Introduction to C

C is a powerful and most popular structured programming language. It is a high level language but it also provides the low level programming features. So, it is called Middle Level Language. This was designed for developing both scientific and business applications. By using C many efficient and economical system software as well as application programs have been developed so far. Its simplicity and powerfulness made it the programmer's first choice to develop any type of application programs.

HISTORY OF C:-

It was developed by **Dennis Ritchie** at AT & T Bell laboratories of U.S.A. in 1972. It was developed as an alternative language for its successor language B developed by Ken Thompson, which is already derived from the language BCPL(Basic Combined Programming Language) developed by Martin Richards.

Many of the features and ideas of C were taken from the language BCPL. As a result of many years research, C language becomes popular. Later different organizations began using their own versions of C and this leads to compatibility problems. To avoid this, in 1983 ANSI standardized the C language.

C language had different versions. The first version was C-Language 1.0 it has the compilation facility in two phases. In 1975 version 2.0 was released which provides both high level and low level features. Later in 1980 version 4.0 with IDE(Integrated Development Environment) was released. It provides a menu driven compilation and debugging facility. Latest versions of the C language are C – 99 and C-11. Different organizations developed compilers for C language such as Borland C, Turbo C, and ANSI C.

FEATURES OF C:-

C is the widely used language. It provides many **features** that are given below.

1.Simple: C is very simple to learn and use. C is a simple language in the sense that it provides a structured approach **(to break the problem into parts),** the rich set of library functions**,** data types**, etc.**

2) Machine Independent or Portable: Unlike assembly language, c programs can be executed on different machines **with some machine specific changes. Therefore, C is a machine independent language.**

3) Mid-level programming language: Although, C is intended to do low-level programming**, it is used to develop system applications such as kernel, driver, etc. It** also supports the features of a high-level language**. That is why it is known as mid-level language.**

4) Structured programming language: C is a structured programming language in the sense that we can break the program into parts using functions**. So, it is easy to understand and modify. Functions also provide code reusability.**

5) Rich Library: C provides a lot of inbuilt functions **that make the development fast.**

6) Memory Management: It supports the feature of dynamic memory allocation**. In C language, we can free the allocated memory at any time by calling the** free() **function.**

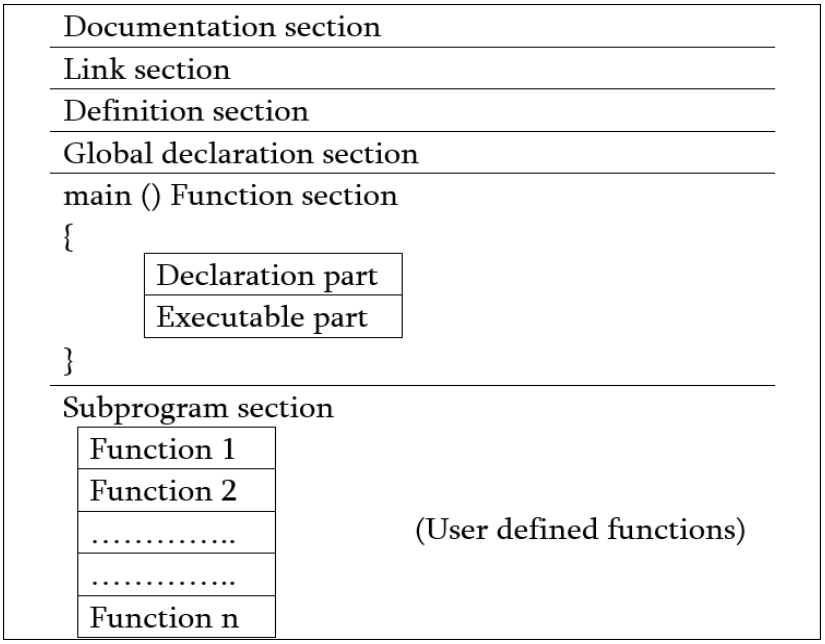
7) Speed: The compilation and execution time of C language is fast.

8) Pointer: C provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array**, etc.**

9) Recursion: In C, we **can call the function within the function**. It provides code reusability for every function. Recursion enables us to use the approach of backtracking.

10) Extensible: C language is extensible because it **can easily adopt new features**.

Basic structure of C program



Documentation section: The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later.

Link Section: The link section provides instructions to the compiler to link functions from the system library such as stdio.h using the *#include directive*.

Definition Section: The definition section defines all symbolic constants using the *#define directive*.

Global declaration section: There are some variables that are used in more than one function.Such variables are called global variables and are declared in the global declaration section that is outside of all the functions. This section also declares all the *user-defined functions*.

main () function section: Every C program must have one main function section. This section contains two parts; declaration part and executable part

Declaration part: The declaration part declares all the *variables* used in the executable part.

Executable part: There is at least one statement in the executable part. These two parts must appear between the opening and closing braces. The *program execution* begins at the opening brace and ends at the closing brace. The closing brace of the main function is the logical end of the program. All statements in the declaration and executable part end with a semicolon.

Subprogram section: If the program is a *multi-function program* then the subprogram section contains all the *user-defined functions* that are called in the main () function. User-defined functions are generally placed immediately after the main() function, although they may appear in any order.

Ex:

```
/* Program to calculate area of given circle */
#include <stdio.h>      /* File inclusion */
#define PI 3.14         /* Defining Section */
float carea(int );      /* Global declaration */
main( )
{
 int r;           /* local declaration */
 float area;
 clrcsr( );
printf("\n Enter the radius of the circle:");
scanf("%d",&r);  /* reading the data from keyboard */
area = carea(r);        /* function call */
 printf("\nArea of the given circle is : %f",area);  /* output on screen */
getch( );
}
float carea( int radius)     /* User defined function */
 {
float area;
area = PI * radius * radius;
return ( area);
 }
```

Using Comments:

Comments are the normal English sentences which explain about the aim of the program or explain about a statement used in the program.

In C we can use single line comments or multi-line (a block of) comments. These comments cannot translate into machine code. Comments are ignored by the compiler.

We can use comments anywhere in the program. These can improve the clarity of the program logic and they made debugging process easy. They do not affect the execution speed and size of the compiled code.

Syntax:

// is used for single line comments. (C- 99 version only)

/* ……..
……. */ is used for block or multi- line comments.

Ex:
```
/* Author: CBDC
Program to find the given number is prime or not */

C = sqrt(16);   // sqrt()function return the square root value
```

'C' character set:-

Different characters allowed in the C language are known as character set. C allows 256 characters approved by ASCII. Each and every character has a defined ASCII value. The characters in 'C' are grouped into the following categories.

1. Letters.
2. Digits.
3. Special characters.
4. White space.

Letters (or) Alphabets:- *A,B,C,D,E,.....X,Y,Z.* and *a,b,c,d,e,.....x,y,z.*

Digits:- *0,1,2,3……9*

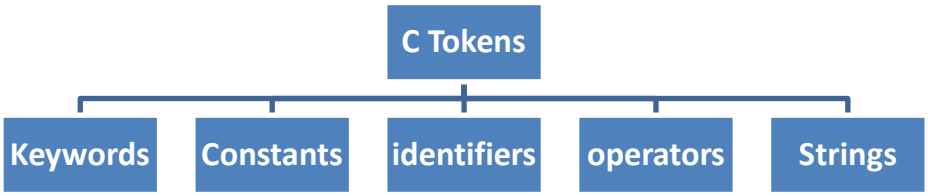
Special characters:-

, (comma)	# (hash)	~ (tide)	. (period)	' (apostrophe)
< (open angle brace)	; (semicolon)	" (quotation)	_ (underscore)	: (colon)
| (vertical bar)	$(dollar sign)	%(percent sign)	?(Question mark)	&(ampersand)
^(caret)	- (minus)	+ (plus sign)	> (closing angle brace)	((left parenthesis)
) (right parenthesis)	[(left bracket)	] (right bracket)	{(left brace)	}(right brace)
/ (slash)	\ (back slash).			

White spaces: Blank space / new line/ carriage return / Form feed/ horizontal tab / vertical tab.

C Tokens

The smallest individual unit of a C program is called token. The C program tokens are classified into 5 categories.



Key words:-

Key words are predefined tokens in 'C'. These are also called as reserved words. Key words have special meaning to the compiler. These keywords must be used only for their specified action. They cannot be used for any other purpose. All these key words must be used in lower case only.There are 32 key words available in C.

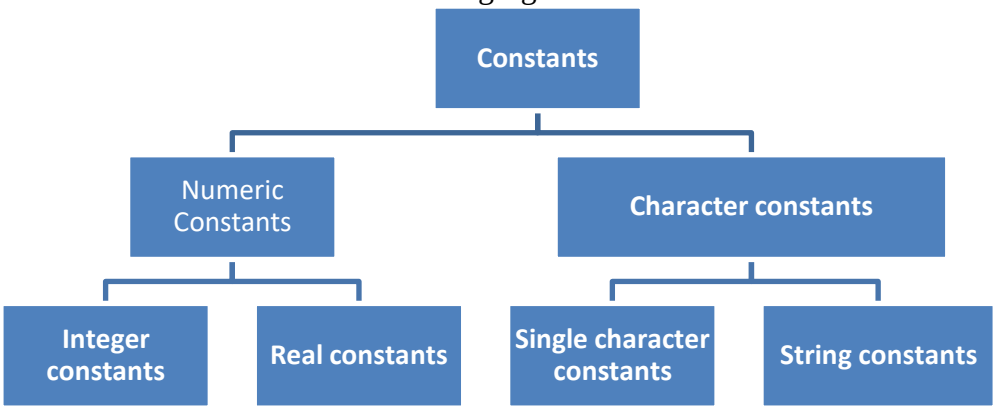
auto	*double*	*int*	*struct*
break	*else*	*long*	*switch*
case	*enum*	*register*	*typedef*
char	*extern*	*return*	*union*
const	*float*	*short*	*unsigned*
continue	*for*	*signed*	*void*
default	*goto*	*sizeof*	*volatile*
do	*if*	*static*	*while*

Some compilers support some additional keywords. They are

ada	*entry*	*fortran*	*near*
asm	*far*	*huge*	*pascal*

Constants:

A constant is a fixed value that cannot be changed during the execution of a program. C supports several types of constants as shown in the following figure.



Integer Constants:

An integer constant is a sequence of digits. The following are the rules for constructing an integer constant.

1. An integer constant must contain at least one digit.
2. It should not contain any decimal point.
3. If a constant is positive, it may (or) may not preceded by a plus (+) sign. If it is negative, it must be preceded by a minus (-) sign.
4. Commas, blank space, and non digit characters are not allowed in integer constant.

The following are valid integer constants

145-50 +516

The following are invalid integer constants

13.86 (decimal point not allowed) -40,000 (Comma not allowed)

Integer constants can also be represented in Octal and Hexadecimal formats.

***Octal Integer Constants*:**

Integer constants consisting of sequence of digits from the set 0 through 7 starting with english alphabet '0' is said to be octal integer constants.

Ex: o23, o345, o, o3452 etc..

Hexadecimal Integer Constants

Hexadecimal integer constants are integer constants having sequence of digits preceded by 0x or 0X. They may also include alphabets from A to F representing numbers 10 to 15.

Ex: 0xD, 0X8d, 0X, 0xbD

It should be note that, Octal and Hexadecimal integer constants are rarely used in programming.

Real Constants:

Real constants are also called as floating point constants. There are two ways to represent a real constant:

1. Decimal form
2. Exponential form.

Real constants expressed in Decimal form must have at least one digit and one decimal Point. The real constants can be either positive (+) or negative (-).

The following are acceptable real constants

0.054+6.165 -3000.0

*Representing a real constant in exponent form***:**

The general format in which a real number may be represented in exponential or scientific form is:

<Mantissa> e Or E < exponent >

The mantissa must be either an integer or a real number expressed in decimal notation.
The letter e or E separates the mantissa and the exponent parts and the exponent must be an integer.
Ex: 25.2E85 15e -10, -3e+8

Here 25.2E85 means 25.2×10^{85}

Single character constant:

A single character constant is a single character enclosed in single quotes. These can be interpreted as ASCII characters.

e.g.: 'c' 'D' '8'

The size of a character constant is always one byte.

String constant:

A string constant is a group of characters enclosed in double quotes. These characters may be either letters or numbers or Blank spaces.

e.g.: *"Welcome" "X+Y=10" "12345"*

Identifiers:

Identifiers are the names given to various program elements like constants, variables, functions, pointers etc.. An identifier is a sequence of letters and may have digits and symbol underscore(_).

ex: *avg, total, max_marks, value1*

Rules for constructing identifiers are:-

1. An identifier's length is minimum one character and the maximum of 31 characters.
2. An identifier contains only alphabets, digits and an underscore character.
3. It must start with a letter or underscore character.
4. Blank spaces are not allowed.
5. Keywords are not used as identifiers.
6. The upper case and lower case letters are treated as different. i.e. Identifiers are case sensitive.
   The following are the examples for valid and invalid identifiers.

Valid identifiers	Invalid identifiers
Name	*1st_value* (First character is digit)
emp_no	*amount$* (Special symbol)
salary	*phone number* (blank space)
MAX	*case* (Keyword)

Data Types

A programmer has to specify in advance to the system which kind of values have been entered from the keyboard and how they can be stored in the memory and which type of operations can perform on them. These can be represented by the data types. So, a data type is a kind of information to the compiler. There are many data types in C language. A C programmer has to use suitable data type as per the requirement.

C language data types can be mainly classified as:

1. Primary / Basic data type (char, int, float, double)
2. Secondary / Derived data type (Array, string ,structure, union)
3. User-defined / Abstract data type (enum , typedef)
4. Empty Data type (void)

Primary data type:

C language has four basic data types. The fundamental or primary data types are:

1. Character (`Char`)
2. Integer (`int`)
3. Real value (`float`)
4. Real values with more precision (`double`)

Many of them can be modified or extended by using some modifiers like:
`long, double, short, unsigned` and `signed.`

1. Character types:

- A single character can be defined as a character (char) data type.
- Characters are usually occupies one byte. (8 bits) of internal storage.
- The qualifier *signed* or *unsigned* may be explicitly applied to this type.
- While unsigned chars have values between 0 to 255 and signed chars have values from -128 to 127.

2. Integer Types:

- Integers are whole numbers that do not have any decimal part.
- We can use an integer in *signed* and *unsigned* forms.
- Generally integers occupy 2 bytes and the range of integer values is: -32768 to +32767.
- A signed integer uses one bit for sign and remaining bits for storing the number.
- Unsigned integers use all the bits for the storage of the number because they are always positive. Therefore, for a 16 bit machine, the range of unsigned integer numbers will be from 0 to 65,535.
- C has different integer values, namely *short int*, *int*, *long int*and *long long int*in both unsigned and signed forms.
- We declare long and unsigned integers to increase the range of values. Long int occupies 4 bytes.

3. **Floating point types**:

- Floating pointor real numbers can contain decimal part.
- Float data type occupies 4 bytes memory with 6 digits of precision.
- Floating point numbers are defined by the keyword *float*.

4. Double data type:

- When the accuracy provided by a float number is not sufficient then we use *double* data type.
- A double data type uses 8 bytes memory and giving a precision of 14 digits.
- To extend the precision further, we may use *long double* which uses 10 bytes.

Note: All data types are *signed* by default. Real values must be used in *signed* format only.

The following table shows more details about the primary data types.

Datatype	Keyword Equivalent	Size (in Bytes)	Range	Input / Output Symbol
Character	char	1	-128 to 127	%c
Unsigned Character	unsigned char	1	0 to 255	%c
Signed Character	signed char	1	-128 to 127	%c
Signed Integer	signed int (or) int	2	-32,768 to 32,767	%i or %d
Signed Short Integer	signed short int (or) short int (or) short	2	-32,768 to 32,767	%i or %d
Signed Long Integer	signed long int (or) long int (or) long	4	-2,147,483,648 to 2,147,483,647	%ld
UnSigned Integer	unsigned int (or) unsigned	2	0 to 65535	%u
UnSigned Short Integer	unsigned short int (or) unsigned short	2	0 to 65535	%u
UnSigned Long Integer	unsigned long int (or) unsigned long	4	0 to 4,294,967,295	%ul
Floating Point	float	4	3.4E-38 to 3.4E+38	%f
Double Precision Floating Point	double	8	1.7E-308 to 1.7E+308	%lf
Extended Double Precision Floating Point	long double	10	3.4E-4932 to 1.1E+4932	%Lf

Secondary / Derived Data types:

These data types are derived by using one or more fundamental data types. Some of them are

1. **Array**: A collection of same data type or homogenous data type elements.
2. **String:** An array of character type elements ended with null character.
3. **Pointers:** It is a special variable which stores the address of the given data item.
4. **Structure**: A collection of different data type elements treated as a single unit.
5. **Union:** It is like a structure but it holds only one element at a time.

User Defined Data types:

In c language users can create their own data types by using keywords like **'*typedef*'** and **'*enum*'**. These can be used like other data types in that program.

Empty data type:

'***void***' data type is known as empty data type. This indicates that this has no value. It is used with functions and pointers but not with normal variables.

1. Variables cannot be declared by using void data type.
   Ex: `void var;` // It is not valid.
2. A void function indicates that it cannot return anything.
   Ex: `void func1()`
3. If a function doesn't have any arguments then it will be indicated like below.
   Ex: `int func2(void)`
4. A pointer can be declared as void. That means it can be converted to any data type.
   Ex: `void *ptr;`

Escape Sequences:

An escape sequence begins with a '\' (backward slash) followed by a character. When a back slash is used in front of selected characters the compiler understood to escape the way that these characters would normally be interpreted

The following are the various Escaping sequences using in C:

Escape Sequence	Meaning
\a	Bell (It will ring a beep)
\b	Back space (It will moves the control one space back)
\f	Form Feed (Moves one page next while printing)
\n	New line (control moves to next line)
\t	Horizontal Tab (It moves 8 spaces of tab)
\v	Vertical Tab (It moves 8 lines down)
\r	Carriage Return (It replaces a word from the beginning of the another word) e.g: printf (" Sky is blue \r shirt"); Output: Shirt is blue
\'	It will display an Apostrophe
\"	It will display Quotation marks
\\	It will display a back slash
\0	Null Character

Format Specifiers:

A format specifier is a special character which begins with % (percentage symbol) character and followed by one or two alphabets. This specifies the compiler which type of data is reading from the keyboard or printing on the screen. These can be used with the functions ***scanf()*** and ***printf()***.(Formatted input and output functions).

Simply format specifiers indicate the data type to the compiler.

Format Specifier	Data type
%c	Character (Either signed or unsigned)
%d	Integer
%i (or) %hi	Short integer
%ld	Long integer
%u	Unsigned integer
%ul	Unsigned long integer
%f	float
%lf	double data type
%Lf	long double
%e (or) %E	Real values in exponential format
%o	Octal format of integer
%x (or) %X	Hexadecimal format of integer
%s	String data
%%	Prints % symbol on the screen

Variables:

A variable is a named memory location in the program. A variable is used to store a data value. A variable value can change any time.

A variable name should be carefully chosen by the programmer in a meaningful way. Variable names are case sensitive.

Any variable declared in a program should confirm to the following:

1. The variable name should contain only alphabets, digits and an underscore characters
2. They must always begin with a letter or an underscore.
3. The length of a variable must not be more than 8 characters.
4. White space is not allowed
5. A variable should not be a Keyword

Valid variable names	Invalid Variable names
Sun	123
number	(area)
Salary	6th
Emp_name	%abc
average1	Student Age

Declaration of Variables:

Every variable used in the program should be declared. The declaration does two things.

1. It tells the compiler the name of variable.
2. It specifies what type of data that the variable will hold.

Syntax:

```
datatype   Variable_name;
```

Or

```
datatype v1, v2, v3, …….. vn;
```

Where v1, v2, v3 are variable names. Variables are separated by commas. A declaration statement must end with a semicolon.

Ex:
```
int sum;
float gpa, salary;
double average, mean;
```

Initialization of Variables:

Initialization means specifying the starting value to the variable. If we cannot specify the starting value then system may store some unwanted values in a variable. So to avoid it we may initialize them.

Syntax:

```
datatype   Variable_name = value;
```

Or

```
datatype v1= value1, v2 =value2, v3=value3, …….. vn=value n;
```

Ex:
```
float avg = 0.00,
int sum = 0, i=1;
```

Types of Variables:

Variables are two types depending on the place where they are declared.

1. Local variables
2. Global variables.

Local variables:-

A variable which is declared with in a function is known as local variable. That variable is available in that function only. It is not available to the other function of that program.

Local variables are created upon the entry into the function (block) and are destroyed upon the exit from the function (block) in which they are defined.

Eg:-
```
fun1( )
    {
      int a;
     char c;
    }
 fun2( )
    {
      float f=1.412;
    double d=1.4;
    }
```

Global variables:-

Global variables are variables those are available any where in the program. They are not confined to a single function.

Generally global variables are declared at the beginning of the program i.e. before the main function.

Eg:-
```
float a=3.14;
main ( )
{
    int a=10;
    char c;
    -----
    ----
}
```

Type conversion:

Generally programmer may use different data type elements in a single expression. If operands of an operator are of different types then type conversion takes place. This conversion is in two ways.

i) Implicit (or) Automatic type conversion
ii) Explicit type conversion /type casting

Automatic type conversion:

In an expression operands are different data type, then operand of the lower type will be converted to the higher type automatically while evaluating the expression. That is

char or *short* is converted to *int* and the result will be the *int* type.

Like that *int* will be promoted to *float* and *float* is to be converted as*double*.. This type of conversion or promotion is called as implicit conversion or Automatic conversion.

e.g:
```
int x = 50;
float y;
y= x;
```
In the above *integerx* value is converted as 50.00 and then assigned *y*.

Explicit Type conversion/ Type Casting:

In this user may force to convert a value according to the requirement by using an operator which is known as casting operator.

Syntax: ***Variable = (data type) exp;***

e.g:
```
int a=15,b = 2;
float c;
```
c= a/ b;/* In this *c* value will be stored as 7.000000*/

c= (float) a/b; /* In this c value will be stored as 7.500000*/

In the above we explicitly converted the values of ***a***and ***b*** into float so that the result is accurate.

Operators

An operator is a symbol which indicates the computer to do a certain mathematical or logical operation. Operators are used in the program to operate on data or variables.

Based on the number of operands that an operator can work, operators are classified into 3 types. They are

1. **Unary Operators** : This type of operators can work with only one operand. (++ , - -, !)
2. **Binary Operators**: This type of operators can work with two operands. (Most of the operators are of this type)
3. **Ternary Operator**: This type of operators can work with three operands. (? :)

Classification of operators:

Depend on the type of operations, Operators are classified as

(ii) Arithmetic operators (+ , - , *, / , %)

(iii) Relational operators (< , > , <= , > =, == , !=)

(iv) Logical operators (&& , | | , !)

(v) Assignment operator (=, +=, - = , *= , /=)

(vi) Bit – wise operators (&, | , ^, ~, <<, >>)

(vii) Increment / Decrement operators (++ , - -)

(viii) Conditional operator (? :)

(i) Arithmetic Operators:-

\+ (Addition)

\- (Subtraction)

* (Multiplication)

/ (Division) (Gives quotient after division)

% (Modular division) (Gives remainder after division)

If both the operands are same type then, the result will also be the same type. If both operands are of different type then the result will be converted to the higher type. For example, if one operand is integer and other operand is float, then the result is a float value.

Note: Modulus operator cannot be applied on float and double values.

(ii) Relational operator:-

Relational operators are used to compare or test two operands.
The operators are

<	(Less than)
>	(Greater than)
<=	(Less than or equal to)
>=	(Greater than or equal to)
==	(Equal to)
!=	(Not equal to)

These operators return two types of values. If the given expression is true, then these return a non-zero integer value (usually treated as 1) otherwise (For False) they return a zero value.

(iii) Logical Operators:-

Logical operators are used to combine more than one relational operator. These are also called as Boolean operators. These are used when we want to test more than one condition. 'C' has 3 logical operators They are

&&	(AND)
||	(OR)
!	(NOT)

Logical operator return 0 if the condition is false, it returns a nonzero value when the given condition is true.

&&(And):-

It returns true if both expressions are true. It returns false when any of two expressions is false.
e.g:- if ((a>b) && (a>c))

|| (OR):-

It returns true if any one of the two conditions is true. It returns false if both expressions are false.
e.g:- if ((a<b) || (a>c))

Truth Table :

Exp1	Exp 2	&&	||
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

! (NOT):- This is an unary operator. It will used to reverse the condition. i.e. It will turns the true condition as false and vice versa.

A	! A
T	F
T	T

e.g : if (!(S = 10))

iv) Assignment operator:-

Assignment operator is used to assign a value or result of an expression to a variable.
In 'C' the assignment operator is '= '

Syntax: ***variable = val (or) exp***;
Eg:- *a = 10;*
a = b+c;

If the two operands of an assignment operator are of different datatypes then the right hand side value will be converted to the left hand side type automatically.
Eg:- *int max;*
float avg = 10.25;
max = f;
In the above *avg* value will be converted into integer and assigned to *max* as 10.

Short hand assignment operators:-

In 'C' we can use the assignment operator in a different way for easy programming. This type of assignment is called short cut assignment. To understand this, observes the following examples.

a = a+1	is written as	*a += 1;*
a = a-1	*is written as*	*a - = 1;*
*a = a*1*	*is written as*	*a *= 1;*
a = a/5	*is written as*	*a /= 5;*

Multiple assignments in a single statement is also possible in C.
a = b = 10;
a = c = b+2;

v) Bitwise Operators:

These operators are used for manipulation of data at bit level. These operators are used for testing the bits, shifting the bits, etc. These operators are not used with float values and double values.

& → **Bitwise and**
| → **Bitwise or**
^ → **Exclusive or**
<< → **Left shift**
>> → **Right shift**
~ → **One's compliment**

Logical 'AND', logical 'OR' are different form Bitwise 'and', Bitwise 'or'.

These operators can operate on integer and character values but not on float and double. In bitwise operator the function showbits( ) function is used to display thebinary representation of any integer or character value.

vi) Increment (OR) Decrement Operators:

These operators can work with a single operand only. So these are called as *unary operators*.

++→ Increment operator. It increases the value by 1.
- -→Decrement operator. It decrements the value by 1.

e.g.: *a++ means a = a+1;*
a-- means a = a-1;

We can use these operators either in prefix or postfix notations like below.

i) ++ *a* (or) -- *a* (Pre-increment or Pre-decrement)
ii) *a* ++ (or) *a* -- (Post- increment or Post-decrement)

In pre- increment the value at the variable is incremented first later the new value will be used in the expression where as in post-increment first the old value in the variable is used in the expression and later it will be incremented. To understand this, observe the following code.

e.g.: 1. *int a = 10;*

int b = ++a; /* Now 11 is assigned to b*/

b = a++; /* Now 10 is assigned to b */

2.. int a = 10;

int b = --a; /* Now 9 is assigned to b*/

b = a--; /* Now 10 is assigned to b */

vii) Conditional Operator:-

The conditional operator is also known as ternary operator. This operator used to carry out three expressions at a time. This operator is used instead of if- else statement.

It is of the form: ***exp1? exp2 : exp3;***

While evaluating the condition, first *exp1* is evaluated. If it is true *exp2* is executed. If *exp1* is false then *exp3* is executed and it becomes the result of the operation. Note that *exp2* and *exp3* are either constant values or a single variable or an arithmetic expression.

Ex**:-** *int a = 5;*

b = (a>0) ? 10 : -10;

here 10 is stored in 'b'.

Special Operators :

'C' supports some special operators apart from the above discussed. They are

1. ***Comma (,)*** – It links two or more expressions.
2. ***Sizeof()*** – It returns the size of the operand i.e., how many bytes that it occupied.

   syntax: ***n =sizeof(variable or expression);***

   e.g: *int i;*

   i = sizeof(float);→ it will assigns 4 to *i* since float data type occupies 4 bytes of memory.
3. ***Pointer operators*** –& (address) , * (value at the address)
4. ***Selection operators*** –•, → (used in structures)
5. ***Unary Plus (+) and Unary Minus (-) :*** These are used to indicate the sign of the value.

   e.g: *int a = +75;* /* This assigns positive value for a */

   int b = -75; /* This assigns negative value for b */

Precedence and Associativity of operators in C:

Operator precedence determines which operator is evaluated first when an expression has more than one operators. For example 100-2*30 would yield 40, because it is evaluated as 100 – (2*30) and not (100-2)*30. The reason is that multiplication * has higher precedence than subtraction(-).

Associativity is used when there are two or more operators of same precedence is present in an expression. For example multiplication and division arithmetic operators have same precedence, let's say we have an expression 5*2/10, this expression would be evaluated as (5*2)/10 because the associativity is left to right for these operators. C operators are listed in order of *precedence* (highest to lowest). Their *associativity* indicates in what order operators of equal precedence in an expression are applied.

Operator	Description	Associativity
() [ ] . -> ++ --	Parentheses: grouping or function call Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement	left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	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 Relational greater than/greater than or equal to	left-to-right
== !=	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 Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

Data Input and Output:-

'C' provides a standard set of input and output library functions. Definitions for these functions are available in the **stdio.h**(Standard Input Output) header file.

So every 'C' program must include this *stdio.h* at the beginning like below

#include < stdio.h > (or) #include "stdio.h"

Data input and output functions are classified into two types. They are

1. Unformatted I/O functions
2. Formatted I/O functions.

1. Unformatted I/O functions:-

By using these functions just we can read and write the data through I/O interfaces in the form of characters only. These cannot support to specify the data types. The main Unformatted I/O functions are

Input functions:` ***getchar()*** ***getche()*** ***getch()*** ***gets( )***

Output functions: ***putchar()*** ***putch( )*** ***puts()***

Input Functions:

getchar():- This is used to read a single character as input from the keyboard.

Syntax: ***variable = getchar();***

e.g:- *char c;*
c = getchar();

This is a buffer function. It will stores input character in the memory buffers temporarily until user press the Enter key. After that the input character is transferred to the corresponding variable of the program.

getche():

It is also used to read a single character from the keyboard. This is a non buffer function. So, it will directly assigns the given character to the variable directly and not wait for enter key press. The given input is visible on the screen so it is as echo function.

Syntax: ***variable = getche();***

```
char ch;
ch = getchar();
```

getch():

This is a non buffer function. This function directly assigns the reading character to the relevant variable. While giving the data to this function the entered character is not visible on screen. (Non – Echo function)

Syntax: ***variable = getch();***

```
e.g:-  char c;
       c = getch();
```

gets():-

gets() is a function which is used to read a string from the keyboard. It will read a string from the keyboard until the user press Enter key then this function adds a null character (\0) at the end of that string.

Syntax: ***gets (string_variable);***

```
e.g:-  char message[30];

       gets(message);
```

Output Functions:

putchar():- The *putchar()*is used to display a single character on the monitor.

Syntax:- ***putchar (const char variable)***

```
e.g:-  char c= 's';
       putchar(c );
```

putch():- The putch()is used to display a single character on the monitor.

Syntax:- ***putch (const char variable)***

```
e.g:-  char c= 's';
       putchar(c );
```

puts():-This is used for display a string on the screen. This function can print the given string in a new line.

Syntax:- ***puts(const char *variable);***

e.g:-
```
char str[20];
puts (str);
```

2.Formatted I/O functions:- Formatted I/O functions are *scanf()* and *printf()*

scanf():-

scanf() is a standard 'C' library function used to read the data from the keyboard. This function is used to read any type of data.

Syntax:- ***scanf("Format specifier", &variable name);***
OR
scanf("Control string ", &arg1,&arg2,.....&arg n);

Control string contains format specifiers(input/output symbols)which helps the compiler to interpret the type of input data. This control string must be enclosed with in double quotations (" ").

"&arg1, &arg2,-----&argn" are the arguments which represents the address of the variables to get their values. Control string and arguments are separated by a comma.

e.g:-
```
int a;
scanf("%d",&a);
float d;
long c;
scanf("%f %ld",&d, &c);
```

☞We can read a specified no.of characters or numbers from the given input by using the width field

Syntax : ***scanf("%wd", &variable);*** --- to read the integer data.
scanf("%wc", &variable); --- to read the character data.

e.g.:
```
int first,second;
scanf("%3d %4d",&first, &second);
```

If we enter the data as 5673241 then, 567 is assigned to first and 3241 is assigned to second.

☞ We can also skip some input values by using * mark along with the format specifier like below.

```
e.g: scanf("%d %*d %d", &first, &second, &third);
```

In the above, the second input value is skipped and the third value is assigned to the second variable.

Assume that we enter the data as 675 89 345 100

Then 675 is assigned to *first*, 89 is skipped, 345 is assigned to second and *third* is assigned with 100.

printf():-

printf() is a standard library function which is used to display the given output of the program on the screen. This function is used to display any combination of numerical values, strings, single characters.

Syntax:- ***printf ("control string", arg1, arg2,.....argn);***

Here the control string contains any of the following

1. Format Specifiers - Which will indicate the data type of the output variable.
2. Normal text characters - These will be displayed as they are appeared in the quotations.
3. Escape sequences – These are used to display the output on the specified position of the screen.

Arguments are the variables whose values are formatted and displayed according to the specification in the control string. The arguments should match in number and in order specified in the control string.

e.g:-
```
printf("%d",a);
printf ("\t Welcome to C ");
printf(" A=%d \t D=%f \t S=%c",a,d,s);
```

☞We can specify the total width of output data like below.

printf("%wd",variable);

e.g: *printf("%5d",a);*

☞We can align the output either to right or left by using + and - symbols.

e.g: *printf("%5d",678);*

Theabove will print total 5 places.It leaves first 2 places blank then prints 678 in the next 3 places.(Right align)

printf("%-5d",678);

This will print total 5 places. It prints 678 in the first 3 places leave the last two places blank.(Left align)

☞For real values we can specify the width like

Syntax: ***printf("%w.pf",variable);***

In this W is for total width and P is for the number of decimal positions.

e.g: *printf("%6.2f",a);*

The above statement prints total 6 places in that 2 places are reserved for decimal part. By default float prints 6 decimal places.

☞Like the above we can also specify the width for characters and strings.

Syntax: ***printf("%wc",variable);*** For Normal characters

printf("%w.ps",variable); - For Strings

In strings W specifies the total number of positions to print and P specifies the number of characters to be print.

e.g: char str[25] = "Welcome to the world of C "

printf("%7c",str); /* This prints only " Welcome " from the given string*/

printf("%20s",str); /* This prints only " Welcome to the World " from the given string*/

printf("%10.7s",str);

/* This prints total 10 places but it prints only 7 characters (Welcome) and they are right aligned.*/

printf("%-10.7s",str);

/* This prints total 10 places but it prints only 7 characters (Welcome) and they are left aligned. */

Decision Control and Looping Statements

Control Structures

A program is defined as a set of executable statements. These statements are executed in the same sequence that they appear in the program from top to bottom. But sometimes programmer may want to change the sequential execution or to jump from a set of statements or may want to repeat execution of some statements. For this C provides various structures those are called as 'Control Structures'. These are also called as 'Logic Constructs'.

Control Structures are classified into three types. They are

1. Decision Making (OR) Selection (OR) Branching Structures (if , switch & conditional operator)
2. Loop Structures (while , do – while , for)
3. Jumping (goto , break , continue)

Decision Making Statements

These are used when the execution of statements is depending on a given condition. The general decision making statements are

1)*if statement* **2) *switch statement*** **3) *Conditional operator***

1) if statement:-

It is a general decision making statement. These are used to specify the conditional execution of statements. Four types of **if** statements are used in C programs. They are

1) **if** 2) **if – else** 3) **Nested if – else** 4) **else – if ladder**

1) if :

This is also called as simple if. It is used to perform basic conditional tests. When we have only one condition and based on that we have only one choice of execution then we choose this **if** statement.

It has the following syntax:

```
if (condition)
 {
    true block statements;
 }
statement –x;
```

When an **if** statement is encountered in a program, then the condition will be evaluated first. If it is true then the statement block associated with **if** is executed. Otherwise the control will skip this block and starts the execution from statement –x (Following statements of if block).

e.g:
```
if( a<0)
{
   printf( "\n A contains  negative value");
}
```

2) if – else:

This statement is used when we have one condition and based on that condition if we have 2 choices. This has two blocks named as **if** block and **else** block.

Syntax:

```
if (condition)
 {
    True block statements;
 }
else
 {
   False block statements;
 }
statement –x;
```

In this first the condition will be evaluated. If the condition is true then, **if** block will be executed. Otherwise the **else** block will be executed.

e.g:

```
if( marks < 35)
  {
      printf( "\n Sorry! You are failed");

  }
else
  {
      printf( "\n Congrats! You pass the exam");

  }
```

3)Nested if – else :-

If the programmer want to test more than one condition then, he uses two or more if – else statements embedded within them this type of if- else statements are called as nested if – else statements. We can use this statement any one of the following three ways.

Syntax: **Case: 1**

```
if( Condition 1)
  {
      if (Condition 2)
        {
            block 1;
        }
      else
        {
            block 2;
        }
  }
else
  {
      block 3;
  }
```

Case: 2

```
if( Condition 1)
  {
      block 1;
  }
else
  {
      if (Condition 2)
        {
            block 2;
        }
      else
        {
            block 3;
        }
  }
```

Case: 3

```
if( Condition 1)
  {
      if (Condition 2)
        {
            block1;
        }
      else
        {
            block 2;
        }
  }
else
  {   if (Condition 3)
        {
            block 3;
        }
      else
        {
            block 4;
        }
  }
```

Case 1:

In this first **condition1** will be evaluated. If it is true then, control entered into **if** block and **condition2** will be evaluated. If this **condition2** is true then **block1** will be executed otherwise **block2** will be executed.

But if **condition1** is false then control enter into the outer else and **block3** will be executed.

Case 2:

In this first **condition1** will be evaluated. If it is true then, **block1** will be executed. Otherwise control entered into outer else block and **condition2** will be evaluated. If this **condition2** is true then **block2** will be executed. If **condition2** is false then **block3** will be executed.

Case 3:

In this first **condition1** will be evaluated. If it is true then, control entered into **if** block and **condition2** will be evaluated. If this **condition2** is true then **block1** will be executed otherwise **block2** will be executed.

But if **condition1** is false then control enter into the outer else and **condition3** will be evaluated. If this **condition3** is true then **block3** will be executed otherwise **block4** will be executed.

```
e.g.:        if(a>b)
                { if(a>c)
                 big = a;
                   else
                        big = c;
                 }
             else
                { if(b>c)
                     big = b;
                   else
                     big = c;
                }
```

4) else-if ladder: -

If we want to test more conditions and each condition has a concern block of statements then, we can use else-if ladder. It has the following syntax.

Syntax:

```
if (Condition1)
        {
            block 1;
        }
else if ( Condition2)
        {
            block 2;
        }
else if ( Condition3)
        {
            block 3;
        }
        ....
        ....
else
       {
            default block;
        }
```

```
e.g.:
        if (income >1000000)
                printf("Millionaire \n");

        else if (income >500000)
                printf("Rich\n");

        else if ( income > 200000)
                printf("Upper Middle Class\n");

        else if (income>100000)
                printf("Middle Class\n");

        else
                printf("Poor\n");
```

Switch Statement :-

This is an alternative to else – if ladder statement. This is used when we have an expression and that expression has many alternative values and according branches . All the alternative values are treated as different cases.

Syntax:

```
switch ( expression )
{
   case   value_1 :  { block 1; }
                     break;
   case   value_2 :  { block 2; }
                     break;
   case   value_3 :  { block 3; }
                     break;
    case  value_4 :  { block 4; }
                     break;
       :        :
       :        :
       :        :

    default        : {   block ; }
                     break;
 }
```

e.g.:

```
switch ( operator )
{
   case   ' + ' :   res = a + b;
                    break;
   case   ' – ' :  res = a - b;
                    break;
   case   ' * ' :  res = a * b;
                    break;
    case  ' / '  :   res = a / b;
                    break;
   case   ' % '  :   res = a / b;
                    break;

    default  : printf( "Invalid Operator");
                    break;
 }
```

This uses four keywords switch, case, break and default. First the expression passed to the switch is evaluated. The resulted value is compared with the case values from top to bottom till a matched value is found. Then the concerned block of matched value is executed. If no case value is matched then the default block is executed. So cases must have distinct values.

At the end of each case or each block there must be a break statement. This will indicate the end of execution and to come out of the switch statement.

Conditional operator (?:):

C provides a different operator which is used for making two way decisions. That is Conditional operator. This is also known as ternary operator.

It uses '**?**' and '**:**'. It requires three operands.

Conditional expression ? expression 1 : expression2;

First the **conditional expression** is executed if it is true then, **expression1** is evaluated and that will be the result of the condition. If the conditional expression is false then, **expression2** is evaluated and that will be the result of the condition.

e.g.**:-** *(no % 2 = = 0) ? printf (" Number is Even ") : printf ("Number is Odd") ;*

This operator is also used in nested format like below.

expression 1 ? (expression 2 ? expression 3 : expression 4) : expression 5;

This is used to check two conditions. Here first the **expression 1** is executed if it is true then, **expression2** will be checked if it is true then **expression 3** is executed otherwise **expression 4** will be executed. But if **expression 1** is false then, only **expression 5** is executed.

e.g.: *big = ((a > b) && (a > c)) ? a : ((b > c) ? b : c) ;*

Loop Statements

Sometimes user may want to execute same set of statements repeatedly. For this C provides a variety of control statements. These are called as loop statements. These statements are also called as **repetitive statements** or i**terative statements**.

Every loop consists two parts namely

1) Control Part or Terminating Condition – This will contain the control condition.
2) Body Part - This will contains the set of statements of the loop.

The execution of the loop is repeated till the given condition is satisfied. Whenever the condition evaluates to false then, the execution of the loop is terminated.
(One time execution of all the statements of a loop is called as ' iteration' or 'pass'.)

C provides three types of loop structures. They are

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

C loop structures are divided into two categories based on the placing of terminating condition.

1) Entry Control loops
2) Exit Control loops

1) Entry control:-

In the entry control loop first the condition will be checked. If it is true then, the statements of the loop are executed. If the condition is false then the body will not be executed.

e.g.: for loop, while loop.

2) Exit control:-

In this type, the body of the loop will be executed first at the end the control condition will be checked. If the condition is satisfied then, the execution is repeated. If the condition is not satisfied control will come out of the loop. So at least one time the body of the loop will be executed.

e.g.: do – while.

While Loop:-

While statement is used to carry out iterative operations till the given condition is satisfied. It is an entry control loop.

Syntax:

```
while (test condition)
 {
  block of statements;
 }
```

e.g:

```
int i=1;
      while (i <= 5)
      {
          printf("\n %d",i );
            i++;
      }
```

In this first the test condition will be checked. If it is true then the body of the loop will be executed. This execution is repeated till the condition will become false.

The body of the loop may have one or more statements. When the loop have multiple statements then using of braces is compulsory. In the body part we will change the values of the controlling variables.

for Loop:-

The ***for*** loop is an entry control loop. This is used for repetitive execution of a block of statements.

Syntax:

```
for ( initialization ; test condition ; increment or decrement of values )
        {
          Body of the loop;
        }
```

This is a one step loop. It has initialization, testing condition and increment or decrement parts in the parenthesis statement.

In the **initialization** part, we can assign starting value to the variable(s) like i = 0 or n=10 etc. These variables are known as loop control variables.

The **test condition** part specifies the conditions(s) that are necessary for loop continuation. Body of the loop will be executed as many times as required till the given condition is satisfied otherwise this loop is terminated.

In **increment or decrement** part, the values of the loop control variable(s) will be changed according to our program logic by using increment or decrement operators.

The initialization, test condition increment or decrements are separated by semicolon (;).

In this loop, first the values are initialized. Next the condition will be checked. If the condition is true then, the body of the loop is executed as many times as required. At last the control goes to the increment or decrement part to change the value of loop control variable.

Note that the variable is initialized only once in the loop.

e.g.:

```
int i, sum=0;
 for (i = 0; i<=10; i++)
   {
     sum = sum+i;
   }
```

do - while loop :-

It is an exit control loop. It is also used to carry out repetitive execution of same statements. In 'do-while' the body of the loop will be executed at least once irrespective of condition checking.

Syntax:

```
do
  {
   block of statements;
   } while (test condition);
```

e.g:

```
int i = 1,sum=0;
   do
     {
        sum = sum + i ;
        i = i+1;
      } while(i<=10);
```

In this first body of the loop will be executed once. Next the given condition will be checked. If the condition is true then the loop will be continuously executed for required number of times. Once the condition becomes false then the loop execution is terminated and execution will be continued with the following statements.

Note that we have to put a semicolon after the while condition.

Jumping Statements

These statements are used to jump the control from one statement to another statement while the program is in execution. Generally three types of statements are used for this purpose. They are

1) goto statement
2) break statement
3) continue statement

Goto statement:-

'C' supports the ***goto*** statement for branch the statements unconditionally from one part to another part in a program. This statement requires a ***label*** in order to identify the place where to jump. Label is any valid name and must be followed by (:) colon. The label is placed immediately before the statement where the control is to be transferred.

This goto is used in two types of jumps. They are

i) Forward jump
ii) Backward jump

Syntax:

```
s1;
s2;
goto label;
s3;
s4;
label :
        s5;
        s6
```

Forward jumping

```
s1;
s2;
label :
s3;
s4;
goto label;
        s5;
        s6
```

Backward jumping

Forward jumping:

In this first *s1*, *s2* are executed. Next the goto statement is encountered then the control jumps to the concerned label and the following statements of that label *s5* and *s6* statements are executed. *s3* and *s4* are skipped.

Backward jumping:

In this first *s1*, *s2* , *s3* and *s4* are executed. Next the goto statement is encountered then the control jumps back to the concerned label place and the following statements of that label *s3* and *s4* statements are executed again. This makes a loop. But it is an infinite loop. So to make it finite we can place some condition with if structure before the goto statement..

```
int a,b,c;
printf(" Enter 2 numbers\n");
scanf( "%d%d",&a ,&b);
goto PRINT;
        c = a * b;
    printf("%d",c);
PRINT:
        c = a + b;
    printf("%d",c);
```

Forward jumping

```
int i=1;
printf(" Numbers 1 to10\n");
PRINT::
        printf("%d",i);
        i=i+1;
    if(i <=10)
    goto PRINT;
```

Backward jumping

break statement:

The break statement is used to quit from a loop or a block. It can be used within *if, switch, for, while*, and do-while.

Syntax:

```
break;
```

When a break statement is encountered in a loop or a block, automatically the control will come out of it.

Continue statement:-

The continue statement will work just opposite to the break statement. That means the control will not come out of the loop. But one of the iteration of the loop will be skipped out.

Syntax:

```
continue;
```

If a continue statement is encountered in a loop, then the control will passed to the beginning of the loop. That means one iteration of the loop will be skipped .In ***while*** and ***do-while*** thecontrol willpass to the conditional part where as in ***for*** loop the control will pass to increment or decrement part.

Observe the following examples for better understanding of **break** and **continue** statements.

```
main()
{
        int i;
        for(i=1;i<=10;i++)
        {
            if(i = =5)
             break;
          printf("%d\n",i);
        }
}
```

```
main()
{
        int i;
        for(i=1;i<=10;i++)
        {
        if(i = =5)
        continue;
        printf("%d\n",i);
        }
}
```

In the above:

First Ex: When *i* value becomes 5, and then the control will come out of the loop. This program can print only 1, 2, 3, 4 numbers on the screen.

Second Ex: When *i* value becomes 5, and then the control will directly goes to the increment part without printing the number 5 . That means only one of the iteration will be skipped. This program can print 1, 2, 3, 4, 6, 7, 8, 9, 10 numbers on the screen except 5.

Arrays

Sometimes user may have to work with more number of variables in a program. It is difficult to specify different names to them. Remembering of all those names are also difficult. To avoid this we use the arrays.

An array is defined as the collection of similar type of data items stored at contiguous memory locations. Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc. It also has the capability to store the collection of derived data types, such as pointers, structure, etc. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as **numbers** and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

Definition: "*An array is defined as a set of homogeneous (same data type) data items. They can have a common name and stored in sequence locations of the memory*."

Properties of Array

- Each element of an array is of same data type and carries the same size, i.e., int = 2 bytes.
- Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Declaration :

Array will be declared in a program like other variables. The declaration of the array has the following syntax:

Syntax: **datatype array_name [size];**

In this

datatype is any one of the C supported data types, which indicates the type of elements of array.
array_name is a valid identifier.
Size indicates the number of elements of the array. It should be a positive integer.
Every array must have a subscript.

Arrays can be classified into

1. Single Dimensional Array (or) Linear Array
2. Multi Dimensional Array. (or) Non Linear Array

Single Dimensional Array (or) Linear Array:

It is a list of items, whose elements are specified by one subscript. It is also called as *one dimensional array*.

Syntax: **datatype array_name [size];**

e.g:- int a[10];
char c[9];

Individual elements of the array are accessed by specifying their subscript number. The subscript number of the array always starts zero.

Initialization:-

The elements of an array can be assigned with initial values in the array definition with a list of values enclosed in braces and separated by commas.

e.g:- *int a[5] = {12 ,18, 44, 85, 96};*

In the above example, we declared an array with a name 'a' and size of '5'.
The 5 integer values are initialized as

a[0]=12, a[1]=18,a[2]=44,a[3]=85,a[4]=96.

It is not necessary to specify the size of an array, in case we provide the initial values for the array. In such cases, the compiler allocates enough memory for all initialized elements.

e.g.:- *int a[ ] = {1,2,3,4};*

If the no.of initial values are less than the size of the array, then the remaining elements will be initialized to zero.

Eg:- *int a[5]={12,18};*

The 5 integer values are initialized as a[0]=12, a[1]=18,a[2]=0,a[3]=0,a[4]=0

eg:- int *a[5]*={0} or *int* a[5]={}

The 5 integer values are initialized as a[0]=0, a[1]=0 ,a[2]=0,a[3]=0,a[4]=0.

If the no. of initial values are greater than the size of the array, then the compiler will generate an error.

Eg:- int a[5]={1,2,3,4,5,6,7} ← Compiler error.

Character arrays may be initialized in the same manner.

e.g.:- *char a[ ] = {'I','N','D','I','A','\0'};*

The above array can be initialized as string constant.

e.g.:- *char a[6] = "INDIA";*

In the above declaration the '\0' (Null character) will be automatically added at the end.

Rules for array initialization:-

1. If the number of initializers are less than the number of elements in an array. Then the remaining elements are set to zero.

   e.g.: *int a[5] = {2,3,4}*;

   In the above example a[3] and a[4] are set to zero.

2. The number of initializers must not be more than number of elements in an array.

   e.g:- *int a[5] = {2,3,4,5,6,7,8};*→ not allowed.

The simplest way to initialize an array is by using the index of each element. We can initialize each element of the array by using the index. Consider the following example.

```
int marks[5];

    marks[0]=80;//initialization of array

    marks[1]=60;  marks[2]=70; marks[3]=85; marks[4]=75;
```

Two – dimensional arrays:-

Arrays whose elements are specified by two subscripts are called two-dimensional arrays. It is also called as double subscripted array.

Each subscript must contain a positive integer and should always start with 0.

Syntax : ***datatype array_name[rows][columns];***

e.g.:- *int a[3][2];*

A two dimensional array is a table of items which had a common name.

The number of elements in a two dimensional array = number of rows × number of columns.

In the above example - Number of elements = 3 × 2 = 6.

Therefore we can store six elements.

Initialization:-

Two dimensional arrays are also initialized like one dimensional array. They can be initialized with a list of values separated by commas and enclosed in braces.

Eg:- *int a[3][2] = {10,12,14,16,18,20};*

(or)

int a[3][2] = { {10,12},{14,16},{18,20}};

In the above example, we declared an array with a name 'a' and size of 3 × 2
The 6 integer values are initialized as

a[0][0]= 10 , a[0][1]= 12, a[1][0] = 14, a[1][1] = 16, a[2][0]= 18, a[2][1] = 20

Two dimension array can also be initialized like below

e.g.: int a[ ][ ] = { {1,2,3},{4,5,6},{7,8,9}};

In the above the size will be interpreted as int a[3][3].

int a[3][2] = { {2},{2},{2} };

In the above example, the first element of each row will be initialized as two , the remaining elements are set to zero.

Multi dimensional arrays:-

The arrays whose elements are specified in more than one subscript are know as multi dimensional arrays. They can be in the following form.

d*atatype array_name [expression1] [expression2]……..[exp n];*

e.g.:- *int a[3][4][5];*

Initialization:-

The multi dimensional arrays are also initialized like other arrays.

```
e.g.:- int a[3][2][2] = { {
                            {1,2},
                            {3,4}
                          },
                         {
                           {3,4},
                           {4,5}
                         },
                         {
                           {2,3},
                           {4,5}
                         }
                        }
                        (or)
int a[3][2][2]  = {  { {1,2},{3,4} },{ {3,4},{4,5} },{ {2,3},{4,5} }  };
```

Working with arrays: -One Dimensional Array:

Reading the elements for an array:

An array can contain a set of elements. So to read the elements generally we use a single **for** loop having a **scanf()** function.

Syntax:

```
for ( i = 0; i < array size ; i ++)
   {
     scanf( "format specifier", &array_name[i] );
    }
```

Displaying the elements of an array:

We can display the elements of an array with the help of a **for** loop having a **printf()** function**.**

Syntax:

```
for ( i = 0; i < array size ; i ++)
   {
     printf( "format specifier", array_name[i] );
    }
```

Working with arrays: - Two Dimensional Array:

Reading the elements for an array:

To read the elements of a two dimensional array, we need two **for** loops having a **scanf()** function.

Syntax:

```
for( i = 0; i < rows ; i ++)
   {    for ( j = 0; j < columns ; j ++)
        {
          scanf( "format specifier", &array_name[i][j] );
        }
   }
```

Displaying the elements of an array:

We can display the elements of a two dimensional array, with the help of two **for** loops having a **printf()** function**.**

Syntax:

```
for( i = 0; i < rows ; i ++)
    {     for ( j = 0; j < columns ; j ++)
          {
             printf( "format specifier", array_name[i][j] );
          }
    }
```

Observe the following examples for better understanding.

Read and display of array Elements:

```
 main()
{
   int i, a[5];
   clrscr();
   printf("enter 5 elements\n");
for( i = 0;i < 5;i++)
          {
          scanf("%d",&a[i]);
          }
   printf("Entered elements are:\n");
      for( i = 0; i < 5; i++)
     {
       printf("\n a[%d] = %d", i, a[i]);
      }
         getch();
}
```

One Dimensional Array

```
main()
{
    int i,j,a[3][2];
     clrscr();
      printf("enter elements\n");
for(i=0;i<3;i++)
          {   for(j=0;j<2;j++)
                scanf("%d",&a[i][j]);
           }
    printf("\n Given elements are :");
for(i=0;i<3;i++)
        {  for(j=0;j<2;j++)
             printf("\n a[%d][%d]=%d", i,j,a[i][j]);
          }
          getch();
}
```

Two Dimensional Array

Strings

"**A string is a collection of characters terminated by a null character ('\0').**

Declaration:-

Syntax: ***char stringname[size];***

e.g.:- char str[10];

1. Each string must be terminated by a null value. i.e."\0".
2. "\0" indicates the ending of the string.
3. If a string doesn't contain "\0" then compiler treats them as a collection of individual characters.

Initialization:-

A string can be initialized like a one dimensional array. We can initialize them in two ways.

1. *char name[10] = {'B','H','A','R','A','T','\0'};*

In this type of initialization each character must be in single quotes and separated by commas at the end of the string user have to insert '\0'. ('\0' is treated as a single character.)

2. *char name[7] = "Bharat};*

(or)

char name[ ] = "Bharat";

In this the string must enclosed be in double quotes. In this type of initialization the compiler automatically adds '\0' at the end of the string.

In the above example 'name' string has the size 7, but it really contains 6 characters only. The last space is occupied by '\0'. So the actual string size is one less than the size of the string.

Reading a string:

To read a string '%s' format specifier is used. Hence the scanf statement is written as

scanf("%s",stringname);

Since string is an array of characters, so it is not necessary to specify the subscript. '&' and is not required in scanf statement.

e.g:- *char name[20];*
scanf("%s",name);

This equals to

for(i=0;i<20;i++)
scanf("%c",&name[i]);

scanf() reads a group of characters without any space between them. So, scanf() does not read a group of words or sentences.

char message[50] = "India Is Heaven on Earth";
scanf("%s", message);

In the above statement only "India" will be read through the scanf() statement. If we want to read the whole sentence we need more scanf() statements. To avoid this situation another function is used to read a group of words or sentences. i.e. "gets()".

Or scanf("%[^\n]",str)

Instead of writing scanf("%s",s), we must write: scanf("%[^\n]s",s) which instructs the compiler to store the string s while the new line (\n) is encountered. Let's consider the following example to store the space-separated strings.

```
1. #include<stdio.h>
2. void main ()
3. {
4.     char s[20];
5.     printf("Enter the string?");
6.     scanf("%[^\n]s",s);
7.     printf("You entered %s",s);
8. }
```

Output

```
Enter the string?This is the best
You entered This is the best
```

Here we must also notice that we do not need to use address of (&) operator in scanf to store a string since string s is an array of characters and the name of the array, i.e., s indicates the base address of the string (character array) therefore we need not use & with it.

Syntax:- ***gets(string);***

gets() can read a multiword string.

e.g.:- *char message[50];*
printf("enter a message\n");
gets(message);

(or)

gets("India Is Heaven on Earth");

Printing a string:-

To print a string we use to **printf()** function with **'%s'** format specifier.

e.g.:- *char message[50] = "A sound mind in a sound body ";*
printf("%s", message);

Another function is also available to print a string i.e. **puts()**

Syntax: ***puts(string name); (OR) puts("String");***

e.g.:- *char message[50] = "Strength is life weakness is death ";*
puts(message);

OR

puts("Strength is life weakness is death");

Generally if the string is read by scanf() statement then we use printf() function to print it. If we read the string by using gets() then we can use puts() to print the string

Array of strings:

If we want to work with much number of strings then we declare them as an array. But string is a character array so we need two dimensional character arrays for this purpose.

Syntax: ***char stringname[numer of strings][size];***

e.g:- *char name[5] [10];*

In the above example we create 5 strings and each string is 10 characters long.

The two dimensional strings can be initialized like below.

char name[5][10] = { "Bhagat","Subhash","Jhansi","Alluri","Kesari"};

OR

char name[0] ="Bhagat";

char name[1] = "Subhash";

char name[2] = "Jhansi";

char name[3] = "Alluri";

char name[4] = "Kesari";

We can access the two dimensional strings by specifying their index numbers.

e.g.: *scanf("%s" , name[0]);* - This will reads the first string from the keyboard.

printf("%s", name[2]); - This will prints the third string on the screen.

Some facts about Strings:

→ We cannot assign a string to another directly.

e.g.:- *char name1[20], name2[20];*
char name1[20]= "Bharathi";

name2 = name1; → *it is not allowed.*

→ We cannot join two strings directly by using arithmetic operators.

e.g.:- *string1[]="Andhra";*
string2[]="Pradesh";
string3 = string1 + string2; → *it is not allowed*

→ We cannot compare two strings directly.

char name1[] = "Kalyani";
char name2[] = "Anil";

if(name1 = = name 2) → *not allowed*

So we cannot apply arithmetic and relational operators directly on the strings.

→ Array elements are stored in continuous memory locations. So we can access them in sequence. Since strings are character arrays, if we knew the starting address of the string then user can access the whole string by incrementing the address with one.

→ For manipulation of strings C provides a set of functions. All these are available in **string.h** file. If we want to use these functions then we must include this header file at the beginning of the program.

String Functions: Some of frequently used sting functions are

1. strlen():- This function is used to find the length of the given string.

Syntax: strlen(string);

This function returns a positive integer value that gives the length of the string.

2. strcpy():-This function is used to copy a string into another string.

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

In the above, string1 contents are copied into string2.

3.strcmp():-

This function is used to compare two strings. It will compare character by character of the supplied two strings. This will treats lowercase letters and uppercase letters differently.

Syntax:- ***strcmp(string1,string2);***

This function returns a numerical value.(i.e ASCII values difference of the comparing characters)

it returns 0 (ZERO) → if string1 and string2 are equal.
< 0 (negative value) → if string1 character is less than string2
> 0(Positive value) → if string1 is greater than string2.

It returns the numerical difference between the ASCII values of the non-matching characters.

4.strcat()- This function is used to append (or) concatenate a string at the end of another string.

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

String2 is joined at the end of string1.

5.strlwr():- It converts all the characters of the string to lower case.

Syntax: strlwr(string);

6.strupr():- It converts all the characters of the string to upper case.

Syntax: strupr(string);

7.strrev():- It reverses the given string.

Syntax: strrev(string);

8.stricmp():- This function compares two strings by ignoring case.

Syntax:- ***stricmp(string1,string2);***

9.strchr():- It returns the first occurrence of the given character in the specified string.

Syntax:- ***strchr(string,'character');***

10.strrchr():- It returns the last occurrence of the given character in the specified string.

Syntax:- ***strchr(string,'character');***

11.strstr():- It finds the first occurrence of the given string in another string.

Syntax:- ***strstr(Main string, searching string);***

12.strset():- This function set all the characters of the given string to a specified character.

Syntax:- ***strset(string,'character');***

13.strrcpy(): This function will copy a specified number of characters from one string to another string.

Syntax:- ***strcpy(string2, string1, n);***

14. strncat(): This function will concatenates specified number of characters at the end of another string.

Syntax:- ***strncat(string1,string2,n)***

POINTERS

Pointer is a variable, which contains the address of another variable.

(or)

A pointer is a variable that references a memory location in which data is stored. Each memory cell in the computer has an address that can be used to access that location

Declaration of pointers:- A pointer can be declared like a normal variable.

Syntax:- datatype *pointername;

E.g.:- *int *p;*

Multiple pointers can also be declared with a single statement like below.

data type *pointer1 , *pointer2, *pointer3,……., *pointer n ;

- Pointer variable name must be preceded by '*'(astrich)

Initialization of pointer:

Pointer variable can be initialized with the address of another variable.

Syntax:- pointer name = & variable name;

Ex:- *int a;*
*int *p;*
p = &a;

- User can also initialize a pointer at the time of its declaration.

*int a, *x = &a;*

Accessing of pointers

A pointer variable can contain the address of another variable, but the real value of the variable not stored at that location.

To get the actual value of a variable which is pointed by a pointer, user has to use the * (Asteric) operator that is known as indirection operator or 'Value at' operator.

e.g:- int x =10;
int *p;
p = &x;

p → gives the address of ***x*** .

p**→ gives the value of ***x .

Suppose **x** is located at 1001 location then **p** gives 1001.
***p** gives the value of **x** 10.

Pointers arithmetic:

Generally we can perform all arithmetic, relational and logical operations on values at pointer variables such as addition, subtraction, division, comparison etc…

Ex:- int a =10, b = 5;
int *p = &a ,*q = &b;

**p + *q; *p -*q, *p * *q, *p / (*q)*

Scale Factor:

The length of data type is called as scale factor.

Ex:- *int x=10;*
*int *p;*
p = &x;

- Suppose *x* is stored at the location 1001 that value is assigned to the pointer '*p'*. If the user increment the pointer by using *p*++ then the address of *p* is incremented by 2. If the user decrements the pointer by using *p*- - then the address of *p* is '999'.

Like that if *p* is float, then p++ means the address is incremented by 4. If *p* is a character variable then *p*value is incremented by one.

Pointers and Arrays:

There is a close relation between pointers and arrays because both are dealing with addresses. When an array is declared then compiler allocates a base address and sufficient memory to that array. Generally array elements are stored in contiguous locations. The base address is the address of the first location of that array.

" *Array name itself is a pointer which holds the address of the first location address."*

```
int arr[10];
int *p;
p = &arr[0];
p = arr;
```

both are same.

Now we can access any element of the array by incrementing are decrementing the pointer *p* value.

Accessing array elements using pointers

Generally user can access array elements by specifying its index numbers with its name. To access the pointer array elements are* *(p + 0), *(p + 1) , *(p + 2).*

P+0 is equal to &a[0]

**(p+0) is equal to a[0].*

Pointers and two dimensional arrays

A two dimensional array is declare by using pointers like below.

data type *pointer name[no.of.rows];

Ex:- int *a[10];

Here 'a' points to the first row of '10' elements array. (a+1) points to the second row of 10 elements array and so on …….

(a+i) points to (i+1)th row of 10 elements array.

Remember the following notations :

***(p+i)+j is equal to &a[i][j]**. Here '**a'** is normal array name and **'p'** is a pointer.

***(*(p+i)+j) is equal to a[i][j]**.

```
main()
{
  int arr[5] = {1,2,3,4,5};
  int *ptr,i;
   clrscr ( );
ptr = arr;
    for (i=0;i<5;i++)
    printf ("\n a[%d]=%d",i, arr[i]);
    for (i=0;i<5;i++)
    printf("\n *(p+%d)=%d",i,*(p+i));
}
```

One Dimensional Array

```
main()
{
 int arr[3][3] = {1,2,3,4,5,6,7,8,9};
   int *ptr,i,j;
  clrscr ( );
ptr = arr;
      for (i=0;i<3;i++)
    {
        for (j=0;j<5;j++)
     printf("\t%d",*(*( arr+i)+j ) );
  }
```

Two Dimensional Array

Array of pointers

Generally a group of same data type elements are declared as an array like that a group of same data type pointers can also be declared as an array.

data type *pointer name[exp];

eg:- int *p[5];

'p' is a group of integer pointers it contains the address of 5 different integer variables.

An array of pointers even can contain the addresses of other array elements.

```
main( )
{ int i,a=5,b=9,c=8,d=9,e=10;
  int *p[5];
p[0]=&a;
p[1]=&b;
p[2]=&c;
p[3]=&d;
p[4]=&e;
for (i=0;i<5;i++)
printf ("\n%d",*p[i]);
}
```

Pointers and strings:-

Strings are characters arrays so we can use pointers on strings like an array. We can declare and initialize the strings directly by using pointers. In this declaration we need not to mention the size of the string.

e.g:
```
char *p= {"welcome to c"};
printf ("%s", p );
```

We can declare a set of strings by using an array of pointers. In this each pointer can point to an individual string.

e.g:
```
char *flower[5]={"Lotus", "Rose", "Lilly", "Jasmine", "Tulip"};
int i;
for(i=0;i<5;i++)
printf("\n%s", name[i]);
```

Pointers and Functions

Generally function calls are two types.

1.Call by value:- In this method , we will pass direct values to the function.

2. Call by reference:-

In this method, we will pass the addresses of the arguments to the function. Whenever the user passes addresses to the functions they must be received by pointers only. So the use of pointers in call by reference is compulsory.

```
#include<stdio.h>
main( )
{
    int a=10;
    clrscr();
    change (&a);
    getch();
}

change (int *x)

 {
   *x = *x+10;

  printf("\n Changed X is:%d",*x);

 }
```

Call by reference

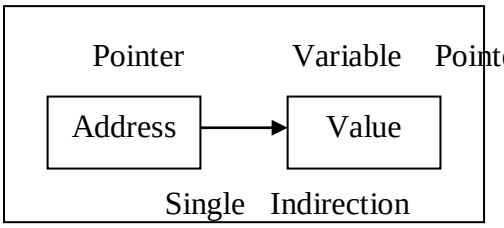
```
#include<stdio.h>
int mul(int ,int );
int (*p) ( int , int);

 main()
{ int a,b,c ;
  printf( "Enter 2 integers");
   scanf( "%d %d", &a,&b);
  p = mul;
  c =  (*p) (a,b);
printf ( "Result = %d", c);
getch();
}

int mul(int x, int y)
 {
     return ( x* y);
 }
```

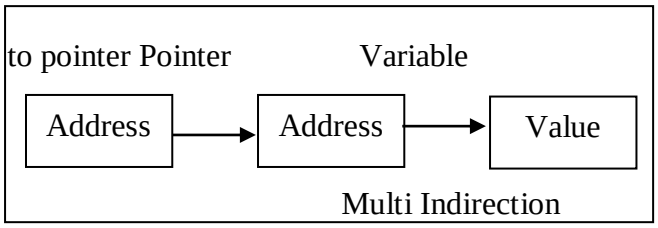
Function pointer

Pointers to Pointers: (Or) Double pointer (Or) Multi indirection

Generally a pointer points to another variable. But user can declare a pointer, which can points to another pointer. This is known as 'multi indirection' (or) "pointer to pointer".

In the case of pointer to pointer the first's pointer contains the address of the second pointer, which points to the actual object that contains the desired value.





We can declare these pointers to pointers by using two asterisks (*) marks like below.

e.g.:
```
int a =10;
int *p, **q;
p = &a;
q = &p;
printf ("\n the value of a is: %d", **q);
```

In the above, **p** is the pointer and **q** is pointer to pointer.

Uses of pointers

1. Pointers reduce the length and complexity of a program.
2. They increase the execution speed.
3. Pointers can easily access the memory elements and they can save the memory space.
4. Pointers get the memory in the run time of a program (Dynamic Memory Allocation).
5. By using pointers it is easy to represent and handle multidimensional arrays.
6. Passing arrays and strings to a function is possible.
7. Pointers can help in returning more than one value from a function.
8. A pointer can also access a variable from the outside of a function.
9. Creating data structures such as linked list, tress, graphs is easy.

Problems with pointers

1. A pointer contains garbage value until it is initialized so, they must be initialized properly.
2. When we work with a wrong pointer, every time you use it may cause read (or) write to the other memory location. It leads to wrong results of the program.
3. Some operators may confuse the users by using with the pointers such as.
   *p++, *p [ ], *(p + i) , * (*(p+i)+j),**p++ etc.
4. Identifying errors in pointers is difficult.

Void Pointer:

Generally we can define pointers with a specific data types. We may confuse to change these pointers from one data type to another. To avoid such situation, C- provides a pointer, which has no data type. That is the void pointer. We can declare this as

Syntax***:*** ***void *pointer name***;

e.g.: void *s:

We can change this pointer to any type.

Dynamic Memory Allocation:

Generally compiler allocates memory for the program variables at compilation time. This is called as ***static memory Allocation***.

But pointer variables can get their memory while in the run time of the program. This is called as "***Dynamic Memory Allocation***".

Memory Management Functions:

The following functions are used in c for purpose of memory management which can be used to allocate and free memory during the program execution.

Function	Task
malloc( )	Allocates memory requests size of bytes and returns a pointer to the Ist byte of allocated space
calloc ( )	Allocates space for an array of elements initializes them to zero and returns a pointer to the memory
free( )	Frees previously allocated space
realloc ( )	Modifies the size of previously allocated space.

1. malloc( ):

This function allocates memory in requested size as a single block and returns the starting address of that block as a pointer. Thisfunction returnsthe "void" memory.

Syntax:

void * malloc (size (or) no. of bytes);

e.g.: *int *p;*
*p = (int *) malloc (20);*

(OR)

*p = (int *) malloc (sizeof(int)*10);*

In the above case we can convert the void pointer into integer by using (int *). The malloc( ) allocates memory to store 10 integers that is 20 bytes.

2. calloc ( ):

This is another memory management function. This will allocate multiple blocks of memory each of same size.

Syntax: **void * calloc (no. of blocks , block size);**

e.g.: *int *p;*
p =calloc (10, sizeof (int));
(Or)
p= calloc (10, 2);

In this example calloc( ) allocates 10 memory blocks and each block is of 2 bytes size.

3. realloc( ):

Sometimes the pre-allocated memory is not sufficient and the user wants to allocate additional memory to a pointer then he can use the realloc ( ) function.

Syntax: ***void *realloc (*p, n);***

Here **p** is a pointer and '**n**' is the number of bytes required as additional memory.

e.g.: *int *p, *q;*

*p = (int *)malloc (200);*

realloc(p, 250);

4. free( ):

This function is used to release the memory, which is unwanted by the program.

It frees the memory which is allocated by either *calloc( )* or *malloc ( )* function.Once the memory is freed that memory may be used by another data contents.

Syntax: ***void free(void *p);***

e.g.: *int *p;*

*p = (int *) malloc (200);*

free (p);

FUNCTIONS

C is a modular or structured programming language. Modular programming means dividing a large program into small maintainable independent parts and each part can performs a specified task. These small segments are known as modules. In C this modular programming is possible with functions.

Definition:

A function is a self-contained block of statements that perform a specific task.

Every C program must contain one or more functions. A function called main ( ) is used in every C program. The execution of C program always begins with the instructions of main ( ).

Functions can be classified into two types.

1. Library functions
2. User Defined Functions. (UDF)

Library functions

The functions that are defined with in the system are called *System Defined* or *Standard in-built functions.*

Library functions are divided into several types depend on their characteristics. Some of them are

1. Mathematical functions (math.h)
2. Character functions (ctype.h)
3. String functions (string.h)
4. Time & date functions (time.h)

1. Mathematical functions:

These functions are used to carryout mathematical operations. These functions are available in " *math.h "* header file. Some of the mathematical functions are

1. abs ( ) 2. sqrt ( ) 3. floor( ) 4. pow( ) 5. ceil ( ) 6. exp( )
7. log( ) 8. sin ( ) 9. cos( ) 10. tan( ) etc..

2. Character functions:

These functions will operate on characters. These functions are defined in "*ctype.h*" file.

1. isupper ( ) 2. islower ( ) 3. toupper ( ) 4. tolower ( ) 5. isalpha( )
6. isalnum( ) 7. isdigit( ) 8. isspace( ) 9. isxdigit ( ) 10. ispunct( ) etc..

3. string functions

The functions which operates on strings are known as string functions. These functions are defined in '*string.h'* file. Some of the string functions are.

1. strlen ( ); 2. strcpy ( ); 3. strcmp( ); 4. strcat( ) etc…

USER DEFINED FUNCTIONS

The functions, which are defined by the user for his own purpose, are known as UserDefined Functions.

Features of Function:

- Use of Functions gives a Structured Programming Approach.
- Functions make the large and complex programs simple.
- Functions use less memory space and increase the program execution speed.
- Reduces Program Size by reusing the same code (reusability)
- Functions can avoid repetition of code and made the program portable.
- Testing and debugging is easy.

User defined function in a program will be implemented in three steps. They are

1. Function declaration (Prototype)
2. Function definition
3. Function call

Function Declaration (Function Prototype):

A function can be declared like other variables in a program. This declaration is also called as **Function Prototype**

The general form of function prototype is:

return _datatype function_name (arguments list if any);

Here ***return_ datatype*** - specifies the data type of the function returning value, ***Function_name*** - is a valid identifier and the ***Arguments / Parameters*** - are the data values that we have to pass to a function.

e.g: **float *simple (int, float, int) ;***

It is good a programming practice to specify the function prototype before the main ( ).

Function Definition: -

The definition is the place where the actual function code is placed.

The function definition will contain two parts.

1. Header part
2. Body part.

The header part contains the function prototype and the body part contains declarations and executable C statements.

return_datatype function_name (arguments list if any)

{

Local variable declarations;

Statements list;

return statement;

}

return statement:-

This is used to return some value from the called function to the calling function

It is of the form.

return (expression); OR **return(value);**

If a function returns nothing then the return statement is of the form

return;

1. A function without a return statement cannot return any value.
2. A return statement can occur any where in the body of the function and a function can have more than one return statement.
3. When a return statement is occurred in a function the compiler terminates the execution of the function and return to called function, the next statement of the call statement.
4. A return statement can return only one value at a time.

Function Call (Accessing functions):

We can call a function by specifying its name followed by a list of variables enclosed in parentheses separated by commas. If the called function doesn't required any arguments then, an empty parentheses must follow the name of the function.

Syntax:

[variable=] function_name([arguments list]);

About functions

- Every 'C' program must have at least one function. main( ) is the function name we found commonly in all C programs.
- A function, which is calling another function, is known as "calling function" and the function, which is called, by another function is known as "called function.
  Forexample, a function clrscr( ) is used in main( ) function, the main ( ) is calling function and clrscr( ) is called function.
- A function returns integer value by default.
- The arguments appeared in the function definition and the function declaration are known as formal arguments. The arguments appeared in the function call are known as actual arguments.

Types of user defined functions:

Based on the no. of arguments and the return values functions are classified into 4 types. They are

1. Functions with arguments with return values.
2. Functions with arguments without return values.
3. Functions without arguments with return values.
4. Functions without arguments and without return values.

1. Functions with arguments and with return values:

This type of function definition looks like below.

```
return_datatype functionName(type arg1,type arg2,type arg3,. ...,type argn)
 {
   local declarations;
    Statements;
    . . . . . .;
  return(value);
}
```

```
include <stdio.h>
//function definition
int sum3(int a,int b, int c)
{
 int sum=0;
 sum=a+b+c;
return(sum);
}

int main()
{
int x,y,z;
int s=0;
printf("Enter three integers:");
scanf("%d%d%d",&x,&y,&z);
s=sum3(x,y,z);  //function call
printf("sum of given 3 numbers is:%d",sum);
getch();
return 0;
}
```

2. Functions with arguments and without return values:

This type of function definition looks like below.

```
void functionName(type arg1,type arg2,type arg3,. ...,type argn)
 /*here 'void' indicates this function can't return any value.*/
  {
    local declarations;
     Statements;
     . . . . . .;
   // No Need of 'return' statement

  }
```

```
include <stdio.h>
//function definition
void sum3(int a,int b, int c)
{
 int sum=0;
 sum=a+b+c;
 printf("sum of given 3 numbers is:%d",sum);

}

void main()
{
int x,y,z;
printf("Enter three integers:");
scanf("%d%d%d",&x,&y,&z);
sum3(x,y,z);  //function call
getch();
}
```

3. Functions without arguments and with return values:

This type of function definition looks like below.

```
return_datatype functionName() // function has no arguments
  {
    local declarations;
     Statements;
     . . . . . .;
   return(value);
 }
```

```
include <stdio.h>
//function definition
int sum3()
{
 int x,y,z;
 int sum=0;
 printf("Enter three integers:");
 scanf("%d%d%d",&x,&y,&z);
 sum=x+y+z;
return(sum);
}

int main()
{
  int s=0;
  s=sum3(x,y,z);  //function call
  printf("sum of given 3 numbers is:%d",s);
  getch();
  return 0;
}
```

4. Functions without arguments and without return values:

This type of function definition looks like below.

```
void functionName()
 /*here 'void' indicates this
function can't return any value.*/
  {
   local declarations;
      Statements;
      . . . . . .;
   // No Need of 'return' statement
        }
```

```
include <stdio.h>
void sum3()       //function definition
{
int a,b,c;
int sum=0;
printf("Enter three integers:");
scanf("%d%d%d",&a,&b,&c);
sum=a+b=c;
printf("sum of given 3 numbers is:%d",a);
}

void main()
{
sum3();  //function call
getch();

}
```

Passing values to functions(Parameter passing mechanism):-

The communication between function will be done through the parameters. Generally we can pass arguments to the functions in two ways. They are

1. Call by Value
2. Call by Reference (address)

Call by Value:-

The default mechanism is pass the values of actual arguments to the formal arguments that is known as call by value.

That means we are passing the direct values to the arguments. So the changes made in the formal arguments (in the called function) will not reflect the values of actual arguments (in the calling function).

```
void   swap(int a, int b);
void main( )
{
 int a=10, b=15;
 clrscr( );
  printf("\n  Before calling the function A=%d \t B = %d", a,b);
 swap(a,b);
printf("\n  After calling the function A=%d \t B = %d", a,b);
getch( );
}
int swap(int a, int a)
{
  int c =a;
   a = b;
   b=c ;
  printf("\n  Within the function  A = %d \t  B =%d",a,b);
}
```

Output *Before calling the function A= 10 B = 15*
Within the function A = 15 B =10
After calling the function A= 10 B = 15

Call by Reference:

By a function call if we pass the addresses of the actual arguments to the formal arguments then it is known as ***call by reference.***

In this, if we change the formal arguments then the corresponding actual arguments will also be changed.

```
void swap(int *, int *);

void main( )

    {   int a=10, b=15;
        clrscr( );
        printf("\n Before Swap A=%d \t B=%d",a,b);
        swap(&a,&b);
        printf("\n After Swap A=%d\t B= %d",a,b);
    }

int swap(int *a, int *b)
{
   int t;
         t = *a;
        *a = *b;
        *b = t;
        printf("\n With in the swap:  A = %d\t,B = %d",*a,*b);
}
```

Output

Before Swap A= 10 B=15
With in the swap : A = 15 B =10
After Swap A= 15 B=10

Passing Array to a Function:

Passing individual elements – We can pass the individual elements of an array either by passing their values or their addresses**.**

Eg:

```
main()
{
    int a[5]={1,2,3,4,5};
    func(a[3]);
}
void func(int num)
{
     printf("%d",num);
}
```

Passing individual element

```
main()
{
    int a[5]={1,2,3,4,5};
    func(&a[3]);
}
void func(int* num)
{
     printf("%d",*num);
}
```

Passing individual element's address

Passing the entire array -

We can pass a single-dimension array as an argument to a function. When an entire array is to be sent to the called function, the calling function just needs to pass the name of the array. We can pass array as a formal argument in one of the following three ways and all three declaration methods gives same results because array name itself is a pointer which holds the address of the starting element of the array so the compiler understands that an integer pointer is going to be received through formal argument. Similarly, you can pass multi-dimensional arrays as formal parameters.

Consider the following function definition syntax to pass an array to the function:

Method 1:

```
return_datatype myFunction(int *array_name)
 {
        local declarations;
         Statements;
        . . . . . .;
         return(value);
 }
```

Method 2:

```
return_datatype myFunction(int array_name[size])
 {
        .
        .
}
```

Method 3:

```
return_datatype myFunction(int array_name[])
 {
        .
        .                .
 }
```

```
// Program to calculate the sum of array elements by passing to a function
#include <stdio.h>
int sum(int marks[]);
int main()
 {
    int result, marks[] = {70,68,72,63,54};
        result = sum(marks);   // marks[] array is passed to sum()function
    printf("Result = %d", result);
    return 0;
}

int sum(int m[]) {
  int sum = 0;
  for (int i = 0; i < 5; ++i) {
                sum =sum+ m[i];
  }
  return sum;
}
```

NOTE: In function call we can pass only the array name to the function.

Inserting an element in an array:

1. First get the element to be inserted, say x
2. Then get the position at which this element is to be inserted, say pos
3. Then shift the array elements from this position to one position forward, and do this for all the other elements next to pos.
4. Insert the element x now at the position pos.

```
void main()
{
    int arr[100];;
    int i, x, pos,n;
    printf("Enter the size of the array:");
    scanf("%d",&n);
    ptrintf("Enter the elements");
    for (i = 0; i < n; i++)
        scanf("%d",&arr[i]);

    // element to be inserted
    x = 50;

    // position at which element
    // is to be inserted
    pos = 5;

    // increase the size by 1
    n++;

    // shift elements forward
    for (i = n-1; i >= pos; i--)
        arr[i] = arr[i - 1];

    // insert x at pos
    arr[pos - 1] = x;

    // print the updated array
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
 }
```

Deleting an element from an array:

```
void main()
{
    int arr[100];;
    int i,pos,n;
    printf("Enter the size of the array:");
    scanf("%d",&n);
    ptrintf("Enter the elements");
    for (i = 0; i < n; i++)
        scanf("%d",&arr[i]);
    printf("enter the position");
     scanf("%d",&pos);

    for (i = pos; i < n-1; i++)
        arr[i] = arr[i + 1];
    n=n-1;
   // print the updated array
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

STORAGE - CLASSES

Generally variables are characterized by means of two types.

1. Data type
2. Storage class

Data type refers to the type of information represented by a variable.

A storage class refers to

- **Scope of the variable.**
  That is in what portions of the program a variable is available to the user to use.
- **Life of the variable**
  How long a variable can exists and returns its value.
- **Storage place of the variable**
  This decides that a variable would be stored in the CPU registers or in the RAM.
- **Initial Value of the variable**
  What will be the initial value of the variable?

In C we have four types of storage classes.

1. Automatic storage class.
2. External storage class
3. Static storage class.
4. Register storage class.

All these storage classes are identified by four keywords they are

1. **auto**
2. **extern**
3. **static**
4. **register**

Generally we will specify the storage class to a variable like below

Syntax:- **storageclass datatype variable_ name;**

Automatic variables

These are also called as local variables. These are declared inside a function. These are confined to that function only. These variables are created when the control entered into the function and are destroyed while exit the function. Any variable declared within the function is by default an automatic variable. These variables are created by using the keyword **"*auto*"**

. It is in the form

auto data type variable name;

The keyword "**auto**" is not compulsory before the variable.

ex:- *auto int x;*

int x;

in above both are same.

- Memory for the local variables is allocated only when needed.
- These variables can be prevented from unwanted changes
- User must declare the local variables at starting of the block.
- Local variables cannot retain their values between function calls.
- Initial value of a local variable may be a garbage value.

Auto variable

```
main( )

{       auto int a=10;
        clrscr( );
        printf("\n  A in main is:%d",a);
        fun_l( );
        fun_2( );
        getch( );
}

fun_1( )
{
        auto int a =30;

        printf("\n  A in fun1 is:%d",a);
}

fun_2( )
{
  auto int a=20;

  printf("\n  A in fun2 is:%d",a);
}
```

Global Variable

```
#include<stdio.h>

int m=10;
void main( )
{
  clrscr( );
   printf("\n M in main  is :%d",m);
        fun1( );
        fun2( );
   getch( );
}

fun_1( )
{
  m = m +10;
  printf("\n M in fun1 is:%d",m);
}

fun_2( )
{
        m = m – 10;
   printf("\n M in func2 is : %d",m);
}
```

External (or) Global variables

These are declared out side of a function. These are available to all functions through out the program execution. These are also known as global variables. These can be accessed by any function of that program.

e.g: See the above example .

```
#include<stdio.h>
int m=10;
void main( )
{
  clrscr( );
   printf("\n M in main  is :%d",m);
         fun1( );
         fun2( );
   getch( );
}
fun_1( )
{
  m = m +10;
  printf("\n M in fun1 is:%d",m);
}
fun_2( )
{
         m = m – 10;
   printf("\n M in func2 is : %d",m);
}
```

Case :1

```
#include<stdio.h>
void main( )
{
  clrscr( );
   printf("\n M in main  is :%d",m);
         fun_1( );
         fun_2( );
   getch( );
}
int m=10;
fun_1( )
{
  m = m +10;
  printf("\n M in fun1 is:%d",m);
}
fun_2( )
{
         m = m – 10;
   printf("\n M in func2 is : %d",m);
}
```

Case : 2

Global variables are available only from the point of declaration to the end of program. If we declared a global variable before the first function then all the functions of that program can access this variable. If we declared that after the first function then it is available from the second function onwards to the end of the program

In the above case: 1 'm' is available for main ( ), fun_1( ) and fun_2( ).

In the case :2 'm' is available only fun_1( ) and fun_2( ). It is not available for main( ).

If we access the '*m*' variable in main( ) then we have to declare that variable as **extern** variable.

```
void main( )
{
extern int m;
  clrscr( );
   printf("\n M in main  is :%d",m);
         fun1( );
         fun2( );
   getch( );
}
int m=10;
fun_1( )
{
  m = m +10;
  printf("\n M in fun1 is:%d",m);
}
fun_2( )
{
    m = m – 10;
   printf("\n M in func2 is : %d",m);
}
```

When a variable is declared as **extern** then the compiler understood that the variable is declared somewhere else. So it doesn't allocate separate memory location for that variable. It just finds the type and name of the variable and automatically it refers to the previous declared variable.

If two files are linking together then the global variables are also available in both the two files. This can be informed to compiler by using ***extern*** keyword.

1. These variables are useful when many functions are using the same data in a program.
2. Using more global variables leads to memory wastage.
3. These are frequently changed by unwanted effects.
4. Global variables stored the latest changed value.
5. These are automatically initialized to zero.
6. If the declaration of the global variable is changed then all functions using that variable will also be changed.
7. All the functions by default extern functions.
8. We declare them in the calling function without the qualifier extern.

Therefore the declaration

void fun_1( ); extern void fun_1( ); both are same

Static Variables:

These variables are created by using the keyword ***static.*** Static variables may be either global variables or local variables depending on the place of declaration. If they are declared inside a function they are static local variables and if they are declared outside a function then they are static global variables. Static local variables are alive through out the program execution. These can retain their values between function calls.

```
void main( )
{ int i;
  clrscr( );
 for ( i =0; i<5; i++ )
    {
        func_1( );
   }
  getch( );
 }
func_1( )
{
  static int a;
    a = a+5;
  printf ( "\n A= %d", a);
}
```

Static Local

```
void main( )
static int a=20;
{ int i;
  clrscr( );
    a = a + 5;
       printf("\n A = %d",a);
  fun_1( );
  fun_2( );
    getch( );
 }
func_1( )
{    a = a +25;
  printf ( "\n A= %d", a);
}
func_2 ( )
{  a = a -10;
   printf ( "\n A= %d", a);
}
```

Static Global

Output:

```
A = 5
A = 10
A = 15
A = 20
A = 25
```

```
A = 25
A = 50
A = 40
```

If we remove static before 'a', in the local case the output is A = 5, A = 5, A = 5, A =5 , A = 5 .

- Static variables are alive throughout the program although they are local variables.
- Static local variables are available in the block of function in which they are declared.
- Static variables can be initialized only once. They cannot reinitialize between function calls.
- Static variables will not be changed by external change.
- We can hide some functions or portions of the program from other functions by declared them as static.

Register Variables

Generally variables of a program can be stored in main memory (RAM). But register variables are stored in CPU registers. These variables can be created by using ***register*** keyword.

Ex: register int x,y;

Registers are the special storage areas with in the CPU. Usually a register can store 16 bits only. A register can be accessed mush faster than a memory location of RAM.

- Register variables are used to speed up the program execution.
- Generally int and char type variables can be used for register variables. Other type of variables cannot allowed to declare as register variables because they requires more than 2 bytes of memory. So they cannot be stored in registers because the size of registers is usually 2 bytes.
- We can create any number of register variables in a program. But all are not stored in CPU registers. Because they are very limited and are engaged with other works (14 to 16 registers are available in a micro computers)
- It is better to declared not more than two as register variables.
- Register and auto variables are closely related if the user declared more register variables than available registers, then some of them are treated as auto variables automatically.
- Global register variables are not allowed.
- These are not referred by the pointers that are their address is not accessible because registers are not addressable by normal programs.
- Usually most frequently used variables can be declared as register variables.

STRUCTURES

Ordinary variables store a piece of information and arrays can store a group of same data type elements. If a user wants to store different data type elements as a single entity then we have to use structures.

"**A structure is a collection of one (or) more variables of different data types grouped together under a single name**".

(Or)

"**A Structure is a set of heterogeneous elements**."

Declaration:

Structures are created by using "**struct**" keyword.

Syntax: **struct struct_name {**
datatype1 variable name;
datatype2 variable name;
:
:
} variables list;

e.g.: *struct student {*
char name[20]
int roll_ no:
float percentage;
} s1,s2;

The variables in a structure are called as members (or) fields (or) elements. Generally all the members of a structure are logically related.

Individual members of a structure can be variables or pointers or arrays or even structures also.

- The member name with in a structure must be unique. [Should not use the same name for two members].
- The declaration must be terminated by a semicolon. [because structure creation is a C-statement].

Creation of structure variables:

After defining a structure the structure variables can be created like below.

Syntax: **struct struct_name var1, var2, ------,var n;**

Where *struct* is the keyword, *struct_name* is the name of the structure *var1, var2, -----,var n* are the different variable names.

```
struct student
 {
   char  name [20];
   int    roll no;
  float   percentage;
 };

Struct student S1, S2, S3;
```

(Or)

```
struct student
 {
   char  name [20];
   int    roll no;
  float   percentage;
 } S1, S2, S3;
```

At the time of structure declaration no memory is allocated for a structure. But if we create a structure variable then the compiler will allocate sufficient memory to that variable. In the above example S1, S2, S3 are the structure variables. Compiler will allocate 26 bytes for each.
[name = 20bytes + roll_no = 2bytes + percentage= 4bytes]

Initialization of Structures:

The members of the structures can be assigned with initial values like an array. The initial values must be appearing in the same order in which the members are. All these values must be enclosed in braces and separated by commas.

Syntax: **struct struct_name variable_name = {Val 1, Val 2... Val n};**

e.g:
```
struct student
{
  char  name[20];
   int    roll no;
  float  percentage;
}S1, S2, S3;

struct student S1 = {"ramu", 999, 90.9};
```

Assignment of structures:

The information contained in one structure variable can be assigned to another structure variable of the same type using a single statement.User need not to assign the value of each member separately.

e.g.:
```
struct student
{
  char  name[20];
   int    roll no;
  float  percentage;
}S1, S2, S3;
struct student S1 = {"Ramu", 999, 90.9};
```

S2 = S1;

In the above example values of S1 are assigned to S2. (because both are same type)

Accessing of structure variables:

Individual members of a structure are accessed through the use of "**.**" (dot) operator.

Syntax: **struct_name . member_name;**

e.g.:
```
# include <stdio.h>
void main ( )

{       struct student
                                        {
          char  name[20];
                                  int     roll no;
          float   percentage;
                                  } S1, S2, S3;

struct student S1= {"RAMU", 999, 90.9};

printf ("\n name of the student is: %s", S1. name);
printf ("\n roll no of the student is: %d", S1.rollno);
printf ("\n percentage of the student is: %f", S1.percentage );
getch ( );
}
```

Nesting of structures:

A structure can be defined as a member of another structure.

e.g.:
```
struct student
{
  char  name[20];
   int    roll no;
  float  percentage;
 };

struct section {
                struct student S1;
                char   f_name [20];
                char  add [50];
                };
```

Here in the above example *section* is a structure, which contains another structure (student) as its member.

The structure, which is used as the member of another structure, must be defined first, then only it can be used as the member of another structure.

User can access this type of variables as

```
main structure_name . member structure_name. variable_name;
```

In the above example we can access *name* as

```
section . student . name;
```

Array of structures:

Generally array stores similar data type elements. But if a user wants to store different data type element as a single group, then he uses the structures as the elements of an array.

To create an array of structures, first user must define a structure and then declare the structure variables as an array.

e.g.:
```
struct student
{
  char  name[20];
   int    roll no;
  float  percentage;
 };

struct student S[10];
```

This creates ten sets of student type of variables.

Like a normal array user can access a specific structure from the array of structures.

e.g.: *S[0] → Indicates the firsts structure of the array.*
S[0]. name → will gives the name of the first student.

All elements of an array are stored in adjacent locations. So, these structures are also stored in adjacent locations.

Passing structures to functions:

Structures can also be passed to a function. Like an ordinary variable, a structure variable can be passed to a function.

User may pass individual structure members(or) the entire structure to a function. They can pass to the function either in value (or) reference.

Passing individual members of a structure to a function:

e.g.:
```
# include <stdio.h>
void main ( )
{
        struct book {
                        char title [15];
                        char author [15];
                        float price;
                        };
struct book b1 = {"A book on c", "Kelly",  350.00};
display (b1. title, b1. author, b1. price);
getch();
}
display (char *t, char  *a, float  p)
{
printf ("\n book name: %s", *t);
printf ("\n author: %s", *a);
printf ("\n price: %f", p);
}
```

Passing entire structure to a function:

```
# include <stdio.h>
void main ( )
{
        struct book {
                        char title [15];
                        char author [15];
                        float price;
                        };
struct book b1 = {"A book on c", "Kelly",  350.00};
display ( b1);
getch();
}
display (struct book b)
        {
        printf ("\n book name: %s", b.title);
        printf ("\n author: %s", b.author);
        printf ("\n price: %f", b.price);
        }
```

Structures and Pointers:

Like a pointer to a normal variable, user can also declare a pointer which points to a structure. Such pointers are known as structure pointers.

Structure members are stored in adjacent locations of the memory. So, user can assign the beginning address of the structure to a pointer. It is of the form:

Syntax: ***struct struct_name *pointer name;***

The variable and pointer declaration can be combined with structure definition.

```
struct book {
                        char title [15];
                        char author [15];
                        float price;
                        } b1, *p;
```

p = &b1;

Through the structure pointer user can access the individual members of the structure by using arrow operator. "→" [one ' – '(minus) symbol one ' > ' (greater than) symbol]

e.g.:
```
# include <stdio.h>
void main ( )
{
        struct book {
                        char title [15];
                        char author [15];
                        float price;
                        } *p;
struct book b1 = {"A book on c", "Kelly",  350.00};
p= & b1;
print f ("\n book title: %s", p -> title);
print f ("\n Author\t: %s", p -> author);
print f ("\n price\t: %f", p -> price);
getch();
}
```

Uses of Structures:

Structures can be used for:

- Database management.
- Formatting a floppy.
- Sending out put to a printer.
- Display the directory of a disc.
- Checking the memory size.
- Interacting with the mouse.
- Hiding a file from the directory.
- Drawing a graphic shape on the screen.
- Clearing the contents of the screen.
- It decreases the length of the program.
- Receiving a key from the keyword.
- To store different data type elements as a single entity.

UNIONS

Union is a derived data type like a structure.

"A union is a derived data type, which contains different data type elements but all the elements can share the same memory location. So, we can access only one member at a time".

Declaration:

Syntax:
```
union  union_name {
                    datatype1 variable name;
                    datatype2 variable name;
                    :
                    :
                   } variables list;
```

In the above ***union*** is the keyword, ***union_name*** is the name of the union.

e.g.:
```
union first
        {
           int a;
           char C;
         } u1, u2, u3;
```

Declaration of union variables:

User can declare a union variable at the time of declaration (or) by using a separate statement.

```
union  union_name {
            datatype1 variable name;
            datatype2 variable name;
            :
            :
       } variables list;
```

(OR)

union union_name var1, var2, var3….., varn;

e.g.:
```
union first
        {
            int a;
            char C;
         } u1, u2, u3;
```

(or)

```
union firstu1, u2, u3;
```

Accessing union variables:

To access a member of a union we use the same syntax that is used for structures.
" **.** " (dot) operator is used for direct operations
"→" (arrow) operator is used for pointer operations.

Points to Remember:

1. Union member can share the same memory location.
2. The union memory can be treated as a variable of one type in one occasion and another type of variable in another occasion.
3. The compiler allocates enough storage space to hold the largest member of the union.

e.g.: In the above example "first" the compiler allocates two bytes of memory because integer requires two bytes.
In the same location both variables *a* and *c* can be stored for different occasions.

e.g.:
```
# include <stdio.h>
main ( )
{
  union key {
              int  a;
             char c[5];
           };
union key u;
u . i = 512;
printf ("\n u .i = %d", u1.i):
printf ("\n u .c[0] = %d", u .c[0]);
printf ("\n u .c[1] =%d", u .c[1]);
getch();
}
```

Differences between Structures and Unions :-

Unions	**Structures**
1. The definitions start with the keyword "union"	1.The definitions start with the keyword "struct"
2. Union cannot call another union within itself.	2. Structure can call another structure within itself.
3. It is of the form : *union tag_name {* *data type 1 variable name;* *data type 2 variable name;* *:* *:* *} variable list;*	3.It is of the form: *struct tag_name {* *data type 1 variable name;* *data type 2 variable name;* *:* *:* *} variable list;*
4. The members of the union can share the same memory location.	4. The members of the structure can stored in different [adjacent] memory locations.
5. A union can include a structure in some cases only.	5. A structure can include unions any time.
6. Files cannot be accessed through unions.	6. Files can be accessed through structure.
7. Union members cannot be initialized.	7. Structure members can be initialized.
8. Consumes less memory.	8. Consumes more memory.

Bitwise Operators

One of the C's powerful is a set of bit manipulation operators. C provides a set of operators, which are used for manipulation of data at bit level. These operators are known as *bitwise operators*.

Bitwise operations refers to testing, setting the actual bits in a byte or word. These are not applying to float and double values. These are applied only on *int* and *char* values.

Bitwise operators are:

& – Bitwise and
! – Bitwise or
^ – exclusive or
~ – one's compliment
<< – left shift
>> – right shift.

Bitwise and (&)

This is a binary operator so it requires two operands. It will return one if both the bits have the value one, other wise it will return zero.

OP1	OP2	&
1	1	1
1	0	0
0	1	0
0	0	0

e.g.: a = 10→ 00001010
b = 8 →00001000
---
a & b → 00001000 = 8

Bitwise and (&) is used to test whether a particular bit is '1' (or) '0'. This is also used to set the parity bit.

Bitwise OR (¦):

This is also called as INCLUSIVE OR. It is a binary operator it will returns one if any of the operant has one otherwise it return zero. This is often used to set a particular bit to one.

OP1	OP2	¦
1	1	1
1	0	0
0	1	0
0	0	0

e.g.: a = 10→ 00001010
b = 8→ 00001000
---
a ¦ b → 00001010 =10

Bitwise exclusive or (^)

It is also a binary operator it returns one only if any one of the bit is one otherwise it returns zero.
(This operator returns one when both the bits are different. Otherwise it returns zero.)

OP1	OP2	^
1	1	0
1	0	1
0	1	1
0	0	0

e.g.: a = 10→ 00001010
b = 8→ 00001000
---
a ^ b → 00000010 =10

One's Complement operator (~)

This is a unary operator. It will inverses the state of each bit in its operand. That is zero as one and one as zero.

e.g.: a = 10→ 00001010
~ (a) → 11110101
~ (~ (a) → 00001010

Notice that sequence of two complements in a row to a value will get the original value.

~ (~ (a)) = = a.

Bitwise shift operators

The shift operators are used to move the bit pattern either to left (or) right. These operators are unary operators.
These are two types:

1. Left shift
2. Right shift.

Left shift operator (<<):

The general form of the left shift operator is

Syntax: ***variable name << no. of positions to shift;***

This operator shifts (moves) all the bits to left by number of specified positions.

e.g.: a =10 →00001010

a << 1→ 00010100 = 20

a << 1→ 00101000 = 40

a<< 3 → 01010000 = 80

As the number of bits is shift off from one end that much number of zeros are added on the other end.

Shifting is not a rotating process. So, the bits shifted off from one end, do not came around to the otherend. Once the bits are shifted off they will be lost.

The left shift of one position is usually multiplies the number by two. (See the above example.)

Right shift operator (>>)

The general form of the right shift operator is:

Syntax: ***variable name >> no. of positions to shift;***

e.g.: a = 10 -> 00001010
a >>3 -> 00000001

It shifts all the bits to right by a specified number of positions. The right shift is equal to the number divided by two.

a = 32 00100000

a>>1 00010000 =16.
a>>2 00001000 =8.
a>>3 00000100 =4.

Uses:

These are very useful when decoding the input from an external device.
These are helpful for reading the status of information.
These are useful for masking.

Masking

Masking is a process in which a given bit pattern is transformed into another bit pattern by means of another logical bitwise operand.

e.g.: Converting hexadecimal number to octal number.

Enumerated Data types

The enumerated data types help the programmer to create new data types of his own. So programmer has to give a suitable name to that data type and define the domain for that type [domain means what values that the variable of this data type can take. This is also called as range.].

The enumerated data type is created by using the keyword 'enum'. It is defined as

Syntax: ***enum tag_ name = {enumerators list (Or) Variables list };***

e.g.: enum section = { physics, chemistry, electronics, statistics, botany ,zoology, geology};
enum marital_status = {single, married, divorced, widowed};
Enumerated variables can be declared as

Syntax: *enum section s1, s2;*

Initially the compiler treats the enumerators as integers. They are automatically initialized from zero. In the above example *marital_status* , *single* is stored as 0, *married* is stored as 1, *divorced* is stored as 2 and
widowed is stored as 3.

We cannot assign a value to an enumerated variable, which is not specified in the original definition.

In the above example,

s1 = physics;
s2 = statistics;

s1 = genetics; → not allowed.

Enumeration variables are generally used within a program. These variables increase the logical clarity of a program.
e.g.: enum day = {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday} d1, d2, d3;

typedef:

C allows the user to define new names to the data types by using *typedef* keyword. Users can change the traditional data type names as the programmer desired.

The general form of *typedef* statement is

Syntax: ***typedef data type newname;***

Where data type is any valid data type, new name is the name which we want to give.

e.g.: *typedef int number;*

The above statement tells the compiler to recognize **number** is another name for ***int***.

Now user can create an integer variable by using number like below.

typedef int number;

number a, b;

File Processing

Def: A file is a place on the disk where a group of related data is stored.

The storage from of the given data in the memory is File. These data files allow us to store information permanently. We can access and alter that information of the file.

These files are of two types. They are

1. Text Files (Stream Oriented (or) Standard)
2. Binary Files (System Oriented (or) Low level)

Like most other languages, C provides an extensive set of library functions to perform some basic operations on files. The general operations we can perform on the files are

1. Opening a File
2. Naming the File
3. Reading the Data from the File.
4. Writing data to the File.
5. Closing the File.

A file can be treated as a stream of characters. A file must be open before it can create or processed. Once the processing is over, the opened file has to be closed.

In C the file control structure is defined as **FILE** structure, which is available in STDIO.H library file. To work with the files, our program needs a pointer that is known as '*file pointer*'. A file pointer is a pointer variable of type **FILE.**

We can obtain a file pointer by using a statement like below.

FILE *fp;

Here **'fp'** is the file pointer.

Opening a File: A file can be opened by using the function **'fopen ( )'**

It is of the form

fp = fopen ("filename", "mode");

Where "**fp**" is the file pointer.

"file name" is the name of the file to be open.

"**mode**" specifies the file type i.e. the manner in which the file will be utilized either for writing, reading or appending.

The modes can be any one of the following.

Mode	Meaning
"r"	Opens an existing text file for reading only
"w"	Opens a new text file for writing only (if the file with specified name already exists, it will be destroyed and a new file is created in its place.)
"a"	Opens an existing text file for appending (i.e. adds new information at the end of the file. A new file will be created, if the specified file doesn't exist.)
"rb"	Opens an existing binary file reading.
"wb"	Opens a new binary file for writing.
"ab"	Opens an existing binary file for appending.
"r+"	Opens an existing text file for both reading and writing. (First reading and next writing)
"w+"	Opens a new files for both writing and reading (First writing and next reading)
"a+"	Opens an existing file for both reading and appending (First reading and next appends the data at the end)

Closing a File:

An opened file can be closed by using the function **fclose ( )**

Syntax:

fclose (fp);

Here 'fp' is a file pointer.

Ex:

```
#include<stdio.h>
{
        char C;
        FILE *fptr;
        fptr = fopen ("sample.dat", "i");
                        :
                        :
                        :
        fclose (fptr);
}
```

Reading and Writing Files:

The I/O operations with the files are divided into two types.

1. Unformatted.
2. Formatted.

Unformatted I/O Functions:

1. getc ( ) – This is used to read a character from a file.
2. fgetc ( ) – This is also used to read a character from a file.
3. putc ( ) – This is used to write a character to a file.
4. fputc ( ) – This is also used to write a character to a file.
5. fgets ( ) – This is used to read a string from a file.
6. fputs ( ) – This is used to write a string to a file.

getc( ) & fgetc( ):

These two functions are used to read a character from a file. These can read a character from a file opened in read mode by using **'fopen( )'**. These can read the character from the file and positions the pointer to the next character in the file.

Syntax:

var = fgetc (file pointer);
var = getc (file pointer);

These functions return **EOF** when the end of the file has been reached, However these also can returns **EOF** if an error occurs.

putc( ) & fputc( ):

These two functions are used to write a character to a file.
These can write a character to a file opened in write mode by using 'fopen( )'.
These can write the character to the file at the current position of the pointer.

Syntax:

putc ('char', file pointer);
fputc ('char', file pointer);

The functions return the character written. Otherwise, it returns **EOF .**

fgets( ) & fputs( ):

These two functions work with strings. These functions work just like **putc ( ) &getc ( )**

Syntax: **fgets (string, no. of chars to be read, file pointer);**
fputs (string of chars, file pointer);

fgets ( ) reads a string from the specified file. It can read until a new line character has been read. If a new line character is read, it will be terminated.
The function will return the string if it is successful, otherwise, it will returns a null pointer.
fputs( ) writes the string to the specified file if it is successful, otherwise, it will returns **EOF** if an error occurs.

Formatted I/O functions: Formatted I/O functions are

i) fscanf( ):- This is used to read data from the standard input device. It is equivalent to normal scanf ( ) function.
ii)fprintf( ):- This writes the number of characters specified, to the standard output device. It is equivalent to normal print ( ) function.

Syntax: **fscanf (file pointer, control string, arguments);**
fprintf (file pointer, control string, arguments);
Ex: **fscanf (fp,"%s%d%f",name,&age,&bs);**
fprintf (fp,"%s%d%f",name,age,bs);

Some other functions

There are some more functions to work with files. They are

1. **feof():** This is used to find the end of the file. It returns true if the end of the file has been reached. Otherwise it returns zero (0).

Syntax: **feof(file pointer);**
Ex: **while(feof(fp))**
ch=fgetc(fp);

2. **ferror():** the ferror() determines whether a file operation has produced an error.

Syntax: **ferror(file pointer);**

3. **rewind():** This function resets the file position indicator to the beginning of the file. That is "it rewinds" the file.

Syntax: **rewind(file pointer);**

4. **remove():** This function is used to delete a specific file. It returns zero if it is successful otherwise returns a non-zero value.

Syntax: **remove(file name);**

5. **fseek():** This function is used for random searching of a file. It returns zero if it is successful otherwise returns a non-zero value.

Syntax: **fseek(file pointer, offset, place);**

Where, offset is long int, which specifies the number of bytes that have to be shifted. Place defines from where it should be shifted. Place has three values they are defined as macros in STDIO.H file. They are

Macro Name	**Place**
SEEK_SET	**Beginning of the file**
SEEK_CUR	**Current position**
SEEK_END	**End of the file**

Ex: fseek(fp, 5, SEEK_CUR);

The above statement moves the file pointer to 5 positions from current position.

6. **fread() & fwrite():** These functions are used with binary files. These are used to read and write data types that longer than one byte. These functions allow the reading and writing of blocks of type of data.

Syntax: fread(buffer pointer, size of the data type, no of variables to read, file pointer);

Where 'buffer pointer' is a pointer to region of memory that will receive the data from the file.

Syntax: fwrite(buffer pointer, size of the data type, no of variables to be read, file pointer);

Where 'buffer pointer' is a pointer to the information that will be written to the file.

Ex: fread(&d,sizeof(double),1,fp);
fwrite(&I,sizeof(int),2,fp);

7. **fflush():** This function is used to flush the contents of an output stream.

Syntax: fflush(file pointer);

This function writes the contents of any buffered data to the file associated with the file pointer. It returns zero if successful otherwise EOF.

8. **freopen():** This function is used to redirect the stream the stream o another file. This function associates an existing stream with a new file.

Syntax: freopen("new file name", "mode", old file stream);
Ex: freopen("OUTPUT", "W", stdout);

The above statement redirect 'stdout' to OUTPUT file.

9. **ftell():** This function returns the current value of the file position indicator. The value represents the no of bytes from the beginning of the file starting from zero.

Syntax: ftell(file pointer);
Ex: int pos;
FILE *fp;
Pos=ftell(fp);

Program to copy the contents of one file into another file

```
#include <stdio.h>
#include <stdlib.h> // For exit()

int main()
{
    FILE *fptr1, *fptr2;
    char filename[100], c;

    printf("Enter the filename to open for reading \n");
    scanf("%s", filename);

    // Open one file for reading
    fptr1 = fopen(filename, "r");
    if (fptr1 == NULL)
    {
        printf("Cannot open file %s \n", filename);
        exit(0);
    }

    printf("Enter the filename to open for writing \n");
    scanf("%s", filename);

    // Open another file for writing
    fptr2 = fopen(filename, "w");
    if (fptr2 == NULL)
    {
        printf("Cannot open file %s \n", filename);
        exit(0);
    }

    // Read contents from file
    c = fgetc(fptr1);
    while (c != EOF)
    {
        fputc(c, fptr2);
        c = fgetc(fptr1);
    }

    printf("\nContents copied to %s", filename);

    fclose(fptr1);
    fclose(fptr2);
    return 0;
}
```

Output: Enter the filename to open for reading
a.txt
Enter the filename to open for writing
b.txt
Contents copied to b.txt