C LANGUAGE

Name	: C
Designation	: Leading computer language
Father's name & Address	: Mr. Dennis Ritchie, AT&T, Bell Laboratories, Murray Hills, New Jersey, USA.
Year of birth	: 1972

Historical Background

Year	Name of the language	Short form	Developed by	Features
1960	ALGORITHMIC LANGUAGE-60	ALGOL-60	International Group	Too theoretical and not much useful in practice.
1963	COMBINED PROGRAMMING	CPL	The Cambridge University, USA. LANGUAGE	Very large and hard to learn as well as to use
1967	BASIC COMBINED PROGRAMMING LANGUAGE	BCPL	Mr. Martin Richards The Cambridge University, USA.	Too specific and less powerful
1970	B	B	Mr.Ken Thompson, AT&T, Bell Labs, Murray Hills, New Jersey, USA.	Too specific
1972	C	C	Mr.Dennis Ritchie, AT&T, Bell Labs, Murray Hills, New Jersey, USA.	Consists of the combined good features of both BCPL and B, and many additional features created by Mr. Dennis Ritchie.

**Status : **Middle level language

- C has fast programming ability like high level languages, but not as much as that of high level languages.
- C has program execution ability like that of low level languages, but not as much as that of low level languages, viz. Assembly language and machine language.
- Since C stands between both of them, to call it a middle level language.

BASIC ELIMENTS

In C , the basic elements include the set of **characters, keywords, identifiers, data types, constants, variables, declarations, expressions, statements and symbolic constants.**

THE BASIC ELEMENTS OF 'C'

Name of the Element	Contents/Characteristics	Name of the Element	Contents / Characteristics
1	**2**	**3**	**4**
Character set	• Alphabets • Digits • Special symbols • Blank spaces	• Upper Case A,B,C…Z(26), Lower Case a,b…z(26) • 0,1,2,3,4,5,6,7,8,9 • # , * ,-, +, \, /, ;, :, ?, ! etc. • Non-printing characters.	Words, expressions and statements can be formed by the combination of alphabets, digits and symbols
Keywords	In C, 32 Keywords are available. They are also called 'Reserved Words' because each word is reserved for a predefined purpose.	Auto, double, if, static, Break else, int, struct, Case, enum long, switch, Char, extern typedef, union, Const, float register, void, Continue return, unsigned, while, Default for, short, do, Goto, signed sizeof, volatile	• Each key word has a standard and pre-defined meaning and a pre-defined purpose as well. • The meaning or purpose of keywords cannot be changed by a user.

CONSTANTS

C constants can be divided into two major categories as:

1. Primary constants and
2. Secondary constants

Primary constants can be divided into three types:

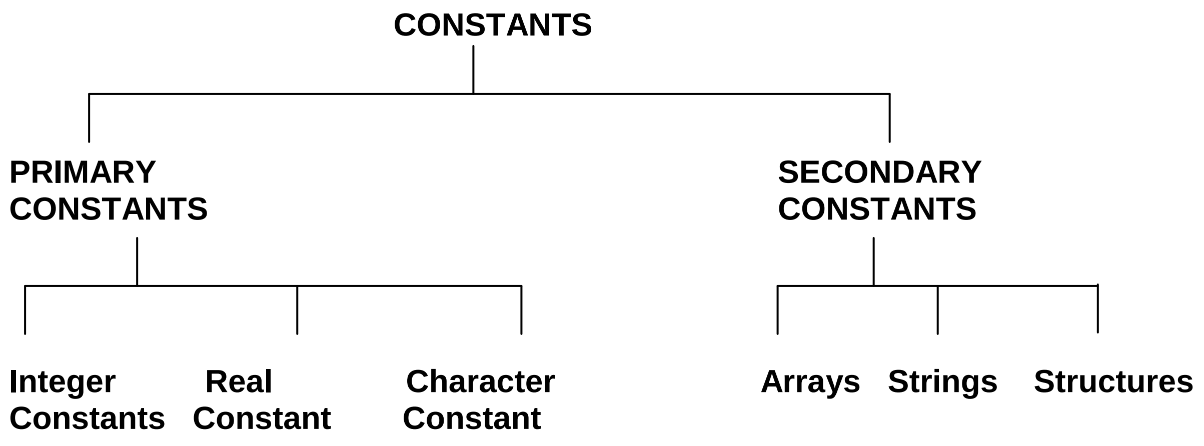
a) Integer constants

b) Real constants

c) Character constants

Secondary constants are divided into several types namely arrays, strings, structures etc.

Classification of Constants



PRIMARY CONSTANTS

Primary constants are mainly of three types. They are:

a) **<u>Integer Constants</u>**

- An integer constant is formed with an integer number.
- Integer constants do not have decimal point.
- The number may be positive or negative.
- It includes zero also.
- If no sign precedes the constant, it is taken as a positive constant.

E.g:- 32768, -25, +25, 1000 , +32767 are valid integer constants.

<u>b) Real Constants</u>

- This is also called Floating point constants.
- Real Constants must have decimal point.
- Integer constants can't express a value in fraction or decimal form. Values of temperature, height, weight, prices etc., are normally in decimal form. In such cases, we should use real numbers.
- These numbers have a decimal point and they are either positive or negative.

- If no sign precedes the number it is always positive.
- No blank spaces, commas or special symbols (except (.) decimal point) are not allowed in real constants.

E.g:- 0.25, -1.25, -0.251, +25.1

- In no value is present either on the left-hand side or the right-hand side of the real number, the normal practice is to write zero/s.

E.g:- 0.25, 5.0, 00.2, 0.20, -0.2, +7.0 (all these are valid real numbers)

C) <u>Character Constants</u>

- A character constant is a single character
- It is either a single alphabet (lower or upper case), a single digit, a special symbol or a blank space enclosed within single quotes (‘) both pointing towards the left.
- The length of the character constant must be one

E.g: ‘X’ ‘a’ ‘5’ ‘#’ ‘ ‘ (The last one is the blank space charater)

d) <u>String constants</u>

- A string constant is sequence of characters enclosed within double quotes.
- The characters include alphabets (lower case or upper case), digits, special characters and blank spaces.
- This can also be defined as an array of (sequence of) character constants, whose last characters is \0 (null) character, which is automatically placed at the end of the string by the C compiler.

E.g:- “VERY GOOD”,   “ 2000 YEAR”,   “$50K”, “A”, “ ” (Null String)

<u>E)Blank slash Character Constants :</u>

- These are collectively called ‘ESCAPE SEQUENCES’ because they are mainly used to create an escape from the present position.
- The common features of all the escape sequences are they are formed with 2 characters. The first one is always the back slash (\) character and the second one is a lower case letter.

E.g:- \n, \b, \t, \r, etc.

The following table shows a complete list of escape sequences for your ready reference.

ESCAPE SEQENCE	NAME	PURPOSE
\n	New line	Moves the cursor position to the first position of the next line
\b	Back space	Moves the cursor position to the previous position on the current line.
\t	Horizontal tab	Moves the cursor position to the next horizontal tab zone.
\r	Carriage return	Moves the cursor position to the, first position of the current line.
\a	Alert	Produces an audible alert
\f	Form feed	Moves the cursor position to the first position of the next page.
\v	Vertical tab	Moves the cursor position to the next vertical tab position.

Explanation

- **\n** (new line) this creates new line. When control encounters \n, it changes it position from the current position to the beginning of the next line.
- **\b** (back space) this moves the cursor one position to back of its current position.
- **\t** (tab) this moves the cursor to the next stop of tab. The screen of the monitor is generally divided vertically in to equal parts and each vertical part consists of 8 columns (means 8 character space) A tropical monitor screen at 80 characters width (width of 10 tab zones). When ever the cursor encounters at \t, it shifts to the beginning of next tab zone. The screen is generally divided into 10 tab zones as shown bellow.
- **\r** this takes cursor a \t in the 4th zone as shown above it's position is immediately shifted to the 5th zone.
- **\a** (Alt) its alts the cursor by making a sound. A speaker inside the computer is used to make such sounds.
- **\f** (form feed) it's used to move to the computer stationary attached to the printer to the top of the next page.
- **\v** (vertical tab) it creates a vertical tab that means the cursor moves down horizontally to next tab zone.

VARIABLES

Def:

- “A quantity which may be varied during the execution of a program is called a variable”.
- For every variable, we have to give a name so as to identify it. Hence,, it is called an identifier.

Rules for formatting variable names

- A variable name is a combination of alphabets and digits.
- The maximum number of characters in a variable name is restricted to eight (8).
- The first character of a variable name should always be an alphabet, and not digit.
- Some version allow more than 8 characters also.

Valid names : y2k, class5, h21350, yr2000, compd_int, day, age,

Invalid names : 5a, 9thclass, 2000yr, 12345.

Note: Upper and lower case case letters are valid in C. though both cases are allowed , usually C variables are written in lower case.

Syntax for variable declaration:

- Data type → Variable List;

Here data type must be a valid data type and variable list may consists of one or more identifier names separated by commas.

```
Ex: int I,j,k;
    float avg;
    char name;
```

Variable initialization

In c, a variable can be assigned with a value during its definition or during the execution of a program.

The value can be assigned to a variable with the use of assignment operator (=).

Syntax:

- Data type → variable name=constant value;

Ex: int a=10;

float pi=3.14;

char name="anitha";

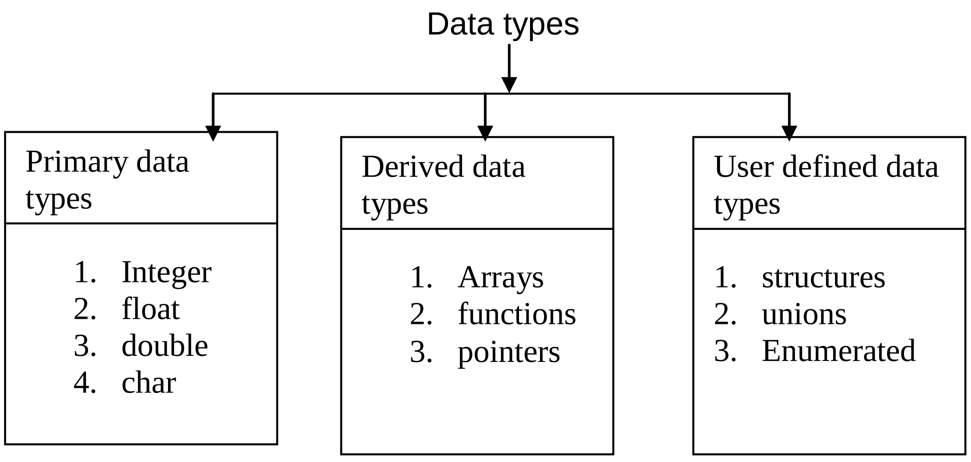
Data Types

Data types allow a programmer to create variables for manipulating data in programmes.

C , supports a wide variety of data types and the programmer can select the appropriate o he needs of the application

C supports the following data types

1. Primary data types or Fundamental data types
2. Derived data types
3. User defined data types



INTEGER TYPE

- Integer numbers include all whole numbers (including their positive values and negative values)as well as 0. They are: ...-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5,..
- The short form of the integer is int, which is keyword.
- The integer data type is basically divided into two types. They are:

  1.**signed int**

  2.unsigned int

1.Signed int:

- It covers both positive integers as well as negative integers. That means, while strong these values in the computer's memory, not only the value of the number but also its sign (plus or minus) should be stored.
- Positive sign is represented by 0 and the negative sign represented by 1. the sign (+or -)occupies a length of one bit in the memory.
- The short form of signed integer is signed int or int.

Unsigned integer

- Some variables take only positive values and they never assume negative values at all.

E.g: Number of students, Number of items etc.

In such cases, the negative sign need not to be used at all. In other words, the sign (+or -) associated with these integers need not to be stored, because, if no sign is associated with a number, we assume that it is a positive number.

- The unsigned integer type is abbreviated to unsigned int.
- It occupies two bytes (16bits) of memory space in a computer.

Short int

- This type represents fairly small values. They can either be **signed** or **unsigned.**
- Signed short integer type should be written as **short int** or **signed short int.**
- Unsigned short integer type should be written as **unsigned short int.**
- This type occupies 1 byte(8bit)of memory space. (Normally, it takes 2 bytes
- A signed short int can store values ranging from -128 to +127.
- An unsigned short int can store values ranging from 0 to 255.

Long int

- These are divided mainly to represent fairly large values. They are also divided into 2 types namely, **signed** and **unsigned.**
- **Signed long int** is abbreviated to **long int.**
- Both these types occupy 4 bytes (32bits) each in the computer's memory.

CLASSIFATION OF INTEGER DATA TYPES & OTHER FEATURES

SIGNED INTEGER TYPE

SIGNED	SHORT FORM	MEMORY SPACE	RANGE OF VALUES	CONVERSION CODE
Int	Int/signed int	2 bytes16 bits)	From -32768 To +32767	%d
Short	Short signed short/ Signed short int	1 byte (8bits)	From -128 To +127	%hd
Long	Long/ signed long/ Signed long int	4 bytes (32 bits)	From -2,147,483, 648 To +2,147,483, 647	%ld

Unsigned integer type

Unsigned	Short form	Memory space	Range of values	Conversion code
Int	Unsigned int/ unsigned	2 bytes (16 bits)	0-65,535	%u
Short	Unsigned short int/ unsigned short	1 byte (8bits)	0 – 255	%hu
Long	Unsigned long/ unsigned long int	4 bytis (32 bits)	0 to 4,294, 967, 295	%lu

FLOATING POINT TYPE

- All numbers which are expressed in decimal form, including their positive and negative values are called floating point numbers or real numbers.
- In C also, it has the same meaning.
- The short form of floating type ids the keyword **float.**
- The numbers occupy a space of 4 bytes (32bits) in memory for all 16 bit and 32 bit computers.
- It can store values ranging from 3.4 e-38 to 3.4e+38.
- In order store greater values of floating point numbers, 2 or more additional float types are created.
- They are: **double** type and **long double** type.

Double type

- It the ordinary float type is not sufficient for the program requirements, C offers a bigger one, which is the **double** data type. It is also of float type but bigger in size.
- The double data type occupies a space of 8 bytes (64 bits) in the computer memory.
- The double data type covers values ranging from 1.7e-308 to 1.7e+308.
- The double data type is represented in the short form by the keyword **double.**
- The float type which is bigger than the **double** type is the **long double.**

Long double type

- The other float type which is bigger than the double type is the **long doubles.**
- The long double type occupies 10 bytes (80 bits) in memory.
- The long double can store values ranging from 3.4e-4932 to 1.1e + 4932.
- It is represented in the short form as **long double.**
- But the double and long double type are rarely used in C programming.

CLASSIFICATION OF Floating point TYPE

Types	Short form	Memory space	Range of values	I/o conversion code
Floating point	Float	4 bytes (32 bits)	3,4e-38 to 3.4e + 38	%f
Double	Double	8 bytes	1.7e-308 to 1.7+308	%lf
Long double	Long double	10 bytes (80 bits)	3.4e-4932 to 1.1e+4932	%Lf

CHARACTER DATA TYPE

- A single character is defined as a character type data. This is represented in the short form by the keyword **char.**
- It occupies a space of 1 byte (8bits)in the computer's memory.
- The character data type can also be divided into **signed** and **unsigned.** The same definition of signed and unsigned (default with in signed and unsigned integers) are also applicable to **char** also.
- **Signed** char and **char** are one and the same. By default, the qualifier **signed** is assumed for **char.** But in the case of unsigned by these two types in the **unsigned** must be used.
- The value represented by these two types in the case of **signed char** varies from -128 to (+)127 and in the case of **unsigned char** varies from 0 to (+) 255.

Classification of C operators:

A large number of operators are available in C to perform different operations, as they are many in number, let us classify them into different categories depending upon the nature of their operations.

Then we shall discuss their features in a tabular form.

The categories of C operators are:

1. **Arithmetic operators**
2. **unary operators**
3. **relational operators**
4. **logical operators**
5. **assignment operators**
6. **conditional operators**
7. **bitwise operators**
8. **special operators**

Classification of C operators

1. **Arithmetic operators**

Operations to perform	**Symbol**	**Name**	**Purpose**
1	**2**	**3**	**4**
Undertake all basic arithmetic operations on numerical values	+ - * / %	Plus Minus Asterisk Slash Modulus operator	Addition Subtraction Multiplication Division Derives the remainder in an integer division
2.logical operators			
When one or more than one condition as to be tested, to make a decision, the logical operators are useful.	&& || !	Logical AND Logical OR NOT	Connects 2 or more conditions to arrive at a decision.
3. Relational operators			
Two quantities can be compared by using relational operators. Eg. Heights of 2 persons, weight of 2 articles, prices of 2 items etc.	< <= > >= == !=	Lesser than Less than or equal to Greater than Greater than orequal to Equal to Not equal to	All these operators are useful to compare to quantities of similar nature only.
4.unary operators			
Commonly for any operation (either mathematical or logical) two or more operands are required. The unary operators are an exception to this rule.	++ - -	Incremental operator Decrement operator	++ adds 1(one) to the operand to which it is prefixed or suffixed. E.g: ++a or a++ Similarly(--a) or (a--) is equal to (a-1) .

5.Assignement operators			
Operations to perform	**Symbol**	**Name**	**Purpose**
The assignment operator is used to assign values. It can assign integer, floating point or character values to a variable. E.g Age=25 Value=3.14159 Sex='M' Area= long *bread are valued expressions.	=	Assignment operator	This operator assigns the value given on the Right hand side of the '=' operator to the variable on the left hand side. FORMAT: VARIABLE=VALUE; The value may be a constant, variable or even an expression.
6.conditional operators			
Consists of 2 symbols , a question mark(?) and a colon(:). They are called TERNARY operators as they deal with 3 arguments.	? : used as x ?z: z format	Conditional operators or TERNARY operators	This provides an alternative and simple way for writing 'if else' statements. They cannot provide alternative statements for all if-else statements. The format Exp 1?exp 2: exp 3; Means if expressional is true, it goes to expression 2 and if expressional is false, it goes to expression 3.
7.Bitwise operators			
Operations to perform	**Symbol**	**Name**	**Purpose**
These are used to access and manipulate individual bits within a piece of data.	& | ^ << >> ~	Bit wise AND bitwise OR EX-OR Shift left Shift right One's complement	These operators are used for testing bits. They are used for shifting bits from left to right and vice versa.

Examples of Operators :

1. Arithmetic Operations

   If a=3, b=4;

Arithmetic Expression	Value
a+b	7
a-b	-1
a*b	12
a/b	1
a%b	1

2. **Relational Operators:**

   If a=10 b=4;

Relational expression	Return
a!=b (10!=4)	True
a<b (10<4)	False
a>b (10>4)	True
a==b (10==4)	False
a<=b (10<=4)	False
a>=b (10>=4)	True

3. **Logical Operators:**
   Truth tables of Logical AND (&&) ,Logical OR (||), Logical NOT (!)

Exp 1	**Exp 2**	**Exp 1 && Exp 2**	**Exp 1 || Exp 2**	**Exp 1 !**
T	**T**	**T**	**T**	**F**
T	**F**	**F**	**T**	**F**
F	**T**	**F**	**T**	**T**
F	**F**	**F**	**F**	**T**

Ex: 1
If a=10, b=4, c=3;

(a>b) && (a>c)

(10>4) && (10>3)

T && T = T

Ex :2

If a=10, b=4, c=3;

(a>b) || (a>c)

(10>4) || (10>3)

T || T = T

4. Increment and decrement operators

- ❖ Pre-increment : first to increment the value and then assigns the value

General form
++ variable;

Ex:
If x is any variable ++x

If A=5 and x = 8	Before assigning	After assigning
X=++a	X=8 a=5	X=6 a=6

- ❖ Post-increment : first to assigns the value and then increments the value

General form
Variable ++;

Ex:
If x is any variable x++

If A=5 and x = 8	Before assigning	After assigning
X=a++	X=8 a=5	X=5 a=6

Decrement operators:

- ❖ Pre- decrement: first decrement the value and then assigns the value

General Form : --variable;

Ex : if x is any variable --x;

If A=5 and x=8	Before assigning	After assigning
X= --a	X=8 a=5	X=4 a=4

❖ Post- decrement: first assigns the value and decrease the value by one

General Form : variable --;

Ex : if x is any variable x--;

If A=5 and x=8	Before assigning	After assigning
X= a--	X=8 a=5	X=5 a=4

5. **Conditional operators :**

If a=5 and b=9 and big is any variable

Big= (a>b ? a:b)
Big=(5>9 ? 5:9)
Now the big value is b i.e., 9.

6. **Bit-wise operators: Truth tables**

Exp 1	**Exp 2**	**Bit wise & (and)**	**Bit wise | (or)**	**Bit wise ^ (XOR)**
T(1)	**T(1)**	**T(1)**	**T(1)**	**F(0)**
T(1)	**F(0)**	**F(0)**	**T(1)**	**T(1)**
F(0)	**T(1)**	**F(0)**	**T(1)**	**T(1)**
F(0)	**F(0)**	**F(0)**	**F(0)**	**F(0)**

Note : Substitute T = 1 and F = 0

Bit-wise & (AND operator):

A=000101

B=100100

```
Then a & b is 000101
              100100
              ______
              000100
              ______
```

Bit-wise | (OR operator):

A=000101

B=100100

```
Then a | b is  000101
              100100
              ______
              100101
```

Bit-wise ^ (XOR operator):

A=000101

B=100100

```
Then a ^ b is 000101
              100100
              ______
              100001
              ______
```

Shift operators :

Bit wise Left Shift : (<<)

Consider an integer variable x whose value is 8. the bit pattern for the storage of this value in a single byte will be as fallow

X – 00001000

X <<0 = 00001000

X<< 1 = 00010000 (shift left side add 0 in right side)

X<< 2 = 00100000 (shift left side add 0 in right side)

X<<3 = 01000000 (shift left side add 0 in right side)

Bit wise Right Shift : (>>)

Consider an integer variable x whose value is 8. the bit pattern for the storage of this value in a single byte will be as fallow

X – 00001000

x>>0 = 00001000

x>>1 = 00000100 (Shift right side add 0 in left side)

x>>2 = 00000010 (Shift right side add 0 in left side)

x>>3 = 00000001 (Shift right side add 0 in left side)

x>>4 = 00000000 (Shift right side add 0 in left side)

Note: Shifting bit may be either 0 or 1 but adding bit must be 0 (zero)

Bit wise complement (~) :

It converts 1 to 0 and 0 to 1

Ex :-

A = 000101

~A = 111010

B= 100100

~B= 011011

8.special operators

C supports some other operators also. They are:

- Comma(,) operator :

The comma operator is used to separate two more expressions written in a statement one after another.

In printf statements

Ordinary case: printf (“%d”, age);

The comma operator(,) is used in the above manner.

FORMATTED CONSOLE I/O FUNCTIONS

- Console I/O functions are those functions which are used receiving the input data from the keyboard and writing the output data to the VDU.
- These two functions are used to read (input) and write (output) data respectively, in the format requirement by a programmer. Hence they are known as formatted functions.

- The **scanf function: For entering the input data**
- The input data can be entered into the computer from the keyboard by using the standard library function scanf().
- The scanf function can be used for entering numerical values, characters, strings or any combination of them into the computer.

The general format of the scanf function is:

Scanf(“control string”, list of the addresses of the variables);

‘&’ sign is called ampersand. It is the **address of** operator. That means if it is prefixed to a variable, the combination indicates the address of the variable in the memory.

Rules of Syntax

1. A scanf statement starts with the word **scanf.**
2. A pair of parentheses follows the word **scanf.**
3. A semicolon follows the closing parentheses.
4. Within the parentheses there are two parts. Namely
    - **a.** Control string, and
    - **b.** List of addresses of variables
5. The control string should be enclosed with in double quotation marks.
6. The two parts should be separated by a comma.
7. The control string is a set of **data conversion codes** and such data conversion codes are separated from one another by a blank space for improving the legibility.
8. The data conversion codes in the control string and the list of addresses of variable should match each other in **number, type** and **order.**

Eg:

```
            Int  a,b;
            float  radius;
            char  name;
scanf (“%d%d%f%c”,&a,&b,&radius,&name);
```

The control string

- The control string consists of a set of characters which are known as **data conversion codes.**
- Each set of characters consists of a percentage sign(%) followed by a conversion character letter in lower case. Together, they are known as data conversion codes.

Data conversion codes: They indicate the data type of the corresponding data items.

The commonly used scanf() conversion codes are listed in the following table.

List of scanf conversion codes

Data type	Conversion codes	Purpose
Integer	%hd →	signed integer data
	%hu →	unsigned short integer data
	%d →	signed integer data
	%u →	unsigned integer data
	%ld →	long signed integer data
	%lu →	long unsigned integer data
	%i →	decimal, hexadecimal or octal integer
Real	%f %lf %Lf	floating double long double
Char	%c %c	signed character unsigned character data
String	%s	Reads a string character should be suffixed, so as to accommodate a Null (/o)character which is automatically added at the end.
Others	%o %x %g	octal integer hexa-decimal integer floating point

Prefix	Meaning
h	For short data types
l	For long data types
L	For long data types(Long Double)

The printf function: for writing the output data

- The printf function is also a standard library function like the scanf function and it comes under the header file **stdio.h**.
- This function is meant for writing the output data in the required format specified in the programs.
- <u>Syntax of printf statement is</u>

  Printf(“control string with text” , list of variables);

  Eg: printf(“ the sum is%d”, sum);

Rules of syntax

1. The printf statement starts with the word printf and is followed by a pair of parenthesis containing the arguments.
2. A semicolon is placed immediately after the closing parenthesis.
3. within the parenthesis, there are 2 parts, namely
   a. control string, and
   b. list of variables.

4. The control string should be enclosed within a pair of double quotation marks.
5. The control string and the list of variables should be separated by a comma.
6. In the list of variables, if there are more than one variable, then the difference variable are separated from each other by a comma.
7. Within the control string, the different data conversion codes are separated from each other by blank spaces, tabs or new line sequences, but not commas.

LIBRARY FUNCTIONS

What are C library functions?

C contains scores of functions in its library. Each of these functions perform either a comely used operation or a calculation.

Classification

- Library functions are classified into different groups depending on the nature of the functions they carry out. Each group contains functionally related functions. Which are stored in separate header files.
- Each header file is given a distinct name for the sake of identification.

E.g. Standard input/output functions header file (stdio.h), standard library function header file, character type function etc.

The following are the names of various header files and their corresponding codes.

Type of functions	**Code**
Mathematical functions	#include <math.h>
Standard input/output functions	#include <stdio.h>
Utility functions	#include<stdlib.h>
String handling functions	#include <string.h>
Character type function	#include<ctype.h>
Time functions	#include<time.h>

How to write a library function?

The procedure for using any library function in a program is as follows:

- The corresponding header file should be #included in the program on the top of the program.
- Where ever it is required, the name of the function, followed by a list of arguments that represent the information being passed on to the function are to be written.
- The argument/s should be written within a pair of parentheses. If there is more than one argument then they should be separated by a comma/s.
- The arguments may be constants, the presence of a pair of parentheses is a must.

STRUCTURE OF C PROGRAMMING:

• Comment	/*This is my first C program */
• Preprocessor Directive (Or) Header File	#include <stdio.h>
• Function Declarator	Void main()
• Function Begin	{
• Body of the function	Printf(”My name is Raju”);
• Function End	}

The various components of the program are discussed below:

1) **Comment Line**: The statement which starts with the symbol /* */ is treated as comment. Hence the compiler ignores the complete line starting from /* character pair. A comment may start anywhere in a line and whatever follows the symbol is ignored. A comment line is a non-executable statement.

2) Preprocessor Directive: #include<stdio.h> is the preprocessor directive.

3) Function Declarator: Main() is called function declarator. Every C program must have one function with name main "**from where the execution of the program begins**." The function name is followed by a pair of parentheses which may or may not contain arguments. Every function is supported to return a value. Suppose a function does not written a value such functions must be preceded by the reserved word or keyword void.

4) Function Begin: The function body in C program, is enclosed between two flower brackets. The opening flower bracket ({) marks the beginning of a function. All the statements in a function, which are listed after this brace can either, be executable or non-executable statements.

5**) Function Body**: The function body consists of statements like input, output, computational, conditional and looping statements.

6**) Function End**: The end of a function body in C program is marked by closing bracket (}). The last line actually marks the end of program and control is transferred to the operating system or termination of the program.

DECISION MARKES IN C (or) Branching

For the purpose of making decisions, there are various powerful of tool with district capabilities available in ‘C’. they are the control flow statements. These statements work efficiently in different complex situations.

With control flow statements **they alter the sequential order of execution**. The control flow statements are mainly of three types. They are:

-if statement

-Conditional cooperators

-switch statements

The control flow mechanism specifies the order in which expressions are evaluated and statements are executed in a program.

1.The’ if’ statement

- ‘if’ is one of the keywords in C.
- it is a powerful decision-making tool in C.
- It controls the order of execution of various statements in a C program.

We used various forms of ‘if’ statement.

- Simple ‘if’ statement
- If-else statement
- Nested if—else statement
- Else if ladder.

SIMPLE IF STATEMENT

GENERAL FORMAT	EXAMPLE

IF (Test condition is TRUE)
{
executable statement I;
executable statement 2;
}
	If (The day is holiday)
	{
	get up late in the morning;
	play cricket the whole day;
	}

Rules of syntax of If:

1.The test condition always follow the ‘if’ and it is always enclosed by a pair of parentheses.

2. No semicolon is placed after the closing parentheses.

3. If the statement block contains of more than one statement, the statement block should be written a pair of braces '{}'. Otherwise, only the first statement in the statement block will be executed.
4. If the test condition is "TRUE', the executable statement or the group of statements are executed sequentially.
5. If the test condition if 'FALSE' the CONTROL simply skips the execution of those statements and executes the subsequent statement, if any.

```
/*DEMO:EVALUTION OF THE STATEMENT BLOCK IN THE if STATEMENT*/
#include<stdio.h>
main()
{
int n;
clrscr();
printf("ENTER AN INTEGER NEUMBER\nn=");   /*ENTER -5 AS INPUT*?
Scanf("%d",&n);
If(n>0)
Printf("THE SQUARE OF %d=%d\n",n,n*n);
Printf("THE CUBE OF %d=%d\n",n,n*n*n);
}
```

2.The if …else statement

- This an a simple extension of the if statement.
- That is, in a simple if statement, we define the course of action if the test condition is "TRUE".
- In case the test condition is "FALSE", we have not defined any course of action. In such a case the **control** simply skips the statement block and goes ahead to executive the subsequent statements, if any.
- The if..else statement defines the course of action in both the cases, that means, when the test condition is TRUE as well as FALSE.
- **when the test condition is TRUE   ----- if block statements executed**
- **when the test condition is FALSE ----- else block statements executed**

The general format of an if..else statement is as follows:

General format	**Example**

If(Test condition is TRUE)
{
executable statement 1;
executable statement 2;
……………………….
……………………….
Executable statement irr;
}
else
{
executable statement 1;
executable statement 2;
……………………….
……………………………
executable statement m;
}
	If(Working day for college)
	{
	get up at 6 a,m.;
	get ready by 8 a.m.;
	catvh the bus by 8:30 a.m;
	be present in the classroom by 9 a.m.;
	
	}
	else
	{
	get up at 9 a.m;
	play cricket the whole day;
	go to a film in the evening;
	have dinner outside;
	}

RULES OF SYNTAX

- Executable statements immediately following the if is called the if block.
- Similarly, the set of all executable statements else is called the else block.
- Else is written exactly below if . this is not terminated by a semicolon(;).
- If else block contains only one statement, the placing of braces of is the optional.
- It contains more than one statement, the placing of bases is a must.
- All the executable statements in the if block as well as the else block should be terminated by a semicolon.

Eg:

```
/*demo if ..esle statement with the relational operators */
#include <stdio.h>
main()
{
int a,b;
clrscr();
printf("enter a value for a \na=");
scanf("%d", &a);
printf("Enter a value for b\nb=");
scanf(“%d”, &b);
If (a==b)
Printf(“a is equal to b”);
Else
Printf(“a is not equal to b\n”);
If (a!=b)
Printf(“a is not equal to b\n”);
Else
Printf(“a is equal to b\n”);
If (a <b)
Printf(“a is less than b\n”);
Else
Printf(“a is more than or equal to b\n”);
If (a>b)
Prinf(“a is more than b\n”);
Else
Printf(“a is lesser than or equal to b\n”);
}
```

Example2:

In a company the salary structure is as fallows.

- DA is 50% of the basic pay.
- HRA is 10% of the basic pay.
- Other perks Rs.500 lumpsum.
- Income tax deduction of 5% of the gross salary, if the grass salary is more than Rs .5000.

Write a program to evaluate the next salary of the employees using if else

Note basic pay is inputted through the keyboard by using the scanf ( ) function.

```
/* PGM:PREPARATION OF THE SALARY STATEMENT*/
#include<stdio.h>
main()
{
float basic, da, hra, gross_sal, net_sal;
float inc_tax, perks;
clrscr();
perks=500.00;
printf(“ Enter the basic salary \n basic=”);
scanf(“%f”, & basic );
da=0.5%*basic;
hara=0.1*basic;
grass_sal=basic+da+hra+perks;
if (gross_sal>=5000.00)
inc_tax=0.05*gras_sal;
else
inc_tax=0;
net_sal=gross_sal-inc_tax;
printf(“\t salary statement \n”);
printf(“---------------------\n”);
printf(“basic pay       :rs.%8.2f\n da            :rs.%8.2f\n”, basic,da);
printf(“hra             :rs.%8.2f\n perks         :rs.8.2f\n”,hra,perks);
printf(“income tax      :rs.%8.2f\n” , inc_tax);
printf(“gross salary    :rs.8.2f\n”, gross_salary);
printf(“net salary      :rs.8.2f\n”, net_salary);
printf(“----------------------------------------------------\n”);
}
```

NESTED if.else

- When we have to take a series of decisions we can use the nested if.lese statement.
    - In a nested if.else statement one if.else statement is written in another if.else statement.
    - The inner if.else statement can be written either in the if block or the else block of the outer if.else statement. It may some times be written in both the blocks too.

Example1:

1. Male candidates who have attained the aged of 21 years are eligible and females are not eligiable .
2. Write a program to decide the eligibility criteria.

```
/*DEMO:NESTED if.else STATEMENT*/
/*PGM: ELIGIBILITY CRITERIA*/
# include <stdio.h>
main( )
{
int age;
char sex:
clrscr( );                              /* M FOR MALE AND F FORFEMALE/*
Printf (“Enter the sex of the candidate/n”)
Scanf ( “/n %c”, &sex);
If (Sex= =’M’)
{
Printf (“AGE:”)                                  /* TEST MALE / NOT*/
Scnaf (“%d”, & age);                             /* ENTER AGE*/
If (Age>= 21)
                      /* TEST AGE IS 21 /MORE OR NOT*/
Printf (“THE MALE CANDIDATE IS ELIGIBLE /n”);
 else                                              /* MALE&<21 YRS*/
printf (:THE MALE CANDIDATE ISNOOT ELIBLE /n”)
}
else                                             /* FEMALE*/
printf (“THE FEMALE CANDIDATE ISNOT ELIGIBLE /n”);   }
```

```
/* DEMO:NESTED if…. Else STATEMENT */
/* PGM:ELIGIBILITY CRITERIA */
# include<stdio.h>
main()
{
int age;
char sex, region;                    /* 'L' for LOCAL, 'N' for Non-local */
clrscr();
printf("ENTER SEX");                 /* INPUT */
scanf("%c", &sex);
if (sex == 'M')                      /* TEST:MALE / NOT */
{
printf("ENTER AGE AND REGION:");     /* INPUT */
scanf("%d\n%c",&age,&region);
if(age>=21)                          /* TEST:AGE 21 OR MORE */
{
if(region=='L')                      /* TEST LOCAL/NOT */
printf("ELIGIBLE\N");
else                          /* MALE, AGE 21/MORE BUT NON-LOCAL */
printf("NOT ELIGIBLE\N");
}
else                          /* MALE BUT AGE IS LESS THAN 21 YRS */
printf("NOT ELIGIBLE\N");
}
else                          /* FEMALE */
printf("NOT ELIGBLE\N");
}
```

```
/* DEMO:NESTED if…. Else STATEMENT */
/* PGM:ELIGIBILITY CRITERIA */
# include<stdio.h>
main()
{
int age,avg_marks;
char sex;
clrscr();
printf(“ENTER SEX AGE AND AVERAGE MARKS:”);
scanf(“%c%d%d”, &sex, &avg_marks);
if (sex==’M’)                              /* TEST MALE/NOT */
{
if (age>=21)                      /* FOR MALE TEST : AGE:21 YRS/MORE */
{
if(aavg_marks>=50)                /* TEST:AVG.MARKS=50% / MORE */
printf(“HE IS ELIGIBLE \N”);
else          /* MALE AGED:21/MORE BUT AVG.MARKS LESS THAN 50 */
printf(“HE IS NOT ELIGIBLE \N”);
}
else                              /* FEMALE */
printf(“HE IS NOT ELIGIBLE\N”);
}
else
{
if (sex==’F’)
{
if (age>=18)                      /* TEST:AGE:18 /MORE */
{
if (avg_marks>=35)                /* TEST AVG.MARKS:35% / MORE */
printf(“SHE IS ELIGIBLE\N”);
else                              /* AVG.MARKS LESS THAN 35% */
printf(“SHE IS NOT ELIGIBLE\N”);
}
else
printf(“SHE IS NOT ELIGIBLE\N”);  }   }   }
```

Switch Statement:

- It is also a branching statement.
- C has a built in multiple branch selection statement called ‘Switch’.
- This statement successively tests the value of an expression against a list of constants.
- When a match is found, the statements associated with that condition are executed.

Syntax: The syntax of switch statement is given below:

```
Switch( expression )
{
Case constant 1:
        Executable statements;
        Break ;
Case constant 2:
        Executable statements;
        Break ;
        ----
        ----
    Case constant n:
            Executable statements ;
            Break ;
    Default :
            Executable statements (wrong statements);
            Break ;
    }
```

The Rules of Syntax:

- The Key word Switch is followed by an expression which is enclosed by a pair of parentheses (…) as follows:

Switch (index)

- The parentheses are not terminated by a semicolon.
- The expression within the parentheses may b e an integer value of a character but it should not be a floating point value.

Case Labels:

- Each value of the case labels should be different from each other as shown below:

  Eg: 1,2,3,…….,n (or)

  A,.B,C,…..X

- They should not be given as

  1,1,2,2,3,3…….n (or)

  A,A,B,B,C,C…..X

- The case labels should be followed by a colon(:)

  eg: case1:

  case2:

- If there is more than one executable statement ,then they should be written within a pair of braces as shown below:

```
Case (1):
    {
        Printf (“THE NUMBER IS GREATER THAN 0 \n”);
         Printf (“THE NUMBER IS A POSITIVE INTEGER \n”);


    }
```

- All these statements should be terminated by a semicolon (;)

example program for Switch statement:

```
/* demo:switch statement */
#include<stdio.h>
{
  Int opt_yr,opt_mt;
  clrscr();
  Printf(“ENTER THE OPT YEAR \n”);
  Scanf(“%d”,&opt_yr);
  Printf(“ENTER THE OPT MONTH \n”);
  Scanf("%d",&opt_mt);
  Switch(opt_mt)
                    Case 1: Printf("January");
                    Break;
                    Case 2: Printf("February");
                    Break;
                    Case 3: Printf("March");
                    Break;
                    Case 4: Printf("April");
                    Break;
                    Case 5: Printf("May");
                    Break;
                    Case 6:Printf("June");
                    Break;
                    Case 7: Printf("July");
                    Break;
                    Case 8: Printf("August");
                    Break;
                    Case 9: Printf("September");
                    Break;
                    Case 10: Printf("October");
                    Break;
                    Case 11: Printf("November");
                    Break;
                   Case 12: Printf("December");
                  Break;
}
        Printf("* * :%d \n",opt_yr);
        Printf("~~~~~~~~~~~~~~~~~~~~~~~~~~ \n");
```

Output:

```
ENTER THE OPT YEAR:   1999
ENTER THE OPT MONTH:12
DECEMBER * * 1999
```

Types of Loops:

- Loops are generally classified into two depending on the position of the test condition in the loop.

  They are:

  1. **Entry Control Loops ------ 1.While, 2.For loop**
  2. **Exit Control Loops ---------- 1.do.. While loop**

1.Entry Control Loops:

- While loop and for loop ,the test condition is placed above the body of the loop.
- Thus in these two loops , the test condition is evaluated first. If it is TRUE, the statement in the loop are executed.
- Since the test condition is evaluated at the entry point of the loop itself, these two are known as entry control loops.

2.Exit Control Loops:

- If the test condition is placed below the body of the loop as in a
do while loop, it is known as exit control loop.
- In this case, the statements in the statement block are executed first and then the test condition is evaluated.
- Hence, irrespective of the test condition being TRUE or FALSE, the statements in the body of the loop are executed at least once.
- If the test condition is TRUE, the loop is executed for the the required number of times until the test condition becomes FALSE. Then the Control goes out of the loop.

Types of loops in C

- In C, there are 3 types of Loop Constructs available for performing loop operations.

  They are:

  1. The **while** loop.
  2. The **do-while** loop
  3. The **for** loop

1) **while loop:**

- The while loop “evaluates the condition first”, and execution starts only if the condition is found to be true.
- The statements in the loop are executed if the “text condition is true” and the “execution continues” as long as it remain true.
- If the condition is false it will comes out of loop.
- Here Condition followed by statements
- It is an entry control loop

Syntax: The syntax of while loop is given below:

```
Initialization;
While (expression)
{
Executable statement 1;
Executable statement 2;
      ………
      ………
Executable statement n;
Incrementation;
}
```

We can find that the three parts in while loop.

1. Initialization
2. Expression
3. Incrementation/Decrementation

1.<u>Initialization:</u>

- The process of giving an initial value to a variable is called initialization. If we initialize a certain value, then that initial value will be taken as the first value. Further incrementation is carried out upon the value.

2.Expresssion:

- A test condition is used in a loop in order to control the repetitive action of the loop. The loop is made to repeat itself until the test condition becomes FALSE(zero).
- Test conditions can be designed in two ways.
    1. To repeat the loop for a predefined number of times.
    2. To repeat the loop until the fulfillment of certain conditions.

3.Incrementation/Decrementation:

The incrementation /decrementation statement is present to enable repeated executions of the statements in the body of the loop, until the test condition becomes FALSE.

Examples:

```
/* program to find the sum of the digits of a given number */
#include<stdio.h>
#include<math.h>
main()
{
long int num,sum;
clrscr();
printf("enter the number");
scanf("%ld",&num);
sum=0;
while(num!=0)
{
sum=sum+num%10;
num=num/10;
}
printf("SUM OF THE DIGITS OF THE GIVEN NUMBER IS %ld",sum);
}
```

Example2:

```
/* program to reverse the given number */
#include<stdio.h>
main()
{
long int num,rev;
clrscr();
printf("enter the number");
scanf("%ld",&num);
rev=0;
while(num!=0)
{
rev=rev*10+num%10;
num=num/10;
}
printf("THE REVERSED NUMBER IS %ld",rev);     }
```

2. Do while loop:

- This construct executes the “body of the loop exactly once” first.
- And then evaluates the “test condition is true the execution is repeated”.
- Test condition false it will come out of loop.
- Here Statements followed by condition
- It is an exit control loop

Syntax: The general syntax of Do while loop is given below:

```
Initialization;
do
{
Executable statement 1;
Executable statement 2;
        ………
        ………
Executable statement n;
Incrementation ;
}
While (condition);
Statement ;
```

We can find that the three parts in do while loop.

1.Initialization
2. Expression
3. Incrementation/Decrementation

1.Initialization:

- The process of giving an initial value to a variable is called initialization. If we initialize a certain value, then that initial value will be taken as the first value. Further incrementation is carried out upon the value.

2.Expresssion:

- A test condition is used in a loop in order to control the repetitive action of the loop. The loop is made to repeat itself until the test condition becomes FALSE(zero).
- Test conditions can be designed in two ways.
    1. To repeat the loop for a predefined number of times.
    2. To repeat the loop until the fulfillment of certain conditions.

3.Incrementation/Decrementation:

The incrementation /decrementation statement is present to enable repeated executions of the statements in the body of the loop, until the test condition becomes FALSE.

Example1:

```
/*  display 1 to n numbers */
#include<stdio.h>
main()
{
int i=0,n;
printf("how many integers to be displayed");
scanf("%d",&n);
 do
{
printf("i",\n);
i++;
}
while(i<n);
}
```

<u>Example2:</u>

```
/* To check the given number is palindrome or not using do while */
#include<stdio.h>
main()
{
int n,num,digit,rev=0;
printf("enter the number");
scanf("%d",&num);
n=num;
do
{
digit=num%10;
rev=rev*10+digit;
num=num/10;
}
while(num!=0);
printf("the reverse number is %d",rev \n);
if(n==rev)
printf("the given number is polindrom");
else
printf("the number is not a polindrom");
}
```

<u>3) For Loop:</u>

- **"** The for loop is useful while executing a statement a fixed number of times".

  <u>Syntax</u>:

  The syntax of for loop is given below:

  For (**initialization ;condition ;updation**)
  {
  Executable statements;
  }
  statement;

- The initialization part is executed only once.
- Next text condition is evaluated, If the test "evaluates to false" , then the next statement "after the for loop is executed."
- If the text expression "**evaluates to true**" the after the "**executing the body of the loop**", the **update part is executed**.
- It is an entry control loop.

We can find that the three parts in for loop.

1.Initialization

2. Expression
3. Incrementation/Decrementation

1.Initialization:

- The process of giving an initial value to a variable is called initialization. If we initialize a certain value, then that initial value will be taken as the first value. Further incrementation is carried out upon the value.

2.Expresssion:

- A test condition is used in a loop in order to control the repetitive action of the loop. The loop is made to repeat itself until the test condition becomes FALSE(zero).
- Test conditions can be designed in two ways.
    1. To repeat the loop for a predefined number of times.
    2. To repeat the loop until the fulfillment of certain conditions.

3.Incrementation/Decrementation:

- The incrementation /decrementation statement is present to enable repeated executions of the statements in the body of the loop, until the test condition becomes FALSE.

Example1:

```
 /* program for Armstrong numbers between 100 and 1000 */
#include<stdio.h>
main()
{
int num,a,b,c;
clrscr();
printf("Armstrong numbers between 100 and 1000 are: \t");
for(num=100;num<1000;num++)

{       a=num/100;
        b=(num%100)10;
        c=num%10;
        num=100*a+10*b+c;
        if(a*a*a+b*b*b+c*c*c)==num)
        printf("%d \t",num);
}  }
```

Example2:

```
/* program to display numbers between 500 and 700 divisible by 13 */
#include<stdio.h>
#include<math.h>'
main()
{
int ctr,i;
ctr=0;
clrscr();
printf("The numbers are \n");
for(i=500; i<700; i++)
{
if(i%13==0)
{
printf("%5d",i);
ctr=ctr+1;
}}
printf("\n the numbers which are divisible by 13 are %d",ctr);   }
```

Example3:

```
/* program to find the sum of all odd numbers less than a given number */
#include<stdio.h>
#include<math.h>
main()
{
int num,sum,limit;
clrscr();
sum=0;
printf("Enter the limit \n limit= ");
scanf("%d", &limit);
for(num=1; num<limit; num++)
{
if(num%2!=0)
sum=sum+num;
}
printf("THE SUM OF ODD NUMBERS LESSTHAN %d IS %d \n",limit,sum);
}
```

C Arrays:

- Arrays constitute a very powerful mechanism or tool in C.
- In an ordinary variable, we can store only value at a time.
- In other words, the current value of an ordinary variable is replaced by the latest value assigned to it.
- Hence, we can always store only one value in an ordinary variable at a time.

Do we have any alternate?

Yes. By using an Array.

- An array can hold and manipulate a set of data consisting of items having similar data type.

What is an Array?

- An Array is a **derived data type** and it is a collective name given to a set of similar quantities of any primary data type namely int, float and char. (or)
- An array is defined as collection of similar elements that shares a Common name. Similar means all are int (or) all are float (or) all are Char (or)
- Array is defined as collection of homogeneous elements that shares a common name.

Derived data type:

- **Int, float** and **char** are the primary data types. Those data types which are created by using the primary data types are called derived or secondary data types.
- An array of ints or an array of **floats** are commonly known as an ARRAY and an array of **chars** is called a STRING or a CHARACTER ARRAY.

How to create an array:

- If we have to create an array with the ages of 30 students, it is to be written as **age[30].**
- **Age** refers to the set if ages if students. The total number of quantities in an array should be given within a pair of square brackets[…] and the quantities are called elements. In other words, the 30 mentioned in the brackets refers to the maximum length of the array. It is also known as the **size** of the array.

Declaration of an Array:

- In C, just like any other ordinary variable, the data type of an array must be declared before it is used in a program.
- The general format of an array declaration is as follows:

Syntax: **data type variable_name[size];**

Eg.: Int age[30];

Data type: It refers to the data type of the elements contained in the array.

Eg.: Ages of students, No. of students.

Variable_name: It refers to the collective name of all the elements (of similar nature and data type) contained in the array.

Eg.: Salaries, heights, weights, marks, temperature in degress etc.

Size: It refers to the maximum number of elements that can be stored in an array.

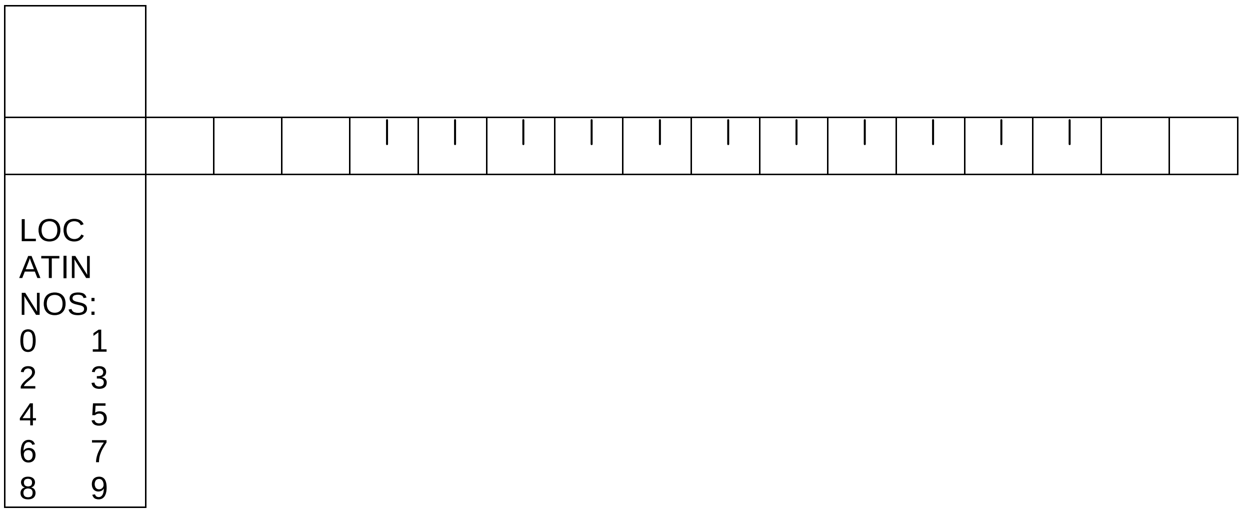
Eg.: float height[50];

- This array declaration indicates that the array height is constructed with 50 elements all having float as their data type.
- Likewise, Int age[30];

What happens if we exceed the limit of 'size'?

- **'size'** represents the **boundary**(maximum number of elements) of an array. It you voluntarily or inadvertently cross the boundary, the compiler gives no error message . However, it returns some unpredictable results. Hence, be cautious. If you declare an array of size 10, then you can fill the array with only 10 or less than 10 values but never more than 10.

MEMORY DIAGRAM



Note: In an array the first location is referred to by ‘0’ and not by ‘1’.

Hence, for the array marks[10] the memory location numbers are 0,1,2,3,----9.

Initialization of Arrays:

- As soon as an array, say, marks[5] is declared, only 5 locations are reserved for the 5 elements in the computer’s memory and the array’s name marks is automatically associated with these locations.
- The elements do not hold any values but they contain only some garbage value initially.
- During the execution of the program, we fill the array locations with the corresponding values.
- We can initialize an array at the time of declaration itself. In this case, the general format of initialization is as follows:

  Data type array_name[size]={list of values};

  Eg.: int marks[5] = {50,55,60,65,70};

  Float price[6] = {10.50, 11.10, 12.00, 13.00, 14.10, 15.00};
- In this manner, if an array is initialized at the time of declaration itself, the following form is also valid.

  Int age[ ] = {10, 15, 20, 25, 30 };
- Giving the size is optional in case of initialization. In such a case, the compiler assumes the number of locations that are required , depending on the number of elements that are initialized in the array.

- We have already learnt that unless and until some values are assigned to the elements of an array they contain garbage values. In order to avoid this, an array can be declared with the **static storage class** as shown below:

  Static int age[5];

- When an array is declared along with the storage class as a Static Variable, during the array, the remaining elements are filled with zeros.
- Now let us see the memory diagram for the different cases.

When an array is declared as: int age[5];

Age

	0	**1**	**2**	**3**	**4**	

Note: Since no storage class is defined, by default the compiler assumes that it is an **auto storage class** and each location ultimately contains some garbage value.

When an array is initialized at the time of declaration as:

int age[5] = {10,20,30,40,50}; age

	10	20	30	40	50	
	0	**1**	**2**	**3**	**4**	

Note: Since values are assigned to the array elements, the location are filled with the Corresponding values.

When an array is declared as **Static int** and initialized with a lesser number of values than the defined size as: **Static int age[5] = {10,20,30};**

Age

	10	20	30	0	0	
	0	**1**	**2**	**3**	**4**	

Actually the array is declared with 5 elements in the static variable declaration, but it is initialized with only 3 values instead of 5. Hence, the last two locations are filled with zero.

Accessing Array Elements:

- By using the printf statement, we can write out the value of any element of an array by referring to the corresponding element number written within square brackets after the array name.
  Eg: The third element is referred to as **age [2]**

- **For reading and writing arrays we use for loop**

For e.g.:

```
# include<stdio.h>
main( )
{
int age[5], c;
printf (“enter the value of ages one by one\n”);
for(c=0;c<5;c++)
{
scanf(“%d”, &age[ c] );
}
for(c=0;c<5;c++)
{
printf(“ the element is %d”, c+1,age[c]);
}
}
```

OUTPUT:

```
Enter the values of ages one by one

23      45      34      26      19

The element 1=23
The element 2=45
The element 3=34
The element 4=26
The element 5=19
```

2-D Arrays:

- In real life, we encounter several situations wherein we deal with a table of values. That means, they have 2 dimensions, say, length and breadth.
- 2-D arrays can be used for storing such tables of values.
- A 2-D array should not be interpreted as an array having 2 lines . But, it is an array with two dimensions (horizontal and vertical) of any size.

Example:

Quality	**Product**		
	A	**B**	**C**
1	200	300	400
2	150	250	300
3	100	200	300

- The above table consists of 9 values. The horizontal side (row) has 3 values and the vertical side (column) has 3 values.
- In mathematics, we call it simply as a 3x3 matrix (reads as 3 by 3 matrix).
- In this table, each row represents the rates of the products A,B,C belonging to a particular quality.

Declaration: The general format of the declaration of a 2-D array is as follows:

Data type array name[row size][column_size];

E.g.: int price_list [3][3];

Rules of Syntax:

- The declaration statement of 2-D arrays is similar to that of single dimension arrays. The only difference is that the array name is followed by 2 pairs of square brackets, instead of one as in a single dimension array.
- In the first set of square brackets, the size (No. of elements) of the row has to be written and in the second set of square brackets the size (No. of elements) of the column has to be written.
- The statement is terminated by a semicolon.

Initialization:

- The general format of the initialization of a 2-D array is as shown below:

```
Datatype array_name [row_size] [column_size] =
{
{I row value 1, value 2, ………value n},
{ II row value 1, value 2, ………value n},
…………………………………
…………………………………
{ nth  row value 1, value 2, ………value n}
};
```

Rules of Syntax:

- All the initializable values should be enclosed within a pair of braces and after the closing brace a semicolon should be written.
- Within the outermost braces for each row of the 2-D array, one pair of braces should be written and within each such pair of braces, the values of the corresponding elements should be written.
- A comma separates each such value. Each row's closing brace is followed by a comma except the last row.
- The table of values is represented by index numbers as shown below:

Column No→	**0**	**1**	**2**
Row No – 0	200	300	400
Row No – 1	150	250	350
Row No -2	100	200	300

The above values in the price list are referred to as shown below:

Row No.	Col. No.	Code	Reference Value
0	0	Price_list [0] [0]	200
0	1	Price_list [0] [1]	300
0	2	Price_list [0] [2]	400
1	0	Price_list [1] [0]	150
1	1	Price_list [1] [1]	250
1	2	Price_list [1] [2]	350
2	0	Price_list [2] [0]	100
2	1	Price_list [2] [1]	200
2	2	Price_list [2] [2]	300

How to access the elements of a 2-D array?

We can access the elements of a 2-D array, either by index numbers or by pointers. The same methodology used in single dimensional arrays is also applicable here.

Example Program:

```
/* To access the elements of 2-D array by using subscripts */
main()
{
int x,y;
int price_list[3][3] = { {200,400,600}, {250,450,650}, {300,500,700}};
clrscr();
for(x=0; x<3, x++)
{
printf("\n");
for(y=0; y<3; y++)
{
```

```
printf("%d \t",price_list[x][y]);
 }
  }
 }
```

OUTPUT:

```
                    200     400     600
                    250     450     650
                    300     500     700
```

Example of 2-D Arrays:

/* C program to add two matrices */

```
#include<stdio.h>
main()
{
int a[10][10], b[10][10], c[10][10], I, j, m, n, p, q;
printf("input row and column of A matrix  \n");
scanf("%d %d", &n, &m);
printf(" %d %d \n", n, m);
printf("input row and column of B matrix  \n");
scanf("%d %d", &p, &q);
printf(" %d %d \n", p, q);
/* checks if matrices can be added */
If((n == p) && (m ==q))
{
printf("matrices can be added \n");
printf("input A – matrix \n");
for(i=0; i<n; ++i)
for(j=0; j<m; ++j)
scanf("%d", & a[I][j]);
/* print A – matrix in matrix form */
for(i=o; i<n; ++i)
{
          for(j=0; j<m; ++j)
                    printf("%5d", a[I][j]);
```

```
        printf("\n");
}
printf("input B – matrix \n");
for(i=0; i<n; ++i)
        for(j=0; j<m; ++j)
scanf("%d", &b[I][j]);
/* print B – Matrix in matrix form */
        for(i=o; i<n; ++i)
        {
        for(j=0; j<m; ++j)
                printf("%5d", b[I][j]);
        printf("\n");
}
/* addition of two matrices */
        for(i=o; i<n; ++i)
                for(j=0; j<m; ++j)
                        C[iI][j] = a[i][j] + b[i][j];
                printf("sum of A & B matrices: \n");
for(I=0; I<n; I++)
{
        for(j=0; j<m; ++j)
                printf("%5d",C[I][j]);
        printf("\n");
}
}
else
printf("matrices cannot be added \n");
} /*main */
```

OUTPUT:

```
Input row and column of A matrix
        3       3
input row and column of B matrix
        1       2
matrices cannot be added
```

```
input row and column of A matrix
          3         3
input row and column of B matrix
          3         3
matrices can be added
Input A – matrix
1         2         3
4         5         6
7         8         9
input B – matrix
1         2         3
4         5         6
7         8         9
Sum of A and B matrices:
2         4         6
8         10        12
14        16        18
```

Example:2

/* C program to subtract two matrices */

```
#include<stdio.h>
main()
{
int a[10][10], b[10][10], c[10][10], I, j, m, n, p, q;
printf(“input row and column of A matrix  \n”);
scanf(“%d %d”, &n, &m);
printf(“ %d %d \n”, n, m);
printf(“input row and column of B matrix  \n”);
scanf(“%d %d”, &p, &q);
printf(“ %d %d \n”, p, q);
/* checks if matrices can be subtracted*/
If((n == p) && (m ==q))
{
printf(“matrices can be subtracted \n”);
printf(“input A – matrix \n”);
for(i=0; i<n; ++i)
for(j=0; j<m; ++j)
scanf(“%d”, & a[I][j]);
/* print A – matrix in matrix form */
for(i=o; i<n; ++i)
{
        for(j=0; j<m; ++j)
                printf(“%5d”, a[I][j]);
        printf(“\n”);
}
printf(“input B – matrix \n”);
for(i=0; i<n; ++i)
        for(j=0; j<m; ++j)
scanf(“%d”, &b[I][j]);

/* print B – Matrix in matrix form */
        for(i=o; i<n; ++i)
{
        for(j=0; j<m; ++j)
                printf(“%5d”, b[I][j]);
        printf(“\n”);
}
/* subtraction of two matrices */
        for(i=o; i<n; ++i)
                for(j=0; j<m; ++j)
                        C[iI][j] = a[i][j] - b[i][j];
                printf(“difference of A & B matrices: \n”);
for(I=0; I<n; I++)
{
        for(j=0; j<m; ++j)
                printf(“%5d”,C[I][j]);
        printf(“\n”);   }    }
else
```

```
printf(“matrices cannot be subtracted\n”);
} /*main */
```

Example:3

/* C program to multiplication of two matrices */

```
#include<stdio.h>
#include<conio.h>
main()
{
int a[3][3], b[3][3], c[3][3], I, j, k, m, n, p, q;
clrscr();
printf(“\n Enter the order of matrix a:”);
scanf(“%d%D”, &m,&n);
printf(“\nEnter the order of matrix b:”);
scanf(“%d%d”,&p,&q);
if (n!=0)
{
printf(“\n Matrix multiplication is not possible”);
for (I=0;I<m;I++)
{
for (j=0;j<n;j++)
scanf(“%d”,&a[I][j]);
}
printf(“\nEnter elements jof b:”);
for (I=0;I<p;I++)
{
for (j=0;j<q;j++)
scanf(“%d”,&b[I][j]);
}
clrscr();
printf(“\n order of matrix a:%d X%d”,m,n);
printf(“\n Elements of a :\n”);
for(I=0;I<m;I++)
{
```

```
for(j=0;j<n;j++)
printf("%d\t",a[I][j]);
printf("\n");
}
printf("\n order of matrix b:%d X%d",p,q);
printf("\n Elements of b :\n");
for(I=0;I<p;I++)
{
for(j=0;j<q;j++)
printf("%d\t",b[I][j]);
printf("\n");
}
printf("\n order of matrix c:%dX%d",m,q);
printf("\n Elements of c:\n");
for(I=0;I<m;I++)
{
for(j=0;j<q;j++)
{
c[I][j]=o;
for(k=0;k<p;k++)
c[I][j]=a[I][k]*b[k][j]+c[I][j];
}
}
for(I=0;I<m;I++)
{
for(j=0;j<q;j++)
printf("%d\t",c[I][j]);
printf("\n");
}
printf("\n");
}
getch();
}
```

C Strings:

- ❖ If an integer values are stored in an array, it is called an ‘Integer array’ , whereas if floating point values are stored in an array, it is called a ‘Floating point array’.
- ❖ A string is an array of characters which is terminated by a Null(‘\0’) character.

Character Array: Char title [ ] = {‘I’, ’N’, ‘S’, ‘I’, ‘D’, ‘E’, ‘R’};
String: Char title[ ] = {‘I’, ’N’, ‘S’, ‘I’, ‘D’, ‘E’, ‘R’};

- ❖ Each character with in a string is stored as an element of an array and each character occupies a length of 1 byte or memory.
- ❖ In order to access and avail of one or more of such string-oriented functions in a program, one should #include the header file containing the string library functions to the program as shown below:

  #Include <string.h>

The salient features of a string are as follows:

- ❖ A string is a one-dimensional array of characters.
- ❖ It is always terminated by a null (‘\0’) character because it indicates the end of the string to those functions, which work on it.
- ❖ A string may contain any set of characters written between a pair of double quotes.

  E.g.: “ A string is always enclosed by a pair of double quotes”.

DECLARATION:

A string is also a valid variable in C. Hence, it should declared like any other variable before it is used in a program.

- ❖ The declaration of a string is similar to that of an array.
- ❖ The general format of the declaration of a string is as shown below.

  Char String_name(size);

  Char title [10];
- ❖ The difference we find in the declaration of a string from that of an array is that an array ‘s data type may be either integer or float, whereas for a string it is always of character type.

<u>Size:</u>

The size indicator is the maximum number of characters that can be stored in a string.

E.g.: char title [10];

The elements of a string are stored in contiguous memory locations in the memory like those of an array.

<u>Initialization:</u>

- ❖ A string can be initialized just like an array at the time of declaration itself.
- ❖ A string may be initialized in either of the tow following ways. They are:

  Char title [12] = “CALLCLEAR”;

  Char title [12] = {‘C’ ‘A’ ‘L’ ‘L’ ‘C’ ‘L’ ‘E’ ‘A’ ‘‘R’, ‘\0’};
- ❖ When we initialize a string as in the second method, by listing each element separately, we must explicitly supply the Null (‘\0’) character as the last character of the string.

<u>String Representation in Memory</u>:

A string is stored in memory by using ASCII codes of the characters that from the string.

The representation of the string HELLO in memory is shown in the following figure.

H
E
L
L
O
/O

“Character string representing in memory” and end with NULL characters.

Reading a String:

The process of reading a string is almost similar to that of an array.

- ❖ The scanf() function is used to read characters or numericals.
- ❖ In arrays, **%d** is the data conversion code, which is used for integer values, and **%f** is used for floating point values. Whereas in strings **%s**is sued as the data conversion code.
- ❖ For reading arrays, in the second part of the scanf statement, that is, in the list of addresses of variables, each variable is preceded by an Ampersand (&) sign, as shown below:
  &marks, &qty, &roll-no etc.
- ❖ Whereas, **for reading strings**, the general format of the **scanf** statement is the same as that arrays but **without the ampersand** (&) sign.
  E.g.: If the declaration statement of a string is
  Char name [10];

It can be read by using the scanf statement as follows:
scanf("%s", name); is a valid scanf statement which is used to read a string.

- ❖ Note that **no '&' sign** is present and also now [ ] is suffixed to the string title in **both the scanf() and printf() functions**.

Writing of a string:

Like an array, a string can also be written by using the printf function, the data conversion code us **%s**.   The general format of the printf statement for writing a string is as follows:

Printf("%s", string_name);
E.g.: Printf("%s', title);

This printf statement is executed as a part of the program and the title is displayed to this screen.

String Handling Standard Library Functions

- ❖ In order to avail of one or more of such functions in a program, the corresponding header file should be #include to the program, at the top of main() as follows:

  #Include <string.h>

- ❖ The following table lists out the most commonly used string-handling functions with brief explanation of the purpose of each function as well as the datatype of its return values.

FUNCTION	PURPOSE
❖ Strlen (s)	❖ Returns the number of character in a string.
❖ Strlwr (s)	❖ Converts the string from the upper to lower case
❖ Strupr (s)	❖ Converts the string from the lower to upper case
❖ Strcmp (s1,s2)	❖ Compares string s1 with s2.
❖ Strnicmpi (s1,s2)	❖ Compares two strings s1 and s2 ignoring their case
❖ Strcmp (s1, s2)	❖ Compares the first n characters of 2 strings by ignoring their case
❖ Strev (s)	❖ Reverses a string
❖ Strdup (s)	❖ Duplicates a string
❖ Strcat (s1, s2)	❖ Appends the string s1 at the end of string s2.
❖ Strncat (s1,s2)	❖ Appends the first n characters of the string s1 at
❖ Strncpy (s1,s2)	❖ Copies the string s2 to string s1.
❖ Strset (s,c)	❖ Sets all characters with in the string s to the character c.

Strlen( ) function:

- This is used to find the **length of a string** actually stored in an array.
- The null character is not included in the calculation of the string.

  General form:

  Strlen(s);
- This returns an integer value for the number of characters present in the strings.

Strcat( ) function:

- This function **joins two strings together**.

  General form:

  Strcat(s1,s2);
- This appends the strings s2 to s1, overwriting the existing null character in the string s1 and providing a new null character after appending s2.
- The Programmer must ensure that the size of s1 sufficiently large to hold both s1 and s2. The string at s2 remains unchanged.

  E.g.: s1= "Good"

  S2="Evening"

  Strcat(s1,s2)

  The result is

  S1= "Good Evening"

It is possible to have nesting of **strcat** functions.

The statement:

```
Strcat(strcat(n1,n2,n3));
```

Joins all the strings n1, n2 and n3 together and result is stored in n1.

Strcmp( ) function:

This function is used to **compare two strings**.

General form:

Strcmp(s1,s2);

This function compares two strings s1 and s2 character by character.

It returns integer value.

It returns

- == 0 – if two strings are identical
- < 0 -- if the first string is less than second string
- >0 -- if the first string is greater than second string

E.g.:

St1 = “raghu”

St2 = “snaju”

Strcmp(st1,st2);

The result is st1 and st2 are not same.

Strcpy( ) function:

General form is:

Strcpy(s1,s2);

- Here the character in the string s2 are **copied to the string** s1, over writing any existing data in s1. The length of s1 should be greater than or equal to s2.

  E.g.: strcpy(name, “sanju”);

Will assign the string “sanju” to the variable name.

- Similarly, the statement strcpy(name1,name2); assigns contents of string variable name2 to name1.

Example1:

```
/* Example for strlen() function */
#include <stdio.h>
#Include <string.h>
main()
{
int n1,n2,n3;
Char goal[ ] = “PERFECTION”;
Char goal[ ] = “SELF CONFIDENCE”;
clrscr();
n1 = strlen(goal);
n2 = strlen(need);
n3 = strlen(“SELF CONTROL”); /* Direct Insertion */
printf(“YOUR GOAL IS %s (LENGTH =%d”) \n”, goal,n1);
printf("YOUR NEED IS %s (LENGTH =%d") \n", need,n2);
printf("ALSO %s (LENGTH =%d") \n", "SELF CONTROL",n3);
}
```

Output:

```
YOUR GOAL IS PERFECTION (LENGTH =10)
YOUR NEED IS SELF CONFIDENCE ( LENGTH = 15)
ALSO SELF CONTROL (LENGTH =12)
```

Example2:

```
/* Program for strcat( ) function */
#include<stdio.h>
#include<string.h>
main()
{
char dont[20] = "SELF";
char have[11] = "CONFIDENCE";
clrscr();
strcat(don't,have);
printf("dont lose; %s \n", dont);
printf("do have; %s ", have);
}
```

Output:

```
Dont lose: SELF CONFIDENCE
Do have : CONFIDENCE
```

Example3:

```
/* program for strcmp( ) function */
#include<string.h>
main()
{
char s1[ ] = "RAVINDRA";
char s2[ ] = “RABINDRA”;
char s3[ ] = “RAVINDRA”;
clrscr();
strcmp(s1,s2) == 0? Printf(“IDENTICAL \N”):printf(“%d \n “, strcmp(s1,s2));
strcmp(s1,s3) == 0? Printf(“IDENTICAL \N”):printf(“%d \n “, strcmp(s1,s3));
}
```

Output:

```
20 [Ascii values v=89 and b = 66 : Difference =20 ]
IDENTICAL [STRING S1.S3 ARE IDENTICAL]
```

Example4:

```
/* program for strcmp( ) function */
#include<string.h>
#include<stdio.h>
main()
{
char s1[ ] = “RAJA”;
char s2[ ] = “RAAJ”;
char s3[ ] = “RAAJA”;
int a,b,c;
clrscr ();
a = strcmp(s1, “RAJA”);
b = strcmp(s1,s2);
c = strcmp(s1,s3);
printf(“%d \t %d \t %d \n”, a,b,c);
}
```

Output:

```
0       9       9
```

Example5:

```
/* program for strcpy( ) function */
#include<stdio.h>
#include<string.h>
main()]
{
char strsong[ ] = “MELODUY”;
char strear[10] ;
char strlove[ ] = “ILU”;
char strheart[10];
clrscr();
strcpy(strear,strong);
printf(“strsong:%s \n “, strong);
printf(“strear:%s \n “, strear);
printf(“strheart:%s \n “, strlove);
printf(“strheart%s \n “, strheart);
printf(“strlove:%s \n “, strlove);
}
```

Output:

```
strong:MELODY
strear:MELODY
strheart:ILU
strlove:ILU
```

C Structures and Unions

- A Structure is **derived date type** (Called as **‘struct’** type) variable which is capable of **holding dissimilar data types** namely integer elements, floating point elements as well as character elements and their derived data types at a time.
- The declaration of a structure is more complicated than that of an array because it should be declared in terms of each element.
- The declaration of structure consists of 2 parts. They are:
  1. Declaration of elements 2. Declaration of variables.

Declaration of Members

General Format	Example

Struct label
{
data type member 1;
data type member 2;
………………………..
……………………….
Data type member n;
};
	Struct book
	{
	int pages;
	char name;
	float price;
	};

Struct: This is a keyword representing the data type **Structure,** and every declaration should start with the keyword **‘Struct’.**

Rules of Syntax:

- Each declaration of the members should be terminated by a semicolon just like any other variable type decalration statements.

- Each member should be given a distinct name.
  E.g.: Roll_no, name[10], hours, tot_wage etc.
- The type declaration of the members should be made within a pair of braces.
- The closing brace is followed by a semicolon.

Note:
The members of a structure are capable of holding not only ordinary variables (int, float and char) but also derived data types namely, arrays, strings, pointers and other structures too.

Declaration of Variables:
Once the structure members are declared as shown above, one or more struct type variables should be declared. The general format of the same is as follows:

Struct label_name	**Variable 1,**	**variable 2,**	**variable 3;**
Sturct report	worker 1,	worker 2,	worker 3;
Struct bill	item 1,	item 2,	item 3;
Struct statement	unit 1,	unit 2	unit 3;

Rules of Syntax:
- This variable declaration statement contains:
- **Struct** which is a keyword.
- Lable name e.g.:report (which is the same we have given in the declaration of members)
- If there is more than one variable the variables are separated by commas.

Note: There is no limit on the number of variable names. Hence, we can declare as many variables as required by the program.

E.g.: Suppose we want to prepare wage_report for 20 workers, then 20 Variables are declared as follows:

Strcut report worker 1, worker 2, ……………………., worker 20.

Initialization:

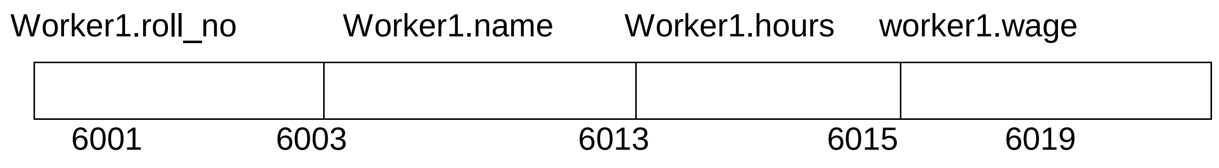
- Structure variables can also be initialized like any other primary variable (int, float, char) and derived variables (arrays, strings, etc.,).

Example program:

```
/ * Initialization of a structure */
main( )
{
    struct report
    {
    int roll_no;
    char[10] name;
    int hours;
    float tot_wage;
};
    struct report   worker1  =  {1, “sudhir”, 50, 1500.00};
                     worker2 =  {2, “sunil”, 60, 1800.00};
                     worker3 =  {1, “suresh”,70, 2100.00};
     ………………………………………………………………
     ………………………………………………………………
    ………………………………………………………………
 }
```

Memory Location:

- The elements of a structure are stored in contiguous memory locations of a computer.



If you analyze the above memory allocation, you can observe the following facts.

Worker1:

Variable	Type	Byte No.s	No. of Bytes	Remarks
Roll_no Name Hours Tot_wages	int string char[10] int float	6001 – 6003 6003 – 6013 6013 – 6015 6015 – 6019	2 10 2 4	For each variable, the same number of bytes are allocated in contiguous memory locations respectively, one by one

- Hence, the data composition of each worker occupies 18 bytes in memory.
- In the same fashion, worker2 and worker3 are also allocated 18 bytes each to store their data in contiguous memory locations one by one.

<u>Accessing the elements of a structure:</u>

- The linking between a variable and an element is made by a **dot(.)** operator, which is called the **member operator** or the **period operator.**

Example:

/ * Structure declaration, initialization, writing, memory * /

```
#include <stdio.h>
main( )
{
struct report
{
int roll_no;
char name[10];
int no_hours;
float tot_wages;
};
struct report worker1  =  {1, “sudhir”, 50, 1500.00};
struct report worker2 =  {2, “sunil”, 60, 1800.00};
clrscr( );
printf(“\t WAGE REPORT    \n”);
printf(“~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ \n”);
printf(“%d \t %s %d \t %d \t \n”, worker1.roll_no, worker1.name, worker1.no_hours,
worker1.tot_wages);
printf(“%d \t %s %d \t %d \t \n”, worker1.roll_no, worker1.name, worker1.no_hours,
worker1.tot_wages);
printf(“~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ \n”);
printf(“%u \t %u \t %u \t %u \n”, worker1.roll_no, worker1.name, &worker1.no_hours,
&worker1.tot_wages);
printf(“%u \t %u \t %u \t %u \n”, worker1.roll_no, worker1.name, &worker1.no_hours,
&worker1.tot_wages);
printf(“~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ \n”);
}
```

Giving values to the elements of a structure:

- With the help of the scanf ( ) function, we can give values to the members of a structure through the keyboard. As usual, in the scanf statement, the address of variable. member should be given as shown below:

  E.g.: scanf(“%d”, &worker1.roll_n);

Example:

```
/ * Reading the values from the keyboard and writing */

#include <stdio.h>
struct report
{
int roll_no;
char name[10];
int no_hours;
float tot_wages;
};
main( )
```

```
{
struct report worker1, worker2;
 clrscr( );
printf("Enter values of all elements one by one  \n");
scanf("%d  %s %d  %f", &worker1.roll_no, &worker1.name, &worker1.no_hours,
&worker1.tot_wages);
scanf("%d  %s %d  %f", &worker2.roll_no, &worker2.name, &worker2.no_hours,
&worker2.tot_wages);
printf("\n %d \t %s \t %d \n", worker1.roll_no, worker1.name, &worker1.no_hours,
&worker1.tot_wages);
printf("\n %d \t %s \t %d \n", worker1.roll_no, worker1.name, &worker1.no_hours,
&worker1.tot_wages);
}
```

Arrays within structure:

Suppose we could write 100 variables like, struct report worker1, worker2, ……….worker100;

Besides, for giving values to the variables, we could write 100 scanf statements.

For writing the values of the variables, we could write 100 printf statements.

but this is a very laborious, cumbersome as well as error-prone exercise.

Is there is any alternative?

Yes . we can construct an array of structures .

Example:

```
/* construct an array of structures with 5 workers*/
#include <stdio.h>
struct report
{
int roll_no;
char name[10];
int no_hours;
float tot_wages;
}; struct report  Worker[5]; /* array declared in structure*/
Int i;
 clrscr( );
```

```
printf("Enter roll no. , name, no of hours hours worked, total wages");
for (i=0;i<5;i++)
{
scanf("%d %s%d %f", &worker[i].roll_no, &worker[i].name, &worker[i].no_hours,
&worker[i].tot_wages);
}
Printf(" ROLL NO, NAME , NO. OF HOURS, TOTAL WAGES);
Printf("¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬");
for (i=0;i<5;i++)
{
printf("%d %s%d %f", worker[i].roll_no, worker[i].name, worker[i].no_hours,
worker[i].tot_wages);
}
}
```

UNIONS

- The basic reason behind development on union data type is **to save memory in the program environment , when a large number of variables are used.**
- Union is a Special data type in C through which objects of different types and sizes can be stored at different times.
- With the help of unions, we can refer to the **same memory location** through different types of variables.
- For example a memory location of 4 bytes occupied by a floating point values can be referred to by a character array of length 4.
- The general format of declaration of a Union as follows.

```
union tag                                Eg:  union price_list
{                                             {
Data type member1;                                            char name[10];
Data type member2;                                            int qty;
--------------------                                    char grade;
--------------------
--------------------
--------------------                                    float price;
Data type member n;
} list of variables;                          } x,y,z;
```

- The tag can be used to declare many other variables of the same union as in a structure.
- In the above example, in x or y, z variables, we can store an integer, a float, a character or a string of size 10 elements.
- Union members are referred in the same way as structures.
- A union can have any numbers of members.
- A union type variable takes as much memory as its largest data type member occupies.
- For example, in the above example the union has 4 members, name,(string 10 bytes), qty (integer-2 bytes_, grade(char-1 byte) and price (float-4 bytes).
- Among the 4 members, the first name (string 10 bytes) is **the largest memory occupying members.** Hence the union occupies 10 bytes**.**

Main differences between Structures and Unions

Sturucture	Union
1. It is a derived data type	1. it is also derived data type
2. we use the keyword struct	2. we use the key word union
3. general format is	3. general format is
Struct label	union lable
{	{
data type member 1;	
data type member 2;	data type member 1;
	data type member 2;
………………………..	………………………..
……………………….	………………………..
Data type member n;	data type member n;
}; struct label variable1, variable2….variable n;	} list of variables ;
4. occupies much memory	4. occupies less memory

File Management

- Store data permanently for further use.
- It is also treated as a data type and it is created in the secondary memory or disks (hard disk or floppy disk), which can store data it in permanently.
-

- A file is a place on a disk, which contains a collection of related records, usually containing data pertaining to a specific area of application. In a file, all the records are identical to each other, in structure.
- We have deal with small volumes of data, and we stored and manipulated them with various primary and secondary data types and we have used standard I/O functions. Mainly scanf() and printf(), for inputting and outputting data respectively, in such data types.
- But in real life we deal with large volumes of data and the data should be stored permanently for further use.
- But the data types we have used so far are incapable of sorting data permanently.
- The standard I/O function, Scanf() and Printf (), etc., are incapable of dealing with large volumes of data. It is very cumbersome and error prone exercise to handle large volumes of data through terminals.
- Because of the aforesaid reasons, there is a serious need for seeking a unique data type with its own batch of I/O specialties.
- Such a data type should have the following features:
  1. It should be able to work on secondary storage device like disks.
  2. It should be able to store data permanently.
  3. It should be able to deal with large volumes of data
  4. we should be able to access, read, write and modify the data from it as and when it is required
  5. there should be a set of I/O specialties in order to handle the data in such data type.
- The data type which possesses all the aforesaid features is called as file.