TASKS OF KODE-BLOOM

- VI EDITOR -

OPENING EDITOR

vi - editor will be opened without any filename

vi filename - editor will be opened with given filename

MODES OF VI EDITOR

command mode - pressing ESC

insert mode   -   pressing i / I

MOVING CURSOR WITHIN FILE

l - moving cursor towards left

h- moving cursor towards right

k-moving cursor up

j-moving cursor down

INSERTING TEXT

i-insert

a-insertion begin after cursor

A-insertion begins at end of line

o- inserts new line after cursor

O-inserts new line above cursor

COPY & PASTE

yy - copies the current line

p - paste the copied line

SAVE & QUIT

go to command mode by pressing ESC

:w - to save the content

:w filename - to save the content with filename

:q - quit the file

:q! - quit the file without saving the content

:wq - save the content and quit the file

- BASIC COMMANDS IN LINUX -

1. mkdir :

Create new Directory with given name.

```
root@LAPTOP-M3171AO3:~# mkdir Practice
```

2. ls :

List the content of Directory.

```
root@LAPTOP-M3171AO3:~# ls
Practice  addNum.c  date.txt  hello.c      hello.cpp.save  library  sharedlib test    test.txt
a.out     copy      hello     hello.c.save  laxmi          mylib.a  static.a   test.c
```

Blue colour indicates the directory.

3. cd :

Used to change to directory.

```
root@LAPTOP-M3171AO3:~# cd Practice
root@LAPTOP-M3171AO3:~/Practice# ls
file.txt   file1.txt  sample sample.c
root@LAPTOP-M3171AO3:~# cd
```

// Coming out of directory

4.touch :

It is used to create a empty file

```
root@LAPTOP-M3171AO3 :~/Practice # touch file1.txt
root@LAPTOP-M3171AO3:~/Practice # ls
file1.txt
```

5.cat > filename :

Create a new file and writes the content and ctrl+D to save the content into the file.

```
root@LAPTOP-M3171AO3 :~/Practice # cat>file2.txt
Hello……
root@LAPTOP-M3171AO3 :~/Practice # ls
file1.txt file2.txt
```

Note : Until U press control d. It asks u data, if u click ctrl d it will store the data in file2.txt

6. cat :

Display content of file.

```
root@LAPTOP-M3171AO3 :~/Practice # cat file2.txt
Hello……
root@LAPTOP-M3171AO3 :~/Practice # cat>file3.txt
Welcome to Linux / Unix Commands.
root@LAPTOP-M3171AO3 :~/Practice #ls
file1.txt file2.txt file3.txt
root@LAPTOP-M3171AO3 :~/Practice # cat file2.txt
Hello……
root@LAPTOP-M3171AO3 :~/Practice # cat file3.txt
Welcome to Linux / Unix Commands.
```

7. cat file2 file3 > file4 :

Creates new file and content of both files will be **copied to new file**.

```
root@LAPTOP-M3171AO3 :~/Practice # cat file2.txt file3.txt > filemerge.txt
root@LAPTOP-M3171AO3 :~/Practice # ls
```

```
file1.txt   file2.txt   file3.txt    filemerge.txt

root@LAPTOP-M3171AO3 :~/Practice #cat filemerge.txt

Hello……

Welcome to Linux / Unix Commands.
```

8. ls *.txt :

List all the files with given extensions

```
root@LAPTOP-M3171AO3 :~/Practice # ls *.txt

file1.txt   file2.txt   file3.txt    filemerge.txt
```

// Our directory having only txt files. Similarly u can try other extentions like ls *.c files, ls *.py, ls *.cpp and other.

9 . pwd :

Current printing working directory

```
root@LAPTOP-M3171AO3:~# pwd

/root

root@LAPTOP-M3171AO3:~# cd Practice

root@LAPTOP-M3171AO3 :~/Practice # pwd

/root/Practice
```

10. cp :

Copy a file or directory

```
root@LAPTOP-M3171AO3 :~/Practice # cp file3.txt file4.txt

root@LAPTOP-M3171AO3 :~/Practice #ls

file1.txt   file2.txt   file3.txt    filemerge.txt   file4.txt

root@LAPTOP-M3171AO3 :~/Practice # cat file3.txt

Welcome to Linux / Unix Commands.

root@LAPTOP-M3171AO3 :~/Practice # cat file4.txt

Welcome to Linux / Unix Commands.

root@LAPTOP-M3171AO3 :~/Practice # cd
```

11. mv :

Moves a file or directory

```
root@LAPTOP-M3171AO3 :~/ # ls
Practice  addNum.c  date.txt   hello.c      hello.cpp.save  library  sharedlib  test    test.txt
a.out     copy      hello      hello.c.save  laxmi          mylib.a  static.a   test.c
root@LAPTOP-M3171AO3:~# mv hello.c  /root/Practice
root@LAPTOP-M3171AO3 :~/ # cd Practice
root@LAPTOP-M3171AO3 :~/ Practice # ls
file1.txt   file2.txt   file3.txt    filemerge.txt  hello.c
```

12. head :

Displays first 10 Lines in given file.

```
root@LAPTOP-M3171AO3 :~/ Practice # cat file1.txt
root@LAPTOP-M3171AO3 :~/ Practice # vi file1.txt
LINE - 1
LINE - 2
LINE - 3
LINE - 4
LINE - 5
LINE - 6
LINE - 7
LINE - 8
LINE - 9
LINE - 10
LINE - 11
LINE - 12
LINE - 13
LINE - 14
LINE - 15
LINE - 16
LINE – 17
LINE - 18
LINE - 19
LINE – 20
root@LAPTOP-M3171AO3 :~/ Practice # head file1.txt
LINE - 1
LINE - 2
LINE - 3
LINE - 4
LINE - 5
LINE - 6
LINE - 7
LINE - 8
LINE - 9
LINE – 10
```

13 . tail :

Display last 10 Lines of a file.

```
root@LAPTOP-M3171AO3 :~/ Practice # tail file1.txt
LINE - 11
LINE - 12
LINE - 13
LINE - 14
LINE - 15
LINE - 16
LINE – 17
LINE - 18
LINE - 19
LINE – 20
```

14 . tac :

Display the content Lines in reverse.

```
root@LAPTOP-M3171AO3 :~/ Practice # cat filemerge.txt

Hello……

Welcome to Linux / Unix Commands.

root@LAPTOP-M3171AO3 :~/ Practice # tac filemerge.txt

Welcome to Linux / Unix Commands.

Hello…….
```

15 . more :

Similar to cat and here we can display large content by using, ENTER, SPACEBAR.

And we can display content screen by screen.

```
root@LAPTOP-M3171AO3 :~/ Practice # more file1.txt

LINE - 1

LINE - 2

LINE - 3

LINE - 4

LINE - 5

LINE - 6

LINE - 7

LINE - 8

LINE - 9

LINE - 10

LINE - 11

LINE - 12

LINE - 13

LINE - 14

LINE – 15

- - more - -(82%)
```

// If we press enter remaining move to next line. It is similar to scroll bar.

16 . id :

Display id of user/group.

```
root@LAPTOP-M3171AO3 :~/ Practice # id
uid=0(root) gid=0(root) groups=0(root)
```

17 . clear :

It is used to clear the screen.

```
root@LAPTOP-M3171AO3:~/Practice# clear
```

18. vi :

Text editor to write programs of text.

19 . grep :

It is a filter to search given pattern in the file content.

```
root@LAPTOP-M3171AO3:~/Practice# cat file3.txt
Welcome to Linux / Unix Commands.
root@LAPTOP-M3171AO3:~/Practice# grep Linux file3.txt
Welcome to Linux / Unix Commands.
root@LAPTOP-M3171AO3:~/Practice# grep o file3.txt
Welcome to Linux / Unix Commands.
```

20 . diff :

Compares the content of two different files.

```
root@LAPTOP-M3171AO3:~/Practice# cat file2.txt
Hello…….
root@LAPTOP-M3171AO3:~/Practice# cat file3.txt
Welcome to Linux / Unix Commands.
root@LAPTOP-M3171AO3:~/Practice# diff file2.txt file3.txt
1c1
< Hello…….
- - - - - -
```

```
> Welcome to Linux / Unix Commands.
```

// So there is no difference in the file.

```
root@LAPTOP-M3171AO3:~/Practice# cat filemerge.txt
Hello…….
Welcome to Linux / Unix Commands.
root@LAPTOP-M3171AO3:~/Practice# diff file2.txt filemerge.txt
1a2
> Welcome to Linux / Unix Commands.
```

// Here we are having common thing that is Hello……. So the first line is automatically matched. So common element is removed remaining diff is dipalyed.

```
root@LAPTOP-M3171AO3:~/Practice# cat file2.txt
Hello……
```

21 . ping :

It Checks the connectivity status of sever

```
root@LAPTOP-M3171AO3:~/Practice# ping google.com
PING google.com (142.250.196.14) 56(84) bytes of data.
64 bytes from maa03s44-in-f14.1e100.net (142.250.196.14): icmp_seq=1 ttl=57 time=15.1 ms
64 bytes from maa03s44-in-f14.1e100.net (142.250.196.14): icmp_seq=2 ttl=57 time=15.0 ms
64 bytes from maa03s44-in-f14.1e100.net (142.250.196.14): icmp_seq=3 ttl=57 time=15.5 ms
^Z
[1]+  Stopped                 ping google.com
```

// **ctrl Z is used to stop.**

If Disconnect the net

```
root@LAPTOP-M3171AO3:~/Practice# ping google.com
ping: google.com: Temporary failure in name resolution
root@LAPTOP-M3171AO3:~/Practice#
```

22 . history :

Review all the commands which you have entered.

23 . hostname :

Display the hostname.

```
root@LAPTOP-M3171AO3:~/Practice# hostname
LAPTOP-M3171AO3
```

24 . hostname -i :

Display host ip.

```
root@LAPTOP-M3171AO3:~/Practice# hostname -i
127.0.1.1
```

25 . chmed :

Change the user/group permissions to access file.

```
root@LAPTOP-M3171AO3:~/Practice# cat file2.txt
Hello……
root@LAPTOP-M3171AO3:~/Practice# vi file2.txt    // adding Updated file change
root@LAPTOP-M3171AO3:~/Practice# cat file2.txt
Hello…….
Updated file change
root@LAPTOP-M3171AO3:~/Practice# chmod u=r file2.txt
root@LAPTOP-M3171AO3:~/Practice# vi file2.txt
```

//There is some warning when U click Insert as Changing a readonly file and we unable to write in read mode.

```
root@LAPTOP-M3171AO3:~/Practice# chmod u=w file2.txt
root@LAPTOP-M3171AO3:~/Practice# vi file2.txt
```

// every thing was erased

26 . nl :

Used to Display the Line Numbers.

```
root@LAPTOP-M3171AO3:~/Practice# cat file4.txt
Welcome to Linux / Unix Commands.
root@LAPTOP-M3171AO3:~/Practice# nl file4.txt
```

1. Welcome to Linux / Unix Commands.

27 . wc :

Given numbers of Lines, Words and Character available in the file content.

```
root@LAPTOP-M3171AO3:~/Practice# wc file3.txt
  1  6  33 file3.txt
root@LAPTOP-M3171AO3:~/Practice# wc file1.txt
30 30 231 file1.txt
```

28 . uniq :

Used to Remove duplicates of file content. There is an exception it can remove only continuous duplicates.

```
root@LAPTOP-M3171AO3:~/Practice# vi file3.txt
root@LAPTOP-M3171AO3:~/Practice# cat file3.txt
Welcome to Linux / Unix Commands.
Welcome to Linux / Unix Commands.
Thank You
And
Thank You
root@LAPTOP-M3171AO3:~/Practice# uniq file3.txt
Welcome to Linux / Unix Commands.
Thank You
And
Thank You
```

29 . rmdir :

Removes the specified directory(Make sure that the directory should be empty.)

30 . rm -r Or rm -f :

It is used to remove the directory with contents/file.

31. rm :

Remove the file.

- - - - - - - - - - - -- - - - - - PREPROCESSOR DIRECTIVES - - - - - - - - - - - - - - -

Macros is a small piece of Code (which is implemented by preprocessor directive #define) → preprocessor directives

Preprocessor is not a part of the compiler but is a separate step in compilation process.

In simple terms a C preprocessor is just a text substitution and it instructs the compiler to do required pre-processing before the actual compilation.

All preprocessor commands begin with a hash symbol(#)

1.MACROS

2.FILE INCLUSION

3.CONDITIONAL COMPILATION

4.OTHER DIRECTIVES

- - - - - - - - - - - - Macros - - - - - - - - - - - - - - -

The "#define" is used to define a macro. Macro are piece of code in a program that is given same name. Wherever this name is encountered by the compiler the compiler replaces the name with the actual value.

Syntax :

#defind macro-name value

1.object like macros :

\\simple implementation like const def

#define N 10

Example :

```
/* OBJECT LIKE MACROS */

#include<stdio.h>

#define A 100

int main()

{

printf("value of A is %d",A);

return 0;
}
```

O/p : value of A is 100

2.chain macros :

```
#define constant pi
#define pi 3.14
```

```
[ constant
        |
        Pi
        |
    3.14 ]
```

Example :

```
/* CHAIN MACROS*/

#include<stdio.h>
#define constant pi
# define pi 3.14
int main()
{
        printf("Value of constant is %f",constant);
        return 0;
}
```

O/p : Value of constant is 3.140000

3.Function like macros :

```
#define sum(a,b)
a=1,b=2
sum(a,b)
1+2=>3
```

// we pass parameters they are implemented & values not store in memory

Example :

```
/* FUNCTION LIKE MACROS*/
#include<stdio.h>
#define sum(a,b) a+b
int main()
{
    int a,b;
    printf("Enter values of a and b");
    scanf("%d %d",&a,&b);
    printf("sum of %d and %d is %d",a,b,sum(a,b));
    return 0;
}
```

O/P :

```
Enter values of a and b
10
20
sum of 10 and 20 is 30
```

- - - - - - - - - - FILE INCLUSION DIRECTIVE - - - - - - - -

When you want to include any files use file inclusion

Definations of pre-defined functions like printf(), scanf() etc…These files must be include to work with this functions. Different functions are declared in different header files.

For example standard I/O functions are in the "stdio.h" file where as functions that perform string oprations are in the "String.h"file.

Syntax :

```
#include<filename>
```

Or

#include “file-name”

Example :

```
#include<stdio.h>
void main()
{
Printf(“Hello…………”);
}
```

- - - - - - - - - - - - - CONDITIONAL COMPILATION - - - - - - - - - - - - -

It’s same as conditional statements

TYPES :

There are six types of conditional statements

1. #ifdef

   The #ifdef preprocessor directive checks if macro is defined by #define If yes only then it executes the code

   Example:

   ```
   /* CONDITIONAL BASED ON if def */

   #include<stdio.h>
   #define a 10
   void main()
   {
   #ifdef a
       printf("Hi......");
   #endif
   }
   O/P:Hi…….
   ```

2. #ifndef

Example :

```
/* CONDITIONAL BASED ON ifndef */
```

```
#include<stdio.h>
#define a 10
void main()
{
#ifndef a
       printf("Hiiiiii");
#endif
}
O/P: Don’t get any output.
```

3. #if
4. #else
5. #elif
6. #endif

Example :

For #if , #else , elif and #endif

```
/* CONDITIONAL BASED ON if, elif  , else and endif  */

#include<stdio.h>
#define a 15
void main()
{
#if a<10
       printf("value less than 10");
#elif a<20
       printf("value < 20");
#else
       printf("No not found");
#endif
}
O/p: value < 20
```

- - - - - - - - - - - - Other Directives(undef) (pragma) - - - - - - - -

#undef :

It’s is used to remove the macro and output as error.

Example :

```
/* undef */
#include<stdio.h>
#define a 10
#undef a
int main()
{
    printf("%d",a);
}
```

O/p :Error message

#pragma directive :

It's used for spl purpose dir and used to turn on or off come features. This type of directives are compiler-specific

They are:

#pragma startup

#pragma exit

#pragma warn

- - - - - - - - - - Structures - - - - - - - - - - - - -

Structures :

Collection of element with diff data types .Here we have single name under this we can save multiple data elements vth diff datatypes. It is a user defined data type that can be used to grp elements of diff types into a single type.

Array →we called as elements

Structure → we called as members.

Steps :

1. Define the structure

2. Declare structure variable
3. Init the members of structure
4. Accessing the members of structure

We use keyword as struct

```
Struct TagName
//TagName ntg but var
{
Datatype variablename;
Datatype variable name;
.
.
.
.
.
};
Or
}Structure var1,structure var2,………;
```

Example:

```
#include <stdio.h>
struct car {
	char *engine;
	char *fuel_type;
	int fuel_tank_cap;
	int seating_cap;
	float city_mileage;
} car1, car2;

/*
  struct {
	  char *engine;
	} car1, car2;
*/

int main()
{
	car1.engine = "DDis 190 Engine";
```

```
        car2.engine = "1.2 L Kappa Dyal VTVT";
        printf("%s\n", car1.engine);
        printf("%s", car2.engine);
        return 0;
}
O/P :
DDis 190 Engine
1.2 L Kappa Dyal VTVTroot
```

- - - - - - - - - - - -Structure Types – Using Structure Tag - - - - - - - - - - - - -

Structure in Global Scope:

If the structure is in the global scope. Hence it is **visible** to all the functions. It can access by user. If we want declare Variable in global scope there is no problem.

Structure in Local Scope:

If the structure is in the local scope. Hence it is **hidden** to all the functions. But cann’t access by user. If we want declare Variable in local scope there is we need redeclare the struct once again.

Instated of writing this once again we use Structure tag(create a type of structure)

- - - - - - - - - - - - -Structure Tag - - - - - - - - - - - -

It is used to identify a particular kind of structure or type of Structure.

Example :

```
/* Structure using structure tag */

#include<stdio.h>

struct employee

{

char *name;

int age;

int salary;

};

int manager()
```

```
{
struct employee manager;
manager.age = 27;
if(manager.age > 30)
manager.salary = 65000;
else
manager.salary = 55000;
return manager.salary;
}
int main()
{
struct employee emp1;
struct employee emp2;
printf("Enter the salary of employee1: ");
scanf("%d", & emp1.salary);
printf("Enter the salary of employee2: ");
scanf("%d", &emp2.salary);
printf("Employee 1 salary is: %d\n", emp1.salary);
printf("Employee 2 salary is: %d\n", emp2.salary);
printf("Manager's salary is %d\n",manager());
return 0;
}
```

O/P :

```
 Enter the salary of employee1: 35000
Enter the salary of employee2: 45000
Employee 1 salary is: 35000
Employee 2 salary is: 45000
Manager's salary is 55000
```

- - - - - - - - - - - - - Using typedef - - - - - - - - - - - - - -

typedef gives freedom to the user by allowing them to create their own types.

Syntax:

typedef existing_data_type new_data_type

It may use Structure Declaration or Separate Declaration and also use global or local scope.

Example :

```
/* Using TPYEDEF KEYWORD */

#include <stdio.h>

typedef int INTEGER;

int main()

{

    INTEGER var = 100;

    printf("%d", var);

    return 0;

}
```

O/P : 100

- - - - - INITIALIZING STRUCTURE VARIABLES AND ACCESSING MEMBERS OF STRUCTURE - - - -

Initialization Structure Variables:
The process of assigning initial values to the members (variables) of a user-defined data type called a "structure" when you first declare a variable of that structure type, essentially setting up the data that will be stored within the structure at the time of creation.

Example :
struct abc

```
{
int p;
int q;
};
int main()
{
struct abc x = {23,34};
}
```

Accessing Members of Structure:
We can access members of the structure using dot(.) operator.

Example :

```
#include <stdio.h>
struct car {
   int fuel_tank_cap;
} c1,c2;
int main()
{
   c1.fuel_tank_cap = 45;
   c2.fuel_tank_cap = 30;
   printf("%d %d", c1.fuel_tank_cap,c2.fuel_tank_cap);
   return 0;
}
O/P: 45 30
```

- - - - - - - - - - - - - - - - DESIGNATION INITIALIZAITON - - - - - - - - - - - - - - - - - -

It allows structure members to be initialized in any order there is no particular order.

Example :

```
struct abc {

int x;

int y;

int z;

};

int main()
```

```
{
struct abc a = {.y=20, .x = 10, .z = 30};
printf("%d\n %d\n %d", a.x, a.y, a.z);
return 0;
}
O/P :
    10
    20
    30
```

Note : Don't forget to use dot operator while accessing the members of the structure.

- - - - - - - - - - - - - - - - - ARRAY OF STRUCTURE - - - - - - - - - - - - - - - - - -

Instead of declaring multiple variables, we can also declare an array of structure in which each element of the array will represent a structure variable.

Example :

```
#include <stdio.h>
struct car
{
int fuel_tank_cap;
int seating_cap;
float city_mileage;
};
int main()
{
struct car c[2];
for(int i=0;i<2;i++)
{
printf("Enter the car %d fuel tank capacity: ",i+1);
scanf("%d", &c[i].fuel_tank_cap);
printf("Enter the car %d seating capacity:", i+1);
scanf("%d", &c[i].seating_cap);
printf("Enter the car %d city mileage:", i+1);
scanf("%f", &c[i].city_mileage);
}
printf("\n");
for(int i=0;i<2;i++)
{
printf("\nCar %d details: \n", i+1);
printf("fuel tank capacity: %d\n", c[i].fuel_tank_cap);
printf("seating capacity: %d\n", c[i].seating_cap);
printf("city mileage: %f\n", c[i].city_mileage);
}
return 0;
}
```

O/P :

```
Enter the car 1 fuel tank capacity: 180
Enter the car 1 seating capacity:258
Enter the car 1 city mileage:3568.45
Enter the car 2 fuel tank capacity: 145
Enter the car 2 seating capacity:256
Enter the car 2 city mileage:456.258


Car 1 details:
fuel tank capacity: 180
seating capacity: 258
city mileage: 3568.449951

Car 2 details:
fuel tank capacity: 145
seating capacity: 256
city mileage: 456.257996
```

- - - - - - - - - - ACCESSING MEMBERS OF STRUCTURE USING STRUCTURE POINT - - - - - - - - - - -

To declare a structure pointer struct keyword is used followed by the structure name and pointer name with an asterisk * symbol. Members of a structure can be accessed from pointers using two ways that are. Using dot and asterisk operator on a pointer. Using arrow operator (->) on a pointer.

Example :

```
 #include <stdio.h>

struct abc {

int x;
int y;
};
int main()
{
struct abc a = {0,1};
struct abc *ptr = &a;
printf("%d\n%d",  a.x , ptr ->  y);
return 0;
}
```

O/P :

```
0
1
```

Note : ptr -> x is equivalent to (*ptr).x

- - - - - - - - - - - - - - STRUCTURE PADDING - - - - - - - - - - - - - -

When an object of some structure type is declared then some contiguous block of memory will be allocated to structure members.

Examples :

```
struct abc {
char a;
char b;
int c;
} var;
//Here var is the object, and  It will allocate memory
```

CALCULATEING THE SIZE OF THE STRUCT :

STRUCTURE PADDING:

Processor doesn’t read 1byte at a time from memory.

It reads 1 word at a time.

If we have a 32 bit processor then it means it can access 4 bytes at a time which means word size is 4bytes

If we have a 64 bit processor then it means it can access 8 bytes at a time which means word size is 8 bytes.

Its an unnecessary wastage of CPU cycles. That’s we can save the number of cycle by using the concept called PADDING.

Example :

```
struct abc {

char a; // 1 byte

char b; //1 byte

int c; // 4 byte

};

int main()

{

printf(" %ld ", sizeof(var));

}
```

O/P :8

[a][b] c[][]|[][]. [][][]

Padding :

[a][b][empty] | c[][][][].

//Total = 1 byte-(a) + 1 byte-(b) + 2 bytes-(empty) + 4 Bytes(c)

Because of structure padding, size of the structure becomes more than the size of the actual structure. Due to this some memory will get wasted.

Example :

```
/* STRUCTURE PADDING  */

#include <stdio.h>

struct abc {
```

```
char a;

int b;

char c;

}var;

int main()

{

printf(" %ld ", sizeof(var));

}

O/P : 12
```

- - - - - - - - - - - - - - - - - STRUCTURE PACKING - - - - - - - - - - - - - - - -

We can avoid the wastage of memory by simply writing #pragma pack(1)

Example :

```
/*  STRUCTURE PACKING OF PRAGMA PACK */

#include <stdio.h>

#pragma pack(1)

struct abc {

char a;

int b;

char c;

} var;

int main() {

printf("%ld", sizeof(var));

return 0;

}

O/P : 6
```

Note : #pragma is a special purpose directive used to trun on or off certain features.

- - - - - - - - - - - - - - STORAGE CLASSES - - - - - - - - - - - - - -

Storage classes :

Storage classes define the **lifetime, scope, and visibility of variables and default values**. They specify where a variable is stored, how long its value is retained, and how it can be accessed which help us to trace the existence of a particular variable during the runtime of a program.

It consists some properties like Default values , location ,scope and lifetime.

Scope having of three types like   a. Block   b. Function/Method   c. Program

There are four primary storage classes :

- auto
- register
- static
- extern

auto :

This is the default storage class for all the variables declared inside a function or a block. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope).

Note : Scope will be block or function But is **Does not have programming scope And Global declaration not allowed.**

Example :

```
#include <stdio.h>

int main() {                          // Function block

                                     // auto is optional here, as it's the default storage class

   auto int x = 10;
{                                   // Block is created
   auto x;                         // Block started and life-time will be within the block.
   printf(“%d”, x);               //If dont init value default val will be print that is grabage val.
}
Printf(“%d”, x);
   return 0;
}
```

O/P :

```
 32767
 10
```

Default val is garbage val

Scope will be block or function But is **Does not have programming scope**. **Global declaration not allowed.**

Life-time will be within block and

Location or visiable store in stack or ram.

static :

This storage class is used to declare static variables that have the property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope.

Note ++: Life-time will be throughout (till the end of prgm) the block , Scope is within the block both declaration allowed global and local , Default value or garbage will be (0) Zero. And the Memory is Ram not CPU register.

Properties of static Storage Class :

- Scope: Local
- Default Value: Zero
- Memory Location: RAM
- Lifetime: Till the end of the program

Example :

```
/* Static storage class */
#include<stdio.h>
void display();
void main()
{
display();
display();
}
void display()
{
static int x;
x += 10;
printf("\nx = %d", x);
}
```

O/P :

```
x = 10
y = 20
```

Example :

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

```
void display()
{
static int x;
 x+=10;
int y =10;
y--;
printf("\n x = %d",x);
printf("\n y = %d",y);
}
```

O/P :

```
x = 10
y = 9
x = 20
y = 9
```

Example :

```
#include<stdio.h>
void display();
void main()
{
	int i;
	for(i =0;i<3;i++)
		display();
}
void display()
{
	static int x = 5; //static storage
	int y = 5; //auto storage
```

```
        x++;

        y++;

        printf("\n x = %d",x);

        printf("\n y = %d",y);

}
```

O/P :

```
x = 6

y = 6

x = 7

y = 6

x = 8

y = 6
```

Example :

```
#include <stdio.h>

void counter() {

   // Static variable retains value between calls

   static int count = 0;

   count++;

   printf("Count = %d\n", count);

}

int main() {

   // Prints: Count = 1

   counter();

   // Prints: Count = 2

   counter();

   return 0;

}
```

O/P :

```
Count = 1

Count = 2
```

register :

This storage class declares register variables that have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the **register of the microprocessor** if a free register is available making it much faster than any of the other variables.

Note : Bcz register having less memory it will be in KB better use variables which we use frequently **like loops, counters we can't use pointers**&, use * is to storage address of another variable(Process will be fast in register but switching will be slow in Ram.

So, its better to **storage in register** to save time runtime will be less and frequency more.)

Note : Scope will be block or function But is **Does not have programming scope And Global declaration not allowed.**

Properties of register Storage Class Objects :

- Scope: Local
- Default Value: Garbage Value
- Memory Location: Register in CPU or RAM
- Lifetime: Till the end of its scope

Note: The compiler may ignore the suggestion based on available registers.

Example 1 :

```
#include <stdio.h>

int main() {

  // Suggest to store in a register

  register int i;

  for (i = 0; i < 5; i++) {
        printf("%d ", i);
      }
      return 0;
    }
    O/P :  0  1  2  3  4
```

```
Example 2:
#include <stdio.h>
int main() {
register   int i,sum =0;
for(i=0;i<10;i++)
sum = sum + i;
printf("%d", sum);
return 0;
}
```

```
O/P : 45
```

extern :

Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well.

Also, a normal global variable can be made extern as well by placing the 'extern' keyword before its declaration/definition in any function/block.

Note : It is mainly focus on scope like GLOBAL No Local variable come under this, Default value is Zero , life-time is throughout or between 2 files and Store in Ram Memory.

Extern keyword can be used with variables and functions. it also called as reuse and linking two files. The extern used to declare not define.

Properties of extern Storage Class Objects

- Scope: Global
- Default Value: Zero
- Memory Location: RAM
- Lifetime: Till the end of the program.

Example :

```
/* Extern Storage class */
#include<stdio.h>
void fun1();
void fun2();
//int a = 10; // Global declaration
int main()
{
	extern int a; //declaratin
	printf("%d\n",a);
	fun1();
	fun2();
}
void fun1()
{
	int a = 2;
	a++;
	printf("a = %d\n",a);
}
void fun2()
{
	printf("Hello from fun2\n");
}
```

```
int a = 10;
```

O/P :

```
10
a = 3
Hello from fun2
```

Example for accessing from diff files using extern keyword :

```
/* support file of extern */
#include<stdio.h>
void display()
{
	extern int x;
	x++;
	printf("Hello from support file...\n");
	printf("x=%d", x);
}

/* Another file is extern1 file */
#include<stdio.h>
#include "support.c"
extern void display();
int x = 10;
void main()
{
	display();
}
```

O/P :

```
Hello from support file...
x=11
```

- - - - - - - - - - - - - - - LIBRARIES - - - - - - - - - - - - - - -

Libraries are a collection of objects that are made available for use by other programs. They are pre-complied functions to avoid repetition. They are not executable but used at runtime or compile time.

Library is just a file.

LIBIO.SO → LIB

PRINTO → FUNCTION

There are two types of libraries

- **Static Libraries**
- **Shared Libraries**

❖ Static Libraries :

They end with **.a a stands for archive**
Used for function that are used frequently or for re-distribution from you or a third party.
static have ***.A,*.LIB**

- Anything linked from the static library is compiled directly into the final object. Any changes needs recompilation to the linked objects

Creation
Use the **ar** command and the name should start with lib

COMPILE TIME : CODE → COMPILED → APP
RUN TIME : APP RUNS → PRINTO → "HELLO WORLD"
IN STATIC LIB we take function and code that we need and we move that to the application just like copy and paste.

Example :
//sum.c
int add(int a, int b)

```
{
	return a+b;
}
```

- -

```
//sub.c
int sub(int a, int b)
{
	return a-b;
}
```

- -

```
// cal.h

int add(int a, int b);

int sub(int a, int b);
```

- -

```
//main.c
#include<stdio.h> //go search in the extended path
#include"cal.h" //It not avaliable in extended path it avalible in local path

int main()
{
	printf("The addition of a and b is %d and sub of a and b is %d\n", add(11,10),
sub(11,10));
	return 0;
}
```

- -

```
root@LAPTOP-M3171AO3:~/library# gcc main.c sum.c sub.c -o main

root@LAPTOP-M3171AO3:~/library# ./main

The addition of a and b is 21 and sub of a and b is 1

 root@LAPTOP-M3171AO3:~/library# ls

 cal.h  main  main.c  sub.c  sum.c
```

- -

```
root@LAPTOP-M3171AO3:~/library# gcc -Wall -c sum.c sub.c
```

//Wall – convert all waring to error (is used for linker and provide .o files)

In-order to make library use to compile without using main

```
root@LAPTOP-M3171AO3:~/library# ls

cal.h  main  main.c  sub.c  sub.o  sum.c  sum.o
```

- -

```
ar -cvq mylib.a *.o    //Used to merge all files into single file.
a - sub.o
a - sum.o
root@LAPTOP-M3171AO3:~/library# ls
cal.h main.c  mylib.a  sub.c  sub.o  sum.c  sum.o
```

- -

```
gcc main.c -o main mylib.a

root@LAPTOP-M3171AO3:~/library# ls

sum.c  sum.o  call.h  main  main.c  mylib.a  sub.c  sub.o

root@LAPTOP-M3171AO3:~/library# ./main

O/P :
The addition of a and b is 21 and sub of a and b is 1
```

- -

```
The sub c and d is -5root@LAPTOP-M3171AO3:~/clibrary# rm add.c sub.c
root@LAPTOP-M3171AO3:~/library# ls
add.o  call.h  main  main.c  mylib.a  sub.o
root@LAPTOP-M3171AO3:~/library# rm sub.o add.o
root@LAPTOP-M3171AO3:~/library# ls
call.h  main  main.c  mylib.a
root@LAPTOP-M3171AO3:~/library# gcc main.c  -o main1  mylib.a
root@LAPTOP-M3171AO3:~/library# ls
call.h  main  main.c  main1  mylib.a
root@LAPTOP-M3171AO3:~/library# ./main1
O/P :
The addition of a and b is 21 and sub of a and b is 1
```

- -

```
root@LAPTOP-M3171AO3:~/library# mv mylib.a ../
root@LAPTOP-M3171AO3:~/library# ls
call.h  main  main.c  main1
root@LAPTOP-M3171AO3:~/library# ./main1
```

O/P :
The addition of a and b is 21 and sub of a and b is 1

- -

- **Shared Libraries :**
  Shared libraries (also called dynamic libraries) are linked into the program in two stages. First, during compile time, the linker verifies that all the symbols (again, functions, variables and the like) required by the program, are either linked into the program, or in one of its shared libraries.
  Shared have ***.SO, *.DLL, *.DYLIB**

- Libraries that can be linked in by dynamic linker when process is loaded into memory (.so) (libc.so)
- Changes do not require binaries to be recompiled
- Requires .so to reside on target platform
- Should be compiled as position-independent code (-fpic or fPIC)
- Global Offset Table
    - Table of dynamically-linked functions
    - And their run-time addresses
- fpic vs fPIC
- fpic – There is a limit on the size of the global offset table
    - limitations are macine-specific
- fPIC – There is no size limit on the global offset table
- For predictable results, use the same option at all stages of compilations.

printf	address
sqrt	address
_libc_start_main	address
srand	address
time	address

Shared Libraries Naming Conventions :

It having three types of names they are :

1. Real Name

- Libname.so.x.y.z

- name = library name
- x = major version
- y = minor version
- z = patch/increment
- libjpeg.so.8.2.2
  - libc : libc-x.yy.so => libc-2.31.

2. Shared Object Name (soname)
   - Lable for a major version of shared object
   - soname : libjpeg.so.8 => libjpeg.so.8.2.2
3. Linker Name
   - libname.so   -Iname

Shared Libraries :

- gcc -fpic -c source1.c source2.c
  -Creates source1.o and source2.o
- gcc -fpic source1.o source2.o -shared -WI, -soname,libsource.so.1
  -o libsource.so.1.1
- fpic, fPIC , and not specifying either one may all result in the same output.
- It's best practice to include either -fpic or -fPIC when compiling for shared libraries.

COMPILE TIME : CODE → COMPILED → APP
RUN TIME : APP RUNS → PRINTO → "HELLO WORLD"
Here the program references the library when it runs , so for your program to work you have to have the library and the application at the same time or nothing's going to work

Note : Shared library that keeps the application very small by taking reference exactly which need, and without using copy and past.

Example :
```
// sum.c
int sum(int a, int b)
{
      return a+b;
}
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
//sub.c
int sub(int a, int b)
{
        return a-b;
}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
//shared.h
int sum(int a, int b);
int sub(int a, int b);
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
// main.c
#include<stdio.h>
#include"shared.h"
int main(void)
{
        printf("Sum of two No's is %d\nDiff of two No's is %d",sum(11,16),sub(11,16));
        return 0;
}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
root@LAPTOP-M3171AO3:~/sharedlib# gcc -c -Wall -Werror -fpic sub.c sum.c
root@LAPTOP-M3171AO3:~/sharedlib# ls
main.c  shared.h  sub.c  sub.o  sum.c  sum.o
// Object files created
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
root@LAPTOP-M3171AO3:~/sharedlib# gcc -shared -o libshared.so sub.o sum.o
root@LAPTOP-M3171AO3:~/sharedlib# ls
libshared.so  main.c  shared.h  sub.c  sub.o  sum.c  sum.o
//shared library created as libshared.so
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
root@LAPTOP-M3171AO3:~/sharedlib# gcc -L/root/sharedlib -Wall -o myout main.c -
lshared
root@LAPTOP-M3171AO3:~/sharedlib# ls
libshared.so  main.c  myout  shared.h  sub.c  sub.o  sum.c  sum.o
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
root@LAPTOP-M3171AO3:~/sharedlib#gcc          -L/root/sharedlib          -Wl+,-
rpath=/root/sharedlib -Wall -o myout2 main.c -lshared
root@LAPTOP-M3171AO3:~/sharedlib# ls
libshared.so main.c  myout  myout2  shared.h  sub.c  sub.o  sum.c  sum.o
root@LAPTOP-M3171AO3:~/sharedlib# ./myout2
Sum of two No's is 27
Diff of two No's is -5
```

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
root@LAPTOP-M3171AO3:~/sharedlib# size myout2
   text    data     bss     dec     hex filename
   1668     648       8    2324     914 myout2
```

- - - - - - - - - - - - - - FUNCTION DECLARATION, CALLS AND DEFINITION - - - - - - - - - - - - - -

Function is a self contained block that carries out a specific well defined task. It enclosed with { }. It’s also known as block of code that can be reused n number of times.

Function declaration, calls and definition :

Function declaration" tells the compiler about a function's name, return type, and parameters before it's used, essentially providing a blueprint for how to call it, while a "function call" is the act of invoking that function within the program to execute its code, passing any necessary arguments to it; essentially, you "declare" a function to define its structure, and then "call" it to use its functionality. The body of the function (code to be executed) is function definition.

Function Declaration :

This informs compiler about the fun name, fun parameters and return values of data types.

Syntax :

Return type fun-name(argument list)

Function Calls :

This calls the actual function.

Syntax :

Fun-name(argument list);

Function Definition :

It contains all the statements to be executed.

Syntax :

```
return type fun-name(argument-list)
{
body of function
}
```

Types of Functions :

There are two types of Functions they are Standard library and Userdefined function.

Standard library function :

This are built-in functions.(Meaning of these functions are already present in header files.)

Example : stdio.h (printf and scanf present in stdio.h)

Userdefined Function :

The function which is created by user for their need.

Example :

```
Sum()
{
}
```

Or

```
fun()
{
}
```

Example :

```
/* Example prgm for Function */
#include<stdio.h>
void fun()
{
printf("Example for Function\n");
printf("Practice for Function\n");
}
void main()
{
fun();
fun();
}
```

O/P :

```
Example for FUN
Practice for FUN
Example for FUN
Practice for FUN
```

Example :

```
/* Addition using function */
#include<stdio.h>
void add()
{
int a,b,c;
printf("Enter Your Values of a and b : \n");
scanf("%d %d",&a , &b);
c = a + b;
printf("Value of c : %d",c);
}
void main()
{
add();
}
```

O/P :

```
Enter Your Values of a and b :

70

70

Value of c : 140
```

Function parameters or arguments :

```
#include<stdio.h>

void fun(int a, char b[ ])

{

printf(“%d  %s\n”, a, b);

}

void main()

{

fun (11, “KodeBloom”);

}
```

O/P :

```
11 KodeBloom
```

Types of Userdefined functions :

They are four types :

1. Function with no arguments and no return values.
2. Function with no arguments and with return values.
3. Function with arguments and no return values.
4. Function with arguments and with return values.

Function with no arguments and no return values :

```
/* Function Definition */

#include<stdio.h>
```

```
void myfun() //fun definition

{

printf("Function with no arguments and no return values");

}

void main()

{

myfun(); //calling fun

}
```

O/P : Function with no arguments and no return values

|

It also written as

```
/* Function Declaration */

#include<stdio.h>

void myfun();  //fun declaration

{

myfun();  //calling fun

}

void myfun()  //fun definition

{

printf(" Function with no arguments and no return values");

}
```

O/P : Function with no arguments and no return values

Function with no arguments and with return values :

```
#include<stdio.h>

int myfun()

{

printf("Function with no argument and with return values");
return 0;
}
void main()
{
myfun();
}
```

O/P : Function with no argument and with return values

Function with arguments and no return values :

```
include<stdio.h>
void myfun(int a, char b[ ])
{
print(“%d %s\n”, a, b);
}
void main()
{
myfun(11,”KODEBLOOM”);
}
```

O/P :

```
11 KODEBLOOM
```

Function with arguments and with return values :

```
#include<stdio.h>
int myfun(int a, char b[])
{
printf(“%d %s\n”, a, b);
return 0;
}
void main()
{
myfun(111 , "KODEBLOOMMM");
}
```

O/P :

```
111 KODEBLOOMMM
```

Note : Function which contains both return type as well as parameters we called as **Signature of Function.**

- - - - - - - - - - - - - FUNCTION POINTER - - - - - - - - - - - - -

Function pointer also a variables which contains address(&) of Function. We can also dereference.

Syntax :

return type of fun (* pointer_name)(Data type of arguments);

Example :

```
/* Declaration of Fun */
int sum(int, int);
/* Definition of Fun */
int sum(int a, int b)
{
return a+b;
}
/* INIT Fun */
int (*ptr)(int, int) = & sum;
```

Example :

```
#include<stdio.h>

int sum(int, int); //Declaration of Fun

void main()

{

int s = 0; //Used to store Variable

int (*ptr)(int, int) = &sum; //Initialization of FUn

s = (*ptr)(10, 10); //Fun calling using ptr_name

printf("Sum = %d\n", s);

}

int sum(int a, int b) //Definition of Fun

{

return a + b;

}
```

O/P : Sum = 20

Callback Function :

A callback is any executable code that is passed as an argument to another code, which is expected to call back (execute) the argument at a given time. In simple terms, a callback is the process of passing a function (executable code) to anther function as argument and then it is called by the passed function. (Or)

One type of application of function pointer. We are passing arguments or address to function and also passing function to another function.

Note : **Normal pointer** contains address of the data or value. And **Function pointer** is like nrml pointer var but it contains the address of code or function.

Example :

```
/* CALLBACK FUNCTION */
```

```
#include<stdio.h>

void sum(int a, int b)

{

printf("%d\n",a + b);

}

void sub(int a, int b)

{

	printf("%d",a - b);

}

void display(void (*ptr)(int, int)) //Passing the address of Function by using ptr as an argument we pass

{

	(*ptr)( 10, 5); //calling the Fun

}

void main()

{

	display(&sum); //Calling display Fun and passing address of name_fun this are callback funtions.

	display(&sub);

}
```

O/P :

```
15

5
```

Returning Pointer from Function :

Example :

```
/* Return String Function */

#include<stdio.h>

char* display();
```

```
void main()
{
	char *str;
	str = display();
	printf("String is : %s ", str);
}
char* display()
{
return "KODEBLOOM";
}
```

O/P :

```
String is : KODEBLOOM
```

Example :

```
/* RETURN POINTER FUNCTION */
#include<stdio.h>
int* returnPointer(int []);
void main(){
	int *p;
	int a[] = {1,2,3,4,5};
	p = returnPointer(a);
	printf("%d",*p);
}
int* returnPointer(int a[])
{
	a = a+2;
	return a; //return a + 2;
}
```

O/P : 3

Return Multiple Values from a Function :

Unfortunately, C and C++ do not allow this directly. But fortunately, with a little bit of clever programming, we can easily achieve this. Below are the methods to return multiple values from a function in C:

1. By using pointers.
2. By using structures.
3. By using Arrays.

Returning multiple values Using pointers:

Pass the argument with their address and make changes in their value using pointer. So that the values get changed into the original argument.

Example :

```
    #include <stdio.h>

   void pointer(int* a, char* c) {

  // Writing data to passed addresses

 *a = 20;

  *c = 'z';

}

int main() {

 int a;

  char c;

  pointer(&a, &c);

 printf("a = %d\nc = %c", a, c);

 return 0;

}
```

O/P :

```
a = 20

c = z
```

Returning multiple values using structures :

As the structure is a user-defined datatype. The idea is to define a structure with two integer variables and store the greater and smaller values in to those variable, then use the values of that structure.

Example :

```
/* MULTIPULE RETURN FUNCTION USING STRUCTURE */
#include<stdio.h>
struct Book
{
    char name[20];
    int pages;
    float price;
};
struct Book inti()
{
    struct Book book = {"Programming with C", 300, 100.0};
    return book;
}
int main()
{
    struct Book book = inti();
    printf("Name = %s\n",book.name);
    printf("Pages = %d\n",book.pages);
    printf("Price = %f\n", book.price);
    return 0;
}
```

O/P :

```
        Name = Programming with C

        Pages = 300

        Price = 100.000000
```

Returning multiple values using an array (Works only when returned items are of same types) :

When an array is passed as an argument then its base address is passed to the function so whatever changes made to the copy of the array, it is changed in the original array.

Example :

```
#include <stdio.h>

int* func() {

    // Creating static array

    static int arr[2] = {10, 20};

    // Returning multiple values using static array

    return arr;

}

int main() {

  // Store the returened array

  int* arr = func();

  printf("%d %d", arr[0], arr[1]);

  return 0;

}
```

O/P : 10 20

- - - - - - - - - - - - - - - - ARRAYS -

It is a simple and fast way of storing multiple values under a single name. Array is collection of more than one data items. All must be carries homogeneous (same type). It always starts with index zero (0).

An array in C is a fixed-size collection of similar data items stored in contiguous memory locations. It can be used to store the collection of primitive data types such as int, char, float, etc., and also derived and user-defined data types such as pointers, structures, etc.

Arrays are two types like Numerical Array(Which contains int,float,double) and Character Array(Which having char data-type).

Syntax :

Data_type name_of_array[size of array];

Or

data_type array_name [size1] [size2]...[sizeN];

Example :

Int a[11/2];

Int a[b=11/2]; //Wrong

Initialization :

1 . At compile time :

When we declare the array that time only we initialize the array.

Int a[5]; //declaration

Different types declaration with initialization at compile time :

1. int a[5] = { 0, -1, 11, 10, 2 }; // declaration vth init
2. int a[ ] = { 0, 1, 2, 0, -1, 6, 7 }; //automatically size calculated
3. int a[5] = { 0, 1, -1 }; //remaining two init vth zero.
4. int a[5]; //Garbage value will be print
5. int a[3];
   a[0] = 1
   a[1] = 2
   a[2] = -1
6. int a[5] = { 0 }; // all memory location fill zero
7. int a[5] = { }; // Error
8. int a[2] = { 1, 2, 3, 4, 5 }; // Error

2. Run-Time :

Here we standard function to initialize array that is like loops.

#include<stdio.h>

```
int i;

int a[5];

printf("Enter Ur Elements of Arr :");

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

{

    scanf("%d",&a[i]);

}
```

Example :

```
#include <stdio.h>

int main() {

   int i;

   int a[5];  // Array declaration

   printf("Enter Your Elements of Array:\n");


   // Loop to take input from the user

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

      scanf("%d", &a[i]);  // Taking input for each array element

   }

   printf("You entered: ");

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

      printf("%d ", a[i]);  // Printing the array elements

   }

   printf("\n");

   return 0;

}
```

O/P :

```
Enter Your Elements of Array:

10  20  11  -11  16  18  29
```

```
You entered: 10 20 11 -11 16
```

When u use compile-time and run-time ?

If Size of array is small we can initialize at Compile-time and if the array size is large we use Run-time .

Overview of Array :

An array is a data structure containing a number of data values (all of which are of same type).

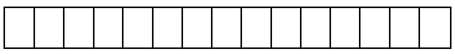
Data Structure :

Data structure is a format for organizing and storing data.

Also, each data structure is designed to organize data to suit a specific purpose.

Example :

Array is data structure which you can visualize as follows :



Imagine an array as a large chunk of memory divided into smaller block of memory and each block is capable of storing a data value of some type.

Types of Array :

1. One Dimensional Array :
   The simplest form of array one can imagine is one dimensional array.
   It’s a large chunk of memory a single row divided into Number of blocks and each block is capable of storing some data value. And all of them must be same type.
   You can also imagine a block of memory as a variable and an array is collection variables.
   Dimension represented as – [ ] → Square Brackets(Know as Subscript)
   Example : marks [ ]
   //It is also called as single subscript Array.

5	4	3	2	1

INITILIZATION OF ARRAY AT RUNTIME :

```
// One Way of runtime init for 1D
int marks[3];
scanf(“%d %d %d “, &marks[0],&marks[1],marks[2]);
```

```
// Other method of init of 1D
 for(i =0;i<n;i++)     //  i ranges from 0 to n-1 and execute upto n-1
{
 Scanf("%d",&marks[i]);
}
```

Display/Print the Array Elements :

Method-1 :

```
//It's difficult for more number of elements.
int marks[3];
printf("%d %d %d ", marks[0], marks[1],marks[2]);
```

Method-2 :

```
for(i=0;i<n;i++)
{
printf("%d ",marks[i]);
}
```

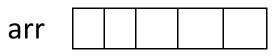
Declaration and Definition of 1D Array :

Syntax :

Data_type   name_of_array[no.of elements];

Example :

int arr[5];



Compiler will allocate a contiguous block of memory of size = 5*sizeof(int)
Size of integer depends on machine to machine either it will be 2bytes or 4bytes

The length of an array can be specified by any integer constant expression.
The length of an array can be specified by any **positive integer** constant expression.

```
int arr[5];
int arr[5+5];
int arr[5*3];
int a; int arr[a = 21/3];
int arr[-5];   //Its not correct length of array bcz length of array cann't be negative.
```

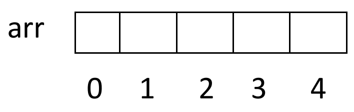
Specifying the length of an array using **macro** is considered to be excellent practice.

```
#define N 10
int arr[N];
```

Accessing elements from 1D array :

To access array element just write :

array_name[index]



It always starts from index 0 if suppose we have an array with length is 5 as we know index starts with 0 and last index of array will be 4**, It always goes up to length-1.**

If you want to access the first element of an array :   arr[0]

Accessing the second element of an array :   arr[1]

-
-
-

And so on…………

Example :

```
/* ARRAY WITHOUT MACRO */
// If we want do any changes we need to update whole prgm
#include<stdio.h>
int main()
{
    int a[10], i;
    for(i=0;i<10;i++)
    {
        printf("Enter Input for Index %d:",i);
        scanf("%d",&a[i]);
    }
    printf("Array elements as follows:\n");
    for(i=0;i<10;i++)
```

```
	{
		printf("%d\n",a[i]);
	}
	return 0;
}
```

O/P :

```
Enter Input for Index 0:11
Enter Input for Index 1:12
Enter Input for Index 2:13
Enter Input for Index 3:14
Enter Input for Index 4:15
Enter Input for Index 5:16
Enter Input for Index 6:17
Enter Input for Index 7:18
Enter Input for Index 8:19
Enter Input for Index 9:20
Array elements as follows:
11
12
13
14
15
16
17
18
19
```

/* ARRAY WITH MACRO */

// If we want do any changes we just update the macro and change will happen everywhere in the prgm where you use macro.

```
#include<stdio.h>
#define N 15
```

```
int main(){
        int a[N], i;
        for(i=0;i<N;i++)
        {
                printf("Enter Input for Index %d:",i);
                scanf("%d",&a[i]);
        }
        printf("Array elements as follows:\n");
        for(i=0;i<N;i++)
        {
                printf("%d\n",a[i]);
        }
        return 0;
}
```

O/P :

```
Enter Input for Index 0:
11
Enter Input for Index 1:
12
Enter Input for Index 2:
13
Enter Input for Index 3:
14
Enter Input for Index 4:
15
Enter Input for Index 5:
16
Enter Input for Index 6:
17
Enter Input for Index 7:
18
```

```
Enter Input for Index 8:
19
Enter Input for Index 9:
20
Enter Input for Index 10:
21
Enter Input for Index 11:
22
Enter Input for Index 12:
23
Enter Input for Index 13:
24
Enter Input for Index 14:
25
Array elements as follows:
11
12
13
14
15
16
17
18
19
20
21
22
23
24
```

Initializing 1D Array :

Syntax :

Data_type Array_name[size] ={value1, value2, value3, ……….};

Example :

 Int marks[6] = { 70, 80, 65, 90, 95, 85 };

//Following way they initilize

marks[0] = 70   marks[1] = 80 marks[2] = 65   marks[3] = 90 marks[4] = 95 marks[5] = 85

Method 1 :

arr[5] = {1, 2, 3, 4, 5};

Method 2 :

//It's much better to use

arr[ ] = { 1, 2, 5, 67, 32 };

Method 3 :

//Not much preferred Bcz of garbage values

```
int arr[5];
arr[0] = 1;
arr[1] = 2;
arr[2] = 5;
arr[3] = 67;
arr[4] = 32;
```

Method 4 :

```
int arr[5];
for(i=0;i<5;i++)
{
scanf("%d",&arr[i]);
}
```

Note :

If Number of elements are lesser than the length specified

Example :

int arr[10] = {45, 6, 2, 78, 5, 6 };

The remaining locations of the array are filled by value 0.

int arr[10] = { 45, 6, 2, 78, 5, 6, 0, 0, 0, 0 };

A Small Tip :

```
int arr[10];
for(i=0;i<10;i++)                   →       int arr[10] = {0}; //preffered
{
arr[i]=0;
}
```

Both are similar but code is much not preferred method as this one line.

Note :

int arr[10] = { };

Should not write this because this is illegal.

You must have to specify at least 1 element. It cann't be completely empty. And it is also illegal to add more elements than the length of an array. You cann't add more number of elements than the length specified.

Exampe :

int arr[5] = { 1, 2, 3, 4, 5, 6 };

Designated Initializers :

Sometimes we want something like this :

int arr[10] = {1, 0, 0, 0, 0, 2, 3, 0, 0, 0};

Certain position we want fill values other than zeros and the rest of the position fill all zeros.

Example :

int arr[10] = {1, 0, 0, 0, 0, 2, 3, 0, 0, 0};

We want :

- 1 in position 0
- 2 in position 5
- 3 in position 6

```
int arr[10] = { [0] = 1, [5] = 2, [6] = 3 };
                 |
```

This way of initialization is called designated initialization. And each number in the square brackets is said to be a designator.

Advantages :

1. No need to bother about the entries containing zeros.
   int arr[15] = { 1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
   We can write it simple as below :
   int a[15] = { [0] = 1, [5] = 2 };
2. No need to bother about the order at all.
   int a[15] = { [0] = 1, [5] = 2 };
   int a[15] = { [5] = 2, [0] =1 };   Both are same

   Note :
   If the length of an array is 'n' then each designator must be between 0 and n-1.
   ```
   int a[5] = { [0] = 4, [4] =78 };     // It's correct
   int a[5] = { [0] = 4, [5] =78 };    //  It's wrong
   ```

If we **won't Mention the length :**

- Designators could be any non-negative integer.
- Compiler will deduce the length of the array from the largest designator in the list.

  Example :

  int a[ ] = { [5] = 90, [20] = 4, [1] = 45, [49] = 78 };

  |

  Because of this designator maximum length of this array would be 50.

Finally we mix both the traditional way of initialization and designator initialization.

int a[ ] = { 1, 7, 5, [5] = 90, 6, [8] = 4};   **Similar to**   int a[ ] = { 1, 7, 5, 0, 0, 90, 6, 0, 4 };

- But if there is a clash then designated initializer will win.

int a[ ] = { 1, 2, 3, [2] = 4, [6] = 45 };   **Similar to**   int a [ ] = { 1, 2, 4, 0, 0, 0, 45 };

Example :

```
/* REVERSE NUMBERS USING ARRAY */

#include<stdio.h>

int main()

{

    int arr[9] = {34, 56, 54, 32, 67, 89, 90, 32, 21};

    //Orginal order

    for(int i=0; i<9; i++)

    {

        printf("%d ", arr[i]);
```

```
        }
        printf("\n");
        //Reverse Order
        for(int i=8; i>=0; i--)
        {
                printf("%d ", arr[i]);
        }
        return 0;
}
```

O/P :

```
     34 56 54 32 67 89 90 32 21
     21 32 90 89 67 32 54 56 34
```

Example :

```
/* TO Check whether any of the digits in a number appears more than once */
#include<stdio.h>
int main()
{
    int seen[10] = {0};     //Part-1

    int N;
    printf("Enter the NO : ");
    scanf("%d", &N);

    int rem;

     while(N>0)     //Part-2
    {
        rem = N%10;
        if(seen[rem] == 1)
        break;
        seen[rem] = 1;
        N = N/10;
    }
    if(N>0)      //Part-3
    printf("Yes");
    else
    printf("No");
    return 0;
}
```

O/P :

```
Enter the NO :  67827
Yes
```

Here Two cases we need to address :

When N > 0 and break

When N == 0

→TO Calculate how many elements are there we use **sizeof o**perator is used.

Syntax :

```
  sizeof(name_of_arr)/sizeof(name_of_arr[0])
```

1 . sizeof(name_of_arr) :

Example :

```
int a[ ] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
sizeof(a);
```

There are total 10 integers. And assume that each integer requires 4 bytes.

Sizeof(a) = 4 * 10 = 40 bytes

2 . sizeof(name_of_arr[0])

Example :

```
int a[ ] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
sizeof(a[0]);
```

sizeof(a[0]) = 4bytes

sizeof(name_of_arr)/sizeof(name_of_arr[0]) = sizeof(a)/sizeof(a[0]) = 40/4 = 10 No.of.Ele

→size of 1 arr element * number of elements = **size of whole array**

→**number of elements =** size of whole arr/size of 1 arr element

= sizeof(name_of_arr)/sizeof(name_of_arr[0])

Example :

```
#include<stdio.h>

int main()

{

int a[ ] = { some numbers……………};

printf("%d",sizeof(a)/sizeof(a[0]));

return 0;

}
```

2 . Multidimensional Array :

Multidimensional arrays can be defined as an array of arrays.
General form of declaration N-dimensional array is as follows :
Syntax :

Data_type name_of_array[size1][size2]…………….[sizeN];

Example :

```
int a[3][4];  //2D array

int a[3][4][5]; //3D array
```

→Sizeof Multidimensional array :

Size of multidimensional array can be calculated by multiplying the size of all the dimensions.

Example :

size of a[10][20] = 10*20 = 200

= 200***4** = 800 bytes

//We can store upto 200 elements in this array

Size of a[4][10][20] = 4*10*20 = 800

= 800*4 = 3200 bytes

//We can store upto 800 elements in this array

2D array :

The basic form of declaring two dimentional array is :

Data_type name_of_array[x][y];

Where x and y are representing the size of the array

Visualizing 2D Array :

Recall that a multideimensional array is an array of arrays

Example 2D array :

int arr[4][5];

4*5 =size of matrix = 20 elements

//[4] → #rows [5]→ #columns

Size of arr[4][5] = 4*5 = 20 elements

If we want to calculate with bytes = 20*4 = 80 //assume 1 element size is 4bytes

Initialize 2D array :

Method 1 :

int a[2][3] = {1, 2, 3, 4, 5, 6}; //Sometimes confusing

	0	1	2
0	1	2	3
1	4	5	6

Method 2 : Row1 Row2

int a[2][3] = {{1, 2, 3}, {4, 5, 6}}; //Better method

Accessing 2D array elements :

Using row index and column index.

Example :

We can access elements stored in 1st row and 2nd column of below array

a[0][1]

Print 2D array elements :

Note :

In 1D array elements can be printed using simple for loop

```
int a[5] = {1, 2, 3, 4, 5};
for(I=0;i<5;i++)
{
```

```
printf(“%d ”,a[i]);

}
```

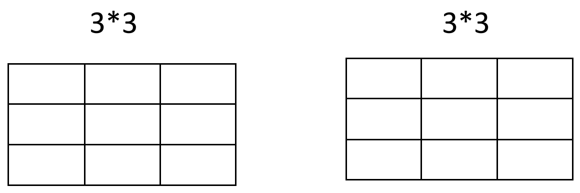
IN 2D array elements can be printed using two nested for loops.

```
#include<stdio.h>

void main()

{

int a[2][3] = {{1, 2, 3}, {4, 5, 6}};

for(i=0;i<2;i++)

{

for(j=0;j<3;j++)

{

printf{“%d “,a[i][j]);

}

}

}
```

```
O/P : 1  2  3  4  5  6
```

3D array :

int arr[2][3][3]; // It means two 3D arrays



If we want to access the element in the 1st Row and 3rd Column of 1st 2D array.

arr[0][0][2]   //1st [0] indicates First 2D arr, 2nd [0] indicates 1st Row and [2] indicates 3rd column.

Initializing 3D array :

Method 1 : //Not much better

Int a[2][2][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

1	2	3
4	5	6

7	8	9
10	11	12

2* 3 2* 3

Method 2 : //better method

Int a[2][2][3] = {

{ {1, 2, 3}, {4, 5, 6}}, //1st 2D array

{{7, 8, 9},{10, 11, 12}} //2nd 2D array

};

1	2	3
4	5	6

7	8	9
10	11	12

2* 3 2*3

Printing 3D array :

```
#include<stdio.h>
void main()
{
   int arr[2][2][3] ={
      {{1,2,3,},{4,5,6}},
      {{7,8,9,},{10,11,12}}
   };
   for( int i=0;i<2;i++)
   {
      for(int j=0;j<2;j++)
      {
         for(int k=0;k<3;k++)
         {
            printf("%d ",arr[i][j][k]);
         }
      }
   }
}
```

O/P : 1 2 3 4 5 6 7 8 9 10 11 12

Example :

```
/*  5*5 array of integers and then prints the row sum and column sum */
#include<stdio.h>
void main ()
{
   int arr[5][5] = {
   {8,3,9,0,10},
   {3,5,17,1,1},
   {2,8,6,23,1},
   {15,7,3,2,9},
   {6,14,2,6,0}
};
   int i,j;
   int sum = 0;
   printf("Row Sum : ");
   for(i=0;i<5;i++)   // i is row index, row constant or static
   {
      for(j=0;j<5;j++)  //j is column index here j incremented
      {
         sum += arr[i][j];
      }
      printf("%d ",sum);
      sum = 0;    // Used to separate and doesn't effect previous sum of other row
   }
   printf("\nColumn Sum : ");
   for(j=0;j<5;j++)  // Putting column index static
   {
      for(i=0;i<5;i++) // moving row index
      {
         sum += arr[i][j];
      }
      printf("%d ",sum);
```

```
        sum = 0;
    }
}
```

O/P :

```
Row Sum : 30 27 40 36 28
Column Sum : 34 37 37 32 21
```

MATRIX MULTIPLICATION :

1	2	3
1	2	1
3	1	2

3*3

1	2	3
1	2	1
3	1	2

3*3

1*1 + 1*2 + 3*3 = 12

1*2 + 2*2 + 3*1 = 9

1*3 + 2*1 + 3*2 = 11

1*1 + 2*1 + 1*3 = 6

12	9	11
6	7	7
10	10	14

3*3 //Resultant matrix

Important Point :

- In order to multiply two matrices, #columns of 1st matrix = #rows of 2nd matrix.

  Therefore, it is mandatory to have no.of columns of 1st matrix to be equal to no.of rows of 2nd matrix.

- Also size of the resultant matrix depends on #rows of 1st matrix and #columns of 2nd matrix.

Example :

1	2	3
1	2	1

2*3

1	2
1	2
3	1

3*2

Explanation for Result :

1*1 + 2*1 + 3*3 = 12

1*2 + 2*2 + 3*1 = 9

1*1 + 2*1 + 1*3 = 6

1*2 + 2*2 + 3*1 = 7

12	9
6	7

2*2

Example :

```
#include<stdio.h>

#define MAX 50

int main()

{

   int a[MAX][MAX], b[MAX][MAX], product[MAX][MAX];

   int arows, acolumns, brows, bcolumns;

   int i,j,k;

   int sum =0;


   //part-1

   printf("Enter the rows and columns of the matrix a : ");

   scanf("%d %d", &arows, &acolumns);


//Part-2

   printf("Enter the elements of matrix a :\n");

   for(i=0;i<arows;i++)

   {

      for(j=0;j<acolumns;j++)

      {

         scanf("%d",&a[i][j]);

      }

   }
```

```
printf("Enter the rows and columns of the matrix b : ");

scanf("%d %d", &brows ,&bcolumns);

if(brows != acolumns)

{

   printf("Sorry! We cannot multiply the matrices a and b");

}

   else

   {

      printf("Enter the elements of matrix b :\n");

      for(i=0;i<brows;i++)

      {

         for(j=0;j<bcolumns;j++)

         {

            scanf("%d", &b[i][j]);

         }

      }

   }

   //Part - 3

   printf("\n");


   for(i=0;i<arows;i++)

   {

      for(j=0;j<bcolumns;j++)

      {

         for(k=0;k<brows;k++)

         {

            sum += a[i][k]*b[k][j];

         }
```

```
                product[i][j] = sum;
                sum = 0;
            }
        }

        //Printing the array elements
        printf("Resultant matrix\n");
        for(i=0;i<arows;i++)
        {
            for(j=0;j<bcolumns;j++)
            {
                printf("%d ",product[i][j]);
            }
            printf("\n");
        }
        return 0;
}
```

O/P :

```
Enter the rows and columns of the matrix a : 3 3
Enter the elements of matrix a :
1 2 3
1 2 1
3 1 2
Enter the rows and columns of the matrix b : 3 3
Enter the elements of matrix b :
1 2 3
1 2 1
3 1 2
```

Resultant matrix

```
12 9 11

6 7 7

10 10 14
```

Constant Array :

Either 1D or multi-dimensional arrays can be made constant by starting the declaration with the keyword const.

Example :

```
const int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

a[1] = 45;
```

O/P : Error

Advantages :

It assures us that the program will not modify the array which may contain some valuable information.

It also helps the complier to catch errors by informing that there is no information to modify this array.

Variable length arrays :

```
#include<stdio.h>

int main()

{

   int n;

   printf("Enter the number of elements you want to reverse : \n");

   scanf("%d", &n);

   int a[n];    //variable length array

   int i;

   printf("Enter all the %d elements:  \n",  n);

   for(i=0;i<n;i++)

   {

      scanf("%d",&a[i]);
```

```
    }
    printf("Elements in reverse order are :\n ");
    for(i=n-1;i>=0;i--)
    {
        printf("%d ",a[i]);
    }
    return 0;
}
```

O/P :

```
Enter the number of elements you want to reverse :
5
Enter all the 5 elements:
11 12 13 14 15
Elements in reverse order are :
15 14 13 12 11
```

Advantages :

- At the time of execution, we can decide the length of the array.
- No need to choose the fix length ehile writing the code.
- Even arbitrary expressions are possible

Example :

```
int a[3*i+5];
int a[k/7+2];
int a[i+j];
```

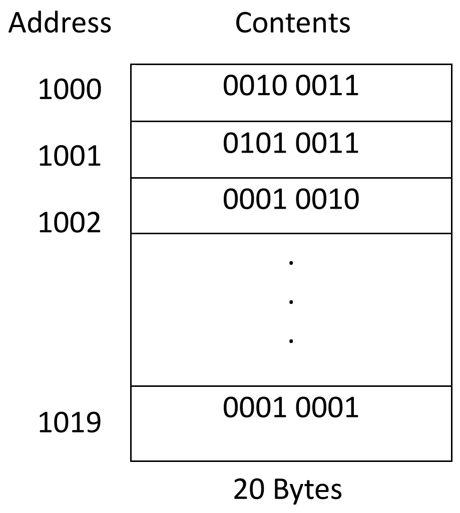
Points to be Noted :

- Variable length arrays **cannot have static storage duration**.
- Variable length array does not have the initializer.

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ POINTERS _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

A **pointer** is a variable that stores the **memory address** of another variable. Instead of holding a direct value, it holds the address where the value is stored in memory. There are **2 important operators** that we will use in pointers concepts i.e.

- **Dereferencing operator**(*) used to declare pointer variable and access the value stored in the address.
- **Address operator(&)** used to returns the address of a variable or to access the address of a variable to a pointer.



Pointer is variable which is capable of storing the initial address of object which it wants to point too.

Pointer is a special variable that is capable of storing some address.

It points to a memory location where the first byte is stored

Declaring pointers variable :

General Syntax :

data_type *pointer_name

Here Data_type refers to the type of the value that the pointer will point to.

Example :

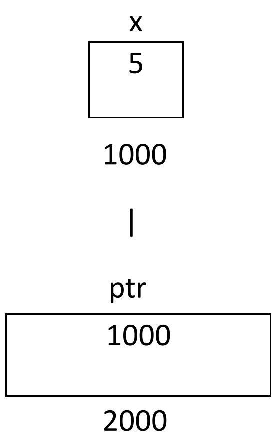
int *ptr;	→	Points to integer value
char *ptr;	→	Points to character value
float *ptr;	→	Points to float value

Initialize pointer :

- Simply declaring a pointer is not enough.
- It is important to initialize the pointer before use.
- One way to initialize a pointer is to assign address of some variable.

Example :

int x = 5;



int *ptr;

ptr = &x;

//& means address of operator

WE can also write all these lines one single line as **int x = 5, *ptr = &x;**

Value of Operator :

Value of operator/indirection operator/deference operator is an operator that is used to access the value stored at the location pointed by the pointer.

Example :

int x = 5;

int *ptr;

ptr = &x;

printf("%d", *ptr);   //*ptr is also called as value of operator(It says go to the address of object and take what is stored in the object.)

We can also **change the value of the object** pointed by the pointer.

Example :

int x = 10;

int *ptr = &x;

*ptr = 4; //By using this assignment statement we can also change this value

printf("%d", *ptr);

O/P : 4

A Word of Caution :

- Never apply the indirection operator to the uninitialized pointer.

Example :

int *ptr;

printf("%d", *ptr); // This is totally illegal

O/P : Undefined behaviour

- Assigning value to an uninitialized pointer is dangerous.

Example :

int *ptr;

*ptr = 1; //This pointer is not pointing to any location.

O/P : segmentation fault (SIGSEGV)

Usually, a segmentation fault is caused by program trying to read or write an illegal memory location.

Assignment of Pointer :

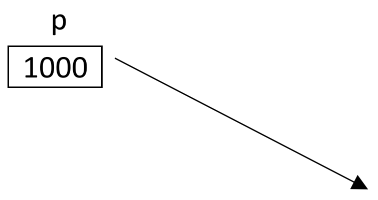
Here we can assign one to another pointer

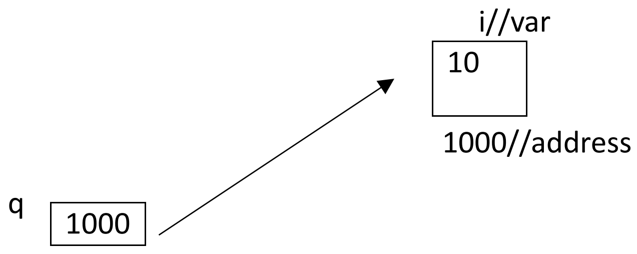
Example :

int i = 10;

int *p, *q;

p =&i;

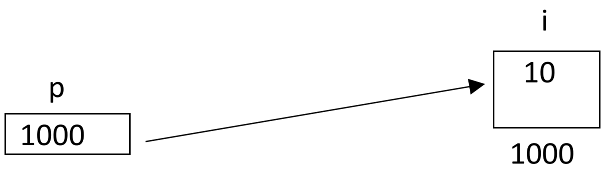


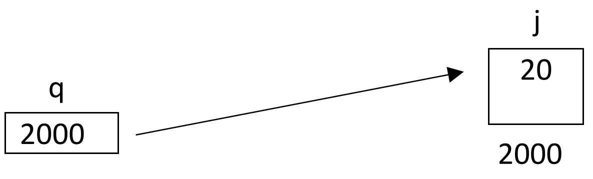


```
q = p;

printf(“%d %d”, *p, *q);
```

O/P : 10 10

Note : q = p is not same as *q = *p





```
int i = 10, j = 20;

int *p, *q;

p = &I;

q = &j;

*q = *p;
```

Example :

```
#include <stdio.h>

  void  main() {

   int i = 1;

   int *p = &i;

   int *q = p; // Declare q as an integer pointer

   *q = 5; // Modify the value of i through q

   printf("%d", *p); // Output the updated value of i
}
```

O/P : 5

- - - - - - - - - - - - - - - - - - QUALIFIERS - - - - - - - - - - - - - - - - - - -

Qualifiers also called as Modifiers. They used to change the meaning of basic data-types.

Qualifiers are keywords that modify the behavior of a data type. They can be used to describe the properties of a variable or pointer.

There are Four types of Qualifiers they are :

Size qualifier, Sign qualifier, Const qualifier and Volatile qualifier.

1.Size Qualifier :

It is mainly useful in-order to change the size of the basic data-type then used Size Qualifier. There are two types of size qualifiers . Short and Long qualifier.

A . Short Qualifier :

Its used to reduce the size of the basic data-type and keeps the as it is.

B . Long Qualifier :

Which increase the basic data-type.

Example :

If int → 2 bytes   then size of long it becomes   long int → 4 bytes

If double → 8 byte   then size of long double becomes   long doble → 10 bytes.

2.Sign Qualifier :

Here there are two types of Sign qualifiers are available.They are signed and unsigned qualifiers.

A. Signed Qualifier :

There is sign before the corresponding value. The value maybe +ve value or -ve value . It may also allow both the signs.

Example :

+10 Or -10

Range of Signed Qualifiers :

If we take n = 3 then 2^3 is 8 so first 4 are -ve and next 4 is +ve numbers.

{ -4,-3,-2,-1, 0, 1, 2,3 } So the Range is -4 to 3 (min is -4 and max is 3).

If n = 4 then $2^4 \rightarrow$ 16 So (-8 to -1 first 8 values of -ve sign) and (0 to 7 is next 8 +ve numbers) Then the range is (-8 to 7).

It is -2^{3-1} to $2^{3-1} - 1$. So, the range of signed is -2^{n-1} to 2^{n-1} -1.

B. Unsigned Qualifier :

There is no signed before the corresponding value then the default sign is plus.

Example :

10 Then the value become plus.

Rang of Unsigned Qualifier :

N=3 bits //N = No.of.bits

Total No of +ve numbers we store is $2^3 \rightarrow$ 8 +ve Numbers (min +ve is 0 and max/last No is 7) Also written as 0 to 2^3 -1

If N = 4 then $2^4 \rightarrow$ 16 (0 -15) it also known as 0 to $2^4 - 1$.

So the rang of unsigned written as 0 to $2^n - 1$.

3.Const Qualifier :

We can maintain a variable as constant. The value of variable may changes during program execution where as value of constant cannot changes during the program execution.

‘const’ keyword is used to declared a variable for storing a constant value. We already having existing constants like integer constants, character constants etc. While are called as literals. User can also create symbolic constant to use in the program as per the need.

Define constant :

const data-type variable = value;

Example :

float PI = 3.14;

Now PI is a variable created which is initialized with the given value. But as it is variable, its value can be changed by the program.

To avoid this variable PI can be declared using ‘const’ keyword.

const float PI = 3.14;

Now any attempt of changing the value of PI will result in compilation error as **Cannot modify a const object**

Example :

```
const int a = 11;

  // It is not possible to change during execution
```

4.Volatile Qualifier :

We can change the value from outside the programs such as interrupts.

That can change **unexpectedly.** No **assumption.** Volatile **prevents** compiler to perform any **optimization** on that object, that can **change** in ways that compiler cannot determine.

The 'volatile' keyword can also be used along with data type while declaring the variable.

volatile int a;

It indicates that a variable can be changed by a background routine.

Every reference to the variable will reload the contents from memory.

It will not use the copy of the variable which can be present in the variable which can be present in the register.

Compiler optimizations are not applied on 'volatile' variables as their value can be changed by code outside the scope of current code at any time. Significant of volatile keyword is to ensure that all the threads always see the latest value even through the cache system or optimization is applied.

Exampe :

```
#define MEMORY_ADDRESS

int main(void)

{

int value = 0;

int *p = (int *) MEMORY_ADDRESS;

while(1)

{

value = *p;

if(value) break;
}
return 0;
}
```

- - - - - - - - - - - - - - - - - - Enumeration -

It's a user-defined type to assign names to integral constants. Enum used to increase the readability of the program. And modification will be easy by using enums.

Enum having both local global and global scope.

- - - - - - - - - - - - - - - - - - GIT COMMANDS -

Introduction :

Source Control (For example surveillance system of the source code) means someone somewhere is watching your source code using source controller.

Source Control tool also called as Version Control Software. VC is all about record changes in the code. Which means I know about your source code I know what you did yesterday, today, few sec back.

By Creating a Surveillance System (a.k.a VCS) comes with a cost. Why we use it because We need it.

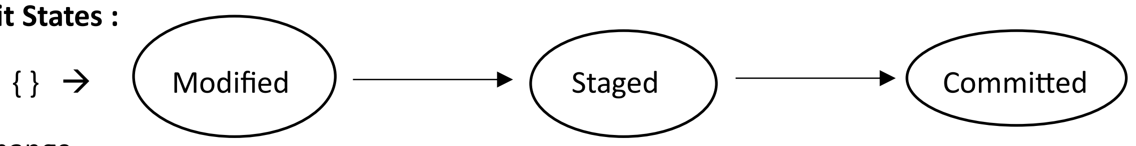
Types of VCS :

There are two types they are : Centralized and Distributed VCS

1. Centralized VCS :
   Your source code and all information about I know what you did last summer is stored in the server. When ever one user want to work on that and it checks out the file and after the working it checks in the file. There is ntg that stores in the users local system is called centralized VCS like SVN.
   SVN is a Centralized VCS
2. Distributed VCS :
   Consider your centralized sever is GITHub you clone the repo which means that the exact replica of everything comes to your local system and your working on your local system. So History of I know what you did yesterday or today is actually stored on your local system. This means that you consider a source control in your local system and U don't need to be connected to be GitHub and to go ahead start working on it checking-out and checking-in your file and changes.
   Git is a Distributed VCS

How Git Stores Data :
Snapshot and NOT the difference.

Git States :



Change

Modified : Files are modified.. Git knows it… But can’t do anything about it

Staged : Git has marked this file for the next snapshot

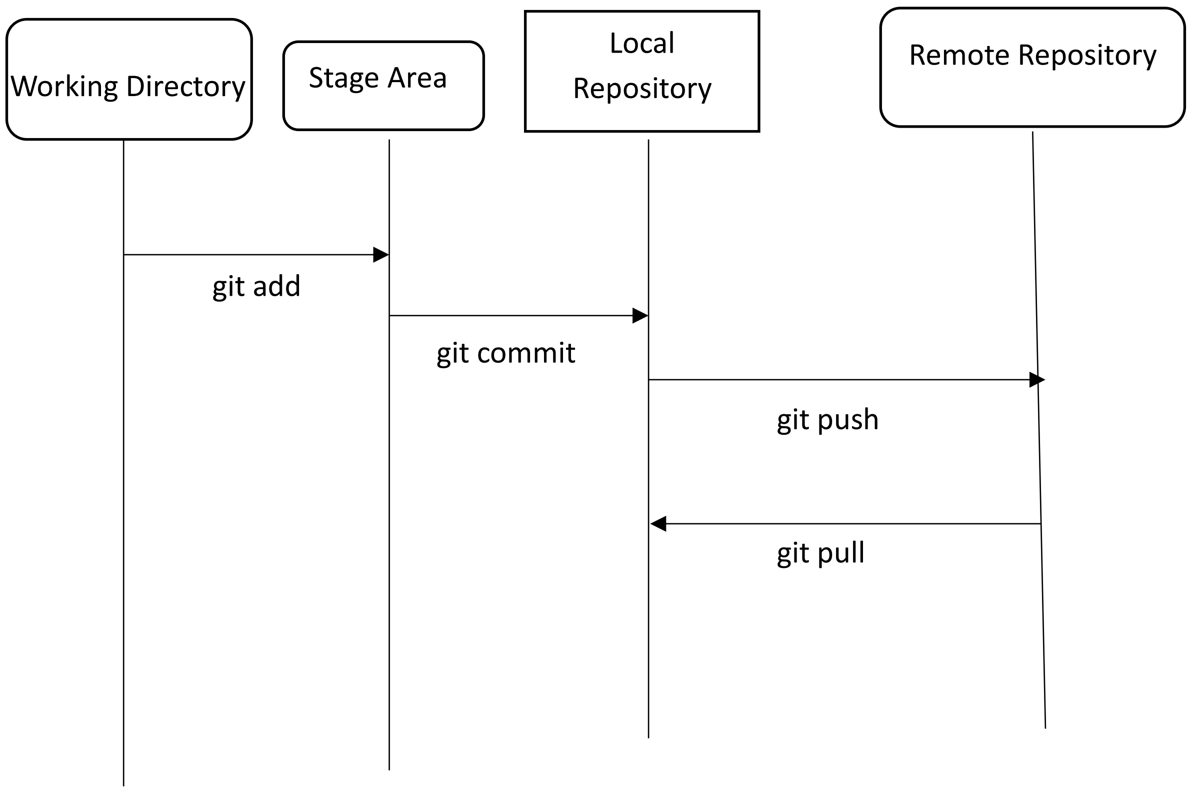
Committed : Snapshot Taken.

HISTORY :

Developed by Linux Community +Linus Torvalds

Born in 2005

GitHub/ Bit-Bucket this are the open source repo



Configure Git With GitHub and Push Files to Repository :

1. Install git

2. Create new Directory in terminal
3. Initialize git
4. Configure values globally
5. Login to github
6. Create new repository
7. Clone the repository to newly created directory
8. Create a file
9. Add file to staging area
10. Commit the file with comment
11. Create token in github
12. Set url with token
13. Push file into repository

Installation of git :

- $ sudo apt-get install git
- $ git –version

Creation of New Directory :

- $ mkdir <<dir_name>>
- $ cd <<dir_name>

Initilize git :

- $ git init
- $ ls -a //Used for hidden files

Configure GIT Globally :

- $ git config –global user.name “<<username>>”
- $ git config –global user.email “<<email>>”
- $ git config --list

Commands :

1. git –version :
   It show the version of git

```
root@LAPTOP-M3171AO3:~# git --version
git version 2.43.0

root@LAPTOP-M3171AO3:~# mkdir Git_Commands
root@LAPTOP-M3171AO3:~# ls
```

Git_ Commands

```
root@LAPTOP-M3171AO3:~# cd Git_Commands
```

```
root@LAPTOP-M3171AO3:~/Git_Commands#
```

2. git init :

   It is used to initialize

   ```
   root@LAPTOP-M3171AO3:~/Git_Commands# git init
   hint: Using 'master' as the name for the initial branch. This default branch name
   hint: is subject to change. To configure the initial branch name to use in all
   hint: of your new repositories, which will suppress this warning, call:
   hint:
   hint:   git config --global init.defaultBranch <name>
   hint:
   hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
   hint: 'development'. The just-created branch can be renamed via this command:
   hint:
   hint:   git branch -m <name>
   Initialized empty Git repository in /root/Git_Commands/.git/
   root@LAPTOP-M3171AO3:~/Git_Commands#ls
   ```

3. ls -a :
   It is used to get some hidden
   ```
   root@LAPTOP-M3171AO3:~/Git_Commands# ls -a
   .  ..  .git
   ```
   //This means init is completed
4. git config --global user.name “<<USER_NAME of github>>” :
   To Confirm the user_id and email id of a github.
   If you got the promt immediately that impulse the command implemented correctly/perfectly.
   ```
   root@LAPTOP-M3171AO3:~/Git_Commands# git config --global user.name
   "ThallapalliLaxmi"
   root@LAPTOP-M3171AO3:~/Git_Commands#
   ```
5. git config - -global user.email “<<USER_EMAIL>>” :
   ```
   root@LAPTOP-M3171AO3:~/Git_Commands# git config --global user.email
   "laxmi.t@kodebloom.com"
   root@LAPTOP-M3171AO3:~/Git_Commands#
   ```
   After this use below command

6. git config --list :
   To check whether they are configured or not by this command we can check.
   ```
   root@LAPTOP-M3171AO3:~/Git_Commands# git config --list
   user.name=ThallapalliLaxmi
   user.email=laxmi.t@kodebloom.com
   core.repositoryformatversion=0
   core.filemode=true
   core.bare=false
   core.logallrefupdates=true
   ```
   //By this we can confirm that the configured successfully completed.
   ```
   root@LAPTOP-M3171AO3:~/Git_Commands# ls
   root@LAPTOP-M3171AO3:~/Git_Commands# cd
   root@LAPTOP-M3171AO3:~# ls
   ```

   Login to GitHub :
   - Create account.
   - Login with your account.

   Create New Repository :
   - Create new Repository in your login

   Clone Repository :
   - Clone the new repository to newly created directory . Use this command
   - $ **git clone <<url>>**

   Create Files :
   - Create any code in the directory.

   Add files to Staging Area :
   - $ git add <<filename_with_extension>>
   - $ git status

   Commit files :
   - $ git commit -m "<<comment>>"
   - $ git status

7. git clone :
   git clone (paste url-link of ur code from git repo)
   git clone is primarily used to point to an existing repo and make a **clone or copy** of that repo at in a new directory, at another location. The original repository can be

located on the local filesystem or on remote machine accessible supported protocols. The git clone command copies an existing Git repository.

```
root@LAPTOP-M3171AO3:~# cd C-Programming-Language-
root@LAPTOP-M3171AO3:~/C-Programming-Language-# ls
Function.c
root@LAPTOP-M3171AO3:~/C-Programming-Language-# git clone
https://github.com/ThallapalliLaxmi/C-Programming-Language-.git
Cloning into 'C-Programming-Language-'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (3/3), done.
root@LAPTOP-M3171AO3:~/C-Programming-Language-# ls
C-Programming-Language-  Function.c
root@LAPTOP-M3171AO3:~/C-Programming-Language-#
```

Use any text editor to create a file (Ex : vi hello.c)

```
root@LAPTOP-M3171AO3:~/C-Programming-Language-# vi sample.c

root@LAPTOP-M3171AO3:~/C-Programming-Language-# ls

C-Programming-Language-  Function.c  sample  sample.c
```

8. git status :
   Which is used to show the status of git (whether updates are done or not, files added or not and any other changes)

```
root@LAPTOP-M3171AO3:~/C-Programming-Language-# git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        C-Programming-Language-/
        sample
        sample.c

nothing added to commit but untracked files present (use "git add" to track)
```

   Note : It means which available in local repository but not available in remote repository.
9. git add :
   It is used to add the files to remote repository

```
git add FILE_NAME
root@LAPTOP-M3171AO3:~/C-Programming-Language-# git add sample.c
root@LAPTOP-M3171AO3:~/C-Programming-Language-# git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   sample.c

// Untracked files:
  (use "git add <file>..." to include in what will be committed)
        C-Programming-Language-/
        sample
```

10. git commit :

    What are the changes are done that should be committed give this command when your changed the file. And in-order to move/go to local repo use this.

```
git commit -m “D-M-Y”       //You can also use this ”D.M.Y”
root@LAPTOP-M3171AO3:~/C-Programming-Language-# git commit -m "11-03-2025"
[main a2aae1b] 11-03-2025
 1 file changed, 6 insertions(+)
 create mode 100644 sample.c
//Now this file is available in the local repository .
```

In-Order push file from local repo to the remote repo we need to **authenticate** it . And that should done by with help of creating Tokens.

Create Token in GitHub :

- Goto Settings
- Click on Developer Settings in the Left Pane
- Click on generate classic token in Left Pane
- Select Create new token
- Give the name and expiry of token
- Select all the options to use generated token
- Copy the token which was generated.

Set Url with token :

- $ git remote set-url origin https ://<<token>>@github.com/<<username>>/<<repository>>

11. git remote set-url orgin
    https://<<token>>@githup.com/<<username>>/<<repository>>
    git remote set-url origin PASTE THE TOKEN
    URL@githup.com/USER_NAME/REPOSITORY_NAME

```
root@LAPTOP-M3171AO3:~/C-Programming-Language-# git remote set-url origin
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@github.com/ThallapalliLaxmi/
C-Programming-Language-
root@LAPTOP-M3171AO3:~/C-Programming-Language-# git remote set-url origin
https://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@github.com/Thallapalli
Laxmi/C-Programming-Language-
```

    We have successfully authenticated so next we need to PUSH file/complete details into Remote repo/Main branch.

    Push file into Remote Repository :

    - $ git push origin main

12. git push :
    Used to add the file to remote repository from the local repository.
    git push origin GIVE BRANCH NAME   (Or)
    git push origin main

```
root@LAPTOP-M3171AO3:~/C-Programming-Language-# git push origin main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 348 bytes | 348.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/ThallapalliLaxmi/C-Programming-Language-
  6579395..a2aae1b  main -> main
root@LAPTOP-M3171AO3:~/C-Programming-Language-# ls
C-Programming-Language-  Function.c  sample  sample.c
```

    //After refreshing we find the file in the remote repo(git). From the local repo.

13. git pull :
    Use to get the modifications done in the remote repo and they visible in local repo or that will be reflected here. Pull is similar to clone.

```
root@LAPTOP-M3171AO3:~/C-Programming-Language-# git pull
Auto-merging sample.c
CONFLICT (content): Merge conflict in sample.c
Automatic merge failed; fix conflicts and then commit the result.
root@LAPTOP-M3171AO3:~/C-Programming-Language-# ls
Function.c  sample  sample.c
root@LAPTOP-M3171AO3:~/C-Programming-Language-# vi sample.c
```

Difference between Git fetch and pull. The key difference between git fetch and pull is that git pull copies changes from a remote repository directly into your working directory, while git fetch does not. The git fetch command only copies changes into your local Git repo. The git pull command does both.

14. ls -la :

Here using this we can see the git initialization**.** But using ls just we see the list of files

```
root@LAPTOP-M3171AO3:~/C-Programming-Language-# ls -la
total 40
drwxr-xr-x  4 root root  4096 Mar 10 19:20 .
drwx------ 10 root root  4096 Mar 10 19:20 ..
drwxr-xr-x  8 root root  4096 Mar 10 19:19 .git
drwxr-xr-x  3 root root  4096 Mar 10 16:37 C-Programming-Language-
-rw-r--r--  1 root root   206 Mar 10 14:55 Function.c
-rwxr-xr-x  1 root root 15960 Mar 10 18:12 sample
-rw-r--r--  1 root root    68 Mar 10 18:11 sample.c
root@LAPTOP-M3171AO3:~/C-Programming-Language-#
```

15. git log :

Here we all the commits that are been performed in this repo.

```
root@LAPTOP-M3171AO3:~/C-Programming-Language-# git log

commit 84b53c5b6ce0f485ffc77d917c23dcdc4fd505f1 (HEAD -> main)

Author: ThallapalliLaxmi <laxmi.t@kodebloom.com>

Date:   Mon Mar 10 18:46:51 2025 +0000

    Save changes to sample.c

commit a2aae1b60eb144f6c934284d2f6bbfcaed541c1e

Author: ThallapalliLaxmi <laxmi.t@kodebloom.com>

Date:   Mon Mar 10 17:15:42 2025 +0000

    11-03-2025

commit 65793958bde37cc440dadaced8de046b86b01339

Author: ThallapalliLaxmi <laxmi.t@kodebloom.com>

Date:   Mon Mar 10 21:09:55 2025 +0530

    Function.c
```

16. git log –patch -1 Or git log –p -1 :

For more details we use this (git log with patch number-of-logs) command rather than git log command.

```
root@LAPTOP-M3171AO3:~/C-Programming-Language-# git log --p -1
fatal: unrecognized argument: --p
root@LAPTOP-M3171AO3:~/C-Programming-Language-# git log --patch -1
commit 84b53c5b6ce0f485ffc77d917c23dcdc4fd505f1 (HEAD -> main)
Author: ThallapalliLaxmi <laxmi.t@kodebloom.com>
Date:   Mon Mar 10 18:46:51 2025 +0000

    Save changes to sample.c

diff --git a/sample.c b/sample.c
index cc083b6..8850405 100644
--- a/sample.c
+++ b/sample.c
@@ -1,6 +1,6 @@
 #include<stdio.h>
 void main()
 {
- printf("WEL_COME TO GIT.....");
+ printf("WEL-COME TO GIT.....");
 }
```

17. git restore :

    It is used to undo the changes from the back staged stste to modified state.

    Use this command :

    $ git restore –staged Name-of-file

    Ex : $ git restore –staged Test.txt

```
root@LAPTOP-M3171AO3:~/C-Programming-Language-# git restore --staged
sample.c
root@LAPTOP-M3171AO3:~/C-Programming-Language-# git status
On branch main
Your branch and 'origin/main' have diverged,
and have 1 and 5 different commits each, respectively.
  (use "git pull" if you want to integrate the remote branch with yours)

All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
```

```
        modified:   sample.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        sample
root@LAPTOP-M3171AO3:~/C-Programming-Language-# git restore sample.c
root@LAPTOP-M3171AO3:~/C-Programming-Language-# clear
```

//There will be no changes after this.

Origin → It is the remote server.
Master → It is the branch server.
If we want to get changes back to github for that we use **git remote -v**
There are two Origin they are Fetch and Push.

TO push changes existing file to the github page we use **git push origin master/main.**
TO add new file we use command as **git add FILE-NAME** or **git add .** → (.) this includes all the files.