

1. Algorithms , Flow chart and Evolution of Programming Languages :

1.1 Algorithm:

An algorithm is a finite step by step list of well defined instructions to solve a particular task. It is a sequence of steps that describe the procedure of a task. It takes one or more inputs and produces one or more output. For example :

To find the area of a rectangle by taking its width and length, the algorithm is as follows:

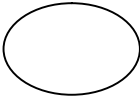
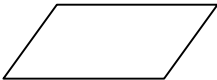
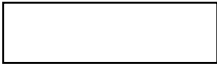
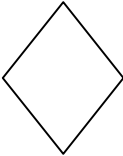

Step1: input W and L
Step2: $A = W * L$
Step3: print A
Step4: exit

Example1: Write the algorithm to find the sum of two numbers.

Step1: input num1 and num2
Step2: $sum = num1 + num2$
Step3: print sum
Step4: exit

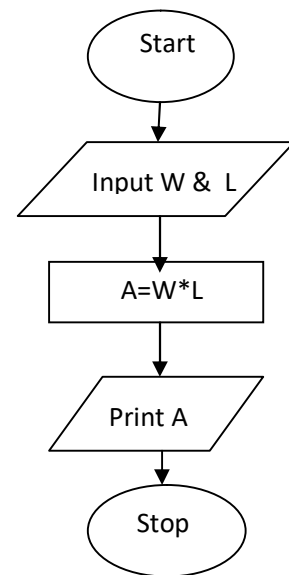
1.2 Flowchart:

It's the graphical representation of an algorithm. It uses the following symbols:

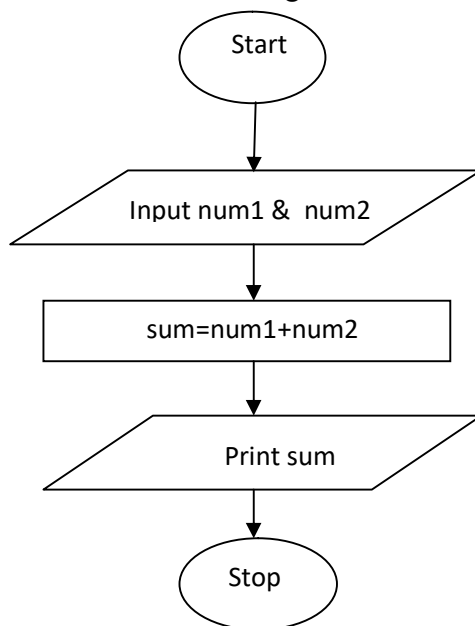
<u>Symbols</u>	<u>Operations</u>
	Start/Stop
	Input/Output
	Calculation Step
	Decision Step
	Control Flow

Example1:

Draw the flow chart to find out the area of a rectangle by taking its width and length.



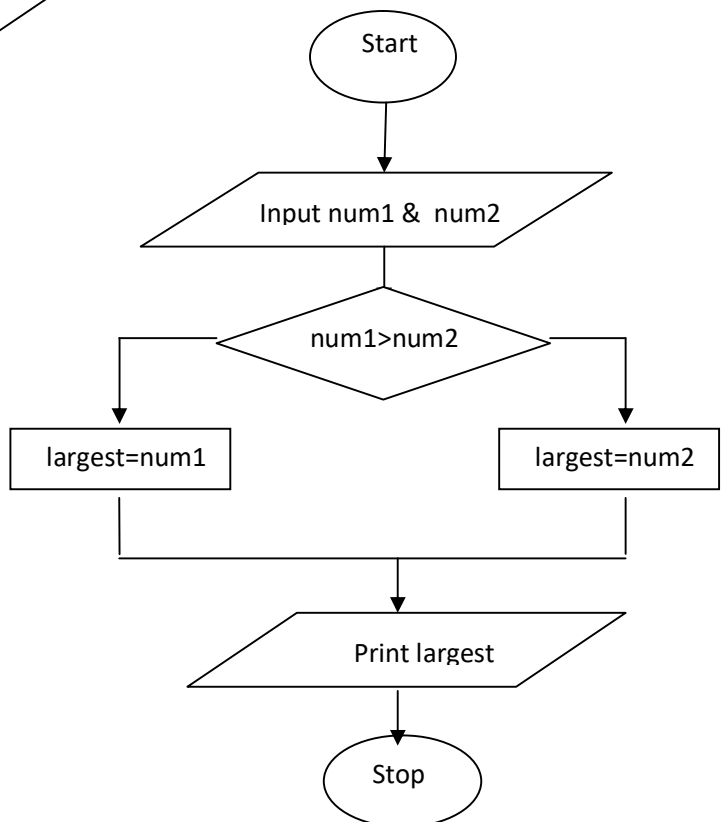
Example2: Draw a flowchart for finding the sum of two numbers.



Example3 :

Write an algorithm to read two numbers and determine the largest number. Draw the flow chart for this algorithm.

Algorithm: Step1: input num1 and num2
Step2: if num1 > num2
 Largest=num1
 Else
 Largest=num2
Step3: print largest



1.3 Evolution Of Programming Language:

1.3.1 Machine Language:

It consist binary number 0's and 1's only. It is readable by computer. It's the 1st generation language.

1.3.2 Assembly Language:

It's the 2nd generation language which consists English like phrases called mnemonics e.g. ADD is used for addition. Assembly Language is not readable by computer, so assembler is used to convert from Assembly Language to Machine Language. Assembler is a translator program used to convert assembly language to machine language

1.3.3 High level Language:

It's a 3rd generation language. Languages like C, C++, FORTRAN, COBOL, BASIC, PASCAL, Java etc comes under it. High level Language is also not readable by computer, so compiler is used to convert from High level Language to Assembly Language.

2. Structure of a C Program :

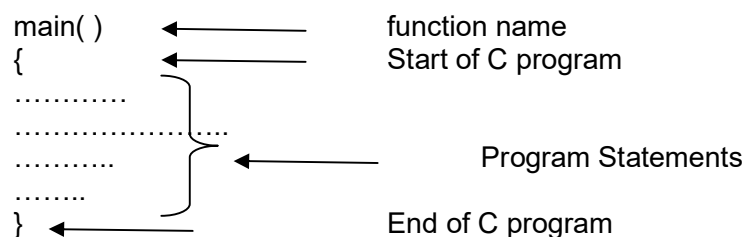
2. Introduction to C:

C is a programming language developed at AT & T's Bell Laboratories of USA in 1972. It was written by Brian Kernighan and Denis Ritchie. This was to be the 1st language that Unix was written to be the first portable language. It evolved from two previous language BCPL and B. BCPL was used for writing OS and compiler and developed in 1967 by Martin Richards used for creating early versions of UNIX OS by Ken Thompson in 1970. Both were type less(No data type) language .

Dialects of C:

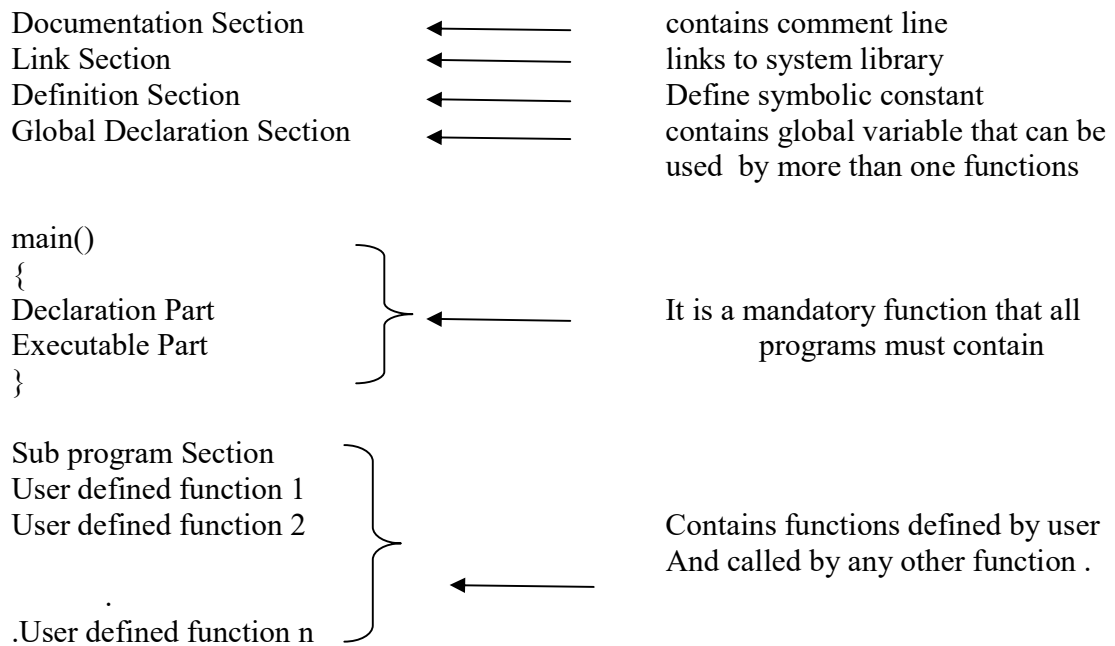
1. Common C
2. ANSI C (American National Standard Institute)

2.1 Structure of a Simple C program:



The main function is the part of every C program. '()' is the symbol of a function that distinguishes from other names. This symbol () is used to pass input to the function. Here it is not taking any input. We may specify void also inside the () .

Overview Of a C program:

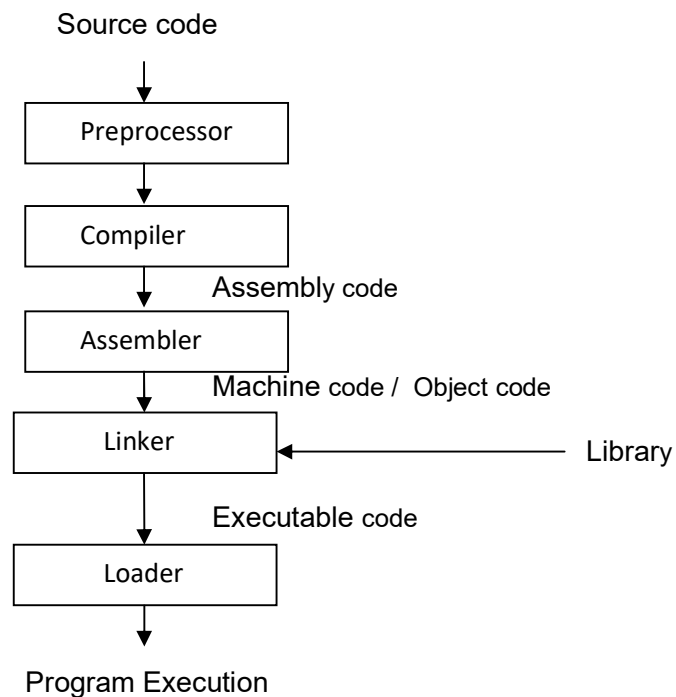


Example : A Simple C program

```
/* First C program to print Hello World */
#include<stdio.h>
void main( )
{
    printf("Hello World") ;
}
```

OUTPUT : The above program will print Hello World on the screen.

2.2 Compilation Process



2.2.1 Preprocessor: The main job of a C preprocessor is to remove comments from the source code and interpret preprocessing statement directly which starts with #. The statement can be either called with `#include<file.h>`, where it searches for file in standard files included header file directory. Sometimes it can be called with `#include"file.h"` where it searches the file to be included in global directory.

2.2.2 Compiler: A compiler is a convector program which converts High level Language to Assembly Language.

2.2.3 Assembler: It's a translator program used to convert assembly language to machine language.

2.2.4 Linker: A linker links other precompiled object file or library and creates the executable file.

Loader: A loader is a program which loads the executable program into primary memory called RAM.

<u>Compiler</u>	<u>Vs</u>	<u>Interpreter</u>
i. Reads entire program and converts it to object code.		i. reads source program one line at a time and execute the specific instruction contained in that line.
ii. Faster execution		ii. Execution is slower than compiler
iii. It is a onetime cost (less overhead)		iii. indicates an overhead each time a program will run

3. Character Set , Keyword , Constants and Variables :

3.1 C character set:

Alphabets : A to Z ,a to z
Digits: 0 to 9
Special Smbols :- ~ ‘ ! @ # % ^ & * () _ - + = | \ { }
 [] : ; " ' < > , . ? /

3.2 Keywords :

Keywords are the reserve words whose meaning has already been defined to the C compiler. These are reserved for doing a specific task. There are 32 keywords in C compiler.

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

3.3 Identifiers:

These are user defined words which are used to give names to variable, function, array and structure.

3.4 Constants:

A constant is an entity whose value doesn't change during the program execution. C constants can be divided into two major categories:

1. Primary Constants
2. Secondary Constants

Primary constants: Integer, Real/Float, Character, String

Secondary Constants: Array, Pointer, Structure, Union, Enum

At this stage we would restrict our discussion to only Primary Constants, namely, Integer, Real and Character constants.

3.4.1 Integer Constants:

Numbers without decimal point are integers. Rules for Constructing Integer Constants :

1. It must not have a decimal point.
2. It can be either positive or negative.
3. If no sign precedes an integer constant it is assumed to be positive.
4. No commas or blanks are allowed within an integer constant.

Example:

465
-323

3.4.2 Real Constants:

Numbers having decimal point are real/float constants. It could be written in two forms.

1. Fractional form
2. Exponential form

3.4.2.1 fractional real Constants:

Rules for constructing fractional real Constants:

1. A real constant must have at least one digit.
2. It must have a decimal point.
3. It could be either positive or negative.
4. Default sign is positive.
5. No commas or blanks are allowed within a real constant

Example :

325.5
-465.6

3.4.2.2 exponential real constants :

It is usually used if the value of the constant is either too small or too large. Here real constant is represented in two parts, the part appearing before 'e' is called mantissa and the part following 'e' is called exponent

Mantissa e^{exponent}

Rules for constructing fractional real Constants:

1. The mantissa part and the exponential part should be separated by a letter e.
2. The mantissa part may have a positive or negative sign.
3. Default sign of mantissa part is positive.

4. The exponent must have at least one digit, which must be a positive or negative integer. Default sign is positive.

Example :

3.2 e 5
-4.5 e -7

3.4.3 Character Constants:

A character constant is a single character from the C character set that enclosed within single inverted commas.

Example :

'A'
'\$'
'5'
'='

3.5 Variables:

Variable names are the names given to the locations in the memory. These locations can contain integer, real or character constants. In other word, a variable is a named container which contains constant values.

Rules for Constructing Variable Names:

1. A variable name is any combination of 1 to 31 alphabets, digits or underscores.
2. The first character in the variable name must be an alphabet or underscore.
3. No commas or blanks are allowed within a variable name.
4. No special symbol other than an underscore is allowed.

Example :

_sal
Si_int
Area
Num

4. Datatype , Block , Statement , Expression :

4.1 Datatypes:

A datatype is a set of values and set of operations that can be applied on those values. It is used to specify the type of values a variable can contain .It also specify the range of values a variable can take. Some basic data type int, float, char.

If a variable is required to contain an integer value then it should be specify to be 'int' type. Similarly if a variable is required to contain a floating point value then it should be specify to be 'float' type. It is similarly the way we specify at our lunch preparation , the bowl should contain Dal (Indian) and the plate should contain Rice(Indian).

Specifying the type of data a variable can contain is called declaration of a variable. The rule (syntax) to declare a variable is as follows.

Syntax:
datatype variable_list ;

Here datatype can be any one from the list of datatype and the variable list can be any number of identifier separated by comma (,).

Example1:

int x ; Here x is called an integer variable which can store integer values.

Example2:

int x , y , z; Here x, y, z are integer variables which can store integer values.

4.1.1 Range of the datatype:

A variable can contain only one value but the value must belong to a specific set of values which is called the range of the datatype. The range of datatype depends on the predefined size of it. The size of a datatype determines the no of bytes a variable occupies in memory.

There are two types of integer variable signed integer (contains both +ve and -ve numbers) and unsigned integer (contains only +ve numbers) .Hence the number range change accordingly. Here signed and unsigned are called qualifiers. Here int, char, float, signed, unsigned are belongs to the 32 keywords in C.

The range of predefined datatype can be increased or decreased by mentioning two quantifiers long and short respectively. If int takes 4 bytes then short int will take 2 byte and long int will take 8 bytes.

4.1.2 Range calculation of a variable :

The range of a variable is calculated from its size. Let's declare an integer variable x as follows.

int x;

Here x is a signed integer. The default sign qualifier of a variable is signed. Its range is -2^{32} to $2^{32} - 1$. Here 16 is the no of bits the variable occupy. To calculate the range of integer and character datatype , we use the following formula.

-2^{n-1} to $2^{n-1} - 1$ where n is the number of bits occupied by the variable.

Example: The range of a short integer is -2^{15} to $2^{15} - 1$

The size and range of all types of data type is given in the following table.

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295
Type	Storage size	Value range
float	4 byte	1.2E-38 to 3.4E+38
double	8 byte	2.3E-308 to 1.7E+308
long double	10 byte	3.4E-4932 to 1.1E+4932

4.2 Statement:

A statement is an instruction to perform a task in a program. Therefore a complete C program comprises a series of statements. All the statements in C program must be written in small case letter. All statements must end with a semicolon (;).

There are different types of instruction as follows.

- i) Type Declaration statement (use to declare the type of variables)
- ii) Arithmetic Statement (Use to perform arithmetic operation)
- iii) Control Statement (Use to control the flow of execution)

4.3 Block:

A block consist of several statements enclosed within a pair of braces { }.

```
{
    .....
    .....
    .....
}
```

If there is a single statement in a block, a pair of braces { } is not required.

4.4 Expression: An expression is a collection of operators and operands.

4.4.1 Operators: Operators are used to perform any operation on data. These are used as tools of a programmer through which the task can be accomplished.

4.4.2 Operands: The data on which the operator is to be applied is called operands.

Based on the number of operands operators are classified as follows

- i. Unary Operators (operates on single operands)
- ii. Binary Operators (operates on two operands)
- iii. Ternary Operators (operates on three operands)

Example :

```
int a;  
a=a+5;  
a=a+a  
a=4+5
```

5. Operators in C (Arithmetic , Relational, Logical, Assignment, Conditional) :

5. Operators:

Operators are used to perform any operation on data. These are used as tools of a programmer through which the task can be accomplished. The data on which the operator is to be applied is called operands. In C language there are several types of Operators, which are mentioned below.

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Conditional Operators
7. Increment / Decrement Operators
8. sizeof Operators
9. Comma Operators

5.1 Arithmetic Operators:

There are five arithmetic operators +, -, *, /, %(modulus).The modulus operator is used to find the remainder of a division operation.

5.1.1 Precedence and Associativity Of operators:

Precedence is the priority level of an operator. When there are more than one operators in an expression, then the operation is to be given highest priority is determined as per the precedence of operators. More than one operators can have the same precedence also. When there are operators having same precedence in an expression, then the operation is to be given highest priority is decided as per the Associativity of operators. There are two types of Associativity

- i. Left to right (operators comes from left first is evaluated first)
- ii. Right to left (operators comes from right first is evaluated first)

The precedence and Associativity of these five operators are given below.

Precedence	operators	associativity
1	% / *	L to R
2	+ -	L to R

These operators are used mostly in arithmetic statement.

5.1.2 Arithmetic statement:

A C arithmetic instruction consists of a variable name on the left hand side of = and variable names & constants on the right hand side of =. The variables and constants appearing on the right hand side of = are connected by arithmetic operators.

5.1.3 Assignment operator (=) :

The operator is used to assign a value to a variable. The syntax of assignment statement is as follows. Assignment can be done anywhere in a program. But assign of value to a variable is done at declaration or after declaration of the variable.

Syntax:

<Variable name> = <constant>

Example:

i. `int x ;` `}` assignment after declaration
 `x = 5 ;`

We can assign more than one variable at time as follows.

`int a, b, c, d ;` `}` assignment after declaration
 `a = b = c = 10 ;`

ii. `int x = 5 ;`

We can assign more than one variable at time in declaration as follows.

```
int a = b = c = d = 10 ;           // assignment at declaration
```

For character variable assignment, the character should be enclosed within single quote as follows.

```
char ch1 = 'a' , ch2;  
ch2 = 'b';
```

Examples of Arithmetic Expression:

- i.

```
int x ;  
x = 5 + 3 ;
```

 Here value of x is 8
- ii.

```
int x = 5, y = 6 , z;  
z = x + y ;
```

 Here value of z is 11
- iii.

```
int x = 5;  
float y = 5.5 , z ;  
z = x + y ;
```

 Here value of z is 10.5

5.1.4 Types of Arithmetic statement:

3 types of Arithmetic statement are there:

- 1. Integer Mode – All operands are of integer type
- 2. Real Mode -- All operands are of real type
- 3. Mixed Mode -- Operands are mix of int and real

Each character has corresponding ASCII (American Standard code for information Interchange) value, which is an integer value. So character operands are considered as integer operand. We can perform arithmetic operation on character values also.

Example:

```
char ch1 , ch2;  
ch1 = 'A';  
ch2 = ch1 + 3;
```

Here value of ch2 is D in char and 68 in integer.

In a mixed mode arithmetic, one type is converted to another which is discussed in later section.

5.2 Relational Operators:

Relational Operators are used to compare two expressions which give either TRUE (1) or FALSE (0) value. There are six relational operators in C. They are < , > , <= , >= , == , != . Operator == is called 'equal to' operator which is used to compare the equality of two expression.

Example:

```
int x=5 , y=6, z;  
z=x<y;
```

Here the value of Z is 1 as x<y is true.

```
z = x==5;
```

Here the value of Z is 1 as x==5 is true.

All relational operators have lower precedence from arithmetic operators.

5.3 Logical Operators:

Logical operators are used to combine the true or false value of expressions. There are three logical operators && , ||, !.

5.3.1 And operator (&&) :

x= expr1 && expr2

If anyone expression either expr1 or expr2 expression has false value then x is false. So if expr1 has false value then expr2 will not be evaluated.

5.3.2 OR Operator (||) :

x= expr1 || expr2

If anyone expression either expr1 or expr2 expression has true value then x is true. So if expr1 has true value then expr2 will not be evaluated.

5.3.3 Not Operator (!):

x= ! expr1

If expr1 has false value then expr2 is true and If expr1 has true value then x is false.

Except the! (NOT) operator both && and || have lower precedence from Logical operators.

5.4 Increment operator ++ and Decrement operator -- :

5.4.1 Increment Operator (++) :

It is an unary Operator. It increases the operand value by one. Two variants are there postfix and prefix. In case of postfix the operator is placed after the operand, where the value of operand is assigned to the LHS variable first then the operand is incremented. . In case of prefix the operator is placed before the operand, where the value of operand is incremented first then the incremented operand is assigned to the LHS variable.

Example:

int x=5 , y ;	<div style="border: 1px solid black; padding: 2px; display: inline-block;">X 5</div>
x++ ;	<div style="border: 1px solid black; padding: 2px; display: inline-block;">6</div>

Here the value of x is changed to 6.

	x	y
y = x++;	7	6

Here value of x is assigned to y first then x is increased by one.

	x
++x ;	8

Here the value of x is changed to 8.

	x	y
y = ++x ;	9	9

Here value of x is increased by one first then x is assigned to y.

5.4.2 Decrement Operator (- -) :

It is similar to the ++ Operator only difference is that It decreases the operand value by one. Two variants are there postfix and prefix. In case of postfix the operator is placed after the operand, where the value of operand is assigned to the LHS variable first then the operand is decremented. In case of prefix the operator is placed before the operand, where the value of operand is decremented first then the incremented operand is assigned to the LHS variable.

Example:

	x
int x = 5, y ;	5
x - - ;	4

Here the value of x is changed to 4.

	x	y
y = x - -;	3	4

Here value of x is assigned to y first then x is decreased by one. So x becomes 3 .

	x
- -x ;	2

Here the value of x is changed to 2.

	x	y
y = - - x	1	1

++ and - - operator are unary operators. They have right to left Associativity and have higher precedence over arithmetic operators.

5.5 Conditional Operators (- ? - : -) :

It is a ternary operator. Its format is as follows.

expr1 ? expr2 : expr3

The whole expression can give a value which can be assigned to a variable.

x= expr1 ? expr2 : expr3

The value of x depends on the true and false value of expr1. If expr1 value is true then x will have the value of expr2. If expr1 value is false then x will have the value of expr3.

Example:

```
int x=2, y=3, z;  
z= x>y ? x : y;
```

Here z value is 3 as x>y is false. We can assign the value to z in the true and false part of conditional operator.

x>y? z=x : z=y ;

6. Operators in C (Bitwise, Sizeof, Comma) :

6.1 Bitwise operators:

These operators work on a bit by bit basis on the operands. The operators are follows.

Op	Operator Name	Rule	Type
&	AND	Anyone 0 is 0	Binary
	OR	Anyone 1 is 1	Binary
~	NOT	If 1 then 0 ,If 0 then 1	Unary
^	Ex-OR	If both are 0 or both are 1 then 0 else 1	Binary
>>	Right shift	rightmost digit was shifted out and a new 0 was shifted into the rightmost position	Binary
<<	Left shift	leftmost digit was shifted out and a new 0 was shifted into the rightmost position	Binary

Example:

6.1.1 And operator (&) :

```
int x = 5, y = 4, z;  
z = x & y;
```

```
5=    101   in binary  
4=    100   in binary  
&
```

100 in binary whose decimal equivalent is 4

The value of z is 4.

6.1.2 Or operator (|):

```
int x = 5, y = 4, z;  
z = x | y;
```

```
5=    101   in binary  
4=    100   in binary  
|
```

101 in binary whose decimal equivalent is 5

The value of z is 5.

6.1.3 EX-OR operator (^):

```
int x = 5, y = 4, z;  
z = x ^ y;
```

```
5=    101   in binary  
4=    100   in binary  
^
```

001 in binary whose decimal equivalent is 1

The value of z is 1.

6.1.4 NOT operator (~) : Its an unary operator.

```
int x = 5, y;  
y = ~ x;
```

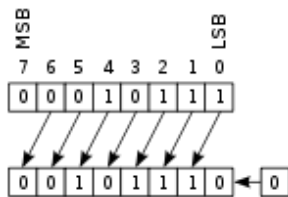
5= 101 in binary
 ~

 010 in binary whose decimal equivalent is 2

The value of z is 2.

6.1.4 left shift operator(<<):

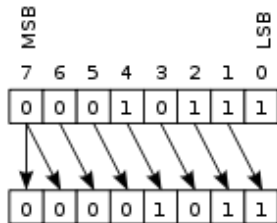
```
int x = 23, y = 1, z;  
z = x << y;
```



The value of z is 46. Its equivalent to multiplying x by 2 for y times.

6.1.5 right shift operator(>>):

```
int x = 23, y = 1, z;  
z = x >> y;
```



The value of z is 11. Its equivalent to dividing x by 2 for y times.

6.2. sizeof Operator (sizeof (datatype)):

It is a unary operator. It returns the size of its operand in terms of byte. The operand can be a variable, constant or any datatype.

Example :

```
int x = 5, y;  
y = sizeof(x);
```

Here the value of y is 4 (Since x is an int variable , its memory requirement is 4 bytes).

6.3 Comma Operator:

It has the Lowest Precedence than all operators. It returns the value of the rightmost operand when multiple comma operators are used inside an expression. Comma Operator Can acts as –

1. **Operator** : In the Expression
2. **Separator** : Declaring Variable , In Function Call Parameter List

Example:

```
int x;  
x= 1,2,3;
```

Here value of x is 1 as Assignment statement will be executed first.

```
int x;  
x=(1,2,3);
```

Here value of x is 3 as Bracket has highest priority and Associativity from Left to Right and return Right most operand.

6.4 Compound Assignment Operators:

Compound assignment operators provide a shorter syntax for assigning the result of an arithmetic or bitwise operator. They perform the operation on the two operands before assigning the result to the first operand. There are eleven compound assignment operators which are as follows.

Operators	Equivalent to
y += x	y = y+x
y -= x	y = y-x
y *= x	y = y*x
y /= x	y = y/x
y %= x	y = y%x
y &= x	y = y&x
y = x	y = y x
y ^= x	y = y^x
y <<= x	y = y<<x
y >>= x	y = y>>x

6.5 Typecast operator:

It is used to convert the datatype of an operand from one type to another type explicitly by the programmer. It is an unary operator. Its format is as follows.

Format: (datatype) operand.

6.6 Precedence and Associativity Of operators:

Precedence is the priority level of an operator. When there are more than one operator in an expression, then the operation is to be given highest priority, is determined as per the precedence of operators. More than one operators can have the same precedence. When there are operators having same precedence in an expression, then the operation is to be given highest priority is decided as per the Associativity of operators. There are two types of Associativity

- i. Left to right (operators comes from left first is evaluated first)
- ii. Right to left (operators comes from right first is evaluated first)

The precedence and Associativity of all operators are given below.

Operator	Description	Associativity
()	Parentheses (function call) (see Note 1)	left-to-right
[]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	Postfix increment/decrement (see Note 2)	
++ --	Prefix increment/decrement	right-to-left
+	Unary plus/minus	
!	Logical negation/bitwise complement	
(type)	Cast (convert value to temporary value of type)	
*	Dereference	
&	Address (of operand)	
sizeof	Determine size in bytes on this implementation	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= =	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right

7. Type Conversion , I/O Statements :

7.1 TYPE CONVERSION :

It is used to convert the datatype of an operand from one type to another type. It is of two types : 1. Implicit type conversion.

2. Explicit type conversion.

Implicit type conversion :

Type conversion that automatically performed by the compiler is called as implicit type conversion

When a C-expression is evaluated, the resulting value has particular data type. If all the operands in the expression are of same type, the resulting type is of same type as well. If both x and y are of int type, the expression x+y is of int type. If the operands of an expression is of different type then type of expression is of largest size data type present in the expression.

Let a is int type, b is float type variable .

Evaluating expression a/b causes a to converted float type before expression is evaluated. This does not mean that type of variable is changed. The value of expression is float type. This type of automatic conversion is known as implicit type conversion. If operands are of different types the lower type is automatically converted to higher type before the operation takes place. This type of conversion is known as implicit type conversion.

Example :

```
char c;  
int j;  
float f;  
double d, r;  
r= (c * j) + (f / j) - (f + d) ;
```

In the above the above example before c is multiplied with j first c implicitly converted to integer . Similarly j implicitly converted to float before division with j. Similarly f implicitly converted to double before addition.

But in assignment expression, the type of expression is the type of left hand side operator. If in the assignment expression two operator may not be of same type, In such case value of right hand side expression is promoted or demoted depending on type of variable on left hand side of =.

```
int i, j;  
float f;  
i=12;  
j=5;  
f= i / j; // f=2.000000
```

Thus while storing the value of an expression always takes type of variable into which its value is being stored.

Example : WAP to covert the temperature in centigrade scale to farahenite scale .

```
main( )
{
    float ctemp, ftemp;
    printf ("Enter temperature in centigrade");
    scanf ("%f", &ctemp);
    ftemp= 9.0 / 5 * ctemp +32; // 9/5=1 not 1.8 but 9.0/5=1.8
    printf ("temperature in farenahite=%f",ftemp);
}
o/p-
Enter temperature in centigrade 70
temperature in farenhite=158
```

EXPLICIT TYPE CONVERSION:

Type conversion also can be done by programmer not by compiler. If the type conversion is done by programmer is called explicit type conversion. To convert an expression to a particular data type expression must be preceded by the name of desired data type enclosed in parenthesis i.e

Syntax : (datatype) expression ;

```
float f=5.5;
((int)f)%2;
float a=7,b=4;
(a+b)%2 ; // (wrong) because % operator takes only integer arguments
((int)(a+b))%2; // (right)
```

Example : WAP to print sum and average of 2 numbers

```
main( )
{
    int input1,input2,sum;
    float avg;
    printf ("Enter two number");
    scanf ("%d%d",&input1,&input2);
    sum= input1+input2;
    avg= ((float) (a+b)) / 2;
    printf ("Sum=%d\n average=%f",sum,avg);
}
o/p-
Enter two number 3 4
Sum=7
Average=3.5
```

7.2 Input output statement

printf () and scanf () are library functions of standard input output header file (stdio.h)

Syntax of printf() : **printf(“format string”, arg1,arg2,.....argn);**

Format string can contain text as well as format specifier. For example %d is the format specifier for integer datatype. Similarly %f , %c for float and char data type respectively.

Example :

```
#include<stdio.h>
main( )
{
    float l,w,area,peri; //variable declaration
    l=5.5; //initialization
    w=2.5; //initialization
    area=l*w;
    peri=2*(l+w);
    printf(“AREA=%f \t PERIMETER=%f ”,area,peri);
}
```

o/p-

AREA=13.75 PERIMETER=16

Syntax for scanf() : **scanf(“format string”, arg1,arg2,.....argn);**

In each argument variable name must be preceded by &

& ->address operator

&l -> address of variable l

Example :

```
#include<stdio.h>
main( )
{
    float l,w,area,peri; //variable declaration
    printf(“\n Enter length of rectangle”);
    scanf(“%f",&l);
    printf(“\n Enter width of the rectangle”);
    scanf(“%d",&w);
    area=l*w;
    peri=2*(l+w);
    printf(“AREA=%f \t PERIMETER=%f ”,area,peri);
}
```

o/p-

Enter length of the rectangle

5.5

Enter width of the rectangle

2.5

AREA=13.75 PERIMETER=16

8. Decision Making : if and if-else :

8.1 if statement :

There are 2 types of control statements in c.

1. Branching or decision making
2. Looping or iteration or repetition

Branching:

Branching can be achieved by if-else statement and switch statement.

There are different ways of writing if –else statement

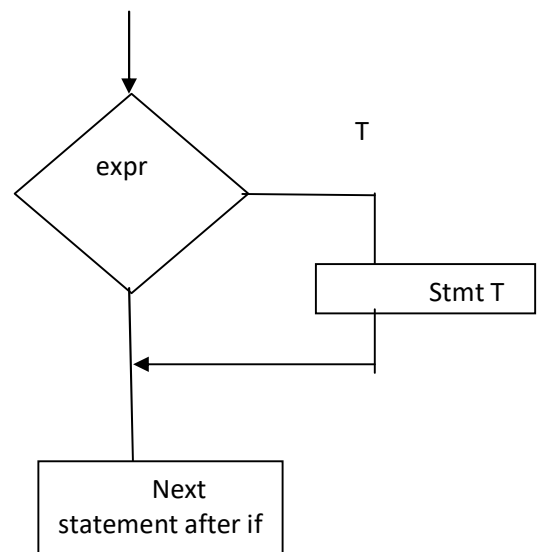
1. if statement
2. if-else statement
3. nested if statement
4. if-else-if statement (else-if ladder)

if statement (one way decision) :

syntax of if statement :

- 1) **if(expr)**
stmt T;
- 2) **if(expr)**
{
 stmt T1;
 stmt T2;

 stmt Tn;
}



The expression following keyword if must be enclosed in parenthesis. If the value of the expression expr is true (that is non zero) statement in if block (stmt T) will be executed. If the value of the expression expr is false (that is zero) then statement in if block (stmt T) will be ignored. The stmt T can be simple or compound statement. If stmt T is compound statement then it must be enclosed within parenthesis. Next statement will be executed if the condition is true or false.

Example : WAP to display a number if it is divisible by 3

```
#include<stdio.h>
int main( )
{
    int input;
    printf("Enter a number ");
    scanf("%d",&input);
    if (input%3==0)                //no semicolon
        printf("%d is divisible by 3\n",input);
    printf("Thanks for using this program");    //next statement after if block
}
```

o/p-
Enter a number 6
3 is divisible by 3
Thanks for using this program

Example : Find the output of the following code

```
main( )
{
    int input;
    printf("Enter a number\n");
    scanf("%d",&input);
    if(input==5) ;                // no error
        printf("\n you have entered 5");
}
```

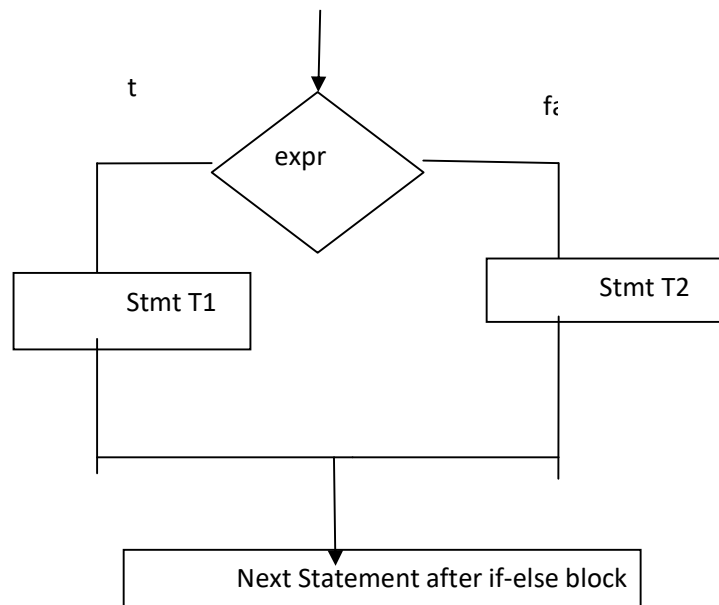
o/p-
Enter a number 10
you have entered 5

Explanation

Irrespective of whether condition is true or false , 2nd printf statement is bound to execute. Here the condition of the if statement is true. So null statement (;) gets executed which does nothing on execution.

8.2 if -else statement (two way decision) :

If- else statement can execute one group of statements. When the if condition is true, a block of statements are executed and the else part is totally skipped. When if the condition is false, if part is skipped and else part will be executed.



Syntax of if-else statement :

```

if (expr)
    stmt T1 ;
else
    stmt T2 ;
  
```

If the expr has non zero value (expr is true) then stmt T1 will be executed otherwise (expr is false) stmt T2 will be executed. Do not give semicolon at the end of if () or else .

Example : WAP to find whether number is even or odd

```

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

    if (input%2==0) //no semicolon
        printf("%d is even number",input);
    else //no semicolon
        printf("%d is odd number",input);
    printf("Thanks for using this program"); //next statement after if-else block
}
  
```

OUTPUT:

```
Enter a number
5
5 is odd number
Thanks for using this program
```

Example :- WAP to calculate the gross salary of an employee. If basic < 1500 then hra= 75% of the basic, da=80 % of the basic. Otherwise hra=85% of the basic , da=87% of the basic.

```
main( )
{
    int basic;
    float da,hra,gross;
    printf("Enter basic salary of the employee ");
    scanf("%d",&basic);
    if (basic<1500)           //no semicolon
    {
        da=0.87*basic;
        hra=0.75*basic;
    }
    else                      //condition for else block to execute basic>=1500
    {
        da=0.80*basic;
        hra=0.85*basic;
    }
    gross=basic+da+hra; //next statement of if –else block
    printf("Gross Salary=%f",gross);
    printf("Thanks for using this program");
}
```

OUTPUT:

```
Enter basic salary of the employee 5000
Gross salary=16100
Thanks for using this program
```

Example : Find the output of the following program

```
#include<stdio.h>
main()
{
    int input1=10,input2 = 25;
    if (input1=input2)
        printf("Equal\n");
    else //no semicolon
        printf("Not Equal\n");
    printf("Thanks for using this program"); //next statement after if –else block
}
```

OUTPUT- Equal
Thanks for using this program

As the value of the expression `input1=input2` is 25 which is non zero value means that expression is true. So if block is executed. Here we are not comparing the values of `input1` and `input2` rather the assignment operator (`=`) will assign value of `input2` to `input1`.

Points- it is not mandatory to have else block for if block. But else block is always preceded by if block.

9. Decision Making : Nested if-else :

9.1 Nested if –else :

We can define an if block or an else block within the body of another if-block or else block. This is called nesting of if-else. Syntax for nested if –else is as follows :

<p>1. if (expr1)</p> <p> {</p> <p> if (expr2)</p> <p> stmt T1 ;</p> <p> else</p> <p> stmt T2 ;</p> <p> }</p> <p> else</p> <p> Stmt T3 ;</p>	<p>// Here the inner if-else block is defined inside the body of the outer if block .</p>
<p>2. if (expr1)</p> <p> stmt T1 ;</p> <p> else</p> <p> {</p> <p> if (expr2)</p> <p> Stmt T2;</p> <p> else</p> <p> Stmt T3;</p> <p> }</p>	<p>// Here the inner if-else block is defined inside the body of the outer else block .</p>

It is also possible that both if block and else block can be nested.

Example : WAP to find largest among 3 numbers

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

```

int input1,input2,input3;
printf("Enter three numbers:");
scanf("%d%d%d",&input1,&input2,&input3);
if (input1 > input2)
{
    if(input1 > input3)
        printf("%d is largest number",input1);
    else
        printf("%d is largest number",input3);
}
else
{
    if (input2 > input3)
        printf("%d is largest number",input2);
    else
        printf("%d is largest number",input3);
}
}

```

OUTPUT-

```

Enter three numbers
5 9 6
9 is largest number

```

9.2 if-else-if statement (else-if ladder) :

Multiple decisions arise when there is multiple condition and different statements are to be executed under different condition. The conditional expression for if could have several possible values. There is if-else statement in every else part except last else part.

Syntax of if-else-if :

```

if ( expr1)
    stmt t1;
else if ( expr2)
    stmt t2;
    .....
    else if (expr n)
        stmt tn;
    else
        stmt f;

```

Expression expr1 is evaluated first if it true then statement stmt t1 is executed. If it is false expr2 is evaluated, if it is true stmt t2 is executed and so on. If none of the expression (1 to n) is true then stmt f is executed. The expr is evaluated in order. If any expr is true the stmt associated with it is executed and this terminates the whole chain. The last else part is executed when none of the above case is satisfied.

Example : WAP to print the grade of a student by inputting marks of 3 subjects through the keyboard. Full mark for each subject is 100. Grade is calculated as follows :

Grade O if avg mark ≥ 90

E if avg mark ≥ 80

A if avg mark ≥ 70

B if avg mark ≥ 60

C if avg mark ≥ 50

D if avg mark ≥ 40

Others F grade.

Use else-if ladder in your program.

```
#include<stdio.h>
main( )
{
    int sub1,sub2,sub3;
    int avg;
    printf("Enter marks in 3 subject");
    scanf("%d%d%d",&sub1,&sub2,&sub3);
    avg = (sub1+sub2+sub3) / 3;
    printf("Average=%d\n",avg);
    if(avg >= 90)
        printf("O Grade");
    else if(avg >= 80)
        printf("E Grade");
    else if(avg >= 70)
        printf("A Grade");
    else if(avg >= 60)
        printf("B Grade");
    else if(avg >= 50)
        printf("C grade");
    else if(avg >= 40)
        printf("D Grade");
    else
        printf("F grade");
}
```

OUTPUT-

```
Enter marks in 3 subject
80 98 91
Average=89
E grade
```

1. If-else- if reduces the indentation.
2. Every else is associated with previous if.
3. Last else is optional.

Example : WAP to print the grade of a student by inputting marks of 3 subjects through the keyboard. Full mark for each subject is 100. Grade is calculated as follows :

**Grade O if avg mark >=90
E if avg mark >=80
A if avg mark >=70
B if avg mark >=60
C if avg mark >=50
D if avg mark >=40
Others F grade.**

Use multiple if statements in your program.

```
#include<stdio.h>
main()
{
    int sub1,sub2,sub3;
    int avg;
    printf("Enter marks in 3 subject");
    scanf("%d%d%d",&sub1,&sub2,&sub3);
    avg=(sub1+sub2+sub3)/3;
    printf("Average=%d\n",avg);
    if(avg>=90)
        printf("O grade");
    if( (avg>=80) && (avg<90) )
        printf("E grade");
    if ( (avg>=70) && (avg<80))
        printf("A Grade");
    if ( (avg>=60) && (avg<70))
        printf("B Grade");
    if ( (avg>=50) && (avg<60))
        printf("C grade");
    if ( (avg>=40) && (avg<50))
        printf("D Grade");
    if( avg <40 )
        printf("F grade");
}
```

OUTPUT-

```
Enter marks in 3 subject
80 98 91
Average=89
E grade
```

10. Decision Making : Switch Statement :

10.1 Switch case statement:

Switch statement is used to choose a particular group statements from several available groups. The selection is based upon the value of an expression. Syntax of switch statement :

```
switch (expression)
{
    case expr1:
        stmt t11;
        stmt t12;

        ..
    case expr2:
        stmt t21;
        stmt t22;
        ....

    .....
    case expr n:
        stmt tn1;
        stmt tn2;
        ....

    .....
    default:
        Stmt tn1;
        .....
}
```

Here switch, case and default are the keywords .

The expression following keyword **switch** must evaluate to an integer value. The expression may also be character type because each character has equivalent integer value. First statement of each group must be preceded by one **case label** . The case label identify different group of statements and distinguish them from one another. So the **case label must be unique** within a switch statement. The case label must be followed by semicolon. The case label must represent constant integer valued expression. Each individual group following the case label may be simple or complex. When switch statement is executed the expression is evaluated and control is transferred to group of statement whose case label matches with the value of the expression. **When a match is found execution start from that case and all the subsequent case and default as well. If none of the case label matches the statement under default case is executed.** The default part is optional in switch statement. If no default case is used and none of the case label is matched , then control is transferred directly to the next statement after switch. Curly brackets are not necessary to group the statement under a particular case, even if multiple statements are there inside a case.

Example :

```
main()  
{  
    int i;  
    printf("Enter a number");  
    scanf("%d",&i);  
    switch( i )  
    {  
        case 1:  
            printf("i am in case1\n");  
        case 2:  
            printf("i am in case 2\n")  
        case 3:  
            printf("i am in case 3\n");  
        default:  
            printf(" i am in Default\n")  
    }  
}
```

OUTPUT- Enter number 2
i am in case 2
i am in case 3
i am in default

This is because , here the matching case block is 2 as the value of variable i is 2. So from case block2 , execution will be started. Case block2 then case block3 and then the default block will be executed.

If you want that only case 2 should be executed and the control should go out of switch statement immediately then use break statement as the last statement in each case block but not in the default block.

Example : WAP to carry out basic arithmetic operations (+, -, *, /) based on option

```
#include <stdio.h>  
main()  
{  
    int input1,input2,result;  
    char choice;  
    printf("Enter 2 number");  
    scanf("%d%d",&input1,&input2);  
    printf("+ -addition\n - -subtraction\n * -multiplication\n \ -division")  
    printf("Enter your option\n");  
    scanf(" %c",&choice)  
    switch ( choice)  
    {  
        case '+':
```

```

        result=input1+input2;
        break;
    case '-':
        result=input1-input2;
        break;
    case '*':
        result=input1*input2;
        break;
    case '\':
        result=input1\input2;
        break;
    default:
        printf("enter a valid choice")
    }
    printf("%d %c %d = %d",input1, choice, input2, result) ;
}

```

OUTPUT- Enter 2 number 7 10
+ -addition
- subtraction
* -multiplication
\ -division
Enter your option +
7+10=17

11. Branching Statements - Loops (while loop and do-while loop) :

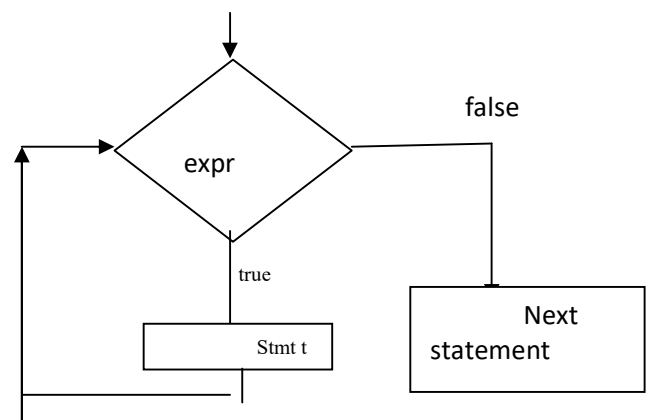
Loops :

Loops are used when programmer wants to repeat some portion of the program either a specified number of times or until a particular condition is being satisfied. There are 3 loop statement in C :

1. **while**
2. **do while**
3. **for**

11.1 while loop

It is an entry control loop.



Syntax of while loop :

```
while ( expr )  
  stmt t;
```

Statement **Stmt t** or body of the loop will be executed repeatedly as long as the expression **expr** is true (i.e. expr has non zero value). **stmt t** can be a simple or compound statement. When the value of the **expr** becomes false the loop terminates and control goes out of the loop. The value of the **expr** must be altered inside body of the loop to avoid infinite looping.

We can also keep multiple number of statements in side the body of the while loop but those statements must be enclosed within a pair of **curly braces { }** . The syntax to keep multiple statements inside the body of the while loop is :

```
while ( expr )  
{  
  stmt 1;  
  stmt 2 ;  
  .  
  .  
  .  
  stmt n ;  
}
```

Example : WAP to print silicon 5 times

```
#include< stdio.h >  
main()  
{  
    int i=0;          // initialization of loop control variable  
    while ( i < 5 )   //condition  
    {  
        printf ("silicon\n");  
        i++;          //update  
    }  
}
```

OUTPUT- silicon
silicon
silicon
silicon
silicon

The number of times body of the loop is executed, is known as **iteration of the loop**. In the above example there are 5 iterations. In each iteration, the value of the loop control variable is updated.

Example : WAP to find out the sum of $1^2+2^2+\dots+5^2$

```
#include<stdio.h>
main()
{
    int i=1;           // initialization of loop control variable
    int sum=0;
    while ( i<=5 )    //condition
    {
        sum = sum + i * i;
        i++;          //update
    }
    printf("SUM=%d",sum);
}
```

OUTPUT- SUM= 55

Here the while loop condition will remain true for i value to be 1, 2, 3, 4, 5. So the body of the loop will be executed for 5 times. When i value will be 6 then the condition will be false and the control will come out of the loop.

Example : WAP to find factorial of any number

```
#include <stdio.h>
main()
{
    int input, i=1;      // initialization of loop control variable
    long int fact=1;
    printf("Enter a number\n")
    scanf("%d",&input);
    if ( input < 0 )
        printf("factorial of negative number is not possible");
    else
    {
        while ( i <= input )    //condition
        {
            fact = fact * i ;
            i++;                //update
        }
        printf("factorial of %d = %ld", input, fact);
    }
}
```

OUTPUT -

```
Enter a number
3
Factorial of 3 = 6
```

fact = 1

it-1
fact=1*1
i=2

it-2
fact=1*2
i=3

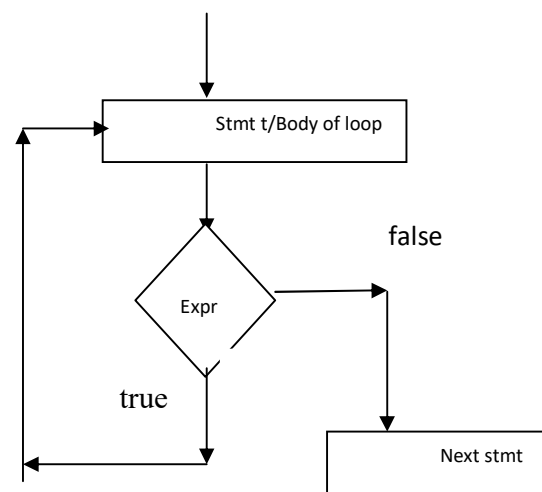
it-3
fact=2*3
i=4

When i is 4, loop condition is false .

Here if we enter the value of input as 3 , the if condition will be false. So the else part will be executed. In the else part, while loop condition will be true as the initial value of i is 1 and input value is 3 (i.e $i \leq \text{input}$ is true) . So body of the loop will be executed and the new value for fact will be 1 and i will be 2. Then in the next iteration of the loop, again the loop condition will be true as i value is 2 and input is 3 (i.e $i \leq \text{input}$ will be true) . Hence the body of the loop will be executed again. So fact will be 2 and i will be 3. Now in the next iteration of the loop, again the condition will be true as i is 3 and input is 3 (i.e $i \leq \text{input}$ is true) . Hence the body will be executed which will cause fact to be 6 and i to be 4 . Now the loop condition will be false as i is 4 and input is 3 (i.e $i \leq \text{input}$ is false) . So the control will come out of the while loop and the printf() will be executed which will print **factorial of 3 = 6** .

11.2 do while loop :

It is an exit control loop.



Syntax of do while:

```
do  
    Stmt t;  
while (expr );
```

First the statement **stmt t** or body of the loop is executed and then the expression **expr** is evaluated. If the expression is false , then the control comes out of the loop. If the expression is true then the control enters into the body again. Stmt t keeps on executing as long as the value of the expression **expr** is true. When the value of the **expr** becomes false the loop terminates and control goes out of the loop. There must be one statement in the body of the loop that alters the loop control variable so that **expr** becomes false ultimately at some point to avoid infinite looping.

Body of the do while loop will be executed at least once. Since the test for the condition does not occur until the end of the first iteration.

We can also keep multiple number of statements in side the body of the do-while loop but those statements must be enclosed within a pair of **curly braces { }** . The syntax to keep multiple statements inside the body of the do-while loop is :

```
do
{
    stmt 1;
    stmt 2;
    ----- ;
    ----- ;
    ----- ;
    ----- ;
    stmt n ;
} while ( condition) ;
```

Example : WAP to print silicon 5 times using do-while loop.

```
#include<stdio.h>
main()
{
    int i=0;           //initialization
    do
    {
        printf("silicon\n");
        i++;           //update
    } while ( i<5 );    //condition checking
}
```

OUTPUT- silicon
 silicon
 silicon
 silicon
 silicon

Difference between while and do while loop :

In while loop first of all the loop condition is tested. If the condition is false then control comes out of the loop and if the condition is true then body is executed. So while loop is an entry-control loop. Where as in do-while loop , first the body is executed once then the loop condition is tested. So do-while loop is an exit-control loop.

Example :
main()
{

```

        int i=0;          //initialization
        do
        {
            printf("silicon\n");
            i++;          //update
        } while (i<0);    //condition checking
    }

```

OUTPUT- silicon

because do while loop body executed at least once then condition is checked for continuation.

```

main()
{
    int i=0;          //initialization
    while (i<0)      //condition checking
    {
        printf("silicon\n");
        i++; //update
    }
}

```

OUTPUT- No output

Because first the condition for continuation is checked then accordingly body of the loop is executed.

There are 2 types 1) Entry control loop 2) Exit control loop

In entry control loop first expression or condition is evaluated and if it is true body of the loop is executed. The process of repeating the execution of body continues until the expression becomes false and then control goes out of the loop to the next statement after the loop. Example- while loop, for loop . In exit control loop first the body the loop is executed then condition is checked for repetition. Example-do while

Example : WAP to count no of digits present in any number.

```

#include<stdio.h>
main()
{
    int input,count=0;
    printf("Enter the number");
    scanf("%d",&input);
    do
    {
        input = input/10;
        count++;
    } while (input>0) ;
    printf("No of digits=%d",count);
}

```

OUTPUT-

Enter the number 100

No of digits=3

Count must be initialized to 0 otherwise initial value for count will be some garbage value.

12. Nesting of Loops :

12.1 Nested while loop

When a loop is defined inside the body of another loop, it is known as nesting of loops. The way if-statement is nested, similarly loops can be nested. Any type of loop can be nested in any loop. A while loop can be nested in a while or do-while or for loop.

Example :

```
main()
{
    int i,j;
    i=1;
    while ( i <= 3 )           //outer loop
    {
        j=1                   //body of outer loop start
        while ( j <= 3 )      //inner loop
        {
            //body of inner loop start
            printf("%d\t",j);
            j++;
        }
        printf("\n");
        i++;
    }
}
```

o/p-
1 2 3
1 2 3
1 2 3

Explanation- inner loop terminates when value of j exceeds 3 and outer loop terminates when value of i exceeds 3. In the beginning of iteration 1 of outer loop i=1 and j value can go from 1 to 3. So 1 to 3 will be printed in the first line In the beginning of iteration 2 of the outer loop i=2 and j value can go from 1 to 3. So 1 to 3 will be printed in the second line. In the beginning of iteration 3 of the outer loop i=3 and j value can go from 1 to 3. So 1 to 3 will be printed in the third line

Example :

```
main()
{
    int i, j;

    for(i=1; i<=3; i++)          //outer loop
    {
        //body of outer loop start
        j=1;
        while ( j <= 3 )        //inner loop
        {
            //body of inner loop start
            printf("%d\t", j);
            j++;
        }
        printf("\n");
    }
}
```

OUTPUT- 1 2 3
1 2 3
1 2 3

Example : WAP to print multiplication table of 2 to 5

```
#include < stdio.h >
main()
{
    int n,i;
    for (n=2 ; n<=5 ; n++ )
    {
        j=1;
        while(j<=10 )
        {
            printf("%d\t", n*j);
            j++;
        }
        printf("\n ");
    }
}
```

OUTPUT -

```
2 4 6 8 ..... 20
3 6 9 12.....30
4 8 12 15 .....40
5 10 .....50
```

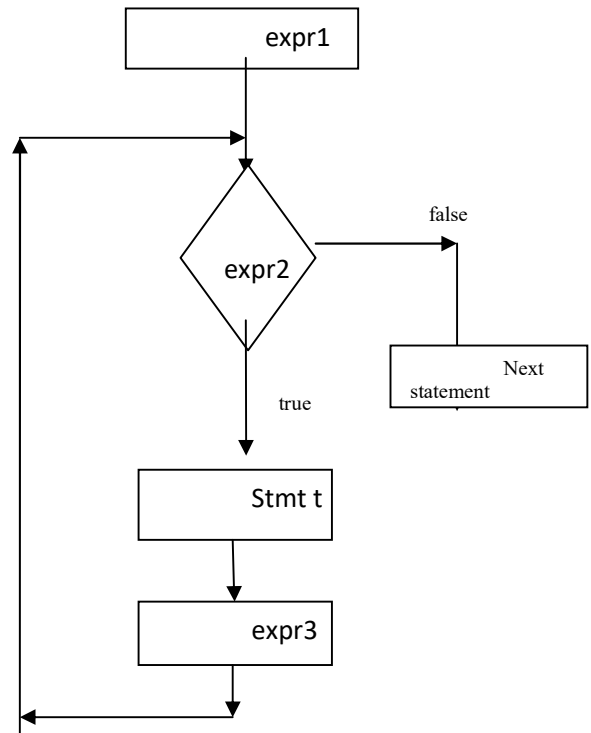
13. Branching Statements - for loop :

13.1 Loops (for loop)

for loop is another entry controlled loop.

Syntax of for loop :

```
for (expr1; expr2 ; expr3)  
    stmt t;
```



Each expression expr must be separated by semicolon. expr1 is executed only once when the loop starts and is used to initialize the loop control variable. expr2 is a condition that determines whether body of the loop is continued or not and it is tested before each iteration. expr3 modifies the loop control variable at the end of each iteration.

Although task of while loop and for loop is same. For loop is used when number of iteration known in Advance and while loop is used when number of iteration are not known.

Example : WAP to print silicon 5 times

```
1) #include<stdio.h>
   main()
   {
       int i;
       for(i=0; i < 5 ; i++)
       {
           printf("silicon\n");
       }
   }
```

// dropping initialization in the for statement

```
2) #include<stdio.h>
   main()
   {
       int i=0;
       for(; i < 5 ; i++)
       {
           printf("silicon\n");
       }
   }
```

// dropping update in the for statement

```
3) #include<stdio.h>
   main()
   {
       int i;
       for(i=0; i < 5 ;)
       {
           printf("silicon\n");
           i++;
       }
   }
```

//dropping both initialization and update in the for statement

```
4) ) #include<stdio.h>
   main()
   {
       int i=0;
       for(; i < 5 ;)
       {
           printf("silicon\n");
           i++;
       }
   }
```

OUTPUT - The above 4 programs produce the same output.

```
silicon
silicon
silicon
silicon
silicon
```

Note : If the 2nd expression of the for loop is omitted, compiler assumes that it has value 1. So for loop is executed indefinitely unless interrupted by some means.

Example of infinite for loop: **for (; ;)**
 printf("silicon");

The initialization expression or update expression of the for loop can contain more than one expression separated by comma. This is called as **multiple initialization** in for loop.

Example :

```
main()
{
    int i, j ;
    for(i=1 , j=5; i<j ; i++ , j--)
        printf("*\t")
}
```

OUTPUT - * *

Example : WAP to print multiplication table of 5

```
#include<stdio.h>
main()
{
    int i,j;
    for(i=5,j=1; j <=10 ; j++)
    {
        printf("%d\t",i*j);
    }
}
```

OUTPUT - 5 10 15 20 25 30 35 40 45 50

Example : WAP to evaluate $x + x/2 + \dots + x/n$

```
#include<stdio.h>
main()
{
    int x,n,count;
    float sum=0.0;
    printf("Enter the value of x and n\n");
    scanf("%d%d",&x,&n);
    for(count=1; count<=n ; count++)
    {
        sum = sum + x / (float) count;
    }
    printf("Sum of the given series=%f",sum);
}
```

OUTPUT-- Enter the value of x and n
 3 2
 Sum of the series=4.5

14. Nested for loops :

14.1Nested for loop

When a loop is defined inside the body of another loop, it is known as nesting of loops. The way if-statement is nested, similarly loops can be nested. Any type of loop can be nested in any loop. A for loop can be nested in a while or do-while or for loop.

Example :

```
main()
{
    int r,c;
    for(r=1 ; r <=2 ; r++)          // outer loop
    {
        for(c=1;c<=2;c++)          // inner loop
        {
            printf("r=%d\t c=%d \t sum=%d \n",r,c,r+c);
        }
    }
}
```

OUTPUT- r=1 c=1 sum=2
 r=1 c=2 sum=3
 r=2 c=1 sum=3
 r=2 c=2 sum=4

Example : WAP to print the following pattern

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

```
main()
{
    int r,c;
    for(r=1 ; r <=4 ; r++)          // outer loop
    {
        for( c=1;c<=4-i+1;c++)      // inner loop
```

```

        {
            printf("* \t");
        }
        printf("\n");
    }
}

```

OUTPUT- * * * *

 * * *

 * *

 *

Example : WAP to print the following pattern

```

1 2 3 4
1 2 3
1 2
1

```

```

main()
{
    int r, c, n ;
    Printf("Enter number of rows");
    Scanf("%d",&n);
    for(r=1 ; r <=n ; r++)
    {
        for( c=1;c<=n-i+1;c++)
        {
            printf("%d \t",c);
        }
        printf("\n");
    }
}

```

OUTPTUT-

```

Enter number of rows  4
1 2 3 4
1 2 3
1 2
1

```

15. Jump statements :

Loops perform a set of operations repeatedly until the control variable fails to satisfy the test condition . The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave as soon as a certain condition occurs. C permits a Jump from one statement to another within a loop as well as a Jump out of a loop.

15.1 The break statement :

We often come across situations where we want to jump out of a loop instantly, without waiting to get back to the conditional test. The keyword **break** allows us to do this

When a **break** statement is encountered inside a loop, the loop is immediately exited and the control automatically passes to the first statement after the loop.

A **break** is usually associated with an **if** statement.

Example1:

```
int i = 10;
while ( i > 0 )
{
    if ( i == 5 )
        break;
    printf("%d\t", i );
    i--;
}
```

The above code segment will display 10 9 8 7 6. Because when i value will be 5, then the if condition will be satisfied and the break statement will be executed which will take the control out of the while loop.

When loops are nested, the **break** would only exit from the loop containing it. That is, the break will exit only a single loop.

Example2:

```
int x, y ;
for( x=5; x>=1; x-- )
{
    for( y=1; y<=2; y++)
    {
        if( y == x )
            break ;
        printf( "x=%d \t y=%d\n", x,y) ;
    }
    printf("\n out of inner loop\n") ;
}
```

The output of the above code segment will be

```
x=5    y=1
x=5    y=2
out of inner loop
x=4    y=1
x=4    y=2
out of inner loop
x=3    y=1
x=3    y=2
out of inner loop
x=2    y=1
out of inner loop
out of inner loop
```

Because when x is 2 and y is also 2, in the inner for loop the if condition will be satisfied and break statement will be executed which will take the control out of the inner for loop. After the inner for loop, the printf() which is present inside the body of the outer for loop will be executed. Similarly when x is 1 and y is 1, in the inner for loop the if condition will be satisfied and break statement will be executed which will take the control out of the inner for loop. After the inner for loop, the printf() which is present inside the body of the outer for loop will be executed.

15.2 The continue statement :

During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions. The keyword **continue** allows us to do this. The **continue** statement causes the loop to be continued with the next iteration after skipping any statements in between. The **continue** statement tells the compiler, " SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION".

A **continue** is usually associated with an **if** statement.

Example1:

```
int i = 10;
while ( i > 0 )
{
    if ( i == 5 )
        continue;
    printf("%d\t", i );
    i--;
}
```

The output of the above code segment will be 10 9 8 7 6 4 3 2 1

Because when i will be 5 the if condition in the while loop will be satisfied and the continue statement will be executed which will skip the execution of the following statements in side the while loop and will take the control to the beginning of the next iteration of the while loop.

Example2:

```
int x, y ;
for( x=1; x<=2; x++)
{
    for( y=1; y<=2; y++)
    {
        if( y= x)
            continue ;
        printf( "x=%d \t y=%d\n", x,y) ;
    }
}
```

The output of the above code segment will be

```
x =1  y=2
x =2  y=1
```

Because when $x = y$, then if condition in the inner for loop will be satisfied and the continue statement will be executed which will skip the execution of the following statements in the inner for loop and will take the control to the beginning of the next iteration of the inner for loop.

15.3 The go to statement :

A **go to** statement can transfer the control to any place in a program. But it is a good practice to avoid **go to** statement. It obscures the flow of control and makes the the program totally unstructured. Therefore the use of **goto** statement should be strictly avoided .

Example:

```
#include<stdio.h>
main( )
{
    int num1 , num2;
    printf(" ENTER ANY TWO NUMBERS");
    scanf("%d%d", &num1 ,&num2);
    if( num1 > num2)
        goto sos;
    else
        printf("num2 is largest");
    sos:
        printf("num1 is largest");
}
```

When if condition is satisfied the goto statement transfer the control to the label 'sos' , causing printf() following sos to be executed. The label can be on a separate line or on the same line as the statement following it , as in ,

```
sos: printf(" num1 is largest");
```

15.4 exit() :

We can jump out of a program by using the library function **exit()**. The **exit()** takes an integer value as its argument. Normally zero is used to indicate normal termination of the program and non-zero value to indicate termination due to some error or abnormal condition. The use of **exit()** requires the inclusion of the header file **<stdlib.h>** .

16. FUNCTION :

16.1 Monolithic vs Modular Programming :

In **Monolithic programming**, it is possible to code the entire program using only **main()**.

Disadvantages :

1. Program becomes too large and too complicated.
2. Testing, debugging and maintenance becomes difficult.

In **Modular programming**, a program is divided into no. of functional parts which may be independently coded and later combined into a single unit. These independently codes subprograms are called as **functions**.

Advantages :

1. Easy to understand
2. Easy to test, debug and maintain.

16.2 Introduction to Function :

A **function** is a self-contained block of code that performs a particular task. C function can be classified into two categories :

1. Library Function
2. User-defines Function

The main differences between them is Library Functions are not required to be written by us because they are pre-defined functions where as User-defined function has to be developed by the user at the time of writing a program.

A Function has the following 3 parts:

1. **Function Declaration or Function Prototype**
2. **Function Calling**
3. **Function Definition**

Function Declaration :

Function Declaration or Function Prototype consists of four parts.

- * Return type of the function
- * Function Name
- * Parameter List
- * Terminating Semicolon

Return type of a function indicates what type of value the function is going to return. If a function returns an integer value, then its return type is int. Similarly if the function returns a floating type value, then its return type is float. When a function does not return any value then its return type is void.

Function Name is a valid identifier. The name should be appropriate to the task performed by the function.

A function can have any no of parameters depending on the requirements. Even a function does not have any parameter also. When a function contains parameters, then the parameter list must be separated by commas. Only the type of the parameter is mandatory in the function declaration part where as parameter name is optional.

Like variable declaration, a function declaration statement must be ended with a semicolon.

Syntax of Function Declaration:

returntype functionname (parameter list) ;

Examples of function declaration:

- | | |
|--------------------------------|--|
| void sum(); | Here the function sum won't take any parameter and won't return any value. |
| int sum (void); | Here the function sum won't take any parameter but returns an integer value. |
| int sum(int, int); | Here the function sum will take two parameters, both of them are of int type and the function will return an int value . |
| float sum (int, float); | Here the function sum will take two parameters, first one is of int type and second one is of float type.
Function sum will return a float value. |

Function Declaration informs the following 4 things to the compiler.

- * What is the name of the function
- * What type of value the function will return
- * How many parameters the function will take
- * What type of parameters the function will take

Function Calling :

A function can be called by using the function name followed by a list of actual arguments (or parameters) if any, enclosed in parentheses.

The syntax of function calling is:

Functionname (actual arguments);

Here only the name of the parameters must be mentioned in the parameter list , not the type. If the function won't take any parameter, then the parentheses must be empty.

Examples of function calling :

```
sum( );  
sum ( x , y );  
sum (10, 20) ;
```

When a function calling statement is executed, the execution of the rest of the statements of the program will be temporarily suspended because immediately after the function calling, control will automatically be transferred to the function definition part.

Function Definition :

Function Definition is an independent program module that is specially written to implement the requirements of the function. A function definition should include the following elements :

1. return type
2. function name
3. list of parameters (both type and name of each parameter)
4. local variable declaration
5. function statements
6. a return statement

All the six elements are grouped into two parts :

- function header (the first three elements)
- function body (the second three elements)

The syntax for function definition is :

```
returntype functionname ( type parametername1 , type parametername2 , .....)  
{  
    ----- ;  
    ----- ;    body of the function  
    ----- ;  
}
```

Here the return type of the function and the function name must be matched exactly with the same which has already mentioned in the function declaration part. The types must match the types of parameters in the function declaration, in number, in types and in order. When types do not match with the types in the declaration, compiler will produce an error. The parameter names do not need to be the same in the prototype declaration and the function definition.

Notes :

1. The function who is calling, is called as the calling function and the function which will be called, is called as the called function.
2. Parameters mentioned in the function calling part are called as the actual parameters or actual arguments. Parameters mentioned in the function definition part are called as the formal parameters or formal arguments.
3. In the function body, the function statements are executed sequentially one after another. When a return statement is executed in the body of a function, the control is transferred back to the calling function. In the absence of a return statement in the body, the closing brace of the function definition acts as a void return which automatically returns the control back to the calling function.
4. A function need not always receive value from calling function. In such cases, function has no parameters. To indicate that the parameter list is empty, we use the keyword void between the parentheses as in

```
void sum ( void )  
{  
    ----- ;  
    ----- ;  
}
```

5. A local variable is a variable that is defined inside the body of a function and used without having any role in the communication between functions.
6. A function may or may not send back any value to the calling function. If it does, it is done through the return statement. The called function can only return one value per call. The return statement can take one of the following forms :

return;	Here only the control will return. It does not return any value.
---------	--

return (expression);	It returns the value.
----------------------	-----------------------

Examples of function definition :

Example1 :

```
void sum( int num1, int num2 )    // Function Header  
{  
    int num3;                    // Local Variable  
    num3 = num1 + num2 ;
```

```
        printf("%d", num3);
    }
```

In the above example function sum will receive two arguments of int type (num1 and num2) . Their addition result will be stored in the local variable num3. Then printf() will display the value of num3. Here sum() does not return any value to the calling function, hence the return type of the function is void.

Example2 :

```
int sum( int num1, int num2 )
{
    int num3;
    num3 = num1 + num2;
    return( num3);
}
```

Here return statement returns the value of num3 to the calling function. Since num3 is an integer variable, hence the return type of the function is int.

17. Simple programs on function :

Program : Write a program to find out the largest between two numbers using function.

```
#include <stdio.h>
int max ( int, int );    // Function Prototype
main( )
{
    int num1, num2, largest ;
    printf(" Enter any two numbers");
    scanf("%d %d" , &num1, &num2);
    largest = max( num1, num2 );        // Function Calling
    printf(" Largest number=%d" , largest);
}

int max( int var1, int var2)            // Function Definition
{
    if( var1 > var2 )
        return var1;                    // Return statement
    else
        return var2;                    // Return statement
}
```

In the above program, main() is the calling function and max() is the called function. When the function calling statement max(num1, num2) inside the main() is executed , execution of the rest of the statements of the calling function i.e main() will temporarily be suspended because control automatically

In C, there are two standard methods to pass the parameters from the calling function to the called function.

1. Pass by value or Call by value Method
2. Pass by address or Call by address Method

1. Call by value Method :

In the call by value method, the called function creates a new set of variables and then copies the values of the actual arguments into them. So the formal arguments are the photocopies of the actual arguments. Value of the actual arguments are passed from the calling function to the called function. So any changes to the formal arguments in the called function won't affect the value of the actual arguments in the calling function.

Example : **Program to swap two variables using call by value method.**

```
#include <stdio.h>
void swap( int, int );           // Function Prototype
main( )
{
    int num1, num2 ;
    printf(" Enter any two numbers") ;
    scanf("%d%d" , &num1, &num2);
    printf(" Before swapping num1= %d and num2= %d", num1,num2);

    swap( num1, num2 );          // Function Calling
    printf(" After swapping num1= %d and num2= %d", num1,num2);
}

void swap( int num1, int num2 )  // Function Definition
{
    int temp;
    temp = num1;
    num1 = num2;
    num2 = temp;
    printf(" In the called function num1=%d and num2= %d" , num1,num2);
}
```

In the above program if we input num1 value as 10 and num2 value as 20 then the output will be :

```
Before swapping num1 = 10 and num2 =20
In the called function num1 =20 and num2 =10
After swapping num1 = 10 and num2 = 20
```

In the above program main() is the calling function and swap() is the called function. variables num1 and num2 in the main() and variables num1 and num2 in the swap() are totally different from each other, only their names are same. Their memory allocations are also different. In the main(), if the user has inputted

10 and 20 as the values of num1 and num2 respectively, then the printf() will display the values of num1 and num2 to be 10 and 20 .Hence the first output of the program will be :

Before swapping num1 = 10 and num2 =20

Then the function calling statement swap(num1, num2) will be executed and the control is transferred to the called function swap() . In the swap(), value of the actual arguments num1 and num2 are copied into the formal arguments num1 and num2 respectively. Hence num1 and num2 in the called function swap() initially becomes 10 and 20. Then as per the logic in the swap(), we have swapped the values of num1 and num2. Hence in the swap(), num1 value will be 20 and num2 value will be 10. The printf() inside the body of the swap() will print 20 and 10 for num1 and num2 respectively. Hence the second output of the program will be :

In the called function num1 =20 and num2 =10

But when the control will be returned back to the main(), inside the main() value of num1 and num2 are still 10 and 20. Hence the last printf() inside the body of the main(), will print 10 and 20 as the values of num1 and num2 respectively. Hence the last output of the program will be :

After swapping num1 = 10 and num2 = 20

2. Call by Address Method :

In the call by address method, address of the actual arguments is passed from the calling function to the called function. So the formal arguments in the called function are basically the pointers to the actual arguments. Therefore any changes to the formal arguments in the called function will directly affect the actual arguments in the calling function.

Example : Program to swap two variables using call by address method.

```
#include <stdio.h>
void swap( int *, int *);           // Function Prototype
main( )
{
    int num1, num2 ;
    printf(" Enter any two numbers") ;
    scanf("%d%d" , &num1, &num2);
    printf(" Before swapping num1= %d and num2= %d", num1,num2);

    swap( &num1, &num2);           // Function Calling
    printf(" After swapping num1= %d and num2= %d", num1,num2);
}
```

```

void swap( int *num1, int *num2 )           // Function Definition
{
    int temp;
    temp = *num1;
    *num1 = *num2;
    * num2 = temp;
    printf(" In the called function num1=%d and num2= %d" , *num1,*num2);
}

```

In the above program if we input num1 value as 10 and num2 value as 20 then the output will be :

Before swapping num1 = 10 and num2 =20
 In the called function num1 =20 and num2 =10
 After swapping num1 = 20 and num2 = 10

In the above program, From main() we are passing the address of the actual arguments num1 and num2 to the called function swap(). Hence the function calling statement is : swap(&num1, &num2); . num1 and num2 in the swap() are basically the pointers to the actual arguments num1 and num2. In the called function swap() , with the help of these pointers we swap the value of num1 and num2 and this modification is also reflected in the calling function main().

19. Introduction to Recursion :

Recursion is the process of defining something in terms of itself. For an example : Suppose we want to find out factorial value of 5. As we know that, the factorial of a number is the product of all integers between 1 and that number. So

$$5! = 5 * 4 * 3 * 2 * 1 \quad \text{This can also be expressed as}$$

$$5! = 5 * 4!$$

Thus factorial of a number can be expressed in the form of itself.

A function is called as a recursive function, if any statement within the body of the function calls the same function. Recursive functions can be effectively used to solve problems where solution is expressed in terms of successively applying the same solution to the subsets of the problem.

Example :

```

int fun( ----- ) ←-----
{
    -----;
    -----;
    -----;
    -----;
    -----;
}

```

Program : Write a program to find out the factorial of a number using recursion.

```
#include <stdio.h>
int rec( int );
main( )
{
    int num, fact;
    printf(" Enter any number ");
    scanf("%d", &num);
    fact = rec( num );
    printf(" Factorial of %d is %d", num, fact);
}

int rec( int x )
{
    int f;
    if ( x == 1 || x == 0 )           // Terminating Condition
        return ( 1 );
    else
        f = x * rec( x - 1 );

    return( f );
}
```

Here is the output of the above program :

```
Enter any number 3
Factorial of 3 is 6
```

Let us see how recursion works ? Assume $x = 3$ in the called function `rec()`. Since the value of x is not 1, the statement `f = x * rec(x - 1);` will be executed with $x = 3$. That is ,

`f = 3 * rec(2);`

will be executed. This expression on the right hand side includes a call to `rec()` with $x = 2$. This call will return the following value :

`2 * rec(1)`

Once again, `rec()` is called with $x = 1$. This time, the function returns 1. The sequence of operations can be summarized as follows :

**`f = 3 * rec(2)`
`= 3 * 2 * rec(1)`
`= 3 * 2 * 1`
`= 6`**

Note : When we write recursive functions , we must have an if statement (**terminating condition**) some where to force the function to return without the recursive call being executed. Otherwise the function will never return .

Program : Write a program to print the Fibonacci series up to the nth term using Recursion .

```
#include <stdio.h>
int fibo( int );
main( )
{
    int n, i;
    printf( " Enter the value of n " );
    scanf( "%d" , &n );
    for( i = 0; i < n; i++ )
        printf( "%d\t" , fibo( i ) );
}

int fibo( int n )
{
    if( n == 1 || n == 0 )           // Terminating Condition
        return n ;
    else
        return( fibo( n - 1 ) + fibo( n - 2 ) );
}
```

Input : Enter the value of n 7

Output : 0 1 1 2 3 5 8

20. Storage Classes in C :

To fully define a variable , one needs to mention not only its 'type' but also its ' storage class' . In other words, In C not only do all variables have a data type , they also have a storage class. A variable's storage class tells us :

- (a.) Where the variable would be stored.
- (b.) What is the default initial value of the variable.
- (c.) What is the scope of the variable (In which functions value of the variable would be available) .
- (d.) What is the life of the variable (how long would the variable exist) .

There are four storage classes in C :

1. **Automatic Storage Class**
2. **Register Storage Class**
3. **Static Storage Class**
4. **External Storage Class**

20.1 Automatic Storage Class :

Storage :	Memory .
Default Initial Value :	An unpredictable value called as Garbage Value .
Scope :	Local to the block in which the variable is defined.
Life :	Till the control remains within the block in which the variable is defined .

Automatic variables are defined inside a function in which they are to be used. They are created when the function is called and destroyed automatically when the function is exited , hence the name automatic. Automatic variables are therefore private or local to the function in which they are defined.

Example 1 :

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

    auto int x;    // Defined but not initialised
    printf( " %d" , x);

}
```

Here the output of the above program will be a **garbage value** because the default initial value of an automatic variable is garbage value.

Example 2 :

```
#include < stdio.h >
main( )
{
    auto int l = 1;
    {

        auto int l = 2 ;
        {
            auto int l = 3 ;
            printf( " %d" , l );
        }
    }
}
```

```

    }
    printf(" %d" , l );
}
printf(" %d" , l );
}

```

Here the output of the above program will be : **3 2 1**

In the above program, compiler will treat three l's as the totally different variables, since they are defined in different blocks. All three l's are available to the innermost printf(). This is because the innermost printf() lies in all three blocks in which the three l's are defined. This printf() prints 3 because even if all three l's are available, but the one which is most local is given a priority. Once the control comes out of the innermost block, the variable l with value 3 is lost (dead) , hence the l in the second printf() refers to l with value 2. Similarly, when the control comes out of the next innermost block, the variable l with value 2 is lost (dead) .Hence third printf() refers to the variable l with value 1 .

Note : A variable defined inside a function without storage class specification , is by default , an automatic variable .

Example :

```

main ( )
{

    int num ;
    -----;
    -----;

}

```

Here the storage class for variable num is automatic .

20.2 Register Storage Class :

Storage :	CPU register .
Default Initial Value :	An unpredictable value called as Garbage Value .
Scope :	Local to the block in which the variable is defined.
Life :	Till the control remains within the block in which the variable is defined .

A value stored in a CPU register can always be accessed faster than the one that is stored in memory. Therefore, for frequently accessed variables, it is better to declare their storage class as register.

Example :

```

#include < stdio.h >

```

```

main( )
{
    register int l ;
    for ( l =1; l <= 10; l++ )
        printf( "%d" , l );
}

```

20.3 Static Storage Class :

Storage :	Memory .
Default Initial Value :	Zero .
Scope :	Local to the block in which the variable is defined.
Life :	Value of the variable persists until the end of the Program .

Example :

```

#include < stdio.h >
void increment( );
main( )
{
    increment( );
    increment( );
    increment( );
}

void increment( )
{
    static int x = 1;
    printf( "%d\n" , x );
    x = x + 1 ;
}

```

The output of the above program will be :

```

1
2
3

```

A static variable is initialized only once, when the program is compiled. It is never initialized again. During the first call to increment () , static variable x is initialized to 1 and it is displayed by the printf(). Then x is incremented to 2. Since x is static, this value persists and therefore, in the second call to increment() ,

printf() will display 2. Then x is incremented to 3. Since x is static, this value also persists. Therefore in the third call to increment() , printf() will display 3.

20.4 External Storage Class :

Storage :	Memory .
Default Initial Value :	Zero .
Scope :	Global.
Life :	As long as the program execution doesn't come to the end .

Variables that are both alive and active throughout the entire program are known as the external variables. They are also known as global variables. Unlike local variables, global variables can be accessed by any function in the program. External variables are declared outside a function.

Example 1 :

```
#include <stdio.h>
int x;                // Global Variable
void increment( );
void decrement( );
main( )
{
    printf("%d", x);
    increment( );
    decrement( );
}

void increment ( )
{
    x = x + 1;         // Accessing the global variable
    printf("\n%d", x);
}

void decrement ( )
{
    x = x - 1;         // Accessing the global variable
    printf("\n%d", x);
}
```

The output of the above program will be :

0
1
0

Here the global variable x is alive and available for use in all three functions i.e main() , increment() and decrement() . The default initial value of x which is 0 will be printed by the printf() inside the main() . Then during the call to the increment() , x is incremented to 1 which will be printed by the printf() inside the increment() . Then during the call to decrement() , x is decremented to 0 which will be printed by the printf() inside the decrement() .

Example 2 : In case a local variable and a global variable have the same name , the local variable will have precedence over the global one in the function where it is declared.

```
int count ;                // Global Variable
void fun( ) ;
main( )
{
    count = 10 ;
    ----- ;
    ----- ;
}

void fun( )
{
    int count = 20 ;        // Local Variable
    printf( "%d" , count ) ;
    ----- ;
    ----- ;
}
```

The output of the above code segment is 20 . In case a local variable and a global variable have the same name , the local variable will have precedence over the global one in the function where it is declared. Hence inside the fun() , printf() will print 20 (the local variable's value) .

Example 3: One other aspect of a global variable is that it is available only from the point of declaration to the end of the program.

Consider the following code segment :

```
main( )
{
    y = 5 ;
    ---- ;
    ---- ;
    ---- ;
}
```

```

int y;

fun()
{
    y = y + 1;
}

```

In the above code segment, `main()` can't access the variable `y` as it has been declared after it. So the compiler will issue an error message. This problem can be solved by declaring the variable with the storage class **extern**. For example :

```

main()
{
    extern int y;    // External Declaration
    -----;
    -----;
}
fun()
{
    extern int y;    // External Declaration
    -----;
    -----;
}

int y;              // Definition

```

Although variable `y` has been defined after both the functions, the external declaration of `y` inside the functions informs the compiler that `y` is an integer type defined somewhere else in the program.

Note : Extern declaration doesn't allocate memory for variables.

21. ARRAY : Concepts, Declaration, Initialization, Accessing elements :

21.1 Basic Concepts :

- An array is a collection of data items, all of the same type, accessed using a common name.
- A one-dimensional array is like a list; A two dimensional array is like a table; The C language places no limits on the number of dimensions in an array, though specific implementations may.
- Some texts refer to one-dimensional arrays as *vectors*, two-dimensional arrays as *matrices*, and use the general term *arrays* when the number of dimensions is unspecified or unimportant.

21.2 Declaration of an array :

- Array variables are declared identically to variables of their data type, except that the variable name is followed by one pair of square [] brackets for each dimension of the array.
- Uninitialized arrays must have the dimensions of their rows, columns, etc. listed within the square brackets.
- Dimensions used when declaring arrays in C must be positive integral constants or constant expressions.

To declare an one-dimensional array, the syntax is:

Datatype array-name[size] ;

- Data type specifies the type of data to be stored (int, float, char etc.).
- Array name must a valid identifier. Array size indicates the number of elements of the array. So array size must be of integer constant. The [] indicates that we are dealing with an array.

Example: `int group[10];` // declares the group as array capable of storing maximum 10 integers.

`char name[10];` //declares name as an array which can store at most 20 characters.

21.3 Initialization

- Arrays may be initialized when they are declared, just as any other variables.
- Place the initialization data in curly {} braces following the equals sign. Note the use of commas in the examples below.
- An array may be partially initialized, by providing fewer data items than the size of the array. The remaining array elements will be automatically initialized to zero.
- If an array is to be completely initialized, the dimension of the array is not required. The compiler will automatically size the array to fit the initialized data. (Variation: Multidimensional arrays - see below.)
- Examples:

```
int a[ 6 ] = { 1, 2, 3, 4, 5, 6 };
float sum = 0.0f, float Array[ 100 ] = { 1.0f, 5.0f, 20.0f };
double piFractions[ ] = { 3.141592654, 1.570796327, 0.785398163 };
```

21.4 Accessing Array elements:

- Elements of an array are accessed by specifying the index of the desired element within square [] brackets after the array name.

- Array subscripts must be of integer type. (int, long int, char, etc.)
- VERY IMPORTANT: Array indices start at zero in C, and go to one less than the size of the array. For example, a five element array will have indices zero through four. This is because the index in C is actually an offset from the beginning of the array. (The first element is at the beginning of the array, and hence has zero offset.)
- The most common mistake when working with arrays in C is forgetting that indices start at zero and stop one less than the array size.
- Arrays are commonly used in conjunction with loops, in order to perform the same calculations on all (or some part) of the data items in the array.
- suppose we take the example `int a={1,2,3,4,5,6}`

1	2	3	4	5	6	
508	510	512	514	516	518	520

- The above array stores 6 integers .The base address of the array is 508. Here we have assumed that integer is occupying 2 bytes. The second cell of the array therefore has the memory location 510.The next 512 and so on.

22.Two Dimensional and Multidimensional Arrays:

22.1 Two dimensional & Multidimensional arrays

- Multidimensional array is one having more than one dimension.
- Two dimensional arrays consist of two dimensions where the first dimension is commonly considered to be the number of rows, and the second dimension is the number of columns.
- `int arr[5][6]` refers to a two dimensional integer array having five rows and six columns.
- One dimensional arrays do not require the dimension to be given if the array is to be completely initialized. By analogy, multi-dimensional arrays do not require the first dimension to be given if the array is to be completely initialized. All dimensions after the first must be given in any case.
- Declaration and initialization can be done simultaneously

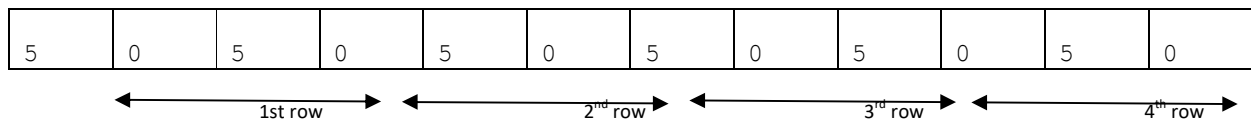
Example1: `int a[2][3] = { { 5, 6, 7 }, { 10, 20, 30 } };`

Example2: `int marks[4][3]= {35,40,45,50,55,60,65,70,75,80,85,90};`

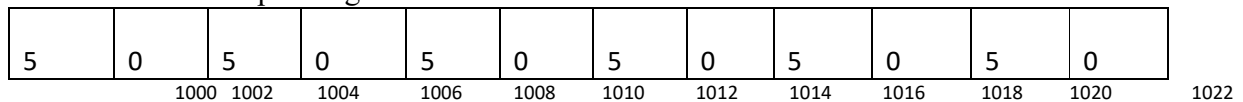
This two- dimensional array is represented by the following matrix

35	40	45
50	55	60
65	70	75
80	85	90

Matrix memory representation



Corresponding addresses



Displaying in matrix format

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

Transpose of a matrix

```
for(i=0; i < col; i++)
{
    for(j=0; j < row; j++)
        b[i][j] = a[j][i];
}
```

22.2 Array Applications

- Array is used to store the number of elements belonging to same data type.
- We can store elements to be sorted in an array and then by using different sorting technique we can sort the elements.

- Matrix operations can be performed using the array. We can use 2-D array to store the matrix.
- CPU Scheduling is generally managed by Queue. Queue can be managed and implemented using the array. Array may be allocated dynamically i.e at run time.
- When the function calls another function or the same function again then the current values are stores onto the stack and those values will be retrieving when control comes back. This is similar operation like stack. Stack can be implemented using array.

23.Programming Examples of Arrays:

Prog 1. To store first 10 natural numbers in an array and then display them.

```
#include<stdio.h>
main( )
{
    int i, arr[10];
    for (i=0; i<10; i++)
    {
        arr[i] = i+1;
        printf("%d\t", arr[i]);
    }
}
```

The above program runs a loop 10 times and then allocates the natural numbers and stores in array arr.

output: 1 2 3 4 5 6 7 8 9 10

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

prog 2. To swap elements of array: Exchanging odd and even positions

```
#include<stdio.h>
main( )
{
    int arr[12] , i , temp;
    printf("enter array elements");
    for( i=0 ; i <=5 ; i++)
        scanf("%d", &arr[i]); //array initialization
    for(i=0; i<=5; i=i+2)
    {
        temp=arr[i]; //swaping the values using a third variable
        arr[i]= arr[i+1] ;
        arr[i+1]=temp;
    }
}
```

```

    for( i=0;i <=5;i++)
        printf("\n %d", arr[i]) ; // printing the new values of the array
}

```

input:

7	1	5	8	3	9
---	---	---	---	---	---

output:

1	7	8	5	9	3
---	---	---	---	---	---

prog3. To calculate the first twenty Fibonacci numbers.

Fibonacci series: 0 1 1 2 3 5 8 11

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

```
main( )
```

```
{
```

```
    int i, fibonacci[ 20 ]; // array named Fibonacci declared
```

```
    fibonacci[ 0 ] = 0; // 1st element initialized
```

```
    fibonacci[ 1 ] = 1; // second element initialized
```

```
    for( i = 2; i < 20; i++ ) // loop runs 18 times
```

```
        fibonacci[ i ] = fibonacci[ i - 2 ] + fibonacci[ i - 1 ]; // 1st & 2nd element are added
                                                                to get 3rd
```

```
    for( i = 0; i < 20; i++ )
```

```
        printf( "Fibonacci[ %d ] = %f\n", i, fibonacci[ i ] );
```

```
}
```

24. Arrays and Functions:

To pass a one dimensional array to a function, it is sufficient to list the name of the array without any subscript and the size of the array as argument.

Example. `large (arr, n);` // where large is the name of the function , arr is the name of the array consisting of n number of elements.

In C the name of the array represents the address of the 0th element of the array which is also called the base address. By passing the name of the array, the address of the 0th element of the array is passed to the called function.

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

```
void modify(int [], int); // function prototype parameter 1 telling that 1st argument passed
                           is an integer array and second argument passed is an integer number.
```

```
main( )
```

```
{
```

```
    int i, arr[5];
```

```
    printf("enter elements");
```

```
    for(i=0 ; i<5 ; i++)
```

```

        scanf("%d", &arr[i]);
        modify(arr,5);           // function calling with arguments

        for(i=0 ; i<5; i++)
            printf("%d", arr[i]);
    }
    void modify( int arr[], int k)    // function definition
    {
        int i ;
        for(i=0 ; i < k ; i++)
            arr[i]=arr[i]+2;
    }

```

Input: enter elements

1 2 3 4 5

Output: 3 4 5 6 7

In the modify function, value of each array element is incremented by 2. Hence output is 3 4 5 6 7.

Passing two dimensional array to function:

Passing two dimensional array can be passed by only passing the array name. Function definition must indicate that array has two dimensions. Size of the second dimension must be specified. The prototype declaration should be similar to the function definition header.

Example: Addition of two matrices

```

void add (int x[][10], int y[][10], int m, int n); // function prototype arguments
main()                                              passed are indicative of two integer
{                                                  matrices shown by the presence of emty [ ]
    int a[10][10], b[10][10];                    bracket. It is mandatory to mention the
    int r ,c;                                     second dimension of the matix
    printf(" Enter row and column");
    scanf(" %d %d", &r, &c);
    printf("\n Enter the first matrix");
    input(a, r, c);
    printf("\n Enter the second matrix");
    input(b, r, c);
    add(a, b, r, c);
}
void input(int x[][10], int m, int n) // input function definition to initialize the matrix
{
    int i, j ;
    for( i=0 ; i<m ; i++ )
    {
        for(j=0 ; j< n; j++)
            scanf("%d", &x[i][j] );
    }
}

```

```

void add ( int x[10][10], int y[10][10], int m , int n) // add function to add the two matrices
{

```



```

int i , j , c[10][10];

for( i=0; i < m; i++)
{
    for(j=0; j<n ; j++)
    {
        c[i][j] = x[i][j] + y[i][j] ;
        printf(“ %d\t” , c[i][j]);
    }
    printf(“\n”);
}
}

```

Input: Enter row and column

3 3

Enter the first matrix

1 2 3

4 5 6

7 8 9

Enter the second matrix

1 2 3

4 5 6

7 8 9

Output:

2 4 6

8 10 12

14 16 18

25. String Handling:

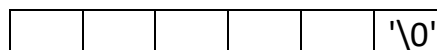
Strings

In C a string is defined as a character array which is terminated by a special character, the null character '\0', as there is no string type as such in C.

Thus the string or character array must always be defined to be one character longer than is needed in order to store the '\0' (null character).

For Example: - string to hold 5 characters

char s[6] ;



A string constant is simply a list of characters within double quotes e.g. "Hello" with the '\0' character being automatically appended at the end by the compiler.

A string may be initialized as simply as follows

```
char s[6] = "Hello" ;
```

'H'	'e'	'l'	'l'	'o'	'\0'
-----	-----	-----	-----	-----	------

as opposed to

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

Again the size specification may be omitted allowing the compiler to determine the size required.

Manipulating Strings

We can print out the contents of a string using printf() as we have seen already or by using puts().

```
printf( "%s", s ) ;  
puts( s ) ;
```

Strings can be read in using scanf() function

```
scanf( "%s", s ) ;
```

where we do not require the familiar & with the array name, as the name of an array without any index or square braces also represents the address of the array.

Example: **Write a C program that will read a string and display it.**

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

```

printf("enter a string\n");
scanf("%s",str); //& is not required when you read string

//Display the string
printf("%s", str);
}

```

Input: Hello
Output: Hello

The above way of reading string using scanf function, will not be the complete string. It will stop reading when it find a whitespace or tab.

To overcome the above problem we can use scanf by passing different arguments
scanf("%[^\n]",s); //This will start reading till getting a new line.

Example: Write a C program that will read a string and display it (String may have white space or tab.

```

#include<stdio.h>

main( )
{
    char s[20];
    printf("enter a string");

    scanf("%[^\n]",s);
    printf("%s",s);
}

```

Input: Hello World
Output: Hello World

A string can also be read in using gets()

Example: gets (s) ;

Write a C program that will read a string and count the length and number of words of the string.

```
#include<stdio.h>
main( )
{
    char str[30];
    int lenOfString, noOfWords ,k;

    printf("Enter the string\n");
    scanf("%[^\n]",s);

    lenOfString = stringLength(str);
    noOfWords= numOfWords(str);

    printf("Length = %d Number of words = %d", lenOfString, noOfWords );
}
```

//This fuction return the length of a string

```
int stringLength( char str[ ])
{
    int i=0,count=0;
    while( str[i++] !='\0')
    {
        count++;
    }
    return count;
}
```

//This fuction return the number of words in a string

```
int numOfWords( char str[ ])
{
    int count=0;

    while(str[i]!='\0')
    {
        if( str[i] == ' ' || str[i] == '\t' || str[i] == '\n')
            count++;
    }
}
```

```

    }
    return count;
}

```

WAP that will read a string and check the string is palindrome or not.

A string is known to be palindrome if it is same as the reverse of the string.

Example: **MADAM, reverse of the string is also MADAM**

/*This function will check the string is palindrome or not. It will take the first parameter as string and second parameter is the length of the string. It will return 1 if it is palindrome otherwise -1 */

```

#include<string.h>
#include<stdio.h>
int checkPalindrom(char [ ], int );

main( )
{
    char str[30];
    int len,k;

    printf("Enter the string\n");
    scanf("%[^\n]",s);

    len = strlen(str);    // compute length of the string

    k=checkPalindrom(str,len);

    if(k==1)
    {
        printf("The string is a palindrome\n");
    }
    Else
    {

```

```

        printf("The string is not a palindrome\n");
    }
}

/*This function return 1 if the string is palindrome else return -1 This function take
parameter as String and length of the String */

int checkPalindrom(char str[ ], int length)
{
    int i,j;

    i=0;j=length-1;

    while(i<j)
    {
        if(str[i] != str[j])
        {
            return -1;
        }
        else
        {
            i++;
            j--;
        }
    }
    return 1;
}

```

26. Arrays of Strings:

An array of strings is in fact a two dimensional array of characters but it is more useful to view this as an array of individual single dimension character arrays or strings.

For Example :-

```
char str_array[ 10 ][ 30 ] ;
```

where the row index is used to access the individual row strings and where the column index is the size of each string, thus str_array is an array of 10 strings each with a maximum size of 29 characters leaving one extra for the terminating null character.

For Example :- Program to read strings into str_array and print them out character by character.

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

```
char str_array[10][30] ;
```

```
main( )
```

```
{
```

```
    int i, j ;
```

```
    puts("Enter ten strings\n") ;
```

```
    for ( i = 0 ; i < 10; i++ )    // read in as strings so a single for
    {                               // loop suffices
```

```
        printf( " %d : ", i + 1 ) ;
```

```
        gets( str_array[i] ) ;
```

```
    }
```

```
    for ( i = 0; i < 10; i++ )    //printed out as individual chars so a
    {                               // nested for loop structure is required
```

```
        for ( j=0; str_array[i][j] != '\0' ; j++ )
```

```
            putchar ( str_array[i][j] ) ;
```

```
        putchar( '\n' ) ;
```

```
    }
```

```
}
```

27.Pointers:

A **pointer** is a **variable** that is used to store a **memory address**. Most commonly the address is the location of another variable in memory.

If one variable holds the address of another then it is said to point to the second variable.

Address	Value	Variable
1000		
1004	1012	ivar_ptr
1008		
1012	23	ivar
1016		

In the above illustration *ivar* is a variable of type *int* with a value 23 and stored at memory location 1012. *ivar_ptr* is a variable of type *pointer to int* which has a value of 1012 and is stored at memory location 1004. Thus *ivar_ptr* is said to point to the variable *ivar* and allows us to refer indirectly to it in memory.

NB : It should be remembered that *ivar_ptr* is a variable itself with a specific piece of memory associated with it, in this 32-bit case four bytes at address 1004 which is used to store an address.

Pointer Variables

Pointers like all other variables in C must be declared as such prior to use.

Syntax : **type *ptr ;**

which indicates that *ptr* is a pointer to a variable of type *type*. For example

```
int    *ptr ;
```

declares a pointer *ptr* to variables of type *int*.

NB : The type of the pointer variable *ptr* is *int **. The declaration of a pointer variable normally sets aside just two or four bytes of storage for the pointer whatever it is defined to point to.

In 16-bit systems two byte pointers are termed near pointers and are used in small memory model programs where all addresses are just segment offset addresses and 16 bits in length. In larger memory model programs addresses include segment and offset addresses and are 32 bits long and thus pointers are 4 bytes in size and are termed far pointers.

In 32-bit systems we have a flat address system where every part of memory is accessible using 32-bit pointers.

Pointer Operators * and &

& is a unary operator that returns the **address** of its operand which must be a variable.

For Example :-

```
int *m ;  
int count=125, i ; /* m is a pointer to int, count, i are integers  
*/  
m = &count ;
```

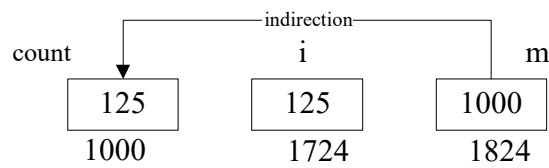
The address of the variable count is placed in the pointer variable m.

The * operator is the complement of the address operator & and is normally termed the indirection operator. Like the & operator it is a unary operator and it returns the **value** of the variable located at the address its operand stores.

For Example :-

```
i = *m ;
```

assigns the value which is located at the memory location whose address is stored in m, to the integer i. So essentially in this case we have assigned the value of the variable count to the variable i. The final situation is illustrated below.



One of the most frequent causes of error when dealing with pointers is using an uninitialised pointer. Pointers should be initialized when they are declared or in an assignment statement. Like any variable if you do not specifically assign a value to a pointer variable it may contain any value. This is extremely dangerous when dealing with pointers because the pointer may point to any arbitrary location in memory, possibly to an unused location but also possibly to a memory location that is used by the operating system. If your program tries to change the value at this address it may cause the whole system to crash. Therefore it is important to initialise all pointers before use either explicitly in your program or when defining the pointer.

A pointer may also be initialised to 0 (zero) or NULL which means it is pointing at nothing. This will cause a run-time error if the pointer is inadvertently used in this state. It is useful to be able to test if a pointer has a null value or not as a means of determining if it is pointing at something useful in a program.

NB : NULL is #defined in <stdio.h>.

For Example :-

```
int var1, var2 ;
int *ptr1, *ptr2 = &var2 ;
int *ptr3 = NULL ;
...
ptr1 = &var1 ;
```

ptr1 and ptr2 are now pointing to data locations within the program so we are free to manipulate them at will i.e. we are free to manipulate the piece of memory they point to.

28.Call by Reference:

Recall when we wanted to swap two values using a function we were unable to actually swap the calling parameters as the call by value standard was employed. The solution to the problem is to use call by reference which is implemented in C by using pointers as is illustrated in the following example.

```
#include <stdio.h>

void swap( int *, int  * ) ;

main( )
{
    int a, b ;

    printf( "Enter two numbers" ) ;
    scanf( " %d %d ", &a, &b ) ;
    printf( "a = %d ;  b = %d \n", a, b ) ;

    swap( &a, &b ) ;

    printf( "a = %d ;  b = %d \n", a, b ) ;
}
void swap ( int  *ptr1, int  *ptr2 )
{
    int temp ;

    temp = *ptr2 ;
    *ptr2 = *ptr1 ;
    *ptr1 = temp ;
}
```

The swap() function is now written to take integer pointers as parameters and so is called in main() as

```
swap( &a, &b ) ;
```

where the addresses of the variables are passed and copied into the pointer variables in the parameter list of swap(). These pointers must be de-referenced to manipulate the values, and it is values in the the same memory locations as in main() we are swapping unlike the previous version of swap where we were only swapping local data values.

In our earlier call-by-value version of the program we called the function from main() as `swap(a,b)`; and the values of these two calling arguments were copied into the formal arguments of function swap.

In our call-by-reference version above our formal arguments are pointers to int and it is the addresses contained in these pointers, (i.e. the pointer values), that are copied here into the formal arguments of the function. However when we de-reference these pointers we are accessing the values in the main() function as their addresses do not change.

29.Pointers and Arrays:

There is a very close relationship between pointer and array notation in C. As we have seen already the name of an array (or string) is actually the address in memory of the array and so it is essentially a constant pointer.

For Example :-

```
char str[80], *ptr ;
```

```
ptr = str ; /* causes ptr to point to start of string str */  
ptr = &str[0] ; /* this performs the same as above */
```

It is illegal however to do the following

```
str = ptr ; /* illegal */
```

as str is a constant pointer and so its value i.e. the address it holds cannot be changed.

Instead of using the normal method of accessing array elements using an index we can use pointers in much the same way to access them as follows.

```

char str[80], *ptr , ch;

ptr = str ;           // position the pointer appropriately

*ptr = 'a' ;          // access first element i.e. str[0]
ch = *( ptr + 1 ) ;   // access second element i.e. str[1]

```

Thus ***(array + index)** is equivalent to **array[index]**.

Note that the parentheses are necessary above as the precedence of * is higher than that of +. The expression

```
ch = *ptr + 1 ;
```

for example says to access the character pointed to by ptr (str[0] in above example with value 'a') and to add the value 1 to it. This causes the ASCII value of 'a' to be incremented by 1 so that the value assigned to the variable ch is 'b'.

In fact so close is the relationship between the two forms that we can do the following

```

int x[10], *ptr ;
ptr = x ;
ptr[4] = 10 ;  /* accesses element 5 of array by indexing a
pointer */

```

30.Pointer Arithmetic:

Pointer variables can be manipulated in certain limited ways. Many of the manipulations are most useful when dealing with arrays which are stored in contiguous memory locations. Knowing the layout of memory enables us to traverse it using a pointer and not get completely lost.

- **Assignment**

```
int count, *p1, *p2 ;
p1 = &count ;          // assign the address of a variable directly
p2 = p1 ;              // assign the value of another pointer variable, an
                        address
```

Addition / Subtraction

The value a pointer holds is just the address of a variable in memory, which is normally a four byte entity. It is possible to modify this address by integer addition and subtraction if necessary. Consider the following we assume a 32-bit system and hence 32-bit integers.

<code>int *ptr ;</code>		Address	Value
<code>int array[3] = { 100,</code>	<code>ptr</code>	1000	2008
<code>101,102 } ;</code>		①	①
<code>ptr = array ;</code>	<code>array[0]</code>	2008	100
	<code>array[1]</code>	2012	101
	<code>array[2]</code>	2016	102

We now have the pointer variable `ptr` pointing at the start of `array` which is stored at memory location 2008 in our illustration. Since we know that element `array[1]` is stored at address 2012 directly after element `array[0]` we could perform the following to access its value using the pointer.

```
ptr += 1 ;
```

This surprisingly will cause `ptr` to hold the value 1012 which is the address of `array[1]`, so we can access the value of element `array[1]`. The reason for this is that `ptr` is defined to be a pointer to type `int`, which are four bytes in size on a 32-bit system. When we add 1 to `ptr` what we want to happen is to point to the **next integer** in memory. Since an integer requires four bytes of storage the compiler increments `ptr` by 4. Likewise a pointer to type `char` would be incremented by 1, a pointer to `float` by 4, etc.

Similarly we can carry out integer subtraction to move the pointer backwards in memory.

```
ptr = ptr - 1 ;
ptr -= 10 ;
```

The shorthand operators ++ and -- can also be used with pointers. In our continuing example with integers the statement ***ptr++*** ; will cause the address in ptr to be incremented by 4 and so point to the next integer in memory and similarly ***ptr--*** ; will cause the address in ptr to be decremented by 4 and point to the previous integer in memory.

NB : Two pointer variables may not be added together (it does not make any logical sense).

```
char *p1, *p2 ;  
p1 = p1 + p2 ; /* illegal operation */
```

Two pointers may however be subtracted as follows.

```
int *p1, *p2, array[3], count ;  
p1 = array ;  
p2 = &array[2] ;  
count = p2 - p1 ;      /* legal */
```

The result of such an operation is not however a pointer, it is the number of elements of the base type of the pointer that lie between the two pointers in memory.

Comparisons

We can compare pointers using the relational operators ==, <, and > to establish whether two pointers point to the same location, to a lower location in memory, or to a higher location in memory. These operations are again used in conjunction with arrays when dealing with sorting algorithms etc.

For Example :- Writing our own version of the puts() standard library function.

1. Using array notation

```
void puts( const char s[ ] )    /* const keyword makes string contents read only */  
{  
    int i ;  
    for ( i = 0; s[i] ; i++ )  
        putchar( s[i] ) ;  
    putchar( '\n' ) ;  
}
```

```
}
```

2. Using pointer notation

```
void puts( const char *s ) { // char *const s would make pointer unalterable
while ( *s )
    putchar( *s++ );
    putchar( '\n' );
}
```

As you can see by comparing the two versions above the second version using pointers is a much simpler version of the function. No extra variables are required and it is more efficient as we will see because of its use of pointer indirection.

For Example :- **Palindrome program using pointers.**

```
#include <stdio.h>

int palin( char * ); /* Function to determine if array is a palindrome. returns 1 if
                      it is a palindrome, 0 otherwise */

main( )
{
    char str[30], c ;

    puts( "Enter test string" );
    gets( str );

    if ( palin( str ) )
        printf( "%s is a palindrome\n", str );
    else
        printf( "%s is not a palindrome\n" );
}
```



```

//Function definition
int palin ( char *str )
{
    char *ptr ;
    ptr = str ;
    while ( *ptr )
        ptr++ ;          /* get length of string i.e. increment ptr while *ptr != '\0' */
    ptr-- ;              /* move back one from '\0' */

    while ( str < ptr )
        if ( *str++ != *ptr-- )
            return 0 ;          /* return value 0 if not a palindrome */

    return 1 ;          /* otherwise it is a palindrome */
}

```

31.Strings and pointers :

Strings can be initialised using pointer or array notation as follows

```

char *str = "Hello\n" ;
char string[] = "Hello\n" ;

```

in both cases the compiler allocates just sufficient storage for both strings.

C's standard library string handling functions use pointers to manipulate the strings. For example the prototype for the strcmp() function found in <string.h> is

```

int strcmp( const char *string1, const char *string2 ) ;

```

where `const` is a C keyword which locks the variable it is associated with and prevents any inadvertent changes to it within the function.

String Library: **String.h**

The following function are predefined function of string library.

1. **size_t strlen(const char *str)**

Computes the length of the string `str`. i.e `char str[10]="Hello"; printf("%d",strlen(str));`

Output: 5

2. **char *strcpy(char *dest, const char *src)**

Copies the string pointed to, by `src` to `dest`.

Example:

```
char str1[10]="Hello";
```

```
char str2[10];
```

```
strcpy(str2,str1);
```

```
printf("%s", str2);
```

Output: Hello

3. **int strcmp(const char *str1, const char *str2)**

Compares the string pointed to, by `str1` to the string pointed to by `str2`. It returns 0 if the string are equal otherwise it returns the ASCII value difference of the first non matching character.

Example:

```
char str1[10]="Hello";
```

```
char str2[10]="Hello";
```

```
int k= strcmp(str1,str2);
```

```
printf("%d",k);
```

Output: k=0;

Example:

```
char str1[10]="Hello";
```

```
char str2[10]="Hgllo";
```

```
int k= strcmp(str1,str2);
```

```
printf("%d",k);
```

Output: k=-2; ASCII value of e - ASCII value of g

4. char *strcat(char *dest, const char *src)

Appends the string pointed to, by *src* to the end of the string pointed to by *dest*. Example:

```
char src[10]="Hello";
```

```
char dest[20]="World"
```

```
strcat(dest,src);
```

```
printf("%s",dest);
```

Output: HelloWorld

5. char *strncpy(char *dest, const char *src, size_t n)

Copies up to n characters from the string pointed to, by *src* to *dest*.

6. int strncmp(const char *str1, const char *str2, size_t n)

Compares at most the first n bytes of str1 and str2. Return 0 for match otherwise return the ASCII value difference of first no matching character

7. char *strchr(const char *str, int c)

Searches for the first occurrence of the character c (an unsigned char) in the string pointed to, by the argument str.

32. Array of Pointers, Pointer to Array :-

32.1 Array of Pointers-

- Like array of integers, array of characters, we can also have array of pointers.
- So array of pointers is nothing but a collection of addresses. These addresses can be addresses of isolated variables or addresses of array elements or any other addresses.

Example - 1:

```
main()
{
    int *arr[3];
    int i=3,j=5,k=10;
    arr[0]=&i;
    arr[1]=&j;
    arr[2]=&k;
    for(m=0;m<3;m++)
        printf("%d\t",*(arr[m]));
}
```

Output: 3 5 10

Example - 2:

```
main()
{
    int a[]={0,1,2,3,4};
    int *p[]={a,a+1,a+2,a+3,a+4};
    printf("\n%u\t%u\t%u\t%u",p,*p,**p);
}
```

Output: Base address of array p Base address of array a 0

32.2 Pointer to an array:

- A pointer pointing to an array is known as pointer to an array.
- **Declaration:**
Syntax: datatype (*pointervar)[sizeofthearray];
Example: int (*p)[2];

➤ **Example - 1:**

```
main()
{
    int arr[3][2]={ {1234,51},{1235,52},{1236,53}};
    int (*p)[2];    //p is a pointer to an array of two integers.
    int i,j,*pint;   //pint is a normal integer pointer.
    for(i=0;i<3;i++)
    {
        p=&arr[i];
        pint=p;
        for(j=0;j<2;j++)
            printf("%d\t",*(pint+j));
        printf("\n");
    }
}
```

Output:

```
1234  51
1235  52
1236  53
```

➤ **Example - 2:**

```
void print(int (*q)[4],int row,int col)
main()
{
    int a[3][4]={
        1, 2, 3, 4,
        5, 6, 7, 8,
        9, 0, 1, 6
    };
    print(a,3,4);
}
void print(int (*q)[4],int row,int col)
{
    int i,j;
    int *p;
    for(i=0;i<row;i++)
    {
        p=q+i;
        for(j=0;j<col;j++)
            printf("%d\t",*(p+j));
        printf("\n");
    }
}
```

Output:

```
1      2      3      4
5      6      7      8
9      0      1      6
```

33. Pointer as Function arguments, Function returning pointer :-

33.1 Pointer as function argument:

- C programming language allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.
- Following a simple example where we pass an integer pointer to a function and change the value inside the function which reflects back in the calling function:

```
#include <stdio.h>
#include <time.h>
void Change(int *);

int main ()
{
    int x=5;

    printf("Before change: x= %d\n", x );
    Change(&x);
    printf("After change: x= %d\n", x );
}

void Change(int *p)
{
    *p = 10;
}
```

When the above code is compiled and executed, it produces the following result:

Before change: x= 5

After change: x= 10

- The function, which can accept a pointer, can also accept an array as shown in the following example:

```
#include <stdio.h>
/* function declaration */
double getAverage(int *arr, int size);

int main ()
{
    /* an int array with 5 elements */
    int balance[5] = {1000, 2, 3, 17, 50};
    double avg;
    /* pass pointer to the array as an argument */
}
```

```

    avg = getAverage( balance, 5 );

    /* output the returned value */
    printf("Average value is: %f\n", avg );

    return 0;
}

double getAverage(int *arr, int size)
{
    int i, sum = 0;
    double avg;

    for (i = 0; i < size; ++i)
    {
        sum += arr[i];
    }

    avg = (double)sum / size;

    return avg;
}

```

When the above code is compiled together and executed, it produces the following result:
Average value is: 214.40000

33.2 Function returning pointers:

- Like a pointer returns an int, float, double it can also return a pointer. If a function returns an address it is known as function returning pointers.
- To make a function return a pointer it has to be explicitly mentioned in the calling function as well as in the function definition.
- **Example:**

```

main()
{
    char *str;
    char *copy(char *,char *);
    char source[]="Silicon";
    char target[10];
    str=copy(target,source);
    printf("%s",str);
}

char *copy(char *t,char *s)
{
    char *r;
    r=t;
}

```

```

while(*s!='\0')
{
    *t = *s;
    t++;
    s++;
}
*t = '\0';
return r;
}

```

Output: Silicon

34. Pointer to Function, Limitations of Pointers :-

34.1 Pointer to function:

- A pointer pointing to a function is known as pointer to function.
- So by using pointer to function we can invoke that function in another way.
- We can assign the address of the function to the pointer to function just like assigning the base address of an array to a function.

➤ **Declaration:**

Syntax: returntype (*functionpointe)(datatype of the arguments);

Example: **int (*p)(int,char);**

➤ **Example:**

```

main()
{
    void display();
    void (*p)();
    p=display; //assigning the address of the function to the pointer to function
    (*p)();    // calling the function display through pointer to function p.
}
void display()
{
    printf("Hello");
}

```


Output: Hello

➤ **Uses of pointer to function:**

- In writing memory resident programs
- In writing viruses, or vaccines to remove the viruses.

Advantages and disadvantages of pointers in c :

Benefits (use) of pointers in c:

- Pointers provide direct access to memory
- Pointers provide a way to return more than one value to the functions
- Reduces the storage space and complexity of the program
- Reduces the execution time of the program
- Provides an alternate way to access array elements
- Pointers can be used to pass information back and forth between the calling function and called function.
- Pointers allows us to perform dynamic memory allocation and deallocation.
- Pointers helps us to build complex data structures like linked list, stack, queues, trees, graphs etc.
- Pointers allows us to resize the dynamically allocated memory block.
- Addresses of objects can be extracted using pointers

Limitations of pointers in c:

- Uninitialized pointers might cause segmentation fault.
- Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to memory leak.
- Pointers are slower than normal variables.
- If pointers are updated with incorrect values, it might lead to memory corruption.

Basically, pointer bugs are difficult to debug. It is programmer's responsibility to use pointers effectively and correctly.

35. Dynamic Memory Management:-

- Dynamic memory management technique permits us to allocate additional memory space or to release unwanted space at run time. So it is the process of allocating/deallocating memory at run time.

Static Memory Allocation

1. Memory is allocated at the time of compilation (before executing)
2. Insertion is not always possible.
3. Deletion of elements will waste the memory.

Dynamic Memory Allocation

1. Memory is allocated at the time of execution (during runtime).
2. Insertion is always possible if there is enough free memory space.
3. No wastage of memory in deletion.

Functions to allocate/deallocate memory dynamically:

➤ **malloc():**

- This function reserves a single block of memory (contiguous bytes) of specified size and returns a pointer of type void.
- If it is successful it returns a pointer (of the cast type) to the allocated memory else it returns **NULL**.
- The allocated memory will be filled with garbage values.
- **Syntax:** ptr=(cast type *)malloc(byte-size);
Example: ptr=(int *)malloc(100);

○ **Example Program:**

```
main()
{
    int *p,n,i;
    printf("Enter the no of elements:");
    scanf("%d",&n);
    p=(int *)malloc(n*sizeof(int));
    for(i=0;i<n;i++)
    {
        printf("Enter a no:");
        scanf("%d",p+i);
    }
    for(i=0;i<n;i++)
    {
        printf("%d",*(p+i));
    }
}
```

➤ **calloc():**

- This function reserves multiple blocks of memory (contiguous bytes) of specified size and returns a pointer of type void.
- If it is successful it returns a pointer (of the cast type) to the allocated memory else it returns **NULL**.
- The allocated memory will be filled with zeros.
- **Syntax:** `ptr=(cast type *)calloc(n,ele-size);`
- **Example:** `ptr=(int *)calloc(10,sizeof(int));`

➤ **free():**

- This function deallocates (frees) the allocated memory.
- **Syntax:** `free(pointervar);`
- **Example:** `free(ptr);` // This frees the memory whose address is at ptr.

➤ **realloc():**

- This function allocates a new memory space.
- **Example:** `ptr=malloc(size);` // allocating a memory of size **size**
 `ptr=realloc(ptr,newsize);`//reallocation of memory of size **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 new memory block may or may not begin at the same place as the old one.
- In case it is not able to find additional space in the same region, it will create the same in an entirely new region and move the contents of the old block into the new block.
- If the function is unsuccessful in locating additional space, it returns a NULL pointer and the original block is freed (lost).

Memory Leak:

- A memory leak is memory which hasn't been freed, there is no way to access (or free it) now, as there are no ways to get to it anymore. (E.g. a pointer which was the only reference to a memory location dynamically allocated (and not freed) which points somewhere else now.)

```
void func()
{
    char *ch;
    ch = (char*) malloc(10);
}
```

In the above function `ch` is not valid outside, no way to access malloc-ed memory
The char pointer `ch` is a local variable that goes out of scope at the end of the function, leaking the dynamically allocated **10 bytes**.

37.Command Line Arguments :-

Command Line Argument:

- It is a parameter supplied to a program when the program is invoked.
- **Example:** ./a.out 2 4
- Here we want to execute a program whose executable code is there in “**a.out**”. This program takes some numbers as input and finds the sum of these numbers. But in spite of taking the input from the user the program takes the input from the command line. Here the inputs are 2 & 4.
- To provide these inputs to the program we need a special type of main of the following structure:

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

- The variable **argc** is an argument counter that counts the number of arguments on the command line.
- The variable **argv** is an argument vector and represents an array of character pointers that points to the command line arguments. The size of this array will be equal to the value of **argc**.
- In the above example the value of **argc** is 3. **argv[0]**, **argv[1]**, **argv[2]** contains ./a.out, 2 & 4 respectively.

- **Example:**

```
main(int argc, char *argv[])
{
    int sum=0, n, i;
    for(i=1; i<argc; i++)
    {
        n=atoi(argv[i]);
        sum+=n;
    }
    printf("Sum=%d", sum);
}
```

Run: ./a.out 2 4

Output: Sum=6

38. Derived Type : Structures :

Array can be used to store a list of data items of same data type and can be accessed through one name. Sometimes it is necessary to store a list of data items of different data type that is needed to be accessed through one name. For example, storing the details (title: string, author: string, page:

integer, price: real) of a book. In such cases array cannot fulfill the requirements. To fulfill this requirement we need to define a different data type to represent book with all data items related to it. This is possible in C by using *structure*.

Structure Definition:-

Structure is a user defined data type, which is defined as a collection of one or more variables possibly of heterogeneous data type. Structures help to organize complicated data by permitting a group of related variables to be treated as a unit instead of as separate entities.

38.1 Defining a Structure:-

Syntax:

```
struct structure_name
{
    data_type variable_name1;
    data_type variable_name2;
    -----;
    -----;
};
```

The keyword **struct** declares a structure to hold the details of multiple data fields. These fields are called structure elements or members. Each member may belong to a different type of data.

Example:

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

Here the keyword **struct** declares a structure **book** to hold the details of four data fields, namely **title** , **author** , **pages** and **price**.

38.2 Declaring Structure Variables:-

After defining a structure format we can declare variables of that type. A structure variable declaration is similar to the declaration of variables of any other data types.

Syntax:

```
struct structurename variable_name1,variable_name3,variable_name3,...;
```

For example, the statement **struct book b1,b2,b3;** declares b1, b2, and b3 as variables of type **struct book**.

Note: It is also allowed to combine both the structure definition and variables declaration in one statement as follows :

```
struct book
{
    char title[20];
```

```

char author[15];
int pages;
float price;
} b1, b2, b3;

```

Note : A structure declaration/ definition statement never allocates any memory rather whenever a structure variable is declared then memory will be allocated.

38.3 Accessing Structure Members:-

Structure members can be accessed using ‘.’ and is called **member operator or dot operator** .

```

Syntax :    structure_variable . structure_element ;
Example:    b1.price=120.50 ;    // represents price of b1
            b1.pages=250 ;      // represents pages of b1

```

We can also use scanf() to enter values through the keyboard.

```

scanf( "%d", &b1.pages ) ;
scanf( "%f", &b1.price ) ;

```

38.4 Structure Initialization:-

Like any other data type structure variable can also be initialized at compile time.

Example: **struct book** b3 = {"C programming", "Balagurusamy", 550, 300.50};

39. Copy and Comparing Structure Variables, Nested Structures :

39.1 Copying and Comparing Structure Variables:-

The content of one structure variable can be assigned to another structure variable of the same type by two ways :

1. copying all elements at one go using assignment operator.
2. piece-meal copy (copy element by element basis)

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

Example: **copy at one go :**

```

struct book b5, b6;           // structure variables b5 and b6 are declared
b5 = {"C programming", "Balagurusamy", 550, 300}, b6; // b5 is initialized
b6 = b5;                      // entire content of b5 will be copied to b6.

```

Example : **piece-meal copy :**

```

struct book b5, b6;           // structure variables b5 and b6 are declared
b5 = {"C programming", "Balagurusamy", 550, 300}, b6; // b5 is initialized
strcpy( b6.title, b5.title );
strcpy( b6.author, b5.author );
b6.pages = b5.pages ;
b6.price = b5.price ;

```

Note: The comparison between two structure variables is **not possible** using relational operators.
To compare structure variables we need to compare manually individual structure members.

39.2 Nested Structure:-

Structures within a structure mean nesting of structures. Nesting of structures is permitted in C. There are two ways of defining nested structure.

1. Structure definition with in structure definition
2. Structure variable inside structure definition

1. Structure definition with in structure definition

Example:

```
struct book
{
    char title[20];
    struct author
    {
        char name[20];
        char affiliation[100];
    };
    int pages;
    float price;
};
```

2. Structure variable inside structure definition

Example:

```
struct author
{
    char name[20];
    char affiliation[100];
};
struct book
{
    char title[20];
    struct author Auth;
    int pages;
    float price;
};
```

Accessing nested structure members:-

Example:

```
struct book b1;  
strcpy(b1.Auth.name,"Balagurusamy"); // dot operator is used twice
```

40. Array of Structures :

Array of Structures:-

An array of structures where each element of the array represents a structure variable.

Declaration syntax:

```
struct structurename arrayname [size] ;
```

Example: **struct book b[10];** Defines an array called b that consists of 10 elements. Each element is defined to be of the type struct book.

Accessing data members:

```
arrayname [index].membername=value;
```

Example: **b[3].price=300;**

Assignment: Define a structure student with elements as name, roll no and total mark. WAP to input the details of 5 students and display the details.

```
#include <stdio.h>  
main()  
{  
    int i;  
    struct student  
    {  
        char name[15];  
        int roll;  
        int mark;  
    };  
    struct student s[5];  
    printf(" ENTER THE NAME, ROLL NO AND TOTAL MARK");  
    for( i=0; i<5; i++)  
        scanf(" %s%d%d", s[i].name, s[i].roll, s[i].mark);  
    printf("\n DETAILS OF THE STUDENTS ARE");  
    for( i=0; i<5; i++)  
        printf("%s\t%d\t%d\n", s[i].name, s[i].roll, s[i].mark);  
}
```


41. Structure and Function, Pointer to Structure :-

41.1 Structure and Function:

C supports the passing of structure values as arguments to functions. The general format of sending a copy of a structure to the called function is:

Function_name(structure_variable_name);

The called function takes the following forms:

```
Return_type function_name (struct_type structure_name)
{
    .....
    .....
    Return (expression);
}
```

Note: the return type can also be a structure type.

41.2 Pointer and Structure:-

Address of structure variables can be assigned to structure pointers of same type. We can also access structure members through pointers. To access structure members through pointers we have to use arrow operator (->).

Example:

```
struct book b, *p;
P=&b;
p->price=300;
```

Self Referential Structure:-

A structure definition can have a pointer of its own type as a member. This type of structure is called self referential structure. This type of structure is helpful to create linked data storage of multiple element of same type of structures.

Example:

```
struct node
{
    int v;
    struct node *next;
};
main()
{
    struct node n1,n2;
    n1.v=5;
    n2.v=10;
    n1.next=&n2;
    printf("n2 value=%d",n1.next->v);
}
```

42. Unions :-

A *union* is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements. Unions provide a way to manipulate different kinds of data in a single area of storage, without embedding any machine-dependent information in the program.

The declaration of union is similar to structure declaration with different semantics.

Union definition:

```
union unionname
{
    ...
    ...
};
```

Example:

```
union item
{
    int x;
    float y;
    char name[20];
};
```

Union variable declaration:

```
union unionname variablename;
```

Example: **union item** a, b ;

Note: The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. We can use only one of them at a time.

Initialization:

```
union item a;
a.x=50;
a.y=123.3;
printf("%d",a.y);    // this will print 123.3
printf("%d",a.x);    // this will give garbage value
```

Note: Union variables can only be initialized with first union member type data at the time of declaration. We cannot initialize other data members at the time of declaration.

Example:

```
union item a={50}; // valid
union item b={123.1}; //invalid
```

43. Bit Fields , Typedef :-

43.1 Bit Fields :

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called status, as follows:

```
struct status
{
    unsigned int widthValidated;
    unsigned int heightValidated;
};
```

This structure requires 8 bytes of memory space but in actual we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situation. If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those numbers of bytes. For example, above structure can be re-written as follows:

```
Struct status
{
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
};
```

Now, the above structure will require 4 bytes of memory space for status variable but only 2 bits will be used to store the values. If you will use up to 32 variables each one with a width of 1 bit, then also status structure will use 4 bytes, but as soon as you will have 33 variables, then it will allocate next slot of the memory and it will start using 8 bytes. Let us check the following example to understand the concept:

Bit Field Declaration:

The declaration of a bit-field has the following form inside a structure:

```
struct
{
    type [member_name1] : width ;
    type [member_name2] : width ;
    type [member_name3] : width ;

};
```

Below the description of variable elements of a bit field:

Elements	Description
type	An integer type that determines how the bit-field's value interpreted. The type may be int, signed int, unsigned int.
member name	The name of the bit-field.
width	The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

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

```
Struct A
{
    unsigned int age : 3;
};
```

The above structure definition instructs C compiler that age variable is going to use only 3 bits to store the value; if you will try to use more than 3 bits then it will not allow you to do so. Let us try the following example:

```
main()
{
    struct A Age;
    Age.age = 4;
    printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
    printf( "Age.age : %d\n", Age.age );
    Age.age = 7;
    printf( "Age.age : %d\n", Age.age );
    Age.age = 8;
    printf( "Age.age : %d\n", Age.age );
}
```

When the above code is compiled it will compile with warning and when executed, it produces the following result:

```
Sizeof( Age ) : 4
Age.age :      4
Age.age :      7
```

Age.age : 0

43.2 Typedef:-

C provides a facility called **typedef** for creating new data type names. For example, the declaration

```
typedef int Length;
```

makes the name Length a synonym for **int**. The type Length can be used in declarations in exactly the same ways that the **int** type can be: Length a,b;

As a more complicated example, we could make **typedef** for the structure declaration in either of the following two forms.

```
typedef struct book Book;  
Book b1;
```

Or

```
typedef struct book  
{  
    ...  
    ...  
}  
Book;  
Book b1;
```

44. File Handling in C :

Concept of file

Why we need File?

We use printf and scanf to write and read data. These are console oriented I/O functions. The console input device is keyboard and console output device is monitor. The console oriented I/O operations have two major drawbacks.

1. It becomes difficult and time consuming to handle large volumes of data through keyboard
2. The entire data is lost when the program is terminated

We frequently use files for storing information which can be processed by our programs. In order to store information permanently and retrieve it when we need.

A file is a place on the disk where a group of related data is stored. Files are not only used for data. Our programs are also stored in files. The editor which you use to enter your program and save it, simply manipulates files for you. The Unix commands cat, cp, cmp are all programs which process your files.

Data File manipulation:

C supports a number of function that have the ability to perform basic file operations, these are:

- Opening a file
- Reading data from a file
- Writing data to a file
- Closing a file

The standard library functions supported by C programming are given in the table below. To use these function we need to include header file `stdio.h`

Function Name	Operations
<code>fopen()</code>	Create a new file or open an existing file
<code>fclose()</code>	Closing a file
<code>getc()</code>	Reading a character from a file
<code>putc()</code>	Writing a character to a file
<code>fprintf()</code>	Reading a set of data from a file
<code>fscanf()</code>	Writing a set of data to a file
<code>getw()</code>	Reading an integer from a file
<code>putw()</code>	Writing an integer to a file
<code>fseek()</code>	Set the file pointer to desired position
<code>ftell()</code>	Give the current position of the file pointer
<code>rewind()</code>	Set the file pointer to beginning of the file

Opening a File:

We will see that file I/O is almost identical to the terminal I/O that we have been using so far. The primary difference between manipulating files and doing terminal I/O is that we must specify in our programs which files we wish to use. As you know, you can have many files on your disk. If you wish to use a file in your programs, then you must specify which file or files you wish to use.

The general syntax of opening the file is

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

Where `FILE` is a pre defined structure which is already defined in `stdio.h`. `fp` is pointer of data type `FILE`. `fopen()` returns the address of this structure. `Filename` is string of characters that make up a valid filename for the operating system. `Mode` specifies the purpose of opening a file. `Mode` can be on the following:

Mode	Purpose
<code>r</code>	Open the file for reading only
<code>w</code>	Open the file for writing only
<code>a</code>	Open the file for adding data to it
<code>r+</code>	Open existing file for both read and write

w+	Same as w but both reading and writing
a+	Same as a but both reading and writing

fopen() is used to establish buffer for the file to be opened.

NOTE: Because you may use a number of different files in your program, you must specify when reading or writing which file you wish to use. This is accomplished by using a variable called a **file pointer**.

Every file you open has its own file pointer variable. When you wish to write to a file you specify the file by using its file pointer variable.

You declare these file pointer variables as follows:

```
FILE *fp1, *fp2, *fp3;
```

The variables fp1, fp2, fp3 are file pointers. You may use any name you wish.

The file <stdio.h> contains declarations for the Standard I/O library and should always be **included** at the very beginning of C programs using files.

Constants such as FILE, EOF and NULL are defined in <stdio.h>.

You should note that a file pointer is simply a variable like an integer or character.

It does **not** *point* to a file or the data in a file. It is simply used to indicate which file your I/O operation refers to.

The function **fopen** is one of the Standard Library functions and returns a file pointer which you use to refer to the file you have opened e.g.

```
fp = fopen( "prog.c", "r" );
```

The above statement **opens** a file called prog.c for **reading** and associates the file pointer fp with the file prog.c.

When we wish to access this file for I/O, we use the file pointer variable fp to refer to it.

RULE: Always check when opening files, that fopen succeeds in opening the files appropriately.

If the file **cannot be opened**, fopen() returns a **NULL** pointer. Thus by checking the file pointer returned by fopen, you can determine if the file was opened correctly and take appropriate action e.g.

```
fp = fopen (filename, "r") ;
```

```
if ( fp == NULL)
{
```

```
    printf("Cannot open %s for reading \n", filename );
    exit(1) ; /*Terminate program: Commit suicide !!*/
```

```
}
```

Closing the file :

The function **fclose** is used to *close* the file i.e. we are finished processing this file.

fclose (fp);

once the file is closed the buffer associated with the file is removed from the memory. Once file is closed you can not do any operation on that file unless the file is reopened.

45. Input and operation on files: standard I/O

Reading from a file :

Once the file has been opened for reading using `fopen()` then file's content are brought into the buffer and file pointer is set to the first character in the buffer. A predefined function `fgetc()` is used to read the content of the file from memory. Thus the statement

`c = fgetc(fp);`

reads the next character from the file referenced by file pointer `fp` .

note:- `fgetc()` is used to read a character from a file that has opened in read mode

Example 45.1- WAP to display contents of file on the screen

```
/* file.c: Display contents of a file on screen */

#include <stdio.h>
main()
{
    FILE *fp;    //declaration of file pointer
    char c ;
    fp = fopen( "prog.c", "r" );    //open a file prog.c

    //read content of the file until end of file has been reached
    while ( (c=fgetc(fp)) != EOF )
    {
        putchar( c );
    }
    fclose( fp );
}
```

OUTPUT-

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



```

        printf("Hello");
    }

```

If prog.c content following information.

In this program, we open the file prog.c for reading. Prog.c must exist in the present working directory

We then read a character from the file.

If the file is empty, we are at the end, so getc returns EOF a special value to indicate that the end of file has been reached.

The while loop simply keeps reading characters from the file and displaying them, until the end of the file is reached.

This program is in effect a special purpose cat command. It displays file contents on the screen, but **only** for a file called prog.c.

Example 45.2- WAP to count no of character in a file

```

/*count.c : Count characters in a file*/
#include <stdio.h>
/* Prompt user for file and count number of characters and lines in it*/

main()
{
    FILE *fp;
    int nc, nlines;
    char filename[40],c ;

    nlines = 0 ;
    nc = 0;

    printf("Enter file name: ");
    gets( filename );

    fp = fopen( filename, "r" );

    if ( fp == NULL )
    {
        printf("Cannot open %s for reading \n", filename );
        exit(1);    /* terminate program */
    }

    while ( ( c =getc(fp))!=EOF )
    {
        if ( c == '\n' )
            nlines++;
    }
}

```

```

        nc++;
    }

    fclose( fp );

    if ( nc != 0 )
    {
        printf("There are %d characters in %s \n", nc, filename );

        printf("There are %d lines \n", nlines );
    }
    else
        printf("File: %s is empty \n", filename );
}

```

The function **exit()** is a special function which terminates your program immediately. **exit(0)** mean that you wish to indicate that your program terminated successfully whereas a nonzero value means that your program is terminating due to an error condition.

Writing to a file :

To write to a file, the file must be opened for writing e.g.

```
fp = fopen( fname, "w" );
```

If the file does not exist already, it will be created. If the file does exist, it will be overwritten! So, be careful when opening files for writing, in case you destroy a file unintentionally. Opening files for writing can also fail. If you try to create a file in another users directory where you do not have access you will not be allowed and **fopen** will fail.

putc() is used to write character to a file opened in write mode.

```
putc(c,fp);
```

the above statement writes the character **c** into file referenced by **fp**.

Example 45.3- WAP to read data from keyboard write it a file called input again read the data from the input file and display it on the monitor.

```

#include<stdio.h>
main( )
{
    FILE *fp;
    char c;
    fp=fopen("input","w");
    while((c=getchar())!=EOF)
    {

```

```

        //write char to inputfile
        putc(c,fp);
    }

    //close the file input
    fclose(fp);
    //reopen
    fp=fopen("input","r");
    //read from file input
    while((c=getc(fp))!=EOF)
    {
        //display character from file
        printf("%c",ch);
    }
    fclose(fp)

}

```

OUTPUT-

```

a
b
c
ctrl+z
a
b
c

```

Example 45.4-: Write a file copy program which copies the file “prog.c” to “prog.old”

```

/* filecopy.c : Copy prog.c to prog.old */

#include <stdio.h>
void main( )
{
    FILE *fp1, *fp2;
    char c ;

    fp1 = fopen( "prog.c", "r" );    /* open for reading */
    fp2 = fopen( "prog.old", "w" );  /* open for writing */

    if ( fp1 == NULL )    /* check does file exist etc */
    {
        printf("Cannot open prog.c for reading \n" );
        exit(1);    /* terminate program */
    }
    else if ( fp2 == NULL )
    {
        printf("Cannot open prog.old for writing \n");
    }
}

```

```

        exit(1);  /* terminate program */
    }

    else          /* both files O.K. */
    {
        while ( (c=getc(fp1)) != EOF)
        {
            putc( c,  fp2);  /* copy to prog.old */

        }

        fclose ( fp1 );      /* Now close files */
        fclose ( fp2 );
        printf("Files successfully copied \n");
    }
}

```

Example 45.5: Write a program to compare two files specified by the user, displaying a message indicating whether the files are identical or different.

```

/* compare.c : compare two files */
#include <stdio.h>
void main( )
{
    FILE *fp1, *fp2, *fopen();
    int ca, cb;
    char fname1[40], fname2[40] ;

    printf("Enter first filename:") ;
    gets(fname1);
    printf("Enter second filename:");
    gets(fname2);
    fp1 = fopen( fname1, "r" );    /* open for reading */
    fp2 = fopen( fname2, "r" );    /* open for writing */
    if ( fp1 == NULL )    /* check does file exist etc */
    {
        printf("Cannot open %s for reading \n", fname1 );
        exit(1);  /* terminate program */
    }
    else if ( fp2 == NULL )
    {
        printf("Cannot open %s for reading \n", fname2 );
        exit(1);  /* terminate program */
    }
    else          /* both files opened successfully */
    {
        ca =  getc( fp1 ) ;
        cb =  getc( fp2 ) ;
    }
}

```

```

while ( ca != EOF  &&  cb != EOF  &&  ca == cb )
{

    ca = getc( fp1 );
    cb = getc( fp2 );
}
if ( ca == cb )
    printf("Files are identical \n");
else if ( ca != cb )
    printf("Files differ \n" );
fclose ( fp1 );
fclose ( fp2 );
}
}

```

46. Input and operation on files: standard I/O

The fprintf and fscanf predefined functions are identical to printf and scanf functions except that they work on files. The first argument of these functions is a file pointer which specifies the file to be used.

fprintf is used to write to a file. The general form of fprintf is

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

Where fp is a file pointer associated with a file that has been opened for writing. The control string is file output specifications list may include variable, constant and string.

```
fprintf(f1,"%s%d%f",name,age,y);
```

In this above example name is an string, age is an int variable and y is float variable

fscanf is used to read data from the file. The general format of fscanf is

fscanf(fp,"controlstring",list);

```
fscanf(fp,"%d",&x);
```

Example 46.1- WAP to write 1 to 30 to a file data and read these numbers and then write all odd numbers to a file odd and all even number to a file even.

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

```

main( )
{

    FILE *f1,*f2,*f3;
    int i,x;
    f1=fopen("data","w");
    for(i=1;i<=30;i++)
    {
        //write 1 to 30 to file
        fprintf(fp, "%d\t",i);
    }
    fclose(f1);
    f1=fopen("data","r");
    f2=fopen("even","w");
    f3=fopen("odd","w");
    //read from data file and write to even and odd file
    while( fscanf(f1,"%d",&x) !=EOF )
    {
        if(x%2==0)
        {
            fprintf(f2, "%d\t",i);
        }
        else
        {
            fprintf(f3, "%d\t",i);
        }

    }
    fclose(f1);
    fclose(f2);
    fclose(f3);
}

```

Example 46.2- WAP to write employee information to a file and read them back

```

#include <stdio.h>

/* structure to hold employee information */
typedef struct employee {
    char name[30];
    int grade;
    float salary;
}emp;

int main( )
{
    FILE *fp;
    char ch;
    emp e;
    fp = fopen("emp_info.txt", "w"); /* open file in write mode */
    if (fp == NULL)
    {
        printf("Error opening file ...");
        return 0;
    }
    do
    {
        printf("Employee Name : ");
        scanf("%s", &e.name);
        printf("Enter Grade : ");
        scanf("%d", &e.grade);
        printf("Enter Salary : ");
        scanf("%f", &e.salary);
        fprintf(fp, "%s %d %f\n", e.name, e.grade, e.salary) ; /* file write */
        printf ( "Do you want to add another record (Y/y) " ) ;
        fflush(stdin);
        scanf("%c", &ch);
    } while(ch == 'y' || ch == 'Y');
}

```

```

fclose(fp); /* close the file after writing */

printf("\nEmployee information saved in the file ... ");
printf("\nLet's read those information ...\n");

fp = fopen("emp_info.txt", "r"); /* open file in write mode */
if (fp == NULL) {
    printf("Error opening file ...");
    return 0;
}
/* read the file one record at a time */
while (fscanf(fp, "%s %d %f", e.name, &e.grade, &e.salary) != EOF) {
    printf("\nName: %s\nGrade: %d\nSalary: %f\n", e.name, e.grade, e.salary);
}
fclose(fp); /* close the file after reading */
return 0;
}

```

47. Random Access to Files: ftell(), fseek(), rewind()

Three predefined functions are used for random access to files

- `rewind()` – return the file pointer to the beginning;
- `fseek()` – position the file pointer;
- `ftell()` – return the current offset of the file pointer.

`ftell()` - It tells the current position in file pointer.

The C library function **long int ftell(FILE *stream)** returns the current file position .

```

n=ftell(fp);
// n stores relative offset(in bytes) of the current position. i.e n bytes have read or write

```

rewind()- it takes the file pointer to start of the file

```

rewind (fp)
n=ftell(fp);

```


It assigns 0 to n file pointer has been set to start of the file by rewind()

Example 47.1- WAP to find size of a file using ftell() and fseek()

```
#include<stdio.h>
main( )
{
    FILE *fp;
    long int len;
    fp= fopen("file.txt", "r");
    if( fp == NULL )
    {
        printf("Error in opening file");
        exit(1);
    }
    fseek(fp,0,SEEK_END);
    len=ftell(fp);
    fclose(fp);
    printf("Total size of file.txt = %d bytes\n", len);
}
```

rewind() - It moves the control to beginning of a file.

The C library function void rewind(FILE *stream) sets the file position to the beginning of the file of the given stream.

fseek() - It is used to moves the file pointer to desired location with the file.

The C library function **int fseek(FILE *stream, long int offset, int positon)** sets the file position of the stream to the given offset.

The offset to use (in bytes) when determining the new file pointer position.

The position can take one of the following 6 values:

Values	meaning
0	beginning of the file
1	current position
2	end of the file
SEEK_SET	offset is relative to the beginning of the file.
SEEK_CUR	offset is relative to the current file pointer position.
SEEK_END	offset is relative to the end of the file.

The offset may be positive meaning move forward or negative means move backward.

fseek(fp,10,1): go forward 10 bytes from current file pointer position

fseek(fp,-5,1): go backward 5 bytes from current file pointer position

fseek(fp,10,SEEK_SET): go forward 10 bytes from beginning of the file

fseek(fp,0,0): go to beginning of the file
fseek(fp,0,2): go to end of the file

fseek(fp,-5,2): go backward m bytes from the end of the file

So, the equivalent of rewind() would be: fseek(f, 0, SEEK_SET);

on success fseek return 0. If programmer attempts to move the file pointer beyond the file boundaries an error occurs and fseek() returns -1.

Example 47.2- WAP to display the content of the file in reverse order

```
/* fseek example */
#include <stdio.h>
main ( )
{
    FILE * pFile;
    char c;
    long int n;
    pFile = fopen ( "example.txt" , "w" );
    while( (c=getchar()) !=EOF)
    {
        putc(c,fp);
    }
    n=ftell(fp);
    printf("No of character entered=%ld",n);
    fclose(fp);
    pFile = fopen ( "example.txt" , "r" );
    fseek(pfile,-1L,2); // move file pointer to last char

    do
    {
        putchar(getc(fp));
    }while( !fseek ( pFile ,-2L , SEEK_CUR ) );
    fclose ( pFile );
}

o/p-
a
b
c
d
ctrl+z
d
c
b
```

a

as after `getc()` file pointer advances 1 character `feek()` moves backward 2 character to bring the file pointer to previous character of the last printed character.

48. The C Preprocessor :

Conceptually, the ``preprocessor" is a translation phase that is applied to your source code before the compiler proper gets its hands on it. Generally, the preprocessor performs textual substitutions on your source code, in three sorts of ways:

- File inclusion: inserting the contents of another file into your source file, as if you had typed it all in there.
- Macro substitution: replacing instances of one piece of text with another.
- Conditional compilation: Arranging that, depending on various circumstances, certain parts of your source code are seen or not seen by the compiler at all.

The syntax of the preprocessor is different from the syntax of the rest of C in several respects.

- First of all, the preprocessor is ``line based." Each of the preprocessor directives we're going to learn about (all of which begin with the # character) must begin at the beginning of a line.
- Secondly, the preprocessor does not know about the structure of C--about functions, statements, or expressions. It is possible to play strange tricks with the preprocessor to turn something which does not look like C into C (or vice versa). It's also possible to run into problems when a preprocessor substitution does not do what you expected it to, because the preprocessor does not respect the structure of C statements and expressions (but you expected it to).
-

Macro Definition and Substitution

A preprocessor line of the form

```
#define name text
```

defines a *macro* with the given name, having as its *value* the given replacement text. After that (for the rest of the current source file), wherever the preprocessor sees that name, it will replace it with the replacement text. The name follows the same rules as ordinary identifiers (it can contain only letters, digits, and underscores, and may not begin with a digit). Since macros behave quite differently from normal variables (or functions), it is customary to give them names which are all capital letters (or at least which begin with a capital letter). The replacement text can be absolutely anything--it's not restricted to numbers, or simple strings, or anything.

The most common use for macros is to propagate various constants around and to make them more self-documenting. We've been saying things like

```
char line[100];
```

but this is neither readable nor reliable; it's not necessarily obvious what all those 100's scattered around the program are, and if we ever decide that 100 is too small for the size of the array to hold lines, we'll have to remember to change the number in two (or more) places. A much better solution is to use a macro:

```
#define MAXLINE 100
char line[MAXLINE];
```

Now, if we ever want to change the size, we only have to do it in one place, and it's more obvious what the words MAXLINE sprinkled through the program mean than the magic numbers 100 did.

Since the replacement text of a preprocessor macro can be anything, it can also be an expression, although you have to realize that, as always, the text is substituted (and perhaps evaluated) later. No evaluation is performed when the macro is defined. For example, suppose that you write something like

```
#define A 2
#define B 3
#define C A + B
```

suppose that you write
`int x = C * 2;`

If A, B, and C were ordinary variables, you'd expect x to end up with the value 10. But, the value of x is 8. Because the pre-processor always substitutes text for macros exactly as you have written it. So it first substitutes the replacement text for the macro C, resulting in

```
int x = A + B * 2;
```

Then it substitutes the macros A and B, resulting in

```
int x = 2 + 3 * 2;
```

To guard against this sort of problem, it is always a good idea to include explicit parentheses in the definitions of macros which contain expressions. If we were to define the macro C as

```
#define C (A + B)
```

then the declaration of x would ultimately expand to

```
int x = (2 + 3) * 2;
```

and x would be initialized to 10.

NOTE: There does not have to be a semicolon at the end of a `#define` line.

```
#define MAXLINE 100;                                /* WRONG */
```

then when you later declare
`char line[MAXLINE];`

the preprocessor will expand it to
`char line[100;];` /* WRONG */

which is a syntax error. **This is what we mean when we say that the preprocessor doesn't know much of anything about the syntax of C.**

Simple macros like `MAXLINE` act sort of like little variables, whose values are constant (or constant expressions). It's also possible to have macros which look like little functions (that is, you invoke them with what looks like function call syntax, and they expand to replacement text which is a function of the actual arguments they are invoked with) but we won't be looking at these yet.

Nesting of Macros:

One macro can be defined inside another macro called nesting of macro.

Example:

```
#define SQUARE(x) ( (x) * (x) )
#define CUBE(x) (SQUARE(x) * (x))
#define SIXTH(x) (CUBE(x) * CUBE(x))
```

File Inclusion

A line of the form

```
#include <filename.h>
```

or

```
#include "filename.h"
```

causes the contents of the file `filename.h` to be read, parsed, and compiled at that point. (After `filename.h` is processed, compilation continues on the line following the `#include` line.)

Example, suppose you got tired of retyping external function prototypes such as

```
extern int getline(char [], int);
```

at the top of each source file. You could instead place the prototype in a header file, perhaps `getline.h`, and then simply place

```
#include "getline.h"
```

at the top of each source file where you called `getline`. (You might not find it worthwhile to create an entire header file for a single function, but if you had a package of several related function, it might be very useful to place all of their declarations in one header file.)

As we may have mentioned, that's exactly what the Standard header files such as `stdio.h` are--collections of declarations (including external function prototype declarations) having to do with various sets of Standard library functions. When you use `#include` to read in a header file, you automatically get the prototypes and other declarations it contains, and you *should* use header files, precisely so that you will get the prototypes and other declarations they contain.

NOTE: The difference between the `<>` and `"` forms is where the preprocessor searches for `filename.h`.

- it searches for files enclosed in `<>` in central, standard directories
- it searches for files enclosed in `"` in the ```current directory,`" or the directory containing the source file that's doing the including.
- The extension ```.h`", by the way, simply stands for ```header,`" and reflects the fact that `#include` directives usually sit at the top (head) of your source files, and contain global declarations and definitions which you would otherwise put there.

Conditional Compilation

The last preprocessor directive we're going to look at is `#ifdef`. If you have the sequence

```
#ifdef name
program text
#else
more program text
#endif
```

in your program, the code that gets compiled depends on whether a preprocessor macro by that name is defined or not.

- If it is (that is, if there has been a `#define` line for a macro called name), then ```program text"` is compiled and ```more program text"` is ignored.
- If the macro is not defined, ```more program text"` is compiled and ```program text"` is ignored.

Just as for the `if` statement, the `#else` in an `#ifdef` is optional. Conditional compilation is useful in two general classes of situations:

NOTE:

Conditional compilation can be very handy, but it can also get out of hand. When large chunks of the program are completely different depending on, say, what operating system the program is being compiled for, it's often better to place the different versions in separate source files, and then only use one of the files (corresponding to one of the versions) to build the program on any given system. Also, if you are using an ANSI Standard compiler and you are writing ANSI-compatible code, you usually won't need so much conditional compilation, because the Standard specifies exactly how the compiler must do certain things, and exactly which library functions it must provide, so you don't have to work so hard to accommodate the old variations among compilers and libraries.