

        clrscr();

        printf("enter the string");
        scanf("%s",str);

        length=strlen(str);

        printf("length of the string is %d\n",length);

        getch();

}
```

Output:

```
enter the string: save class

length of the string:10
```

Strcmp

This function is used for comparison of two strings . if the two string match, strcmp() returns a value 0,otherwise it returns a non zero value. This function compares the strings character by character. The comparison stops when either the end of string is reached or the corresponding characters in the two strings are not same. The non-zero value returned on mismatch is the difference of the ASCII values of the non-matching characters of the two strings-

Strcmp(s1,s2)return a value-

< 0 when s1 < s2

= 0 when s1 = = s2

> 0 when s1 > s2

Generally we don’t use the exact non-zero value returned in case of mismatch. We only need to know its sign to compare the alphabetical positions of the strings. We can use this function to sort the strings alphabetically.

7.7 Program to understand the work of strcmp() function

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
        char str1[10],str2[10];
        clrscr();
        printf("enter a 1st string");
        scanf("%s",str1);
        printf("enter a second string");
        scanf("%s",str2);

        if((strcmp(str1,str2))==0)
        printf("strings are same\n");
        else
        printf("strings are not same\n");
        getch();
}
```

Output:

```
enter the first string: bangalore enter the second
string: mangalore strings are not same
```

Strcpy

This function is used for copying one string to another string(str1,str2) copies str2 to str1. Here str2 is the source string and str1 is destination string. if str2 = "suresh" then this function copies "suresh" into str1. This function takes pointers to two strings as arguments and returns the pointer to first string.

7.8 Program to understand the work of strcpy() function

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
        char str1[10],str2[10];
        clrscr();
        printf("enter a 1st string");
        scanf("%s",str1);
        printf("enter a second string");
        scanf("%s",str2);
        strcpy(str1,str2);


        printf("first string %s \t\t second string %s\n",str1,str2);
        strcpy(str1,"delhi");
        strcpy(str2,"calcutta");
        printf("first string %s \t\t second string %s\n",str1,str2);
        getch();
}
```

Output:

```
enter the first string: bombay
enter the second string: mumbai
first string: Mumbai
second string : Mumbai
first string: Delhi
second string: Calcutta
```

Strcat()

This function is used for concatenation of two strings. If first is “king” and second string is “size” then after using this function the first string becomes “kingsize”.

Strcat(str1, str2); /*concatenates str2 at the end of str1*/

The null character from the first string is removed, and the second string is added at the end of first string. the second string remains unaffected.

7.9 Program to understand the work of strcat() function

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
        char str1[20],str2[20];
        clrscr();
        printf("enter the 1st string");
        scanf("%s",str1);
        printf("enter the 2nd string");
        scanf("%s",str2);


        strcat(str1,str2);
        printf("1 st string %s \t second string %s\n",str1,str2);
        strcat(str1,"_one ");
        printf("now first string is %s \n",str1);
        getch();
}
```

Output:

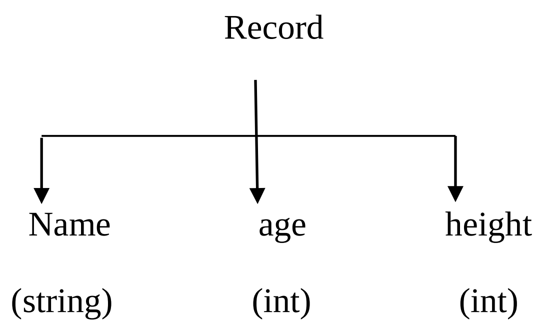
```
enter the first string: data
enter the second string: base
first string : database
second string : base
now first string is : database_one
```

CHAPTER 8

STRUCTURE AND UNION

Structure is a collection of dissimilar datatype.

Array is a collection of same type of elements but in many real life applications we may need to group different types of logically related data. For example if we want to create a record of a person that contains name, age height of that person, then we can’t use array becomes all the three data elements are of different type.



To store these related fields of different data types we can use a structure, which is capable of storing heterogeneous data. Data of different types can be grouped together under a single name using structures. The data elements of a structure are referred to as members.

Defining a Structure

Definition of a structure creates a template or format that describes the characteristics of its members. All the variables that would be declared of this structure type, will take the form of this template. The general syntax of a structure definition is-

```
structure tagname{
            datatype mamber1;

            datatype member2;

            …………
            …………
            Datatype member;
            };
```

Here struct is a keyword, which tells the compiler that a structure is being defined. Member1, member2, ………. memberN are known as members of the structure and are declared inside curly braces. There should be a semicolon at the end of the curly braces. These members can be of any data type like int, char, float, array, pointers or another structure type. Tag name is the name of the structure and it is used further in the program to declare variables of this structure type.

8.1 Program to display the values of structure members

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct student
{
        char name[20];
        int rollno;
        float marks;
};

main()
{
        clrscr();
        struct student stu1={"mary",25,68};
        struct student stu2,stu3;
        strcpy(stu2.name,"john"); stu2.rollno=26;
        stu2.marks=98;
        printf("enter name,rollno and marks for stu3");
        scanf("%s %d %f",stu3.name,&stu3.rollno,&stu3.marks);

        printf("stu1 : %s %d %.2f\n",stu1.name,stu1.rollno,stu1.marks);
        printf("stu2 : %s %d %.2f\n",stu2.name,stu2.rollno,stu2.marks);
        printf("stu3 : %s %d %.2f\n",stu3.name,stu3.rollno,stu3.marks);
        getch();
        }
```

Output:

```
enter name,rollno and marks for stu3: tom 27     79.5

Stu1: Mary 25   68.00

Stu2: john  26     98.00          Stu3: tom  27      79.50
```

8.2 Program to assign a structure variables to another structure variable

```
#include<stdio.h>
#include<conio.h>
struct student
{
        char name[20];
        int rollno;
        float marks;
 };
main()
{
        struct student stu1={"oliver",12,98};
        struct student stu2;
        clrscr(); stu2=stu1;
        printf("stu1 : %s %d %f\n",stu1.name,stu1.rollno,stu1.marks);
        printf("stu2 : %s %d %f\n",stu2.name,stu2.rollno,stu2.marks);
        getch();
}
```

Output:

```
Stu1: Oliver 12    98.00

Stu2: Oliver 12    98.00
```

8.3 Program to show that members of structure are stored in consecutive memory locations

```
#include<stdio.h>
#include<conio.h>
main()
{
struct student
{
	char name[5];
	int rollno;
	float marks;
}stu;

clrscr();
printf("address of name=%u\n",stu.name);
printf("address of rollno=%u\n",&stu.rollno);
printf("address of marks=%u\n",&stu.marks);
getch();
}
```

Output:

```
address of stu.name=65514
address of stu.rollno=65519
address of stu.marks=65521
```

8.4 Program to understand array of structures

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

```
{
        char name[20];
        int rollno;
        int marks;
};
main()
{
        int i;
        struct student stuarr[10];
        clrscr();
        for(i=0;i<10;i++)
{
        printf("enter name,rollno and marks");
        scanf("%s %d %d",stuarr[i].name,&stuarr[i].rollno,&stuarr[i].marks);
}
for(i=0;i<10;i++)
printf("%s %d %f \n",stuarr[i].name,stuarr[i].rollno,stuarr[i].marks);
getch();
}
```

8.5 Program to understand arrays within structures

```
#include<stdio.h>
#include<conio.h>
struct student
{
char name[15];
int rollno;
int submarks[4];
};
```

```
main()
{
	int i,j;
	struct student stuarr[3];
	clrscr();

	for(i=1;i<3;i++)
	{
		printf("enter data for student %d\n",i);
		printf("enter name");
		scanf("%s",stuarr[i].name);

		printf("enter roll num");
		scanf("%d",&stuarr[i].rollno);

		for(j=1;j<4;j++)
		{
		printf("enter marks for subject %d:",j);
		scanf("%d",&stuarr[i].submarks[j]);
		}
		}
	for(i=1;i<3;i++)
	{
		printf("data of student %d\n",i);
		printf("name:  %s,roll number  %d\n marks:  ",stuarr[i].name,stuarr[i].rollno);
		for(j=1;j<4;j++)
		printf("%d\t",stuarr[i].submarks[j]);
		printf("\n");
		}
		getch();
	}
```

UNION

Union is a derived datatype like structure and it can also contain members of different datatypes. The syntax used for definition of a union, declaration of a union variable and for accessing members in similar to that used in structures, but here keyboard union is used instead of struct. The main difference between union and structure is in the way memory is allocated for the members. In a structure each member as its own memory location, whereas members of union share the same memory location.

When a variable of type union is declared, complier allocates sufficient memory to hold the largest member in the union. Since all members share the same memory location hence we can use only one member at a time. Thus union is used for saving memory. The concept of union is useful when it is not necessary to use all members of the union at a time.

The syntax of definition of a union is-

```
Union union_name

{

        Datatype member1;

        Datatype member2;

        …………………..

};
```

8.6 Program for accessing union members

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

main()

{

union result

{
        int marks;
        char grade;
        float per;
}

        res;
        clrscr();
        res.marks=90;
        printf("marks %d\n",res.marks);
        res.grade='a';
        printf("grade %c\n",res.grade);
        res.per=85.5;
        printf("percentage:%f\n",res.per);
        getch();
}
```

Output:

```
marks : 90
grade : a
percentage: 85.500000
```

8.7 Write a program to compare the memory allocated for a union and structure variable.

```
#include<stdio.h>
#include<conio.h>
struct stag { char c; int i; float f; };
union utag
{
        char c;
        int i;
        float f;
};

main()
{
union utag uvar;
struct stag svar;
clrscr();

printf("size of svar=%d\n",sizeof(svar));
printf("address of svar: %u\t",&svar);

printf("address of members %u %u %u\n",&svar.c,&svar.i,&svar.f);

printf("size of uvar=%d\n",sizeof(uvar));
printf("address of uvar: %u\t",&uvar);

printf("address of members: %u %u %u\n",&uvar.c,&uvar.i,&uvar.f);
getch();
}
```

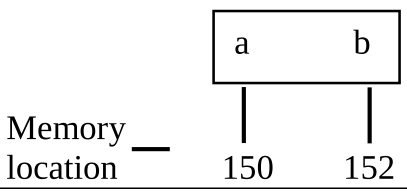
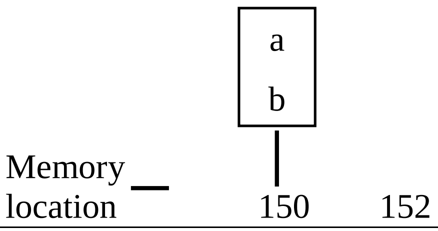
Output:

```
size of svar =7
address of svar:65514
adress of members : 65514   65515   65517
size of uvar =4
address of uvar: 65522
address of members:   : 65522    65522     65522
```

Difference between structure and union

	STRUCTURE	**UNION**
Memory Allocation	Members of structure do not share memory. So A structure need separate memory space for all its members i.e. all the members have unique storage.	A union shares the memory space among its members so no need to allocate memory to all the members. Shared memory space is allocated i.e. equivalent to the size of member having largest memory.
Member Access	Members of structure can be accessed individually at any time.	At a time, only one member of union can be accessed.
Keyword	To define Structure, '**struct**' keyword is used.	To define Union, '**union**' keyword is used.
Initialization	All members of structure can be initialized.	Only the first member of Union can be initialized.
Size	Size of the structure is > to the sum of the each member's size.	Size of union is equivalent to the size of the member having **largest** size.

Syntax	struct struct_name
	{
	structure ele 1;
	structure ele 2;
	———-
	———-
	structure ele n;
	}struct_variable_name;
		union union_name
		{
		union ele 1;
		union ele 2;
		———-
		———-
		union ele n;
		}union_variable_name;

Change in Value	Change in the value of one member can not affect the other in structure.	Change in the value of one member can affect the value of other member.
Storage in Memory	Struct { int a, float b) 	union { int a,float b) 

CHAPTER 9

PYRAMID

9.1 program to understand the pyramid

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,n;
clrscr();
printf("Enter n");
scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=i;j++)
                {
                printf("*");
                }
        printf("\n");
        }
getch();
}
```

Output:

```
*
**
***
****
```

9.2 program to understand the pyramid

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,n;
clrscr();
printf("Enter  n");
scanf("%d",&n);

for(i=1;i<=n;i++)
        {
        for(j=1;j<=i;j++)
        {
        printf("%d",i);
        }
        printf("\n");
        }
getch();
}
```

Output:

```
1
22
333
4444
55555
```

9.3 program to understand the pyramid

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,n;
clrscr();
printf("Enter n");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
        for(j=1;j<=i;j++)
        {
        printf("%d",j);
        }
printf("\n");
}
getch();
}
```

Output:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

9.4 program to understand the pyramid

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,n,p=1;
clrscr();
printf("Enter n");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
        for(j=1;j<=i;j++)
        {
        printf("%3d",p++);
        }
printf("\n");
}
getch();
}
```

Output:

```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
```

9.5 program to understand the pyramid

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,n;
clrscr();
printf("Enter n");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
        for(j=1;j<=i;j++)
        {
```

```
        printf("%d\t",i+j);
}
printf("\n");
}
getch();
}
```

Output:

```
2
3 4
4 5 6
5 6 7 8
6 7 8 9 10
```

9.6 program to understand the pyramid

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,n,k;
clrscr();
printf("Enter n");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
for(j=1;j<=i;j++)
{
k=i+j;
if(k%2==0)
printf("1");
else
printf("0");
}
printf("\n");
}
getch();
}
```

Output:

```
1
0 1
1 0 1
0 1 0 1
1 0 1 0 1
```

9.7 program to understand the pyramid

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,n;
clrscr();
printf("Enter n");
scanf("%d",&n);
for(i=n;i>=1;i--)
{
for(j=1;j<=i;j++)
{
printf("*");
}
printf("\n");
}
getch();
}
```

Output:

```
*****
****
***
**
*
```

9.8 program to understand the pyramid

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,n;
clrscr();
printf("Enter n");
scanf("%d",&n);
for(i=n;i>=1;i--)
{
for(j=1;j<=i;j++)
{
printf("%3d",i);
}
printf("\n");
}
getch();
}
```

Output:

```
5 5 5 5 5
4 4 4 4
3 3 3
2 2
1
```

9.9 program to understand the pyramid

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,n;
clrscr();
printf("Enter n");
scanf("%d",&n);
for(i=n;i>=1;i--)
{
        for(j=1;j<=i;j++)
        {
        printf("%3d",j);
        }
printf("\n");
}
getch();
}
```

Output:

```
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

9.10 program to understand the pyramid

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,n,p;
clrscr();
printf("Enter n");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
	for(j=1;j<=n-i;j++)
		printf("");
		p=n;
	for(j=1;j<=i;j++)
		printf("%d",p--);
		p=p+2;
	for(j=1;j<i;j++)
		printf("%d",p++);
	printf("\n");
}
getch();
}
```

Output:

```
5
5 4 5
5 4 3 4 5
5 4 3 2 3 4 5
5 4 3 2 1 2 3 4 5
```

9.11 program to understand the pyramid

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,n,p;
clrscr();
printf("Enter n");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
```

```
for(j=1;j<=n-i;j++)
printf("");
p=1;
for(j=1;j<=i;j++)
printf("%d",p++);
for(j=1;j<i;j++)
printf("%d",p++);
printf("\n");
}
getch();
}
```

Output:

```
1
1 2 3
1 2 3 4 5
1 2 3 4 5 6 7
1 2 3 4 5 6 7 8 9
```

9.12 program to understand the pyramid

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,n;
clrscr();
printf("Enter n");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
for(j=1;j<=i;j++)
printf("");
for(j=1;j<=(n-i);j++)
printf("*");
for(j=1;j<(n-i);j++)
printf("*");
printf("\n");
}
getch();
}
```

Output:

```
1
2 3 2
3 4 5 4 3
4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
```

CHAPTER 10

FILES

File is a collection of bytes that is stored on secondary storage devices like disk. There are two kinds of files in a system. They are,

1. Text files (ASCII)
2. Binary files

- Text files contain ASCII codes of digits, alphabetic and symbols.
- Binary file contains collection of bytes (0's and 1's). Binary files are compiled version of text files.

BASIC FILE OPERATIONS IN C PROGRAMMING

There are 4 basic operations that can be performed on any files in C programming language. They are,

1. Opening/Creating a file
2. Closing a file
3. Reading a file
4. Writing in a file

Declaration of opening a file

FILE *open(const char *filename,const char *mode);

fopen( ) function takes two strings as arguments, the first one is the name of the file to be opened and the second one is the mode that decides which operations (read, write,append etc) are to be performed on the file. On success,fopen( ) returns a pointer of type FILE and on error it returns NULL. The second argument represents the mode in which the file is to be opened. The possible values of mode are-

1. **"w" (write):** If the file doesn't exist then this mode creates a new file for writing, and if the file already exists then the previous data is erased and the new data entered is written to the file.

2. **"a" (append):** If the file doesn't exist then this mode creates a new file, and if the file already exists then the new data entered is appended at the end of existing data. 3. "r" (read): This mode is used for opening an existing file for reading purpose only. The file to be opened must exist and the previous data of file is not erased.

3. **"r" (read):** this mode is used for opening an existing file for reading purpose only. This file to be opened must exist and the previous data of file is not erased.

4. **“w+”(write+read):** This mode is same as “w” mode but in this mode we can also read and modify the data. If the file doesn’t exist then a new file is created and if the file exists then previous data is erased.

5. **“r+”(read+write):** This mode is same as “r” mode but in this mode we can also write and modify existing data. The file to be opened must exist and the previous data of file is not erased. Since we can add new data and modify existing data so this mode is also called update mode.

6. **“a+”(append+read):** This mode is same as “a” mode but in this mode we can also read the data stored in the file. If the file doesn’t exist, a new file is created and if the file already exists then new data is appended at the end of existing data. We cannot modify existing data in this mode.

Declaration of closing a file

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

The functions used for file I/O are

- Character I/O – fgetc( ), fputc( ), gets( ),putc( )
- String I/O – fgetc( ), fputc( )
- Integer I/O- getw( ),putw( )
- Formatted I/O- fscanf( ),fprintf( )
- Record I/O- fread( ), fwrite( )

Character I/O

i) fputc: int fputc(int c, FILE *fptr)

This function writes a character to the specified file at the current file position and then increments the file position pointer. On success it returns an integer representing the character written, and on error it returns EOF.

10.1 program to understand the use of <u>fputc()</u> function

```
#include<stdio.h>
#include<conio.h>
main()
{
FILE *fptr;
int ch;
if((fptr=fopen("myfile.txt","w"))==NULL)
{
	printf("file does not exits\n");
}
else
{
printf("enter text: \n");
while((ch=getchar())!=EOF)
fputc(ch,fptr);
}
fclose(fptr); getch();
}
```

Output:

```
enter text : i am a student
^z
```

after the execution of this program,this text along with the ^z character will be written to the file myfile.txt.

ii) fgetc( ) : int fgetc(FILE *fptr);

This function reads a single character from a given file increments the file pointer position. On success it returns the character after converting it to an int without sign extension.

10.2 Program to understand the use of fgets()

```
#include<stdio.h>
#include<conio.h>
main()
{
FILE *p; char ch;
if((p=fopen("myfile.txt","r"))==NULL)
printf("file does not exits\n");
else
{
while((ch=fgetc(p))!=EOF)
printf("%c",ch);
}
fclose(p);
getch();
}
```

Output:

```
i am a student
```

iii) getc( ) and putc( ):

The operations of getc( ) and putc( ) are exactly similar to that of fgetc( ) and putc( ), the only difference is that the former two are defined as macros while the latter two are functions.

Integer I/O

i) putw( ): int putw(int value,FILE *fptr)

This function writes an integer value to the file pointed to by fpts. It returns the integer written to the file on success ,anf EOF on error

10.3 Program to understand the use of putw( ) function

```
#include<stdio.h>
#include<conio.h>
main()
{
FILE *fptr;
int value;
fptr=fopen("num.dat","wb");
for(value=1;value<=30;value++)
putw(value,fptr);
fclose(fptr);
getch();
}
```

ii) getw( ): int getw(FILE *fptr);

This function returns the integer value from the file associated with fptr. It rturns the next integer from the input file on success, and EOF on error or end of file.

10.4 Program to understand the use of getw() function

```
#include<stdio.h>
#include<conio.h>
main()
{
FILE *fptr;
int value;
fptr=fopen("num.dat","rb");
while((value=getw(fptr))!=EOF)
printf("%d\t",value);
fclose(fptr);
getch();
}
```

String I/O

i) fputs: int fputs(const char *str,FILE *fptr);

This function writes the null terminated string poited to by str to a file. The null character that marks the end of string is not written to the file. On success it returns the last character written anf on error it returns EOF.

10.5 Program to understand the use of fputs()

```
include<stdio.h>
#include<conio.h>
main()
{
FILE *fptr;
char str[80];
fptr=fopen("test.txt","w");
printf("enter the text\n");

printf("to stop entering,press ctrl+d in unix and ctrl+z in dos\n");

while(gets(str)!=NULL) fputs(str,fptr);
fclose(fptr);
getch();
}
```

ii) fgets: char *fgets(char *str, int n, FILE *fptr);

This function is used to read characters from a file and these characters are stored in the string pointed to by str.

10.6 Program to understand the use of <u>fgets()</u>

```
#include<stdio.h>
#include<conio.h> main()
{
FILE *fptr; char str[80];
fptr=fopen("test.txt","r");

while(fgets(str,80,fptr)!=null)

puts(str);
fclose(fptr);
getch();
}
```

Formatted I/O

i) <u>fprintf( )</u>: fprintf(FILE *fptr,const char *format[,argument,…….]);

This function is same as the printf( ) function but it writes formatted data into the file instead of the standard output(screen).

10.7 Program to understand the use of fprintf()

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

```
char name[20];
float marks;
}stu;

main()
{
FILE *fp;
int i,n;
fp=fopen("students.dat","w");
printf("enter number of records");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
printf("enter name and marks");
scanf("%s%f",stu.name,&stu.marks);
fprintf(fp,"%s %f",stu.name,stu.marks);
}
getch();
}
```

ii) fscanf: fscanf(FILE *fptr,const char *format[,address,…..]);

This function is similar to the scanf ( ) function but it reads data from file instead of standared input, so it has one more parameter which is a pointer of FILE type and it points to the file from which data will be read.

10.8 Program to understand the use of <u>fscanf( )</u>

```
#include<stdio.h>
#include<conio.h>
struct student
{
char name[20];
float marks;
}stu;

main()
{
FILE  *fp;
fp=fopen("students.dat","r");
printf("NAME\tMARKS\n");

while(fscanf(fp,"%s%f",stu.name,&stu.marks)!=EOF)


printf("%s\t%f\n",stu.name,stu.marks);
fclose(fp);
getch();
}
```