UNIT 1

History of C

- ALGOL was the first computer language to use a block structure.
- In 1967, Martin Richards developed a language called BCPL [Basic Combined Programming Language] primarily for writing system software.
- In 1970, Ken Thompson created a language using many features of BCPL and called it simply B.
- C was evolved from ALGOL, BCPL and B by Dennis Ritchie at Bell laboratories in 1972.
- C was many concepts from these languages and added the concept of data types and other powerful features.
- Since it was developed along with the UNIX operating system, It is strongly associated with UNIX.
- During 1970's C had evolved into what is now known as " Traditional C".
- To assure that the C language remains standard in 1983; American National Standards Institute [ANSI] appointed a technical committee to define a standard for C.
- The committee approved a version of C in 1989 which is now known as ANSI.

1960	ALGOL	International Group
1967	BCPL	Martin Richards
1970	B	Ken thompson
1972	Traditional C	Dennis Ritchie
1989	ANSI C	ANSI Committee
1990	ANSI / ISO C	ISO Committee

Importance of C :

- It is a robust language whose rich set of built – in – functions and operators can be used to write any complex program.

- The C compiler combines the capabilities of an assembly language with the features of a high – level language and therefore it is well suited for writing both system software and business package.
- It is many times faster then BASIC.
- Several standard functions are available which can be used for developing programs.
- C is highly portable this means that c program written for one computer can be run on another with little or no modifications.
- C language is well suited for structured programming thus required the user to think of a program in terms of function modules or blocks.
- Another important feature of c is its ability to extend itself.

Evaluation of the Structure of a C program

- Every "C" program must have one main() function section.
- This section contains two parts, declaration part and executable part.
- The Declaration part declares all the variables used in the executable part. There is at least one statement in the executable part.
- These two parts must appear between the opening and the closing braces. The program execution begins at the opening brace and ends at the closing brace.
- All statement in the declaration and executable parts end with semicolon.

Example :

```
DOCUMENTATION SECTION
/* Addition of 2 numbers */
LINK SECTION
#include <stdio.h>          //    Standard I/O File    //
#include <conio.h>          //    Console I/O File     //
MAIN() FUNCTION SECTION
void main()
{
DECLARATION PART
int a = 10;
int b = 12,c;
//EXECUTABLE PART
C = a+b;
Printf("Addition = %d"'c);
```

}

OUTPUT

Additon = 22

Constants, Variables and Data Types :

CHARACTER SET :

- The characters in C are grouped into the following categories
  - i) Letters
  - ii) Digits
  - iii) Special Characters
  - iv) White Spaces

Letters :

- ★ Upper Case A.....Z
- ★ Lower Case a.....z

Digits :

- ★ All decimal digits 0.....9

Special Characters :

,	Comma	&	Ampersand
;	Semicolon	*	Asterisk
:	Colon	!	Exclamation mark
~	Tilde	‘	Apostrophe
/	Slash	_	Underscore

White Space :

- ★ Blank Space
- ★ Horizontal tab
- ★ Carriage return
- ★ New line
- ★ Form Feed

Trigraph Characters :

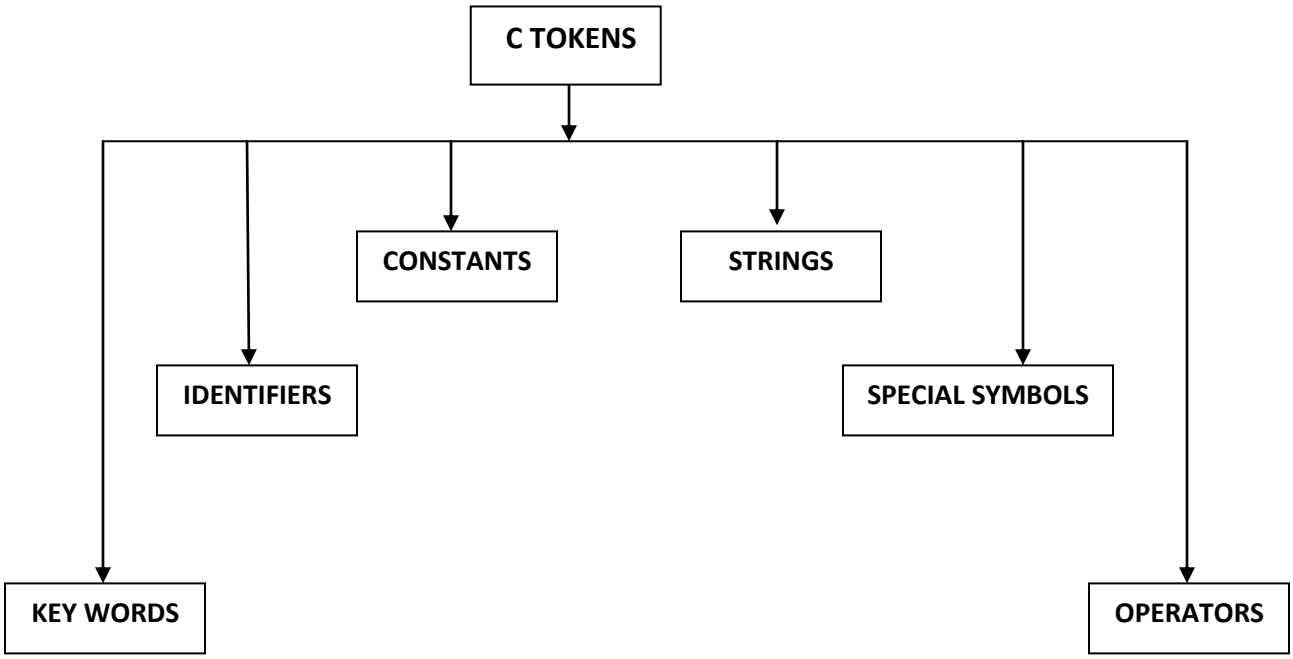
- Ansi C introduces the concept of “trigraph” sequence to provide a way to enter certain characters that are not available on some keyboards.

- Each trigraph sequence consists of three characters [two question mark followed by another character].

Trigraph Sequence	Translation
?? =	# number sign
?? (	[left bracket
?? <	{ left brace
?? /	\ back slash
?? \	^ caret

C Tokens :

- In a C Program the smallest individual units are known a C Tokens.
- C has six types of C Tokens.



Key Words :

- Every C word is classified as either a keyword or an identifier.
- All keywords have fixed meanings and these meanings cannot be changed.
- All keywords must be written in lower case.
    - Eg : int, float, char, double, if, do, else, static, main, scanf, printf.

Identifiers :

- Identifiers refer to the names of variables, function and arrays.

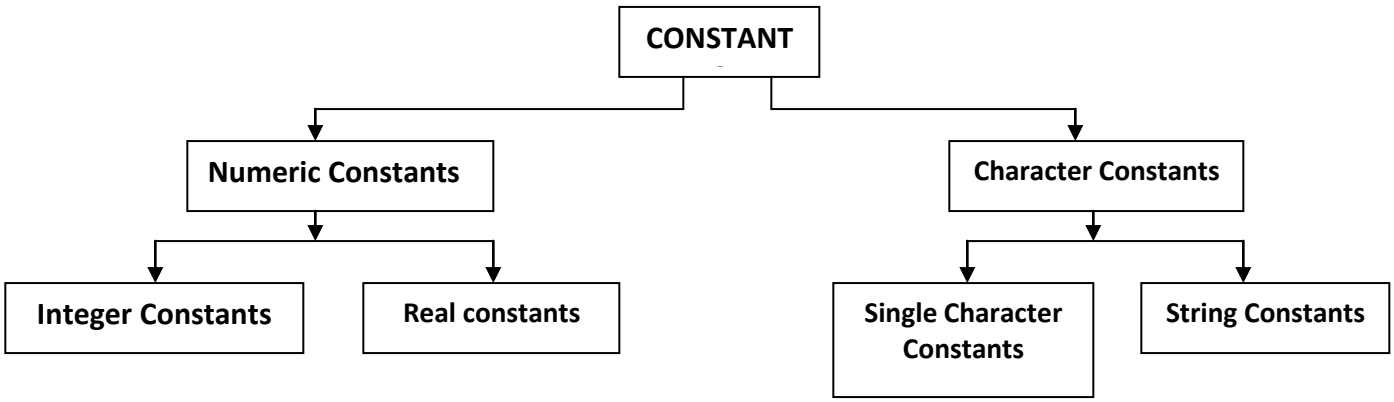
- These are user-defined names and consists of sequence of letters and digits, with a letter as a first character.
- Both uppercase and lowercase letters are permitted. The underscore characters is also permitted in identifiers.

Rules for Identifiers :

1) First character must be a letter or underscore.
2) Only first 31 characters are significant.
3) Cannot use keyword.
4) White spaces are not allowed.

Constants :

- Constants in C refers tp fixed values that do not change during the execution of a program.
- C supprot several types of constants.



Integer Constants :

- An integer constants refers to a sequence of digits. There are three types of integers namely Decimal Integer, Octal Integer and Hexadecimal Integer.
- Decimal Integers consists of a set of digits 0 through 9 preceeded by optional – (or) + sign.

  Eg : 123, -321, 0...

- An Octal Integer constant consist of any combination of digits from 0 to 7 with leading zero.

  Eg : 037, 0435..,

- A sequence of digits preceded by OX or ox is considered as hexadecimal integer. They also include alphabets A to F or a to f. The letter A to F represents the numbers from 10 to 15.

  Eg : ox2, OXA1

Real Constants :

- A number with decimal point is called real or floating point.

  Eg : 12.4, 0.25.

- A real number may also be expressed in exponential or scientific notation.
- The general form is **"mantissa e exponent"** the mantissa is either a real or an integer. The exponent is an integer number with an optional + or _ sign.

  Eg: 0.65e4, 1.4e+2.

Single Character Constants :

- A single character constants contains a single character enclosed within a pair of single quote mark.

  Eg : '5' , 'A'

String Constants :

- A string constant is a sequence of characters enclosed in double quotes.

  Eg: "Hai"

VARIABLES :

- A variables is a data name that may be used to store a value.
- A variable may take different values at different times during execution.

Rules for naming a variable :

- They must begin with an alphabet.
- Some system permits underscore as the first character.
- It should not be a keyword.
- White space is not allowed.

Valid => Sum, +1, S-value,

Invalid => % sum, 5+0+, int.

DATA TYPES:

- C supports three classes of data types.
    i) Primary [or] Fundamental [or] Built-in data types.
    ii) Derived data types.
    iii) User – Defined data types.

Primary Data Types:

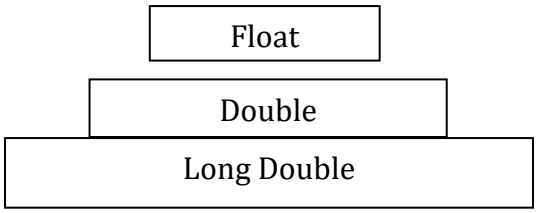
- C Supports four fundamental data types, They are
    i) Integer
    ii) Floating Point
    iii) Character
    iv) Void.

Integer types:

- Integers are whole number C has three classes of Integer storage namely,
    - ★ Short int –> 1 byte –> (range) -128 to +127
    - ★ int –> 2 byte –> (range) -32768 to +32767
    - ★ Long int –> 4 byte –> (range) -2,147,483,648 to +2,147,483,647.

Floating Point Type :

- In C floating point numbers are defined by the keyword float. The storage capacity of float is 4 bytes or 32 bits, The range of float data type is from 3.4e – 38 to 3.4e + 38.
- When the accuracy of float is not sufficient the type double can be used. Double data type uses 64 bits.
- To extend the accuracy we may use long double which uses 80 bits.



Character Data Type:

- In C character are defined by the keyword char.
- Characters are usually stored in 8 bits or 1 byte.

- It may be signed or unsigned
- Signed char values from -128 to 127 unsigned char values from 0 to 255.

Void Type :

- It has no values.
- This is usually used to specify the type of function.
- The type of function is said to be void, when it does not return any value to the calling function.

Declaration of Variables:

- Declaration does two things
  - i) It tells the name of the variable to the compiler.
  - ii) It specifies what type of data the variable will hold.
- The variable must be declared before they are used.
- There are two types of declaration.
  - i) Primary type declaration.
  - ii) User _ defined type declaration.

Primary Type Declaration :

- A variable can be used to store the value of any data type

Syntax :

```
Datatype v1,v2,.....vn;
```

- Where v1,v2 are the names of variable they are separated by commas.

  Eg : float a, xy;

User - Defined data type Declaration :

C supports user defined data types like typedef, enum, structures, union etc...

Operators :

- An operator is a symbol that tells the computer to perform certain mathematical or logical manipulation.
- Operators are used in programs to manipulate data and variables.
- C operators can be classified into a number of categories.
- They include,

i) Arithmetic operators
ii) Relational operators
iii) Logical operators
iv) Assignment operators
v) Increment and Decrement operators
vi) Conditional operators
vii) Bitwise operators
viii) Special operators

Arithmetic Operators :

- C provides all the basic arithmetic operators. They can operate on any built –in data type allowed in C.

Operator	Meaning
+	Addition or Unary plus
-	Subtraction or Unary minus
*****	Multiplication
/	Division
%	Modulo division

- The integer division truncates any fractional part. The modulo division operation produces the remainder of an integer division.

Integer Arithmetic :

- When both the operands in a single arithmetic expression such as a+b are integers, the expression is called as an integer expression and the operation is called ARITHMETIC..,

Eg : a-b = 10

a+b = 18

Relational Operators :

- C supports six relational operators in all

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

- A simple relational expression contains only one relational operator and takes the following forms.

ae – 1 relational operator ae -2

Where,

ae – 1 and ae – 2 are arithmetic expressions which may be simple constants, variables or combination of them.

Logical Operators :

In addition to the relational operators, C has the following three logical operators.

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

- The logical operators && and || are used when we want to test more than one condition and make decisions.

  Eg : a>b && x == 10;

- An expression which combines two or more relational expressions, is termed as a logical expression.

  Eg : if (age > 55 && salary < 1000).

Assignment Operators :

- Assignment operators are used to assign the result of in expression to a variable. In addition, C has a set of SHORTHAND assignment operators of the form,

```
v op = exp
```

Where, **v** is a variable, **exp** is an expression and **op** is a C binary arithmetic operator.

- The assignment operator statement v op = exp; is equivalent to v = v op (exp);

Eg : x+ = y+1;

Same as x = x+(y+1);

Increment and Decrement Operator :

++ and –

- The operator ++ adds 1 to the operand, while – subtracts 1 from the operand. Both are unary operators and takes the following form.

++m ; or m++;

--m; or m--;

i) ++m is equivalent to m=m+1

ii) --m is equivalent to m=m-1

m=5; y=m++

then the value of y would be 5 and m would be 6.

- A prefix operator ++m first adds 1 to the operand and then the result is assigned to the variable on left.

- A postfix operator m++ first assign the value to the variables an left and then increases 1 to the operand.

Conditional Operator :

- A ternary operator pair "?" is available to construct conditional expressions of the form.

```
exp1? exp2 : exp3;
```

Where, exp1, exp2 and exp3 are expressions.

- The operator ?: works as follows :
  - i) exp1 is evaluated first. If it is non zero (true), then the expression exp2 is evaluated and becomes the value of the expressions.
  - ii) If exp1 is false, exp3 is evaluated and its value becomes the value of the expression.

Eg : a = 10; b = 15;

x = (a>b) ? a : b ;

Bitwise Operator :

- C has a distinction of supporting special operators known as bitwise operators for manipulation of data at bit level.
- These operators are used for testing the bits, or shifting them right or left.
- Bitwise operator may not be applied to float or double.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	Shift Left
>>	Shift Right

Special Operators :

- C supports some special operators of interest such as comma operator, size of operator, pointer operators (& and *) and member selection operator (. and ->).

Comma Operator :

- The comma operator can be used to link the related expressions together.

Eg: value = (x=10, y=5, x+y);

Sizeof Operator :

- The sizeof is a compile time operator and when used, with an operand, it returns the number of bytes the operand occupies.

Eg : m = sizeof (sum);

n = sizeof (long int);

k = sizeof (235L);

Arithmetic Expressions:

- An arithmetic expressions is a combination of variables constants and operators arranged as per the syntax of the language.

Evaluation of Expressions :

- Expressions are evaluated using an assignment statement of the form.

Variable = expressions

- Variable is any valid C variable name. The expression is evaluated first and the result then replaces the previous value of the variable on the left hand side.

Eg : x = a * b - c;

y = b / c * a;

z = a – b / c + d;

Program :

```
Main()
{
      Float  a, b ,c, x, y, z ;
      a = 9;  b =12; c = 3;
      x = a – b / 3 \ = c * z - 1
      y = a – b / (3 + c) *  (z - 1);
      z = a - (b / (3 + c) * z) - 1;
      Printf("X = %f  \n", x);
      Printf("Y = %f  \n", y);
      Printf("Z = %f  \n", z);
}
```

Precedence of arithmetic operator :

- An arithmetic expression without parenthesis will be evaluated from left to right using the rules of precedence of operators.
- There are two distinct priority levels of arithmetic operators in C.

i) High Priority * ? %

ii) Low Priority + -

★ *X = a – b / 3 + c * 2 – 1*

When a = 9 b = 12 and c = 3 the statement becomes,

★ *X = 9 – 12 / 3 + 3 * 2 – 1 and is evaluated as follows.*

Step 1 : x = 9 – 4 + 3 * 2 – 1

Step 2 : x = 9 – 4 + 6 – 1

Step 3 : x = 5 + 6 – 1

Step 4 : x = 11 – 1

Step 5 : x = 10

Rules for evaluation of Expression:

- First, parenthesized sub expression from left to right are evaluated.
- If parenthesis are nested, the evaluation begins with the innermost sub-expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub expression.
- The associativity rule is applied when two or more operators of the same precedence level appear in a sub expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When parenthesis is used, the expressions with in parentheses assume highest priority.

Type Conversion in Expression :

- There are two types of conversions, they are as follows :
  - i) Implicit type conversion
  - ii) Explicit type conversion

Implicit Type Conversion :

- C permits mixing of constants and variables of different types in an expression.
- This automatic conversion is known as implicit type conversion.
- If the operands are of different types, the “lower” type is automatically converted to the “higher” type before the operation proceeds.

Rules :

- All short and char are automatically converted into int; then
  1) If one of the operands is long double and the other will be converted to long double.
  2) Else, if one of the operands is double, the other will be converted and the result in double.

3) Else,. If one of the operands is float, the other will be converted to float and the result will be float.
4) Else, if one of the operands is unsigned long int, the other will be converted to unsigned long int and the results will be unsigned long int.
5) Else, if one of the operands is long int and the other is unsigned int, then the unsigned int can be converted into long int.

Explicit Conversion :

- There are instances when we want to force a type conversion in a way that is different from the automatic conversion is known as explicit conversion.
- The process of such a local conversion is known as Explicit Conversion or CASTING A VALUE.

Syntax :

```
(type-name) expression
```

Example	Action
X = (int) 7.5	7.5 is converted to integer by truncation.
A = (int)21.3 / (int)4.5	Evaluated as 21/4 and the result would be 5.

Built-in Functions:

- The C standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

- A function is known with various names like a method or a sub-routine or a procedure etc.These functions can perform task such like string handling, mathematical computations, input/output processing, memory allocation and several other operating system services.

- These functions are included in the c programm by adding their header file in the starting of the c program.
- Some of the header files are:

- Math.h - Defines mathematical functions.
- String.h - Defines string manipulation functions.
- Time.h - Containg time and date handling functions.

UNIT II

Managing Input and Output Statements

Input and Output Operations:

- ★ Unformated I/P and O/P Operations
- ★ Formated I/P and O/P Operations

Unformated I/P and O/P Operations:

Reading a Character :

- ★ Reading a single character can be done by using the function get char()
- ★ The getchar() takes the following form
  - Variable-name = getchar();
- ★ Variable-name is a valid C name that has been declared as char type.

Eg : char name

Name = getchar();

Writing a Character :

- ➢ A putchar() is used for a single character to the terminal.
- ➢ The putchar() takes the following form.

Putchar (variable-name);

Where, variable-name is a type char variable containing a character.

Eg : answer = 'Y';

Purchar (answer);

Example :

```
# include <stdeio.h>
```

```
# incluce<conio.h>
Main()
{
      Char alphabet;
      Printf("Enter an alphabet");
      Putchar ("\n");
      Alphabet = getchar();
      If  (is lower (alphabet))
            Putchar (to upper (alphabet));
      Else
            Putchar(to lwer (alphabet));
}
```

Ouput:

Enter an alphabet: b

B

Formatted I/P and O/P Operations:

Formatted Input :

- Formatted input refers to an input data that has been arranged in a particular format.
- The general form of scanf() is

  scanf("control string", &arg1, &arg2,.....& argn)

- The control string specifies the field format in whch the data is to be entered and the arguments arg2, arg2,.....argn specifies the address of locations wh3re the data is stored.
- Control string and arguments are separated by commas.

Inputting Integer Numbers :

- The field specification for reading an integer number is %wd
- W is an integer number that specifies the field width of the number to be read and d known as datatype.

  Eg : scanf("% 2d %sd", & num1, & num2);

  50 31426

The value 50 is assigned to num1 and 31426 to num2. Suppose the input data is as follows
31426 50

- The variable num1 will be assigned 31 because of %2d abd num2 will be assigned 426.
- The value 50 that is unread will be assigned to the first variable in the scanf next call.

Inputting Character Strings:

- Follwoing are the specifications for reading character strings

  %ws (or) %wc

Reading mixed datatype:

- It is possible to use one scanf statement to input a data line containing mixed mode data.

Eg: scanf("%d %c %f %s", &count, &code, &ration; name);

15 p1.575 coffee

Rules for scanf

- Each variable to be read must have field specification.
- For each field specification there must be a variable address of proper type.
- Any non-white space character used in the format string must have a matching character in the user input.
- Never end the format string with whitespaces. It is a fatal error!
- The scanf reads until :
  - A white space character is found in a numeric specification (or)
  - The maximum number characters have been read (or)
  - An error is detected (or)
  - The end of file is reached.

Scanf code format	
code	meaning
%c	read a single character
%d	read a decimal integer
%e, %f, %q	read a floating point value
%h	read a short integer
%i	read a decimal hexadecimal or octal
%o	read a octal integer
%s	read a string
%u	read a unsigned decimal integer
%x	read a hexadecimal integer

Formatted Output :

- The general form of printf statement is printf("control string", arg1, arg2,.... arg n);

Output of Integer Numbers:

- The format specification for printing an integer no is %wd, where w specifies the minimum field width for the output d specifies that the value to be printed is an integer.

Format	Output
Printf ("% d", 9876)	9 8 7 6
Printf ("% 6d", 9876)	_ _ 9 8 7 6
Printf (""% -6d", 9876)	9 8 7 6 _ _

Output of Real Numbers:

- The output of a real number may be displayed in decimal notation using the following format specification.

  % wpf

- The integer w indicates the minimum number of positions that are to be used for the display of the value and the integer p indicates the number of digits to be displayed after the decimal point.

- It can also display a real number in exponential notation by using the specification.

% wpe

Format	Output
Printf (“% 7.4f ”, y)	9 8 . 7 6 5 4
Printf (“% 7.2f ”, y)	9 8 . 7 6
printf (“% -7.2f ”, y)	9 8 . 7 6
printf (“% f “, y)	9 8 . 7 6 5 4
printf (“%f 10.2e“, y)	9 . 8 8 e + 0 1

Printing A Single Character:

- A single character cab ne displayed in a desired position using the format %wc. The character will be displayed right-justified in the field of w columns. We can make the display left-justified by placing a minus sign before the integer w. The default value of 2 is 1.

Printing of strings :

- The format specification for outputting string is similar to that of real numbers.

  %wps

  Where, w specifies that field width for display and p instructs that only the first p characters of the string are to be displayed. The display is right justified.

Specification	Output
%s	N E W D E L H I 1 1 0 0 0 1
% 20s	N E W D E L H I 1 1 0 0 0 1
% 20.10s	

									N	E	W	D	E	L	H	I

% -20.10s

N	E	W	D	E	L	H	I										

Mixed Data Output:

- It is permitted to mix data type in one printf statement for example the statement of the type.

```
Printf ("%d %f %s %c", a, b, c, d);
```

Control Statement :

- The statement which is used to change (or) alter the flow of execution is called by the name control statement.
- There are two types of control statement.
  1) Selection (or) Branching and decision making.
  2) Looping (or) Iteration.

Selection (or) Branching :

- This is further classified into two types.
  - If
  - Switch

If :

- The different form of if statement is as follows
  - Simple if statement
  - If .....else statement
  - Nested if .....elst statement
  - Eles if .... ladder.

Simple if Statement :

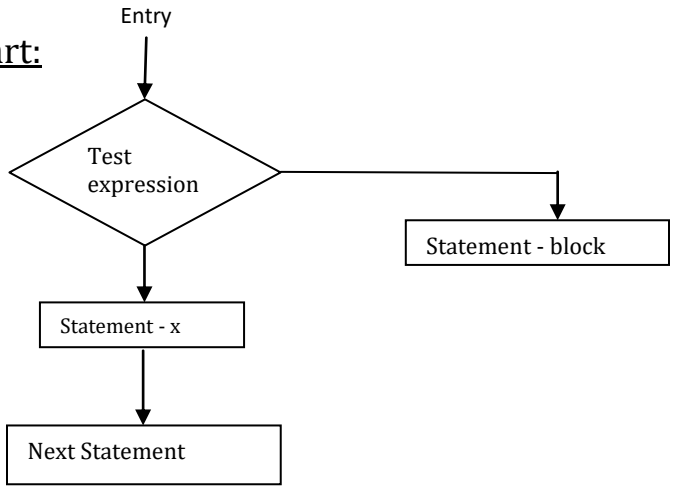
- The General form of a if statement

```
If (test expression)
{
```

```
        Statement-block ;
    }
        Statement – x;
```

- The statement – block may be a single statement of a group of statements. If the test expression is true the statement - block will be executed otherwise the statement - block will be skipped and the execution will jump to statement - x.

Flow chart:



Example:

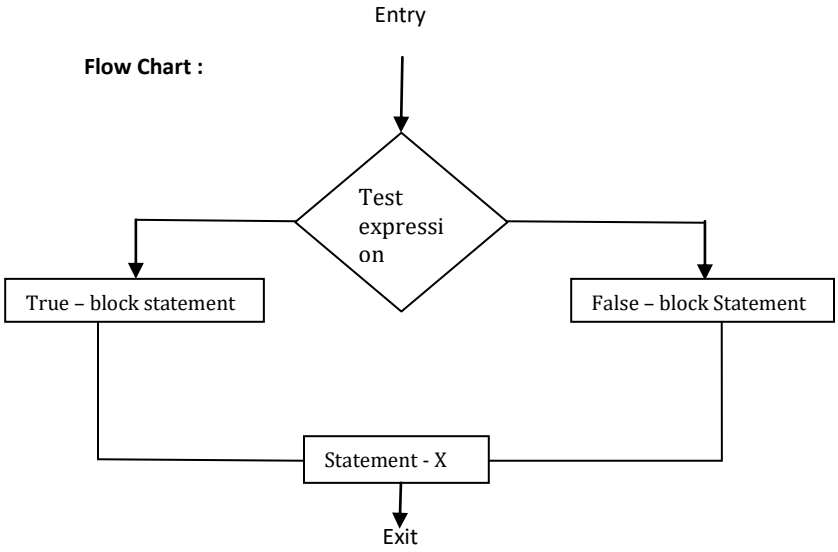
```
# include <stdio.h>
# include <conio.h>
Void main()
{
Int a,b;
Clrscr();
Printf ("Enter the No. ");
Scanf ("%d %d", &a, &b);
If (a>b)
{
Printf ("a is greater");
}
Getch();
}
```

The If ……Else Statement :

- The if....else statement is an extension of the simple if statement.
- The General form is

```
If (test expression)
{
    True – block statement;
}
Else
    {
    Flase – block statement;
    }
Statement – x;
```

Flow Chart :



- The Test expression is true, then the true – block statement, immediately following the if statement are executer; otherwise the false – block statement are executed in either case, either true or false – block will be executed not both in both the cases, the control is transferred subsequently to the statement –x.
- Example :

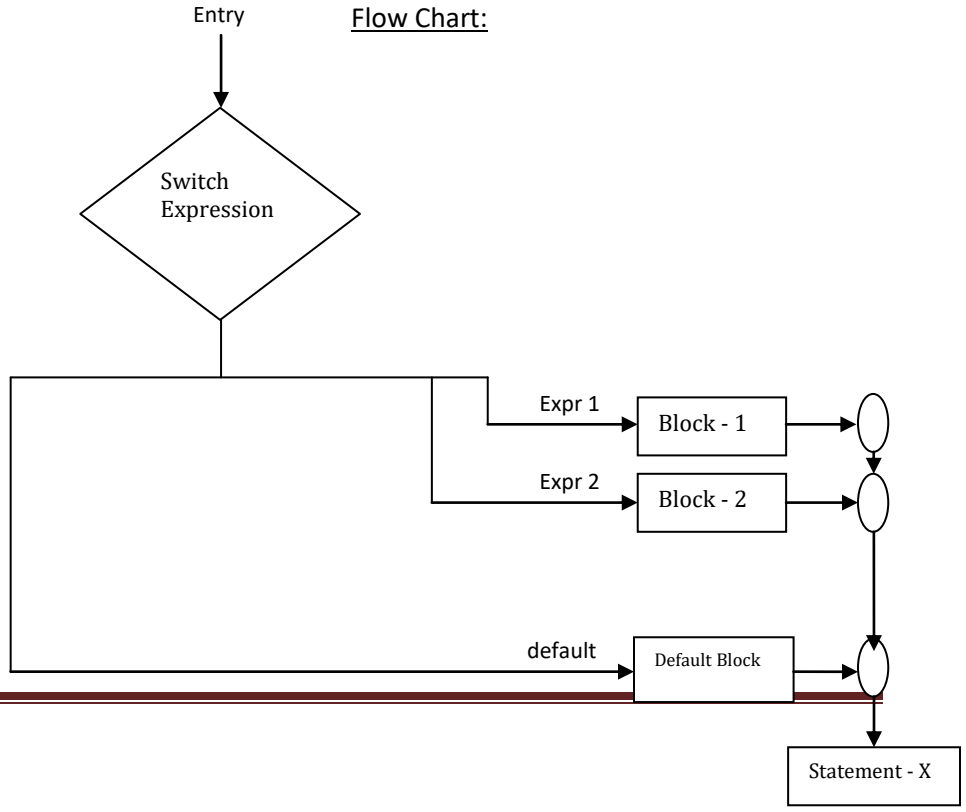
```
    # include <stdio.h>
    #inclide <conio.h>
Void main()
{
Int a,b;
Clrscr();
Printf("Enter the Number");
Scanf("%d D", &a, &b);
If (a>b)
{
    Printf(" a is greater");
}
Else
{
    Printf("b is greater");
}
Printf("Process is Completed");
Getch();
}
```

The Switch Statement :

- C has a built in multi way decision statement known a SWITCH. The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed.
- General Syntax:

```
    Switch (expression)
{
case value-1;
block -1;
break;
case value -2;
block -2;
break 2;
.....
.....
default ;
```

Flow Chart:



default-block

break;

}

statement –x;

- Expression is an integer expression or characters value-1, value-2 are constants or constant expressions are known as <u>case labels</u>. Each of these values should be a unique within a switch.
- Statement block-1, block-2..... are statement lists and may contain zero or more statements. There is no need to put bracer around these blocks. Note that case labels end with a count.
- When the switch is executed the value of the expression is successfully compared against the value, value-1, value-2,,..... if a case is found whose value matches with the value of the expressoion then the block of statements that follows, the case are executed.
- The break statement at the end of each block signals the end of a particular case and causes an exit from the switch sttatement transferring the control to the statement-x following the switch.

<u>Example :</u>

```
# include <stdio.h>
#include<conio.h>
Voidmain()
{
  Int res a, b ,c;
  Clrscr ();
  Printf ("Enter the a, b, value \n");
  Scanf(" %d %d", &a, &b);
  Switch();
  {
     Case 1:
         C = a + b;
          Break;
     Csse 2:
          C = a – b;
          Break;
     Case 3:
          C = a * b;
          Break;
     Case 4:
          C = a / b;
          Break;
     Default:
          Printf ("Exit");
     }
Printf ("The Result is %d",c);
```

```
getch ();
}
```

Rules for Switch Statement:

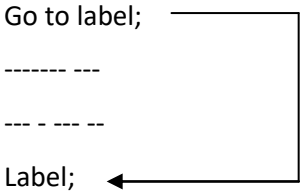
- The switch expression must be an integral type.
- Case labels must be constants.
- Case labels must be unique. No 2 labels can have the same value. It must end with colon.
- The break statement transfers the control out of the switch statement.

GoTo Statement:

- The go to statements is used to branch unconditionally from one point to another in the program. The goto requires a label in order to identify the place where the branch is to be made. A label is any valid variable name, and must be follows by a colon.

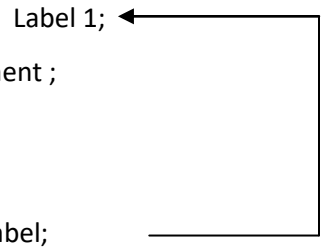
The G.F.:

Forward jump



Statement;

Backward jump



- The label : can be anywhere in the program either before or after the fotolabel; statement.
- Ex :

```
#include <stdio.h>
# include<conio.h>
Void main()
{
  Int i,x;
  Printf (" goto example program \n");
  Scanf ("%d", & x);
  For (i = 0; i < 100; i = i + 2)
```

```
   {
      If ( i==50)
           Goto x;
     Printf ("i  = %d \n",');
  }
Printf("End");
getch();
}
```

Decision Making AND Looping:

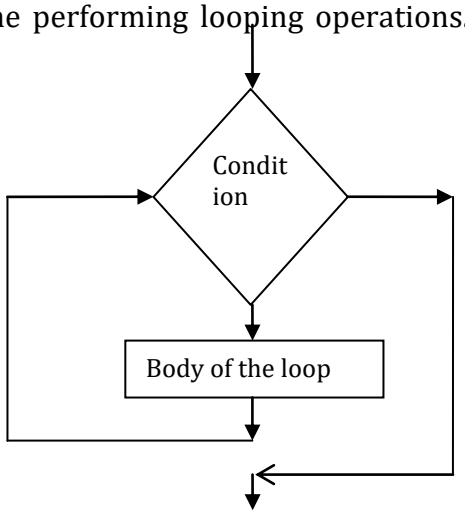
- The loop in a program consists of two parts one is body of the loop another one is control statement
- Any looping statement includes the following steps.
  1) Initialization of a condition variable.
  2) Test the control statement
  3) Executing the body of the loop depending on the condition.
  4) Updating the condition variable.
- There are two types of looping.
  i) Entry – controlled loop
  ii) Exit – controlled loop.

Entry controlled loop:

- In the entry controlled loop the condition are tested before the start of the loop execution.

Exit controlled loop:

- In the exit – controlled loop the test is performed of the end of the body of the loop. Therefore body is executed unconditionally for the first time.
- The 'C' language provides for three constructs for the performing looping operations. They are,
  1. While Statement
  2. Do _while Statement
  3. For loop Statement.



The While Statement:

G.F:

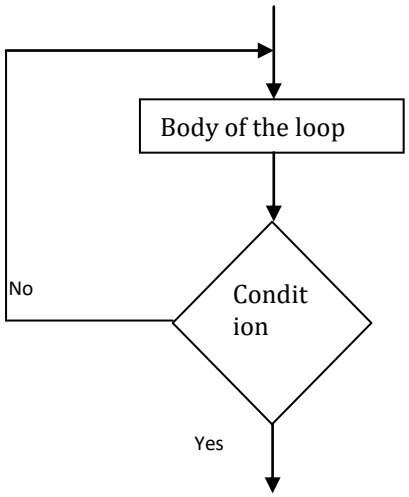
```
While (test condition)
{
   body of the loop;
}
```

- The while is an entry-controlled loop statement. The best condition is evaluated and if the condition is true then the body of the loop is executed. After execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again.
- This process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the loop.

Example:

```
# include <stdio.h>
#include ?<conio.h>
void main()
{
        int i, sum;
        Clrscr ();
        i=1;
        Sum = 0;
        While (i<=10)
        {
                Sum = sum+1;
                i++
        }
        Printf("The sum of the numbers is =%d",sum);
        getch();
}
```

Do - While Statement:



Syntax:

```
Do
        {
        Body of the loop;
        } While (test-condition);
```

- The program proceeds to evaluate the body of the loop first. At the end of the loop the test-condition in the while statement is evaluated. If the condition is true the program continue to evaluate the body of the loop once again. The process continues as long as the condition is true when the condition become false the loop will be terminated.

Ex :

```
# include <stdio.h>
# include <conio.h>
void main()
{
        int i, sum;
        Clrscr();
        i = 1;
        Sum = 0;
        Do
        {
```

```
                Sum = sum + 1;
                i++;
        }
        While ( i < = 10);
        Printf (“The sum of the no is =%d”, sum);
        getch ();
}
```

For Loop:

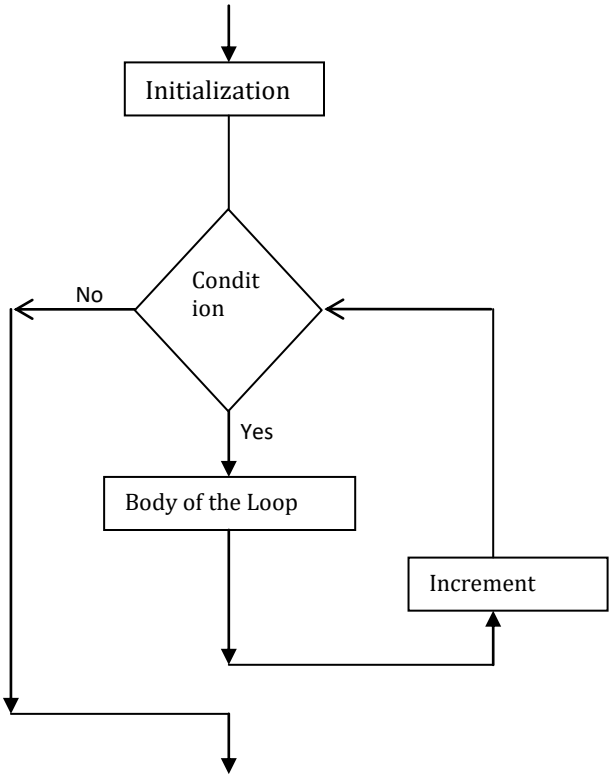
Syntax:

For (initialization; test-condition; increment)

-- - - - -- ----

---

body of the loop



Flow chart :

Example:

```
# include <stdio.h>
# include <conio.h>
void main()
{
 int i, sum;
Clrscr();
i = 1;
Sum = 0;
for ( i = 1; i < = 10; i + +)
{
        Sum = sum + i;
}
Printf (“The sum of the no is = % d”, sum);
getch ();
}
```

Difference between while and do-while statement:

While	Do-While
This is the top tested loop [Entry Controlled]	This is the bottom tested loop [Exit Controlled]
This condition is first tested. If the condition is true then the block is executed until the condition is false	If executes the body once after it check the condition. If it is true the body executed until condition becomes false.
Loop is not executed. If the condition is false	Loop is executed at least once even though the condition is false.

It has no termination	It has termination

Break Statement :

- Break statement is used to terminate the loop

Syntax:

break:

Eg :

```
# include <stdio.h>
# include <conio.h>
void main()
{
 int i;
for (i = 1; i < = 10; i + +)
{
        for (i == 6)
        break;
        printf("%d" , i);
}
getch ();
}
```

The Continue Statement:

Syntax :

continue;

Eg :

```
# include <stdio.h>
# include <conio.h>
void main()
{
 int i, n,sum = 0;
Clrscr();
for ( i = 1; i < = 5; i + +)
{
        Printf ("The sum of the no is = % d", sum);
        Scanf  ("%d", &n);
        If (n < 0)
                continue;
        Else
                Sum = sum + n;
}
Printf ("The sum of the no is = % d", sum);
getch ();
}
```

Break	Continue
Break statement takes the control to the outside of the loop	Continue statement takes the control to the beginning of the loop
It is also used in switch statement	This can be used only in loop statement
Always associated with if condition in loops	This is also associated with if condition in loops

UNIT – III

(Arrays- Character Arrays and Strings –User defined functions.)

ARRAYS

An array is a sequential collection of related data items which are stored in consecutive memory locations that share a common name. In other words an array is a group or table of values referred by the same variable name.The individual values in an array are called elements.

Arrays are sets of values of the same type,which have a single name followed by an index. In C , square brackets appear around the index right after the name. Eg:- salary[10].

Declaring the name and type of an array and setting the number of elements in the array is known as dimensioning the array.

Depending upon the number of subscripts in the array, arrays can be classified into

i. One-dimensional arrays
ii. Two-dimensional arrays
iii. Multi-dimensional arrays

One-dimensional Arrays:

A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or a one dimensional-array.

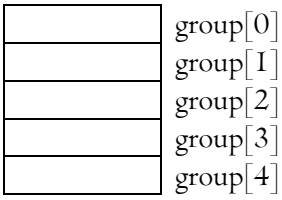
An array must be declared before it ish used.

Syntax:- **data_type array_name[size];**

The data type specifies the type of the element, whether int,float or char. The size indicates the maximum number of elements that can be stored inside the array.It should be positive integer.

Eg:- int group[5];

group is an array which contains 5 integer constants.The computer reserves 5 storage locations as shown below.



The first element in the array is numbered 0, so the last element is 1 less than the size of the array i.e for 5 it is 0,1,2,3 and 4.

After an array is declared , its elements must be initialized. Array can be initialized either at two stages : a) At compile time b) At run time.

At compile time:

Syntax: **data_type array_name[size] ={ list of values};**

The values in the list are separated by commas.

Eg:- int group[5] = {20,30,40,10,50};

i.e

20	group[0]
30	group[1]
40	group[2]
10	group[3]
50	group[4]

If the number of values is less than the number of elements, then only that many number are initialized. The remaining elements will be set to zero automatically.
Sometime the size may be omitted , in such cases, the compiler allocates enough space for all initialized elements.

E.g:- int counter[]={2,4,1,0};

This would declare counter array to contain four elements with initial value 1.

At run time:

For large arrays e.g.

```
for(i=0;i<100;i++)
{
  if(i<50)
     sum[i]=0;
 else
     sum[i]=1.0;
}
```

The first 50 elements of the array sum is initialized to zero while the remaining 50 elements are initialized to one at run time.

We can also use scanf function to initialize an array.

Eg. int x[3];

scanf ("%d%d%d",&x[0],&x[1],&x[2]);

This will initialize array elements with the values entered through the keyboard.

Example for one-dimensional Arrays:

1.
```
main()
{
 int a[5]={11,12,13,14,15};
int i;
printf("contents of the arrya\n");
for(i=0;i<=4;i++)
{
printf("%d \t" , a[i]);
}
}
```

Output: contents of the array

11 12 13 14 15

2.
```
main()
{
int a[100];
int i,n;
printf("how many numbers are in the array? :");
scanf("%d",&n);
printf("Enter the elements");
for(i=0;i<=n-1;i++)
```

```
{
scanf("%d",&a[i]);
}
printf("contents of the array\n");
for(i=0;i<=n-1;i++)
{
printf("%d\t",a[i]);
}
}
```

Two-dimensional array:

When a table of values are to be stored, two-dimensional arrays are used. A two-dimensional will require two pairs of square brackets.

Syntax:- **data_type array_name[row-size][column-size];**

Eg:- int x[2][2];

It is stored in memory as follows:

	Column 0	Column 1
Row 1	[0][0]	[0][1]
Row 2	[1][0]	[1][1]

Two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. The initialization is done row by row.

```
Eg:  int x[2] [2] = {0,0,1,1};
                (or)

int x[2] [2] = { {0,0},{1,1}};
                (or)
int x[2] [2] = {
                        {0,0},
                        {1,1}
                    };
```

When an array is completely initialized with all values, explicitly we need not specify the size of the first dimension.

```
Eg:- int x[ ] [2] = { {0,0},{1,1}};
```

If the values are missing in an initialize, they are automatically set to Zero.

Example for two-dimensional arrays:

```
1.      void main()
    {
    int i,j;
    int a[3][4] = {
                    {1,2,3,4} ,
                    { 5,6,7,8},
                    {9,10,11,12}
                } ;
```

output:

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

```
printf( "Contents of Array:\n");
 for(i=0;i<=2;i++)
{
     for( j=0;j<=3;j++)
     {
          printf( "%d \t" , a[i][j]);
     }
printf("\n");
}
}
```

2.
```
main()
{
int a[50];
int i,j,n,m;
printf("Enter how many rows and cols in the array?");
scanf("%d%d",&n,&m);
printf("Enter the elements");
for(i=0;i<=n-I;i++)
{
  for(j=0;j<=m-I;j++)
   {
    scanf("%d",&a[i][j]);
   }
}
printf("contents of the array\n");
for(i=0;i<=n-I;i++)
{
for(j=0;j<=m-I;j++)
   {
    printf("%d",&a[i][j]);
   }
printf("\n"]);
} }
```

CHARACTER ARRAYS AND STRINGS:

A string is a sequence of characters that is treated as a single data item.Any group of characters defined between double quotation is a string constant. Eg: " India".

C allows us to represent strings as character arrays.Therefore a string variable is any valid C variable name and is always declared as an array of characters.

Syntax:-

char string_name [size];

The size determines the number of characters in the string_name.

Eg:- char city[10];

When the compiler assigns a character string to a character array, it automaticallysupplies a null character ('\0') at the end of the string. Therefore , the size should be equal to the maximum number of characters in the string plus one.

i. char city[9] = "NEW YORK" ;
ii. char city[9] = {'N', 'E','W',' ','Y','O','R','K'};

We can also initialize a character array without specifying the number of elements.In such cases the size of the array will be determined automatically based on the number of elements initialized.

Eg: char string[]={“good”};

Here there are 5 elements in the array string.

We can also declare the size much larger than the string in the initialize.

Eg: char string[10]= {“good”};

Reading strings from terminal:

A) Using scanf function:

scanf can be used with %s format specification to read a string of characters and & is not required before the variable name.

```
Eg:- char address[10];
     scanf('%s',address);
```

The scanf function terminates its input on the first white space it finds.

Eg:- “NEW YORK” . Only NEW will be read into the arrays. If we want to read the entire line“NEW YORK” ,then we must use two character arrays of appropriate sizes.

```
Eg: char addr1[5],addr[5];
    scanf(“%s%s”,addr1,addr2);
```

This would assign the string “NEW” to addr1 and”YORK” to addr2.

scanf function will read only strings without white spaces. To read a text containing more than one word, C supports a format specification known as edit set conversion . code %[].

```
Eg: char line[80];
    scanf(“%[^\n]” , line);
    printf(“%s”, line);
```

B) Using getchar and gets function:

getchar cab be used to repeatedly read successive single character from the input and place them into a character array.Thus, an entire line of text can be read and stored in an array.The reading is terminated when the newline character is entered.

```
Eg:- char ch;
     ch=getchar();
```

The getchar function has no parameter.

gets function can also be used to read a text containing white spaces.This is a simple function with one string parameter and called as under:

```
gets(str);
```

str is a string variable declared properly.It reads character into str from the keyboard until a newline character is encountered and then appends a null character to the string.

```
Eg:- char line[80];
     gets(line);
     printf(“%s”,line);
```

Writing strings to screen:

A) Using printf function:

The format %s can be used to display an array of characters.

Eg:- printf(“%s”,name); can be used to display the entire contents of the array name.

We can also specify the precision with which the array is displayed.

Eg:- %10.4 indicates that the first 4 characters are to be printed in a field width of 10 columns. If we include minus sign (% - 10.4), the string will be left justified.

B) Using putchar and puts functions:

putchar is used to output the values of character variables. It takes the following form:

```
char ch ='A';
putchar(ch);
```

The function putchar requires one parameter. This statement is equal to printf("%c",ch);

Another way of printing string values is to use the function puts.

```
puts(str);
```

where, str is a string variable containing a string value.This prints the value of the string variable str and then moves the cursor to the beginning of the next line on the screen.

```
Eg:- char line[80];
     gets(line);
     puts(line);
```

STRING HANDLING FUNCTIONS:

C library <stdlib.h> supports many string-handling functions that can be used to carry out many of the string manipulations. The following are the most commonly used string-handling functions.

i. strcat() - concatenates (joins) two strings
ii. strcmp() - compares two strings
iii. strcpy() - copies one string over another
iv. strlen() - finds the length of a string

I. strcat () function:

The strcat function joins two strings together.

Syntax: **strcat(string1, string2);**

Strign1 and string2 are character arrays.When the function strcat is executed, string2 is appended to string1.It does so by removing the null character at the end of string1 and placing string2 from there.

Eg:-

partI =

0	1	2	3	4	5	6	7	8	9
V	E	R	Y		\0				

part2=

0	1	2	3	4	5	6	7	8	9
G	O	O	D	\0					

part3=

0	1	2	3	4	5	6	7	8	9
B	A	D	\0						

strcat(partI,part2); will result in

partI =

0	1	2	3	4	5	6	7	8	9
V	E	R	Y		G	O	O	D	\0

part2=

0	1	2	3	4	5	6	7	8	9
G	O	O	D	\0					

We must make sure that the size of the string2(to which string2 is appended) is large enough to accommodate the final string.

strcat may also append a string constant to a string variable.

Eg:- strcat(part1,"hello");

C permits nesting of string functions. Eg:- strcat(strcat(str1,str2),str3);

II. strcmp() function:

The strcmp function compares two strings identified by the arguments and has a value 0 if they are equal.If they are not, it has the numeric difference between the first non-matching characters in the strings.

Syntax:

strcmp(string1,string2);

string1 and string2 may be string variables or string constants.

Eg:- strcmp(name1,name2);
strcmp(name1,"john");
strcmp("RAM",ROM");

III. strcpy() function:

This copies one string over another. It works like a string assignment operator.

Syntax:

strcpy(string1,string2);

This assigns the contents of string2 to string1. String2 may be character array variable or a string constant.

Eg:- strcpy(city,"Delhi");

This assigns Delhi to the string variable city.

IV. strlen() function:

This function counts and returns the number of characters in a string.

Syntax:

n = strlen(string);

where n is an integer variable, which receives the value of the length of the string.The counting ends at the first null character.

USER DEFINED FUNCTIONS:

C functions can be **classified i**nto two categories.

i) Library functions or Built-in functions (Eg:- printf and scanf)
ii) User-defined functions. (main())

A complex C Program can be decomposed into small or easily manageable parts. Each small module is called a function. In C main function itself is a function which invokes the other functions to perform a task.

Advantages:- 1. Writing a function avoids rewriting the same code over and over.
2.Simple to write correctly a small function.
3.Easy to read , write and debug a function.

Elements of User-defined function:

There are 3 elements:

i) Function definition
ii) Function call
iii) Function declaration

Function definition:

Syntax:

```
Function_type  Function_name (Parameter list)      } function header
{
        Local variable declaration;
        Executatble statement-1;                   } function body
        ……
        return  statement;
}
```

Function Call:

A function can be called by simply using the function name followed by a list Of actual parameters enclosed in paranthesis.

Eg: y = mul(10,5);

The actual parameters must match the functions parameter in type,order and number.

Function Declaration: (or) Function Prototype :

All functions in a C program must be declared before they are called.

Syntax:

Function_type function_name(Parameter list);

Eg:- int mul(int m, int n);

THE return STATEMENT:

The Keyword return is used to terminate function and return a value to its caller. The return statement may also be used to exit a function without returning a value. The return statement may or may not include an expression.

Syntax:

return;

return(expression);

TYPES / CATEGORIES OF FUNCTION:

There are three categories:

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with argument and return values.

Functions with no arguments and no return values:-

It is the simplest way of writing a user defined function in C. There is no data communication between a calling portion of a program and a called function block.The function is invoked by a calling environment without any formal arguments and the function also does not return back anything to the caller.

Eg:-

```
main()
  {
      message();
  }
void message()
  {
      printf("hello");
  }
```

Functions with arguments and no return values.

Some formal arguments are passed to the function, but the function does not return back any value to the caller.It is a one way data communication between a calling portion of a program and the function block.

Eg:

```
main()
  {
```

```
        int i, max;
        printf("Enter Value");
        scanf("%d",&max);
        printf("Number\t Square");
        for(i=0;i<=max-I;i++)
        square(i);
    }
square(int n);
    {
        int temp;
        temp = n * n;
        printf("%d\t%d\n",n,temp);
    }
```

Functions with argument and return values:

Here some formal arguments are passed to a function from calling portion of the program and the computed values, if any, is transferred back to the caller. Data are communicated between calling portion and a function block.

Eg:
```
main()
    {
        int i, temp,max;
        printf("Enter Value");
        scanf("%d",&max);
        printf("Number\t Square");
        for(i=0;i<=max-I;i++) {

        temp = square(i);
        printf("%d\t%d\n",i,temp);

    }
square(int n);
    {
        int value;
        value = n * n;
        return(value);
    }
```

ACTUAL AND FORMAL ARGUMENTS:

Sometimes a function may be called by a portion of a program with some parameters and these parameters are known as actual arguments.

The formal arguments are those parameters present in a function definition. It is also called as dummy arguments or parametric variables.

Eg:
```
main()
{
 int a,b,c,sum;
 …….
 sum = calsum(a,b,c);  ——————→ actual arguments
 …….
}
calsum( int x, int y, int z)  ——————→ formal arguments
{
 ……..
}
```

LOCAL AND GLOBAL VARIABLES:

Local variables are defined inside the function b**lock or compound statement and are referred only to the particular part of the block or function.The same variable name may be** given to the different part of the function or a block and each variable will be treated as **different** entity.

```
func( int i, int j)
{
```

```
        int k , m;  ——————→ local variables
    }
```

The global variables are declared outside the main function block. These variables are referred to the same datatype and the same name throughout the program in both calling portion of a program and a function block.

Eg:
```
int x,y=4;
main()
{
 x = 10;
…….
 funI();
}
funI()
{
 int sum;
 sum = x + y;
 …….
}
```

CALL BY VALUE AND CALL BY REFERENCE:

Arguments can generally passed to functions in one of the two ways.

1. Sending the values of the argument.
2. Sending the address of the argument.

In the first method the value of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function. The changes made to the formal arguments in the called function have no effect on the values of the actual arguments in the calling function.

Eg:-
```
main ()
    {
    int a=10,b=20;
    swap(a,b);
    printf("a=%d b=%d",a,b);
    }
swap(int x, int y)
    {
    int t;
    t = x;
    x = y;
    y = t;
    printf("x = %d y=%d",x,y);
    }
```

The value of a and b remain unchanged even after exchanging the values of x and y.

In the second method the address of actual arguments in the calling function are copied into formal arguments of the called function.

Eg:-
```
main ()
    {
    int a=10,b=20;
    swap(&a , &b);
    printf("a=%d b=%d",a,b);
    }
swap(int  *x, int *y)
    {
    int t;
    t = *x;
    *x = *y;
    *y = t;
    printf("x = %d y=%d",x,y);
    }
```

Using addresses we would have an access to the actual arguments and hence we would be able to manipulate them. We cannot alter the actual arguments, if desired it can be achieved through call by reference.

RECURSION:

The function which calls itself directly or indirectly again and again is known as the recursive function. There is much difference between the normal function and the recursive function. The normal function will be called by the main function whenever the function name is used. On the other hand the recursive function will be called by itself directly or indirectly as long as the given condition is satisfied.

Eg:-
```
main()
{
        int a,fact;
        printf("Enter a number");
        scanf("%d", &a);
        fact = rec (a);
        printf("Factorial =%d",fact);
}
rec ( int x)
{
        int f;
        if (x = =I)
        return;
        else
        f=x*rec(x-I);
        return f;
}
```

UNIT-IV

(*Structures and Unions – Pointers – File Management in c.)*

POINTERS

Definition:

A pointer is a variable which holds the address of another variable. The Pointer is a powerful technique to access the data by indirect reference because it holds the address of that variable where it has been stored in the memory.

Accessing the address of a variable:-

Declaring pointer variable:

Like any other variable, a pointer variable must be declared as pointer before we use them:

General form: **datatype *pt_name;**

The asterisk (*) tells that the variable pt_name is a pointer variable. pt_name points to a variable of type datatype. pt_ name needs a memory location.

Eg/-
```
int *p;
float *p;
```

Initialization of pointer variable:-

The process of assigning the address of a variable to a pointer variable is known as initialization. Accessing The address of a variable is done through &(ampersand) operator i.e address of operator. The pointer variable must be initialized before they are used in the program. We can initialize the Variable using assignment operator.

Eg\-
```
int quantity;
int *p;     /* declaration */
P=&quantity;   /* initialization */
```

Accessing a variable through its pointer:-

After assigning the address of a variable to a pointer, We can access the value the value of a variable using the pointer. This is done by using another *operator or indirection operator or dereferencing operator.

Eg\-
```
int quantity, *p,n;
quantity=179;
p= & quantity;
n=*p;     /* use of indirection operator */
```

The last line contains the indirection operator *. When the operator * is placed before a pointer variable in an expression (on the right hand side of the equal sign), the pointer returns the value of the variable of which the pointer value is the address.

Here *p returns the value of the variable quantity i.e the value of n would be 179. The * can be remembered as "value at address".

Valid pointer declaration:-

```
int x,y;
int *p1,*p2;
```

1) p1=&x;

The memory address of x is assigned to a pointer variable p1.

2) y=*p1;

The contents or value of the pointer variable *p1 is assigned to the variable y, not the memory address.

3) p1=&x;
   p2=p1; /* address of p1 is assigned to p2 */

The address of p1 is assigned to pointer variable p2. The contents of both p1 and p2 will be same.

Invalid pointer Declaration:-

1) int x;
   int x_ptr; /* error */
   x_ptr =&x;

Error: pointer declaration must have prefix of *.

2) float y;
   float *y-ptr;
   y-ptr=y;   /* error */

Error: While assigning variable to the pointer variable the address operator (&) must be used along with the variable y.

3) int x;
   char *c-ptr;
   c-ptr =&x;

Error: Mixed data type is not permitted

Eg/-

```
#include <stdio.h>
main()
{
int x=10;
int *ptr;
ptr=&x;
printf(" Address of x=%u",&x);
Printf("Address of x=%u",ptr);
Printf("Address of ptr=%u",&ptr);
```

```
                Printf("Address of ptr=%u",ptr);
                Printf("Address of x=%d",x);
                Printf("Address of x=%d",*(&x));
                Printf("Address of x=%d",*ptr);
                }
```
o/p:

```
Address of x=6485
Address of x=6485
Address of ptr=5276
Address of ptr=6485
Address of x=10
Address of x=10
Address of x=10
```

Pointer Arithmetic:

Some arithmetic operation can be performed with pointers. The C language four arithmetic operators that may be used with pointers such as,

addition +
subtraction –
incrementation ++
decrementation –

C allows to add integers to or subtract integer from pointers,as well as to subtract one pointer from another

p1+4, p2=2 and p1=p2 are valid but p1+p2 is illegal because no two pointers can be added .Pointers can be compared using relational operator.

p1>p2,p1==p2 and p1!=p2 are valid.

The following pointers variables can be used in expression. If p1 and p2 are properly declared and initialized, then the following statement are valid.

```
y=*p1 X  *p2;
sum=sum+*p1;
Z=5* - *p2/*p1;     //  (5*(-(*p2)))/(*p)
```

In case of pointer incrementation.

```
Eg/-
int value,*p1;
value=100;
P1=&value;
P1++;
```

If the address of value is assigned as 2800 to p1, then after p1=p1+1.the value of p1 will be 2802, that is when we increment, a pointer, its value is increased by "length" of the datatype that is points to. This length is called "*scale factor*".

Character 1 byte
Integer 2 bytes
Float 4 bytes
Longint 4 bytes
Double 8 bytes

Pointer and arrays:

In C, there is a close correspondence between array datatype and pointers. An array name in C is very much like pointer but there is difference between each others. The pointer is a variable that can appears on the left side of an assignment operator. The array name is a constant and can't appear on the left side of an assignment operator

The following declaration is valid

```
int value [20];
int*ptr;
```

can be declared as

value [0]

which holds the address of the Zeroth element of the array value,

the pointer variable pointer is also an address, so the declaration value[0]and pointer is same because both holds the address.

The following statement is value [0];

The address of the zeroth element is assigned to a pointer variable pointer.If the pointer is incremented to the next data element then the following expression of the equality operator is same.

Eg/-

```
Ptr++  == value[I]
The following equalities are valid
Ptr+6  =&value[6];
Ptr ==&value[0];
*(ptr+6)==&value[6];
Ptr++==&value[I];
```

Array subscripting is defined in terms of pointer arithmetic.

(or) The expression a[i] is defined to be the same as *((a) + (i)) which is to say the same as *(&(a)[0]+(i)).

Eg\- /* to display the contents of the array using a pointer arithmetic */

```
#include<stdio.h>
main()
{
int a[4]={11,12,13,14};
int I,n,temp;
n=4;
printf("contents of the array \n");
for(i=0;i<=n-1;i++)
{
temp=*((a)+(i));
printf("value=%d\n",temp);
}
}
```

o/p:

Contents of the array,

Value =11

Value=12

Value=13

Value=14

Array of pointers:

Pointers may be arrayed like any other datatype.The declaration for an integer pointer array of size 10 is,

```
int*ptr[10];
```

makes ptr[0],ptr[1],ptr[2]….ptr[n] an array of pointer.

Eg/-

```
#include<stdio.h>
main()
{
char*ptr[3];
ptr[0]="hai";
ptr[1]="hello";
ptr[2]="bye";
printf("Contents of printer1=%s\n",ptr[0]);
printf("Contents of pointer2=%s\n",ptr[1]);
printf("Contents of pointer3=%s\n",ptr[2]);
}
```

STRUCTURES:

C supports a constructed data type known as structures. Structure consist of a group of variable of different datatypes placed in a common name. A structure is a heterogeneous datatype where as an array is a homogeneous data type.

Defining a structure:

The format of a structure must be defined first.

Syntax:

```
struct tag_name
{
        datatype member1;
        datatype member2;
        ……
};
```

Here, struct is a keyword and it declares a structure to hold different data fields.Each field is called a structure element or member. Each member may belong to different data types. The tag_name is the name of the structure and is called structure tag. The tag_name is used for declaring variables. The template should end with a semicolon.

Eg:-

```
struct book_bank
{
        char title[20];
        char author[15];
        int pages;
        float price;
};
```

book_bank is the name of the structure. It holds four data fields namely title,author,pages and price. Each of the members belong to different type of data.

Declaring Structure variables:

The above definition has not declared variables. We can declare structure variables like any other variable. Declaring includes

i. The keyword struct
ii. The structure tag name
iii. List of variable names separated by commas
iv. A terminating semicolon

Eg:-

```
struct book_bank book1,book2,book3;
```

This statement declares book1,book2 and book3 as variables of type struct book_bank. The complete definition is as follows.

```
struct book_bank
{
char title[20];
char author[15];
int pages;
float price;
} b1,b2b3;
```

(or)

```
struct book_bank
{
char title[20];
char author[15];
int pages;
float price;
}
struct book_bank b1,b2b3;
```

Accessing Structure members:

The members are not variables.So they have to be linked to the structure variables in order to make them meaningful. The link is established using the member operator '**.**' which is also known as dot operator or period operator. eg:- b1.price

Structure Initialization:

Like any other variable structure be can also be initialized.

At Compile time:

```
main()
{
 struct st_record
 {
      int weight;
      int height;
 };
struct st_record s1={50,70,80);
struct st_record s2={55,75,85);
……
}
```

We can also initialize a structure variable outside the function.

```
struct st_record
 {
      int weight;
      int height;
 };
main()
{
 struct st_record s1={50,70,80);
 struct st_record s2={55,75,85);
 ……
}
```

We can also assign values as follows.

Eg:- i. strcpy(s1.name,"Ram"); ii. s1.weight = 60;

At Run time:

We can also use scanf to give values through keyboard at runtime.

Eg:- scanf("%s%d",s1.name, &s1.weight);

Example:

```
struct person
{
 char name[20];
 int day;
 char month[10];
 int year;
 float salary;
};
main ( )
 {
```

```
struct person p;
printf("Input values");
scanf("%s%d%s%d%f " , p.name, &p.day, p.month, &p.year, &p. salary);
printf("%s%d%s%d%f " , p.name, &p.day, p.month, &p.year, &p. salary);
}
```

Copying and Comparing Structure Variables:

Two variables of the same structure types can be copied the same way as the ordinary variables.

For example , if b1 and b2 belong to the same structure then

b1 = b2;

b2 = b1; are valid statements.

C does not permit any logical operations on structure variables.

b1!=b2;

b2==b1 are invalid.

We can compare them only by comparing the members individually.

b1.author==b2.author

Arrays of Structures:

Similar type of structures placed in a common heading or a common variable name is called an array of structures.

Eg:-

```
struct book
{
char name[10];
int pages;
float price;
} b[100];
```

In the above example b is an array of 100 elements b[0] …..b[99]. Each element is defined to be of type struct book.

Eg:-

```
struct book
{
char name[10];
int pages;
float price;
};
main ()
{
int i;
struct book b[100];
printf("enter the values for each element\n");
for(i=1;i<100;i++)
{
        scanf("%s%d%f",b[i].name,&b[i].pages, &b[i].prices);
}
printf("The structure values are \n");
for(i=1;i<100;i++)
{
         printf("%s%d%f",b[i].name,b[i].pages,b[i].prices);
} }
```

The array is declared just as it would have been with any other array. Since b is an array, we use the usual array accessing methods to access individual elements and then the member operator to access members.

Arrays within Structures:

C permits the use of arrays as structure members. We can use character arrays, single or multidimensional arrays of type int and float inside a structure.

Eg:-
```
struct marks
        {
        int number;
        float subject[3];
        };
struct marks student[2];
```

Here the member subject contains three elements, subject[0],subject[l] and subject[2]. These elements can be accessed using appropriate subscripts.

Eg: student[I] ,subject[2];

Would refer to the marks obtained in the third subject by the second student.

Eg:
```
main()
        {
        struct marks
        {
        int sub[3]; int total;
        };
        struct marks student[3]={45,67,81,0,75,53,69,0,57,36,71,0};
        struct marks total;
        inti,j;
        for(i=0;i<=2;i++)
        {
        for(j=0;j<=2;j++)
        {
        student[i] ,total+=student[i] .sub[j]; total. sub[j]+=student[i].sub[j];
        }
        total. total+=student[i] .total;
        *
        }
        printf("student Total\n"); for(i=0;i<=2;i++)
        printf("student[%d] %d\n" , i+I , student[i] .total);
        printf("subject TotalVn");
        for(j=0;j<=2;j++)
        printf("subject %d %d\n",j+l ,total.sub[j]); printf("grand total=%d\n", total.total);
        }
        O/P:
        student    total
        student[ I ]          I93
        student[2]            I97
        student[3]            I64
        subject total
        subject I  I77
        subject 2  I56
        subject 3  22I
        Grand Total = 554
```

Structures within structures :

A structure within a structure means nesting of structures. The syntax of the structure within structure is as follows.

```
struct time
{
int second;
 int minute;
 int hour;
};
struct t
{
```

```
int carno;
struct time st;
struct time et;
};
struct t player;
```

Structures and Functions:

Structures can be passed to functions. There are three methods:

i. The first method is to pass each member of the structure as the actual argument of the function call. The actual argument are then treated independently like ordinary variables.

Eg:

```
struct date
{
int day; int month; int year;
};
main()
{
struct date a; a.day=I0; a.month=2; a.year=I992;
output(a.day, a.month, a.year);
}
output(n)
struct date n;
{
printf("Today's date is: %d/%d/%d/n'\n.day,n.month,n.year);
}
0/P:
Today's date is: I0/I2/I992
```

ii. The second method involves passing a copy of the entire structure to the called function.

eg:

```
struct date
{
int day; int month; int year;
};
main()
{
struct date a={ I0,2,2017}; output(a);
}
output(n) struct date n;
{
printf("Todav's date is: %d/%d/%d",n.day,n.month,n.year);
}
O/P:
Today's date is: I0/2/2017
```

iii. The third method is by using pointers to pass the structure as an argument. In this case the address location of the structure is passed to the called function.

eg:

```
struct book
{
char name[35]; char author[35]; int pages;
};
main()
{
struct book bl={"Java", "P. Naughton",886}; show(&bl);
}
show(struct book *bl)
{
printf("\n%S by %S of %d pages", bl-> name, bl-> author, bl-> pages);
}
O/P:
Java by P. Naughton of 886 pages
```

UNIONS

Union is a variable which is similar to the structure. It contains a number of members like structure but it holds only one object at a time. In the structure each member has its own memory location, whereas members of union have same memory locations.The union requires bytes that are equal to the number of bytes required for the largest members. For example, if the union contains char, int and longint then the number of bytes reserved in the memory for the union is 4 bytes.

eg:

```
main()
{
union result
{
int marks; char grade;
};
struct res
{
char name[15]; int age;
union result perf;
}data;
printf("size of union:%d\n", sizeof(data.perf)); printf("size of structure:%d\n", sizeof(data));
}
O/P:
size of union: 2 size of structure: 19
```

FILE MANAGEMENT IN C

Introduction:

The console oriented I/O operations done through terminals(keyboard and screen) using printf and scanf functions works fine as long as the data is small .But,

- handling larger volumes of data is difficult and time consuming
- data is lost when either the program is terminated or the computer is turned off

These drawbacks can be overcome using files, where we can store data and read it whenever necessary.

Definition:

"A file is a place on disk where a group of related data is stored".

Basic file operations:

naming a file
opening a file
reading data from a file
writing data to a file
closing a file

File handling functions:

The above said file operations can be performed by using the file handling functions available in the C library. They are,

i. fopen()-creates a new file for use and opens an existing file for use.
ii. fcloseQ-closes a file which has been opened for use.
iii. getc()-reads a character from a file.
iv. putc()-writes a character to a file.
v. getw()-reads a integer from a file.

vi. putw()-writes an integer to a file.
vii. fprintf()-writes a set of data values to a file.
viii. scanf()-reads a set of data values from a file.
ix. fseek()-sets the position to a desired point in the file.
x. ftell()-gives the current position in the file.
xi. rewind()-sets the position to the beginning of the file.

Defining and opening a file:

If we want to store data in a file,we have to specify the

- filename
- data structure and
- purpose

Filename:

It is a stimg of characters.lt contains two parts

i. a primary name
ii. an optional period with the extension.

Data structure:

Data structure of a file is FILE. All files should be declared as type FILE before they are used.

Purpose:

When we open a file,we must specify whether to read or write a file.

The general format for declaring and opening a file is

```
FILE*fp;
fp=fopen("filename","mode");
```

- The first statement declares the variable fp as a pointer to the data type FILE.
- The second statement opens the file named filename and assings an identifier to the FILE type pointer fp.
- The second statement also specifies the purpose of opening this file.The mode does this job.

r - open the file for reading only,
w - open the file for writing only,
a - open the file for adding or appending data to it.
r+ - the existing file is opened to the beginnig for both,reading and writing.
w+ - same as w except both for reading and writing.
a+ - same as a except both for reading and writing.

When trying to open a file,one of the following things may happen:

i. When the mode is 'writing',a file is with the specified name is created if the file does not exist.The contents are deleted,if the file already exists
ii. When the purpose is 'appending',the file is opened with the current contents safe.A file with the specified name is created if the file does not exist.
iii. If the purpose is 'reading',and if it exists,then the file is opened with the curret contents safe otherwise an error occurs.

Closing a file:

A file must be closed as soon as all operations on it have been completed.

The general form is

```
fclose(file_pointer);
```

Example:

```
FILE *pl,*p2;
pl=fopen("data","w");
p2=fopen("results","r");


fclose(pl);
fclose(p2);
```

Input and Output Operations:

Once a file is opened,reading out of or writing to it is accompolishes using the standard I/O routines.There are three functions.

- The getc and putc functions
- The getw and putw functions
- The fscanf and fprintf functions,

i.The getc and putc functions:

These functions can handle one character at a time. If a file is opened in "w" mode and the file pointer is fpl,then

c=getc(fp2);

This statement would read a character from the file whose file pointer is fp2.
Example:

```
# include<stdio.h> mainQ
{
FILE*fl; char c;
printf("Data Input\n");
fl =fopen("Input","w");
while((c=getchar())!=EOF)
putc(c,fl);
fclose(fl);
printf("Data OutputVn");
fl=fopen("Input","r");
wh i le((c=getc(f I)) !=EOF)
printf("%c",c);
 fclose(fl); »
}
Output:
Data Input
This is an example to demonstrate file handling in C.
Data Output
   This is an example to demonstrate file handling in C.
```

ii. The getw and putw functions:

The getw and putw are integer oriented functions.lt is used to read and write integer values. The general forms are

putw(integer,fp);
getw(fp);

iii. The fprintf and fscanf functions:

It is similar to printf and scanf except that they work on files.They can handle group of mixed data simultaneously.

General form of printf is,

fprintf(fp,"control string",list);

Example:

fprintf(fl,"%s%d%f',name,age,7.5);

General form of scanf is,

fscanf(fp,"control string",list);

Example:

```
fscanf(f2,":%s%d",item,&quantity);
```

Error handling during I/O operations:

Error may occur during I/O operations on a file like

- trying to read beyond end-of-file.
- device overflow.
- trying to use a file that is not opened.
- opening a file with invalid file name.
- trying to perform an operation on a file,when the file is opened for another type of operation.
- attempting to write a write-protected file.

There are 2 functions to detect I/O errors in files.They are,

i. feof
ii. ferror

feof:

1. Used to test for ann end of file condition.
2. it takes file pointer as its argument
3. returns non-zero integer value if all data is read and zero otherwise.

Example:

```
if(feof(fp))
printf("end of data");
```

ferror:

i. reports the status of the file.
ii. it takes file pointer as its argument.
iii. returns a non-zero integer if an error has been detected and zero otherwise.

Random access to files:

Accessing only a particular part of a file can be achieved using

fseek()
ftell( )
rewind()

ftell() :

It gives the current position in the file.

n=ftell(fp);

i. ftell takes a filepointer as its arguments
ii. it returns a number of type long
iii. this function will be useful for saving the current position of a file

rewindQ:

This function sets the position to the beginning of the file.

rewind(fp);

it takes the filepointer as its argument and resets the position to the start of the file.

fseek():

This function is used to move the file position to a desired location within the file.

fseek(file_ptr,offset,position);

i. file_ptr is the file pointer to the file concerned.
ii. offest is a number/variables of type long,it specifies the number of position (bytes) to be moved from the location specified by position.
iii. position can take one of the following 3 values.

0 - beginning of file
1 - current position

2 - end of file

iv. if offset is positive then moves forward and if negative moves backward

v. operations on fseek function:

i. fseek(fp,ol,0); - go to the begining
ii. fseek(fp,ol,l); - stay at current position
iii. fseek(fp,ol,2); - go to the end of the file
iv. fseek(fp,m,0); - move to m+lth byte in the file
v. fseek(fp,m,l); - go forward by m bytes
vi. fseek(fp,-m,l); - go backward by m bytes from the current position.
vii. fseek(fp,-m,2); - go backward by m bytes from the end

vi. fseek returns zero if the operation is successful.-l otherwise.

Command Line Arguments:

A command line argument is a parameter supplied to a program when the program is invoked.

For example,if we want to execute a program to copy the contents of a file named x-file to another one named y-file,then we may use a command line like

c > program x-file y-file

where,

program -> is the filename where the executable code of the program is stored.

Advantages:

This would eliminate the need for the program to request the user to enter the filenames during execution.

These parameters can be supplied to the program by passing arguments to the main functions.

The main functions can take two arguments i)argc and ii)argv

i. argc is the argument counter that counts the number of arguments on the command line.The above example has 3 arguments.
ii. argv is an arguments vector and represents an array of character pointers that point to the command line arguments.The size of the array will be equal to the value of argc.

for the above example,

argv[0]- program
argv[l]- x-file argv[2]-
y-file

In order to access the command line arguments,we must declare the main functions and its parameters as follows:

In order to access the command line arguments,we must declare the main functions and its parameters as follows:

```
main(int argc,char*argv[])
```

Example:

```
# include<stdio.h>
main(int argc,char*argv[])
{
FILE* fs,* ft;
char ch;
if(argc!=3)
  {
puts("Insufficient arguments");
 exit();
  }
fs=fopen(argv[1],"r");
if(fs==NULL)
  {
```

```
puts("cannot open source file");
exit();
  }
ft=fopen (argv [2]," w");
 if(ft==NULL)
  {
puts("cannot open target file");
  exit();
  }
  }
```

UNIT-V

(Dynamic memory allocation – Linked lists- preprocessors-programming guidelines.)

DYNAMIC MEMORY ALLOCATION:

The process of allocating memory at runtime is known as dynamic memory allocation. There are four library routines called as "memory management functions" that can be used for allocating and freeing memory during program execution.

Function	Task
malloc	Allocates requested size of bytes and returns a pointer to the first byte of the allocated space.
calloc	Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
Free	Frees previously allocated space.
realloc	Modifies the size of previously allocated space.

1. Allocating a block of memory: **malloc**

A block of memory may be allocated using the function malloc. The malloc function reserves a block of memory of specified size and returns a pointer of type void.

Syntax: **ptr =(cast-tyye*)malloc(byte-size);**

Ptr is a pointer of type cast-type. The malloc returns a pointer to an area of memory with size byte-size.

Example: X=(int *) malloc (100 x sizeof(int));

A memory space equivalent to 100 times the size of an int bytes is reserved and the address of the first byte of the memory all ocated is assigned to the pointer X of type int.

2. Allocating multiple blocks of memory: **calloc**

Calloc is a memory allocation function that is used for requesting memory space at runtime for storing derived data types such as arrays and structures. Calloc allocates multiple blocks of storage, each of the same size, and then sets all bytes to zero.

Syntax: **ptr =(cast-tyye*)calloc(n, elem-size);**

This allocates continous space for n blocks, each of size elem-size bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned.If there is not enough space, a NULL pointer is returned.

3. Releasing the used space: **free**

When the data is stored in a block of memory is no longer needed , we can release that memory for future use , using thee free function.

Syntax: **free(ptr);**

Ptr is a pointer to a memory block, which has already been created by malloc and calloc.

4. Altering the size of the block: **realloc**

The memory size already allocated can be changed with the help of realloc function. Memory size can be reduced or extended using this function. For example if the original allocation is done by the statement **ptr=malloc(size);** then reallocation of space may be done by the statement

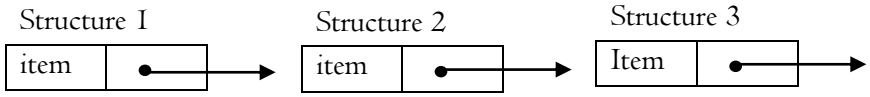
ptr=realloc (ptr,newsize);

This function allocates a new memory space of size newsize to the pointer variable ptr and returns a pointer to the first byte of the new memory block.The newsize may be longer or smaller than the size.

LINKED LISTS:

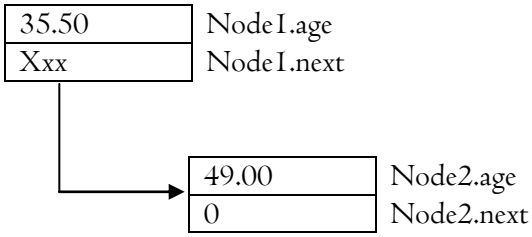
List is a set of items organized sequentially.Eg:-Arrays. In arrays we use the index for accessing and manipulation of array elements. But the major problem with the arrays is that , the size of the array must be specified precisely at the beginning.

Linked list is a dynamic data structure. Each item in the list is a part of a structure and also contains a "link" to the structure containing the next item. This type of list is called a linked list because it is a list whose order is given by links from one item to the next.



Each structure of the list is called a node and consists of two fields, one containing the item and the other containing the address of the next item in the list.

Example:



Example:

```
struct link_list
{
float age;
struct link_list *next;
};
struct  link_list node1, node2;
node1.next = &node2;
node1.age=35.50;
node2.age=49.00;
node2.next=0;
```

Advantages of Linked List:

1. Linked Lists can grow or shrink in size during the execution of a program.
2. It does not waste memory space.
3. It provides flexibility by allowing the items to be rearranged efficiently.

4. It is easier to insert or delete items by rearranging the links

Disadvantages:

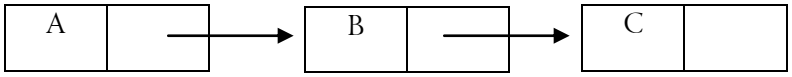
1. Accessing an arbitrary item is time consuming.
2. It uses more storage than an array with the same number of items.

Types of linked lists:

The following are the different types of linked list.

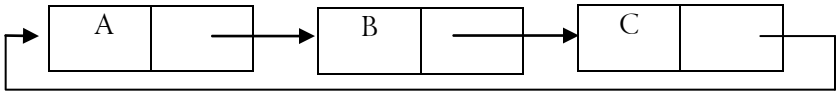
i. Linear singly linked list.
ii. Circular linked list.
iii. Two-way or Doubly linked list.
iv. Circular Doubly linked list.

Linear singly linked list.



Circular linked list.

It has no beginning and no end. The last item points back to the first item.

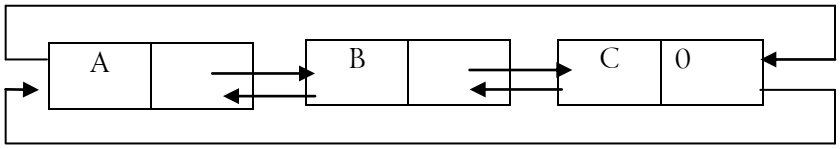


Doubly linked list.

It uses double set of pointers, one pointing to the next item and other pointing to the preceding item. This allows us to traverse the list in both directions.

Circular Doubly linked list.

It employs both the forward pointer and backward pointer in circular form.



GUIDELINES FOR DEVELOPING A C PROGRAM:

The program development process includes three stages:

1.Program design
2. Program Coding
3. Program Testing

Program Design:

Before coding a program the following steps should be followed.

a. program analysis
b. outlining the program structure
c. algorithm development

d. Selection of control structure.

Program Coding:

The algorithm developed during program design must be translated into a set of instructions that a computer can understand. The program created should be readable and simple to understand. Complex logic and tricky coding should be avoided.

Program Testing and Debugging:

Testing and debugging refers to the task of detecting and removing errors in a program, so that the program produces desired result.

TYPES OF ERRORS:

Errors can be classified under four types: Syntax errors, runtime errors, logical errors and latent errors.

a. Syntax Errors: Any violation of the rules of the language results in syntax errors. The compiler can detect and isolate such errors. When syntax errors are present, the compilation fails and is terminated after listing the errors and the line numbers in the source program.
b. Run-time Errors: Errors such as mismatch of data types or referencing an out of-range array element go undetected by the compiler.A program with these mistakes will run, but produces no erroneous results.
c. Logical Errors: These errors are related to the logic of the program execution like taking a wrong path, failure to consider a particular condition, incorrect order of evaluation etc.,
d. Latent Errors: It is a hidden error that shows up only when a particular set of data is used.

COMMON PROGRAMMING ERRORS:

The following are some of the common mistakes while coding a program.

i. missing semicolon: every C statement must end with a semicolon.
ii. misuse of semicolon : eg/- for(i=1;i<10;i++)**;** while(x<10)**;**
iii. use of = instead of = =
iv. missing braces with multiple statements.
v. missing quotes for string and characters.
vi. improper comment characters.Nesting of comments is not allowed.
vii. undeclared variables.
viii. forgetting the precedence of operators.
ix. forgetting to declare function parameters.
x. mismatching of actual and formal parameter types in function calls.
xi. non-declaration of function.
xii. missing **&** in scanf parameters.
xiii. crossing the bounds of array.
xiv. forgetting a space for null characters in a string.
xv. using uninitialized pointers.

PREPROCESSORS:

The preprocessor is a program that modifies the C source program according to directives supplied in the program. An original source program is stored in a file.The preprocessor does not

modify this program file, but creates a new file that contains the processed version of the program.This new file is then submitted to the compiler.

Preprocessor does the following actions.

1. replacement of defined identifiers by macros.
2. inclusion of other files.
3. renumbering and renaming of source files.

The general rules for defining a preprocessor are

1. all preprocessor directives begin with **#.**
2. they must start in the first column and no space between # and the directive.
3. the preprocessor is not terminated by a semicolon.
4. only one preprocessor directive can occur in a line.
5. it may appear in any place in the source file, outside functions, inside functions or inside compound statements.

The common preprocessor directives and their uses are:

S.No.	DIRECTIVE	USES
1.	#include	Inserts text from another file.
2	#define	Defines preprocessor macros.
3	#undef	Removes macro definitions.
4	#if	Conditionally include some text, based on the value of the constant expression.
5	#error	Terminate processing easily.
6	#ifdef	Conditionally include some text based on whether a macro name is defined.
7	#else	Alternatively include some text.
8	#endif	Combination of #if and #else.
9	#line	Gives a line number for compiler message.

MACROS:

The macro definition is the use of the #define directive to define constants in a C program. The macros can be classified into two groups. They are

a. Simple macro definition
b. macro definition with arguments.

The advantages of using macros are;

1. Easy to read and write
2. Easy to check string constants.
3. Easy to transfer from one machine to another
4. gives good look to the program.

Simple macro definition:

A macro is simply a substitution string that is placed in the program.

Eg:-
```
#define MAX 100
 main()
 {
 char name[ MAX];
for (i=0;i<MAX-1;i++)
 …….
}
```
The MAX is internally replaced with the following program.
```
main()
 {
 char name[100];
for (i=0;i<100-1;i++)
 …….
}
```
Each occurrence of the identifier MAX as a token is replaced with the string 100 that follows the identifier in #define line.

MACRO WITH PARAMETERS:

The more complex form of macro definition declares the names of formal parameters within paranthesis, separated by commas.

Syntax: **#define Name(var1,var2,…var n) substitution string**

Example: #define PRODUCT (a,b) ((a) x (b))

#define MIN(a,b) ((a)<(b) ?(a): (b))