FUNCTIONS

Self contained block of 1 or more statements or a sub program which is designed for a particular task is known as function

ADVANTAGES

1) Module Approach :- By using functions we can develop the application in module format ie procedure oriented language concept.

2) Reusability :- By using functions we can create reusability blocks i.e
develop once and use Multiple times

3) By using functions, we can easily Debug the program.

4) Code Maintainence - When we are developing the application by using functions, then it is easy to maintain code for future enhancement.

- The basic purpose of function is Code reuse
- A 'C' prog. is a collection of functions.
- Always compilation process will take place from Top to Bottom.
- Execution process starts from main and ends with main only.
- When we are working with functions, functions can be defined randomly.
- From any func. we can invoke (call) any another function.
- When we are calling a function, which is defined later for avoiding the compilation error, we required to go for forward declaration

- Declaration of a function means required to specify return type, name of the function & parameter type information
- In Function Definition, first line is called function header / function prototype.
- Always function declaration must be required to match with function declaration.
- When the function is not returning any value then specify the return type as void.
- Void means nothing i.e no return type or value.
- Default returntype of func^n is an int.
- Default parameter type of func is void

- When the function is returning the value, we required to specify the return type is other than void i.e what type of data it is returning, same type of return statement is required to be specified.
- When the function is returning the value, specifying the return statement is optional.
  In this case compiler will give a warning message i.e function should return a value.
- When the function is returning the value, collecting the value is always optional
  In this case compiler will not give any warning message or error.

Syntax :-

```
return_type FunctionName(parameters)
{
  function_statement;
  —
  —
  return statement;
}
```

- Acc. to syntax, specifying the return type, parameters and return statements are optional.
- All the rules of variable declarations are applicable to function name also.

Functions are classified into 2 types-

1) Library functions/ Built-in functions/ Pre-defined funcⁿs.
2) User-defined functions

1) **Library Functions** :- Library functions are set of pre implemented functions which are available along with compiler.

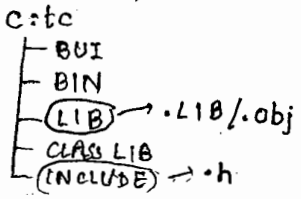
↔ The implementation part of library functions are available in .lib or .obj files which is available in C:\tc\LIB directory

↔ .Lib or .obj files contains pre-compiled object code

↔ When we are working with pre-defined functions for avoiding the compilation error we required to go for forward declaration i.e prototype is required

↔ When we required to provide of prototype of pre-defined functions, then required to go for header-files

↔ .h files does not provide any implementation part of predefined functions, it provides only prototype, i.e forward declaration of function.



When we install C software, some pre-defined functions are automatically come with that in C: directory.

Limitations

- All predefined functions contain Limited task only i.e for what purpose func^n is developed, for same purpose we required to use.
- As a programmer, we doesn't having control on predefined functions.
- As a programmer, it is not possible to alter or modify the behaviour of any pre-defined functions.

eg:- printf(), scanf(), clrscr(), getch()
strcpy(), pow(), sqrt()

- When pre-defined functions are not supporting user requirement then go for user-defined functions

2) **User-defined Functions** :-

→ As per client or project requirements, the functions we are developing are called user-defined functions.

→ Always user-defined functions are client specific functions ~~or~~ project specific functions only.

→ As a programmer, we are having full control on user-defined ~~defined~~ functions.

→ As a programmer, it is possible to alter or modify the behaviour of any user-defined functions if it is required because coding part is available.

→ Depends on return type and parameter type.

→ User-defined functions are classified into 4 types -

1) No return type with no parameters
2) No return type with parameter
3) With return type without parameters
4) With return type with parameter.

NOTE: All predefined functions are user-defined functions only bcoz somewhere else another programmer developed these functions & we are using in the form of object code or compile code.

- Whenever we are using any functions in compiled format then it is called pre-defined functions.

```
void printf()
 {
 }
void main()
  {
    printf("Hello");
  }
```

O/P:- [Blank]

It is compiled & executed, don't give any error but don't access pre-defined function access only user-defined function.

- It is possible to place user-defined function name as similar to pre-defined function name also.
- When both are same then it is not possible to call predefined function.

About main() :-

- Main is an identifier in the program which indicates startup point of an Application (Every function name is identifier)
- Main is a user-defined functions with pre-defined signature for Linker.
- A Linker is a Assembly Language Program which always decides startup point of the program is main.

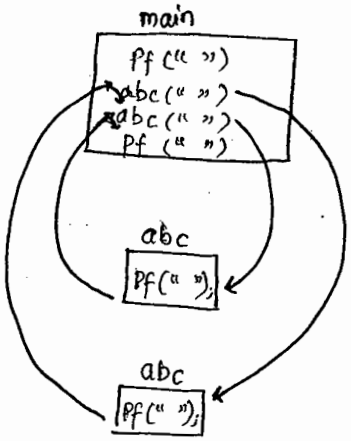
C	CPP	Java	C#
Void main()	int main()	p s v main()	void main()
{	{	{	{
}	}	}	}

- It is possible to change name of the main function if it is required but to execute the program, we required to place one more function with the name called main()
- It is possible to develop a program without using main function also. In this case Compilation is success but Linking is failure.
- In any kind of Application, only one ~~kind~~ of function is required to place with any 1 unique name i.e. multiple main functions are not possible.
- When we are developing .lib/.obj files for other projects then doesn't required to include main function (reusable components for multiple projects)
- Generally main function doesn't returns any value, that's why return type of main function is void.
- In implementation, when we required to provide exit status of an application then recommended to specify the return type as an int.
- Void main function doesn't provides any exit status back to the OS.
- int main function provides exit status back to the OS i.e success or failure.
- When we required to inform the exit status as success, then return value is 0, i.e return 0; or return EXIT_SUCCESS.
- When we required to inform the exit status as failure, then return value is 1, i.e return 1; or return EXIT_FAILURE.
- When the return type of main funcn is integer, then it is possible to return the values from the range of -32768 to 32767 but except 0 and 1, remaining values doesn't having any meaning.
- When the user is explicitly terminating the program, then return value is -1

Prog1 →
```
void abc()      //&abc
{
  printf("Hello abc\n");
}
void main()     //&main
{
  printf("Hello main1\n");
  abc();        //calling &abc();
  abc();        //calling &abc();
  printf("Hello main2\n");
}
```

O/P:-
```
Hello main1
Hello abc
Hello abc
Hello main2
```

Compilation starts from Top to Bottom while execution from main()



⇒ A 'C' program is a combination of pre-defined and User-defined functions Always compilation process starts from Top to Botton and execution process starts on main() and ends with main() only.

⇒ In order to compile a program, if any functions is occured then with that function name one unique identification value is created called address of func^n.

⇒ When we are calling any function, that calling statement is substituted with corresponding func^n address called **Binding process.**

⇒ With the help of Binding process only, compiler will recognize that which func^n required to call at the time of execution.

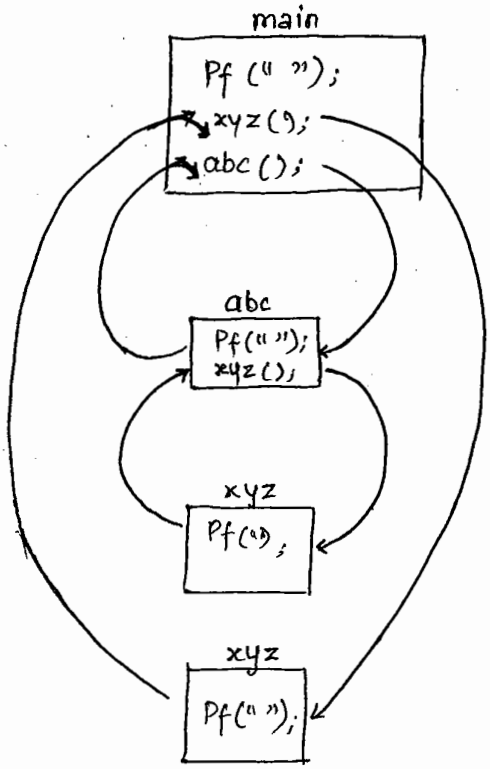
⇒ In 'C' prog. lang., always <u>static binding takes place</u> i.e Compile-Time Binding Process.

⇒ Dynamic Binding is a OOPL concept which works with the help of polymorphism.

Prog2 →
```
void xyz()      //&xyz
{
  printf("Welcome xyz\n");
}
void abc()      //&abc
{
  printf("Welcome abc\n");
  xyz();        //calling &xyz();
}
```

address is not given to printf because printf definition is not there all are user-defined functions

```
void main()        // & main
{
 printf (" Welcome main \n");
 xyz();            // calling  & xyz();
 abc();            // calling  & abc();
}
```



O/P:-
```
Welcome main
Welcome xyz
Welcome abc
Welcome xyz
```

- When we are working with functions, functions can be implemented randomly, i.e in any sequence functions can be defined.
- From any function we can call any other function
- After execution of any function, automatically control roill pass back to the calling place i.e from which location we called the func to same location it will pass.

Prog3 -
```
void main()
{
  printf (" Hello main \n");
  abc();
}
void abc()
{
  printf (" Hello abc \n");
}
```

O/P :- Error Type mismatch in redeclaration of 'abc'

Case 1:

```
Void abc()
{
}
Void main()
{
  abc();
}
```

Valid

1) Address
2) Type information
   - return type - Void
   - Parameters - void
   - No. of parameters - 0

Case 2:

```
Void main()
{
  abc();
}
Void abc()
{
}
```

Error

1) Address
2) Type information (default)
   - return type - int
   - Parameters - void
   - No. of Parameters - 0

On this, if we write int instead of void

Soln 1

```
Void main()
{
  abc();
}
int abc()
{
}
```

O/P :-

valid

- When we are working with functions, at the time of compilation along with the address, Type of information also maintain by compiler i.e return type, parameter type and no. of parameters.
- If function is compiled before calling, then along with the address, Type information also available but if func^n is calling b4 compilation then address is not available and automatically compiler take default type information i.e return type - int
  parameter type - void
  no. of parameters - 0
- In previous prog., at the time of compilation, we are getting Type mismatch error because return type is expecting an int but actual return Type is void.
- When we are calling a func^n which is defined later for avoiding the compilation error, we required to provide forward declaration i.e prototype of the function.

- Forward declaration means required to specify return type, function name and parameter type info
- Forward declaration statement always provides Type information explicitly so compiler doesn't take default Type info.

Soln2
```
void main()
{
  void abc(void);   // declaration
  printf("Hello main\n");
  abc();
}
void abc()
{
  printf("Hello abc\n");
}
```
O/P:-
```
Hello main
Hello abc.
```

Prog4 :-
```
void xyz();   //global decla
void abc()
{
  printf("Welcome abc\n");
  xyz();
}
void main()
{
  printf("Hello main\n");
  xyz();
  abc();
}
void xyz()
{
  printf("Welcome xyz\n");
}
```
O/P :-
Error Type Mismatch in redeclaration of 'xyz'

- In order to call the xyz function in abc(), main(), we required to provide forward declaration of xyz() in abc(), main() also.
- In implementation, when we required to provide forward declaration more than once then recommended to go for Global declaration
- When we are declaring a func at top of the prog. before defining first function then it is called Global declaration.
- When the Global declaration is available then doesn't required to go for local declaration. If local declaration also available then Global declaration is ignored.

Soln
```
void xyz();     // global declaration
void abc()
{
  //void xyz();  // local declaration
   printf("Welcome abc\n");
   xyz();
}
void main()
{
  //void xyz();  // local declaration
   printf("Hello main\n");
   xyz();
   abc();
}
void xyz()
{
   printf("Welcome xyz\n");
}
```

O/P:-
```
Hello main
Welcome xyz
Welcome abc
Welcome xyz
```

STORAGE CLASSES

Storage Classes of C will provide following information to the compiler i.e
1. storage area of a variable.
2. Scope of a variable i.e in which block that variable is visible.
3. Lifetime of a variable i.e how long that variable will be there in active mode.
4. Default value of a variable if it is not initialised

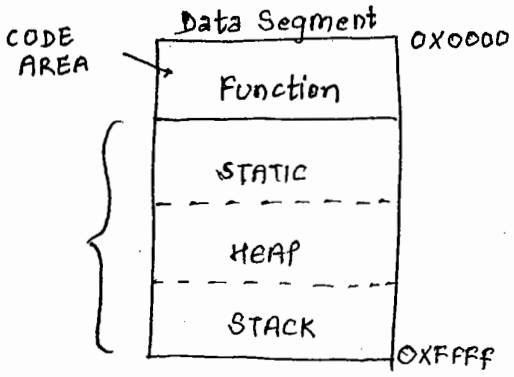
* depends on storage area and behaviour, storage classes are classified into 2 types-
  1. Automatic storage class -
  2. Static storage class

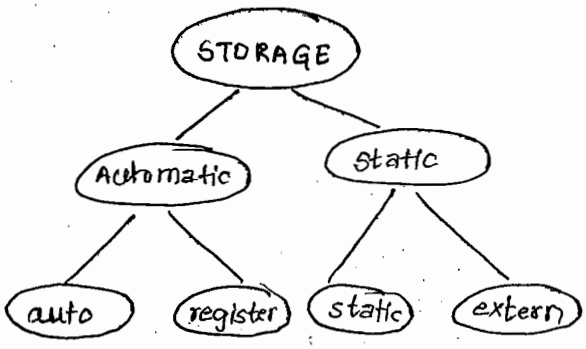
1) AUTOMATIC STORAGE CLASS

- Automatic Storage class Variables are created automatically and destroyed automatically.
- This storage class variables stored in stack area of Data Segment.
- Under automatic storage class, we are having 2 types of storage class specifiers auto, register

2) STATIC STORAGE CLASS

- This storage class variables are created only once and throughout the program it will be there in active mode.
- Static Storage Class variable are stored in static area of data segment.
- Under Static Storage Class, we are having 2 types of static storage specifiers i.e static, extern.





TYPE	SCOPE	LIFE	DEFAULT VALUE
auto	body	body	Garbage value
static	function	program	0
extern	Program	program (or) All functions	0
register	body	body	Garbage value

⇒ In 'C' prog lang. we are having 4 types of scope
- body scope
- function scope
- file scope
- program scope

⇒ By default any variable ~~is~~ storage class specifier is auto within the body.

REGISTER VARIABLES

- It is a special kind of variables which stores in CPU register
- The basic advantage of register variable is it is faster than normal variables

- In implementation, when we are accessing a variable throughout the program n no. of times then go for register variable

Limitations

⇒ Register m/m is limited so it is not possible to create n no of register variables.

⇒ On register variables we can't apply POINTERS because register variables doesn't allows to access address.

NOTE :- Register storage class specifier just recommends to the compiler that variable need to be stored in CPU Register if memory is available or else store in stack area of data segment.

```
void main()
{
  register int r = 10;
  ++r;
  printf ("\nr = %d", r);

  printf ("\nEnter a value: ");
  scanf ("%d", &r);
  ++r;
  printf ("\nr = %d", r);
}
// Enter a value: 100
```

O/P :- Error must take address of a memory location.

In Interviews, they give

```
scanf ("%d", r);
```

O/P :

```
r = 10
r = 12
```

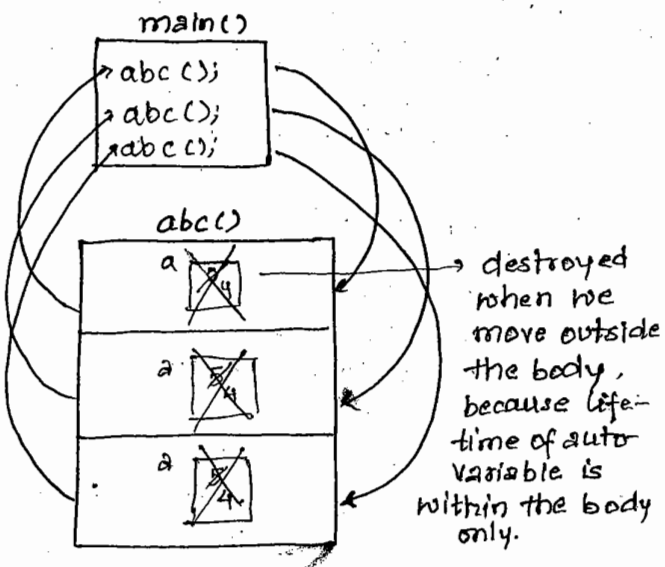
```
register int r = 10;
int *ptr;
ptr = r;          // ptr = &r;

*ptr = 100;
printf ("%d", r);
```

Error

WORKING WITH AUTO VARIABLES

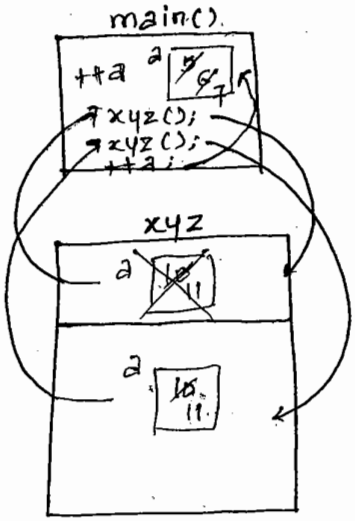
```
void abc()
 {
   int a=5;
    --a;
    printf("%d",a);
 }
void main()
 {
   abc();
   abc();
   abc();
 }
```

O/P :- 4 4 4



⇒ Acc. to storage classes of C, by default any type of variable storage class specifier is auto and lifetime of autovariable is restricted within the body that's why how many times we are calling the function that many times it is constructed.

```
void xyz()
 {
   auto int a = 10;
   ++a;
   printf("\na = %d", a);
 }
void main()
 {
   int a = 5;
   ++a;
   xyz();
   xyz();
   ++a;
   printf("\na = %d",a);
 }
```



O/P: a = 11
a = 11
a = 7

→
```
void abc()
{
  auto int a = 10;
  --a;
  printf("\na = %d", a);
}
void main()
{
  abc();
  abc();
  printf("\na = %d", a);
}
```

O/P:
```
Error
Undefined symbol 'a'
```

- Scope of the autovariable is restricted within the body only that's why abc() related data we can't access in main function.

→
```
void main()
{
  int a = 10;
  clrscr();
  int b = 20;
  ++a;
  ++b;
  printf("\na = %d  b = %d", a, b);
}
```

should be declared here.

O/P:
```
Error
declaration is not
allowed here.
```

- In 'C' prog. lang., variables required to declare on top of the program after opening the body before writing first statement

→
```
void main()
{
  auto int a = 5;
    int a = 10;
  ++a;
  printf("a = %d", a);
}
```

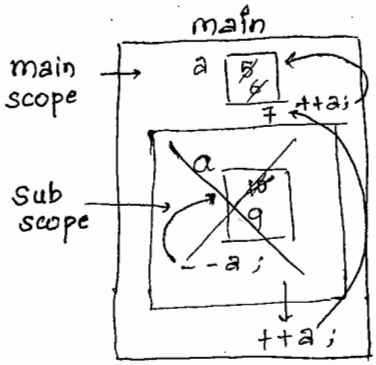
O/P: Error
Multiple declaration for 'a'

⇒ In any type of scope only one variable required to place with any one of unique name, Multiple variables. if we are creating with same name then it gives error

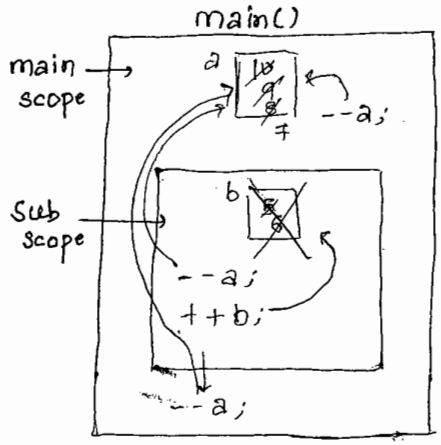
```
void main()
{
  auto int a = 5;
  ++a;
  printf ("\na = %d", a);
  {
    int a = 10;
    --a;
    printf ("\na = %d", a);
  }
  ++a;
  printf ("\na = %d", a);
}
```

O/P :

```
a = 6
a = 9
a = 7
```

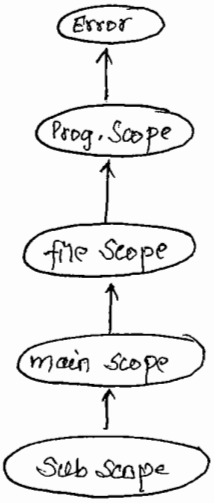



```
→ void main()
  {
    int a = 10;
    --a;
    printf ("\na = %d", a);
    {
      auto int b = 5;
      --a;
      ++b;
      printf ("\na = %d  b = %d", a, b);
    }
    --a;
    printf ("\na = %d", a);
  }
```



O/P :-

```
a = 9
a = 8    b = 6
a = 7
```



⇒ It is possible to access main scope variable directly in subscope, if subscope doesn't having same variable.

⇒ Acc. to K & R-C standard, previous prog. is error because it is not possible to create variables directly in sub-scope.

⇒ Acc. to NCC standard, it is possible to create variables anywhere within the prog. after opening the body.

→
```
void main()
{
  auto int a = 5;
  ++a;
  printf("\na = %d ", a);
{
  int a = 10;
  auto int b = 20;
  ++a;
  ++b;
  printf("\na = %d b=%d", a, b);
}
++a;
++b;
printf("\na = %d b=%d", a, b);
}
```

O/P: Error undefined symbol 'b'.

It is not possible to access subscope variable in mainscope, if main scope doesn't having same variable.

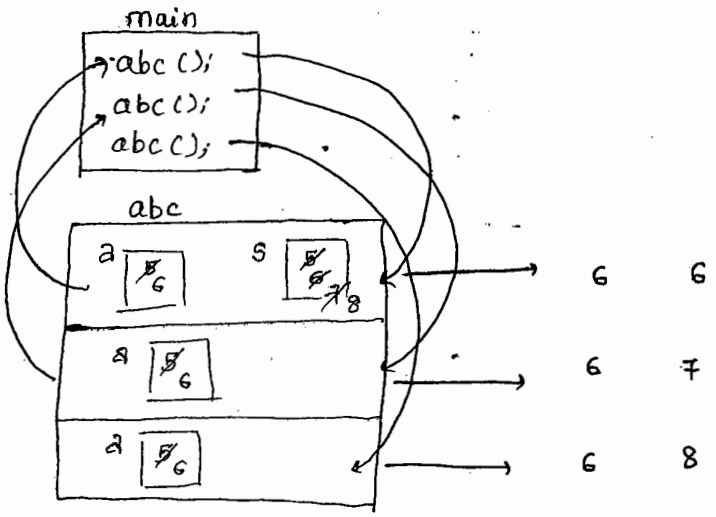
11/9/15

WORKING WITH STATIC VARIABLES

→
```
void abc()
{
  int a = 5;
  static int s = 5;
  ++a;
  ++s;
  printf("\n%d %d", a, s);
}
void main()
{
  abc();
  abc();
  abc();
}
```

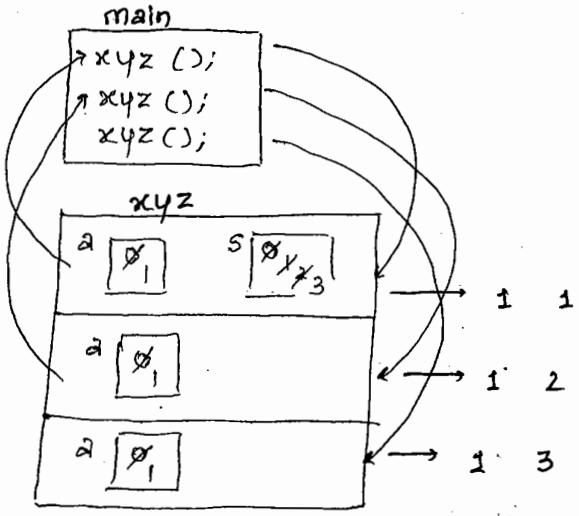
O/P :-

6	6
6	7
6	8



- When we are working with static variable, it is created only once when we are calling the function first time and throughout the program the variable will be there in active mode only

```
void xyz()
{
 auto int a=0;
  static int s;
  ++a;
  ++s;
  printf ("\n%d %d", a, s);
}
void main()
 {
   xyz();
   xyz();
   xyz();
 }
```



O/P:

1	1
1	2
1	3

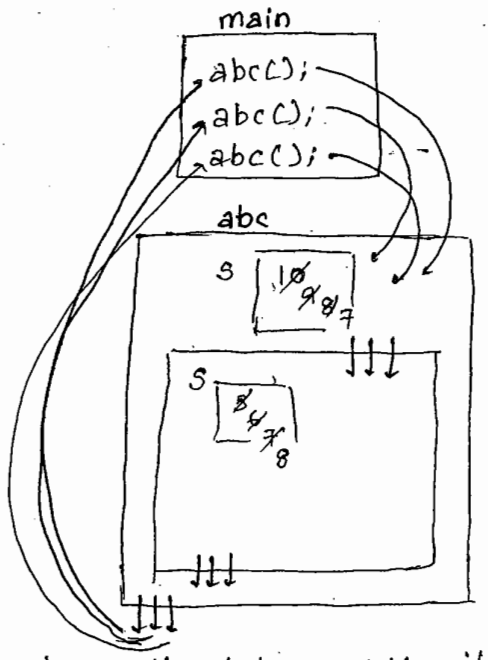
- When we are working with static variable, by default value of static variable is 0 if it is not initialized

→
```
void abc()
 {
  static int s= 10;
  --s;
  printf ("\nstatic 1: %d", s);
    {
     static int s = 5;
     ++s;
     printf ("\nstatic 2: %d", s);
    }
 }
```

O/P:-

static c1: 9
static c2: 6
static c1: 8
static c2: 7
static c1: 7
static c2: 8

```
void main()
{
 abc();
 abc();
 abc();
}
```



- static c1 : 9
  static c2 : 6

  static c1 : 8
  static c2 : 7

  static c1 : 7
  static c2 : 8

- When we are working with static variable, it can be created within function scope and subscope/body/local also.
- When we are creating the static variable within the subscope then it is accessible in subscope only, if we are creating in functionscope then throughout the function it can be accessed.

→
```
void abc()
{ static int s = 1947;
  ++s;
  printf("\ns = %d", s);
}
void main()
{ abc();
  abc();
  printf("\nstatic data : %d", s);
}
```

o/p: Error undefined symbol's!

⇒ When we are working with static variable, scope is restricted within the function only that's why abc function related data, we can't access in main function.

⇒ When we are working with auto variable, scope & lifetime both are restricted within the body only.

⇒ When we are working with static variable, scope is restricted within the function but lifetime is not restricted.

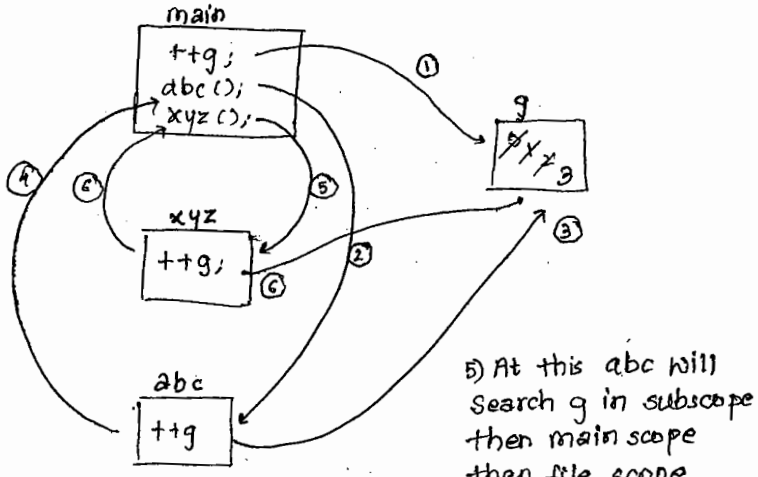
⇒ In implementation, when we required to access a data in more than function then go for extern variables, i.e Global variable is required.

⇒ When we are declaring a variable, outside the function, then it is called Global variable

⇒ When we are working with global variables then scope and lifetime is not restricted.

WORKING WITH GLOBAL VARIABLES :-

→
```
int g;  // global variable
void abc()
{
  ++g;
}
void xyz()
{
  ++g;
}
void main
{
  ++g;
  abc();
  xyz();
  printf("g = %d", g);
}
```



5) At this abc will search g in subscope then main scope then file scope

O/P : g = 3

⇒ By default, any type of variable storage Class specifier is auto within the body, extern outside the body

⇒ When we are working with global variable, it can be created anywhere within the prog. i.e top or middle or end of the prog. also

→
```
void abc()
{
  ++g;      → this creates error
}
int g;
void xyz()
{
  ++g;
}
void main()
{
  ++g;
  abc();
  xyz();
  printf("g=%d", g);
}
```

O/P :- Error Undefined symbol 'g'

During compilation process, from Top to Bottom [++g] treated as local variable but there is no local and global variable defined. So throws error.

⇒ When we are working with Global variable, it can be created anywhere within the program but if we are accessing before creation then we will get an error.

⇒ In previous program, due to Top-Bottom compilation process, when we are compiling ++g statement in abc() then we will get an error because memory is not created for global variable.

⇒ To avoid the compilation error of global variable, we required to provide forward declaration by using extern Keyword.

Soln -
```
void abc()
{ extern int g;   // declaration
  ++g;
}
int g;
Void xyz()        // defining
{ ++g;
}
void main()
{ ++g;
  abc();
  xyz();
  printf("g= %d", g);
}
```
O/P: g=3

NOTE: The basic difference between declaration and defination of Global variable is-

- In declaration of global variable, it doesn't occupies any physical memory, it avoids only compilation error.
- In defination of global variable, actual memory is constructed.

```
void abc()
{
  --g;
}
void xyz()
{
  --g;
}
  void main()
  { abc();
    xyz();
    --g;
    printf("g=%d", g);
  }
  int g = 10;
```
O/P: Error, Undefined symbol 'g'.

⇒ In order to access global variable g, in abc(), xyz() and main(), we required to provide forward declaration of global variable in abc(), xyz() and main() also.

⇒ In implementation, when we required to provide declaration of a global variable more than once, we required to go for Global declaration.

⇒ When we are declaring a global variable, Top of the program, before defining first function then it is called Global declaration if the global declaration is available then doesn't required to go for local declaration, if local declaration also available then global declaration is ignored.

Soln :-
```
extern int g;  // global declaration
void abc()
{
  // extern int g; local declaration
  --g;
}
void xyz()
{
  // extern int g; local declaration
  --g;
}
void main()
{
  // extern int g; local declaration
  abc();
  xyz();
  --g;
  printf("g = %d", g);
}
int g = 10;
```

O/P: g = 7

→
```
extern int g;
void abc()
{
  ++g;
}
void main()
{
  ++g;
  abc();
  printf("g = %d", g);
}
```

O/P: Error Undefined Symbol -g (Linking Error)

- When we are working with global variable, if declaration statement is not available, we will get Error at the time of Compilation.
  If definition is not available, then we will get the error at the time of Linking

→
```
extern int g = 10;
void abc()
{
  --g;
}
void main()
{
  --g;
  abc();
  printf("g=%d", g);
}
int g;
```

O/P: g = 8

- Initialisation of the global variable can be placed in declaration and definition also.

→
```
extern int g = 5;
void xyz()
{
  ++g;
}
void main()
{
  ++g;
  xyz();
  printf("g=%d", g);
}
```

O/P: g = 7

- When declaration statement contains initialisation, then physical memory is constructed in declaration only. So doesn't required to go for definition. if definition also available then it will ignore automatically.

→
```
extern int g = 5;
void abc()
{
  ++g;
}
void main()
{
  abc();
  ++g;
  printf("g=%d", g);
}
int g = 10;
```

O/P: Error Variable 'g' is initialised more than once

→ In 'C' programming lang. when local variable and Global variables names are same, then we can't access global varial in local variable In C++, in order to access the variable we required to use scope resolution operator i.e ::

```
extern int g = 5;
void abc()
{
  int g = 420;
   ++g;
  printf ("\ng = %d", g);
}
void main()
 {
   ++g
   abc();
   printf ("\ng = %d", g);
 }
```

O/P: 421
g=6

FORMAL ARGUMENTS, ACTUAL ARGUMENTS :-

- In function declarator or in function header, what variables we are creating those are called formal arguments, Parameters
- In function calling statement what data we are passing those are called

- In order to call a function if it is required specific no. of parameters then it is not possible to call the function with less than or more than no. of arguments.
- Where we are implementing the logic of the function it is called function definition.
- In function definition, 1st line is called function header / function declarator
- Where we are executing the logic of a function it is called function calling statement
- When we are providing type information explicitly then it is called function prototype or forward declaration.
- Always function declaration must be required to match with function declarator.

CALLING CONVERSIONS

→ As a programmer, it is our responsibility to indicate in which sequence parameters required to create.

→ Always Calling conversions will decides that in which sequence parameters created i.e left to Right or Right to left.

→ In 'C' prog. lang., calling conversions are classified into 2 types-

1) cdecl (_cdecl)
2) pascal (_pascal)

1) cdecl calling conversion - In this calling conversion, parameters are created towards from right to left.

- Bydefault any function calling conversion is cdecl.
- When we are working with cdecl calling conversion that recommed to place the function name in lower case.

2) pascal calling conversion :- In this calling conversion, parameters are created towards from left to right.

- In pascal calling conversion, recommeded to place ~~the calling conversion~~ the function name in UPPERCASE

- Generally in database pascal calling conversion is available like orade.

PARAMETER PASSING TECHNIQUES

In 'C' prog. lang. we are having 2 types of parameter passing Techniques-

1) Call by value
2) Call by address

1) Call by Value

- When we are calling a function by passing value type data then it is called Call by Value
- In call by value, actual arguments and formal arguments both are value type variables only.
- In call by value, if any modifications are occur on formal arguments then those modifications doesn't pass to actual arguments.

ex- printf(), pow(), sqrt(), textcolor()etc

2) Call by address

- When we are passing address type data to a function then it is called call by address.
- In call by address, actual arguments are address type and formal arguments are pointer type.
- In call by address, if any modifications occured on formal arguments then those changes will pass to actual arguments.

  eg:- scanf(), strcpy(), strupr() ... etc

NOTE: • C prog. lang. doesn't supports **call by reference**

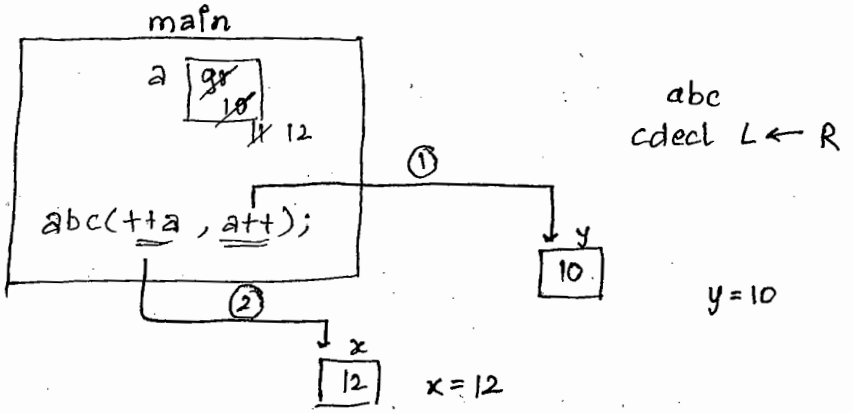
- Call by reference is a OOPL concept which is used to access the data by using reference type.

• C prog. lang. doesn't support reference type

- That's why call by reference is not possible.

```
void cdecl abc (int x, int y)
{
 printf ("\n x = %d  y = %d", x, y);
}
void main ()
{
  int a ;
  a = 10;
  abc (++a, a++);   - Here 2 arguments need to be passed because
  printf ("\n a = %d", a)        of 2 parameters.
}
```

O/P: x = 12 y = 10
a = 12

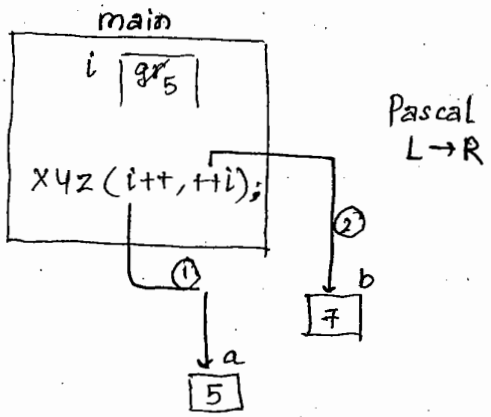


In this above prog.,

If cdecl is not then automatically by default it will take cdecl

→ void pascal XYZ (int a, int b)
```
{
  printf ("\n a = %d  b = %d ", a, b);
}
void main ()
{
  int i;
  i = 5;
  XYZ (i++, ++i);
  printf ("\n i = %d", i);
}
```

O/P:
```
a = 5  b = 7
i = 7
```



→ void swap()
```
{
  int t;
  t = a;
  a = b;
  b = t;
  printf ("\n a = %d  b = %d", a, b);
}
void main()
{
  int a, b;
  a = 10; b = 20;
  swap (a, b);
  printf ("\n a = %d  b = %d ", a+10, b+10);
}
```

O/P: Error undefined symbol 'a', 'b'

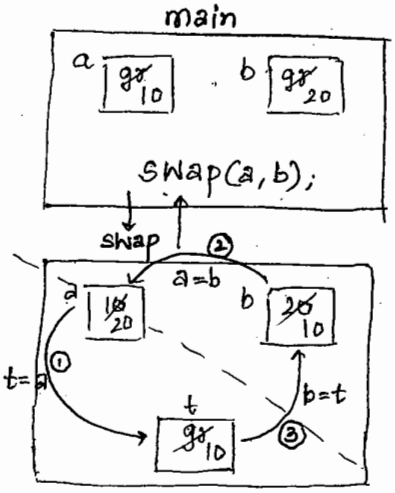
- In order to call the swap function it doesn't required any parameters but we are calling the function by sending 2 arguments
- Acc. to storage classes of 'C', a, b are autovariables, but we are toying to access in swap func^n ∴ Invalid

```
void swap (int a, int b)
{
  int t;
  t = a;
  a = b;
  t = b;
  b = t;
```

```
    printf("\na = %d  b = %d", a, b);
}
void main()
{
    int a, b;
    a = 10; b = 20;
    swap(a, b);
    printf("\na = %d  b = %d", a, b);
}
```

O/P:

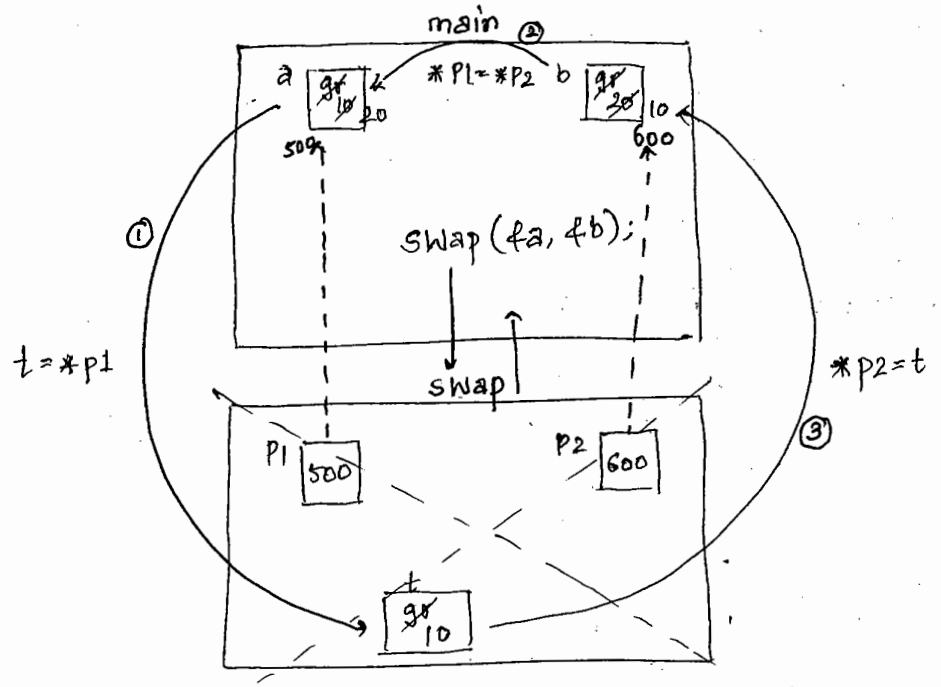
a = 20	b = 10	in swap
a = 10	b = 20	in main



- In previous prog., swap() is working with the help of call by value mechanism. That's why no any modification of swap() is passing back to main.
- In implementation, when we are expecting the modifications, then go for call by address.
- In 'c' prog. lang., call by address can be implemented by using POINTERS only

→ The basic advantage of pointer is accessing the data which is available outside of the function

→
```
void swap(int *p1, int *p2)
{
    int t;
    t = *p1;          // t = a
    *p1 = *p2;        // a = b
    *p2 = t;          // b = t
    printf("\nData in swap a = %d b = %d", *p1, *p2);
}
void main()
{
    int a, b;
    a = 10; b = 20
    swap(&a, &b);
    printf("\nData in main a = %d  b = %d", a, b);
}
```



- In previous program, array elements are passing by using call by address mechanism that's why all the modifications of swap() is passing back to main() function.
- The relation b/w formal arguments & parameter is always parameters are initialised with arguments.

14/7/2015.

```
int a=10, b=20;
a=a+b;
a=a-b
a=a-b;
```

O/P: a=20, b=10

a: 10 30 20 | b: 20 10

```
int a=10, b=20,
    c=30;
a=a+b+c;
b=a-(b+c);
c=a-(b+c);
a=a-(b+c);
```

a=30 b=10 c=20

O/P:

a: 10 60 30 | b: 20 10 | c: 30 20

```
int a=10, b=20,
    c=30, d=40;
a=a+b+c+d;
b=a-(b+c+d);
c=a-(b+c+d);
d=a-(b+c+d);
a=a-(b+c+d);
```

O/P:-

a: 10 100 40 | b: 20 10 | c: 30 20 | d: 40 30

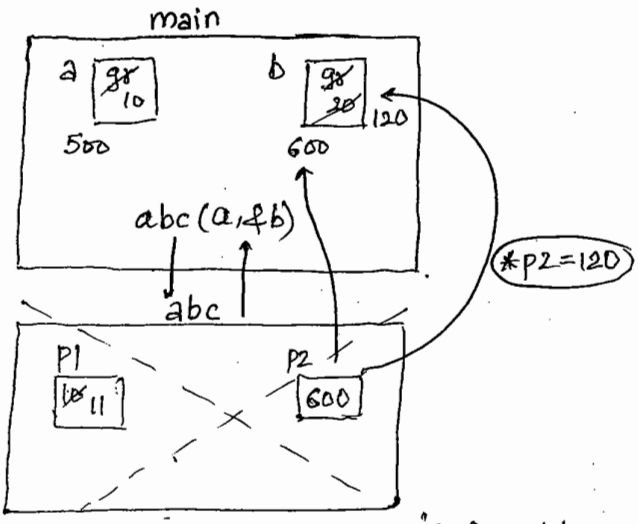
a=40 b=10 c=20
d=30

```
int a=10, b=20;
a=a^b;
b=a^b;
a=a^b
```

O/P:- a=20 b=10

→
```
void abc (int p1, int *p2)
{
 ++p1;
 *p2 = 120;
}
void main()
{
  int a, b;
  a = 10; b = 20;
  abc (a, &b)
  printf ("\na = %d  b = %d ", a, b);
}
```

O/P: a = 10 b = 120



→ In a single func^n, it is possible to pass value & address at a time.
⇒ When we are passing value type data then it is not updated but if we are passing address type data then it is updated.

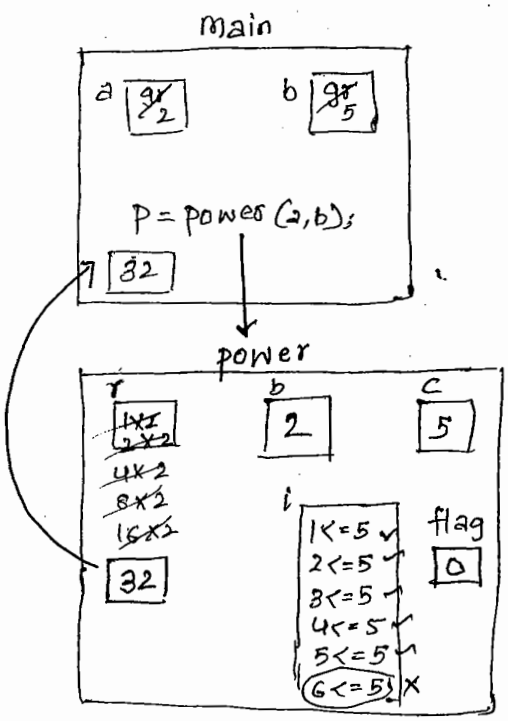
Return by value

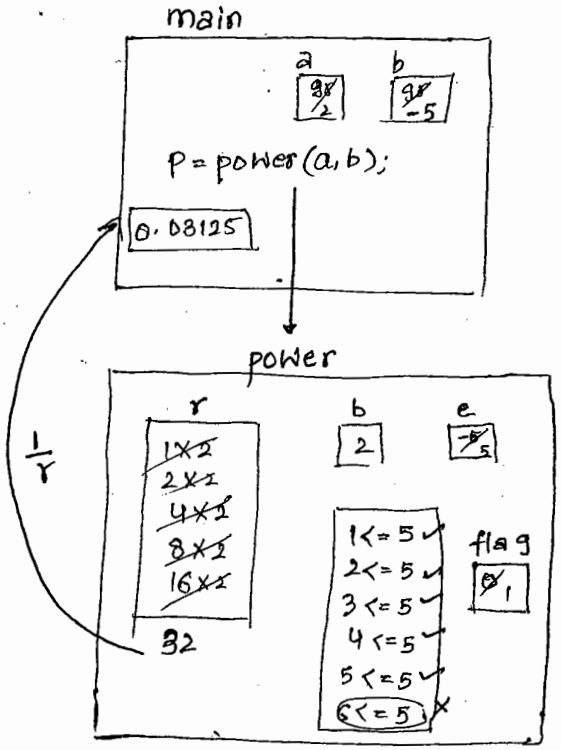
→ When the function is returning value type data then it is called return by value
→ When the function is not returning any values then specify the return type as void
→ Void means nothing i.e function doesn't return any value.
→ When the function is returning
i.e what type of data it is returning, same type of return statement required to specified
→ In implementation, when the function is returning an integer value, specify the return type as int, i.e function returning value type called return by value.

POWER PROGRAM -

```
* float power (int b, inte)
{
  int i, flag=0;
  float r=1;
  if (e<0)
  {
    flag=1;
    e=e*-1
  }
for (i=1; i<=e; i++)
  r=r*b;
  if (flag==0)
  return r;
else
    return (1/r);
}
```

```
void main()
{
  int a,b;
  float p;
  clrscr();
  printf ("Enter value of a:");
  scanf ("%d", &a);
  printf ("Enter value of b:");
  scanf ("%d", &b);
  p= power (a, b);      // p=pow(a, b);
                           <math.h>
  printf ("\n %d^%d value is %g",
                        a,b,p);
  getch();
}
```





→ return is a Keyword, by using return keyword we can pass the control back to the calling place with arguments or without arguments.

→ return is a exit control of a function which will stop the execution process of a function.

→ In a function it is possible to place any no. of return statements, but at any given point of time, only one return statement can be executed.

→ function can return only one value.

→ By using return statement, it is possible to return only one value.

→ From a function, when we required to return multiple values, then use call by address

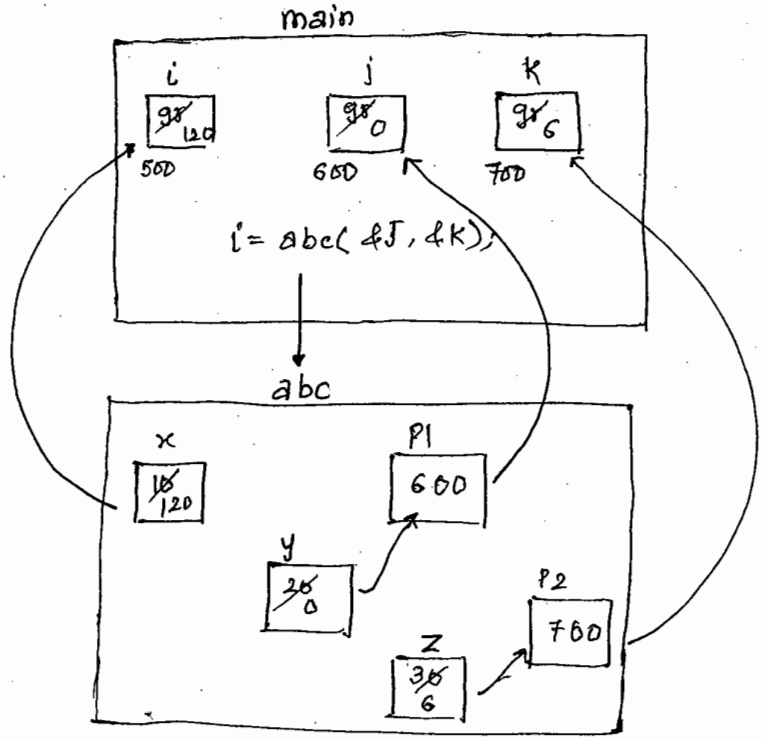
➤ By using call by address, it is not possible to return multiple values but it is possible to pass multiple values by using POINTER.

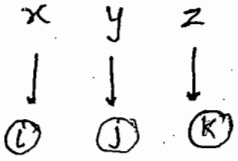
```
→ int max (int x, int y)
  {
    if (x>y)
     return x;
    else
      return y;
    }
  Void main()
   { int a, b, m;
     clrscr();
     printf ("Enter 2 values:");
     scanf ("%d %d", &a, &b);
     m = max (a, b);
     if (a!=b)
      printf (" Max value is : %d", m);
    else
      printf (" Both values are same);
     getch();
  }
```

O/P :- Enter 2 values : 10 20
Max value is : 20

```
int abc (int *p1, int*p2)
{ int x=10, y=20, z=30;
  x* = 10+2;  // x = x*(10+2);
  y% = 1+3;  // y = y%(1+3);
  z/ = 2+3;  // z = z/(2+3);
*p1 = y;  // j = y;
*p2 = z;  // K = z;
  return x;
}
```

```
Void main ()
  { int i, j, k;
    i = abc (&j, &K);
    printf (" i = %d  j = %d  K = %d", i, j, K);
  }
```

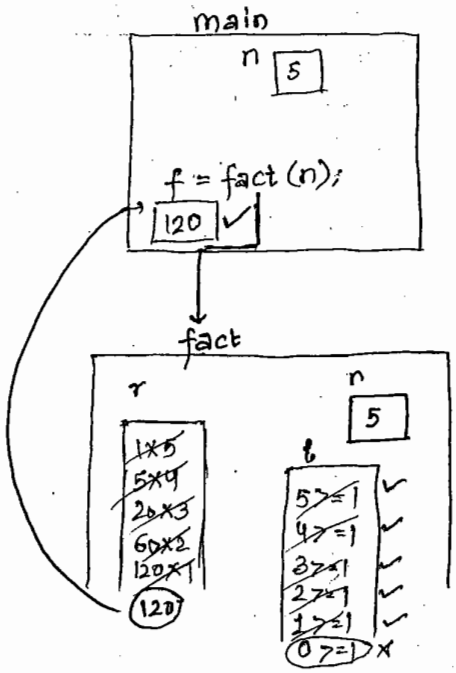




➤ By using POINTERS, we can collect any no. of values as per the requirement of a function

→ FACTORIAL

```
void main()
{
  int n,f;
  //int fact(int);
  clrscr();
  printf("Enter a value");
  scanf("%d",&n);
  f= fact(n);
  printf("%d fact value is : %d", n,f);
  getch();
}
int fact (int n)
  {
    int r=1, i;
    if (n<0)
     return 0;
    for (i=n; i>=1; i--)
     r = r*i;
     return r;
  }
```



- Acc to K and RC standard, when the return type is integer and parameter type is other than float then doesn't required to go for forward declaration
- As per ANSI standard when we are calling a function which is defined later for avoiding the compilation error, we required to go for forward declaration i.e prototype is required.

return by address.

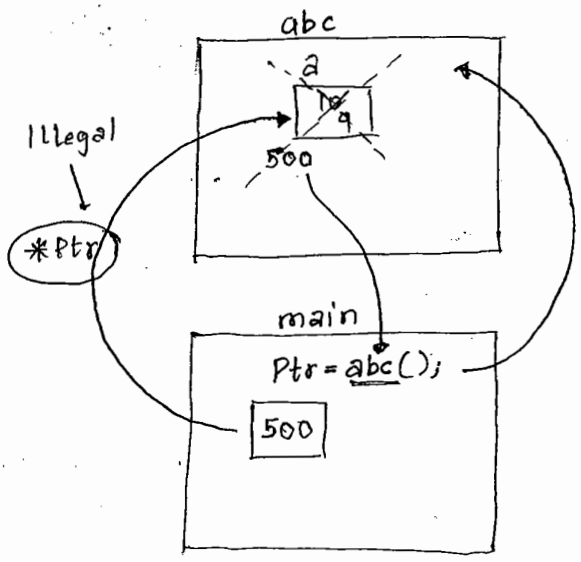
→ When the function is returning address type data then it is called return by address.
→ When the function is not returning any values then specify the return type as void
→ When the function is returning the int value then specify the return type as int i.e function returning value called return by value
→ When the func^n is returning an integer value address then specify the return type as an int * i.e function returning address called return by address.
→ The basic advantage of return by address is 1 func^n related local data can be accessed from outside of the function.

DANGLING POINTER

→ The pointer variable which is pointing to a inactive or dead location called Dangling pointer

```
→ int *abc()
  {
    int a = 10;
    --a;
    return &a;
  }
  void main()
    {
      int *ptr; //dangling pointer
      ptr = abc();
      printf("value of a = %d", *ptr);
    }
```

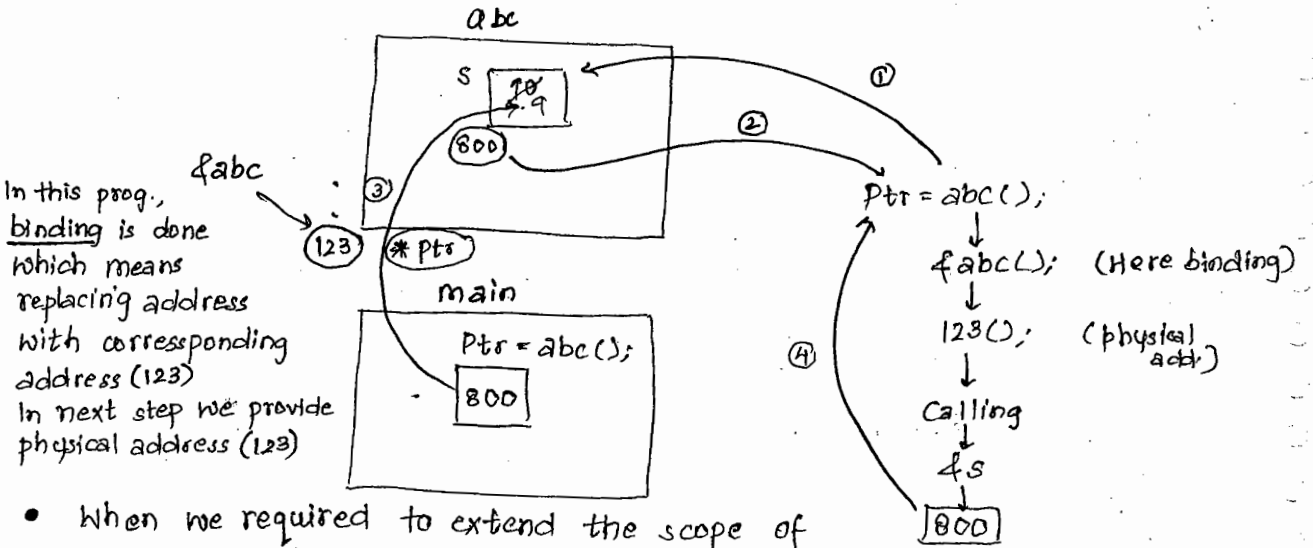
O/P: value of a = 9 (illegal)



- In previous program, ptr is called Dangling Pointer because acc. to storage classes of C by default any type of variable storage class specifier is void and lifetime of the auto variable is within the body only. But in previous prog control is passing back to main func^n it is destroyed but still pointer is pointing to that inactive variable only.

- Solution of dangling Pointer is In place of creating auto variable, recommended to create static variable because lifetime of the static variable is entire program.

Prog
```
int *abc() {
  static int s=10;
  --s;
  return &s;
 }
void main()
 {
  int *ptr;
  ptr = abc();
  Printf (" static data : %d", *ptr);
 }
```
O/P: static data : 9



- When we required to extend the scope of static variable then recommended to go for return by address

FUNCTION POINTER

- The pointer variable which holds the address of a function it is called function Pointer.
- The basic advantage of function pointer is 1 function ~~related~~ can be passed as a parameter to another function.
- Function pointer calls are faster than normal functions.
- Function pointers are similar to deligates in c#.

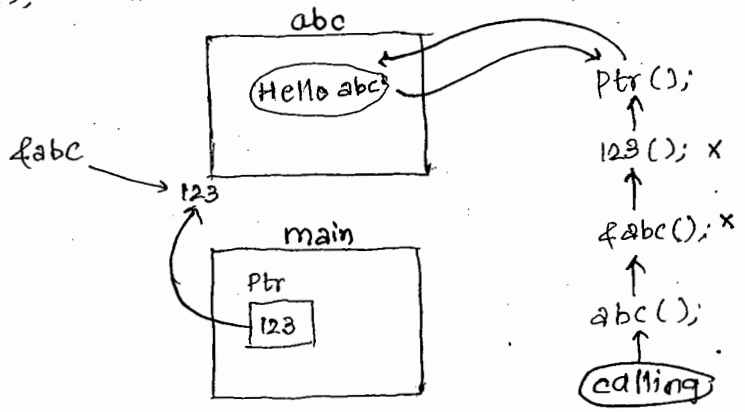
Syntax :- `Datatype (*ptr)(parameters);`

Ex :-

```
void (*ptr)(int);
int (*ptr)(int,int);
int *(*ptr)(int);
Datatype (*ptr)();
void (*ptr)();
int (*ptr)();
int*(*ptr)();
```

- If function takes parameters then go for Parameterized function Pointer.
- If function doesn't takes any parameters than go for non-parameterized function pointer.
- Acc to syntax, function pointer datatype must be required to match with return type of the function.

```
void abc()
{
 printf("Hello abc\n");
}
void main()
{
  void (*ptr)(void)      //Pointer Function
   ptr = &abc;
   ptr();      //abc();
}
```



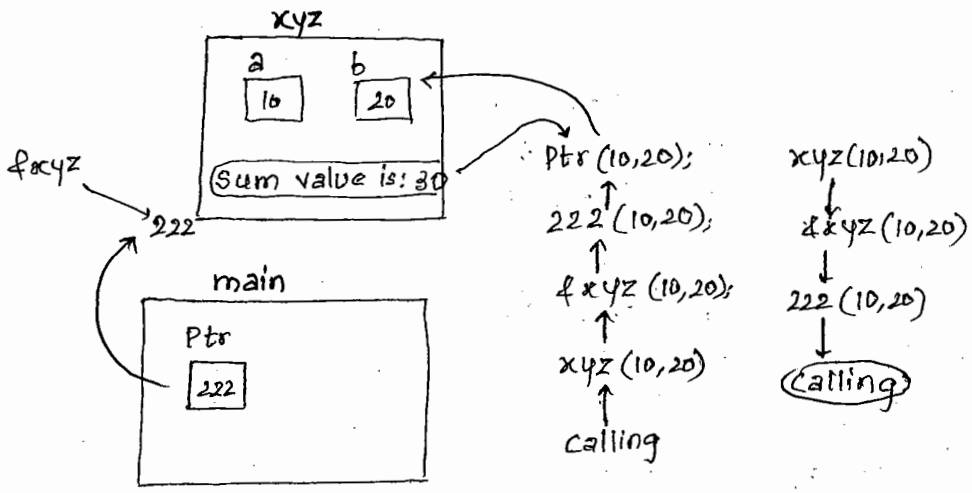
Here in function Pointer binding is not required because Ptr is already pointing to 123.

→ void xyz (int a, int b)

To create function pointer

```
{
 printf ("sum value is: %d", a+b);
}
void main()
{
 void (*ptr)(int, int);
  ptr = &xyz;
   ptr (10, 20);   // xyz (10, 20)
}
```

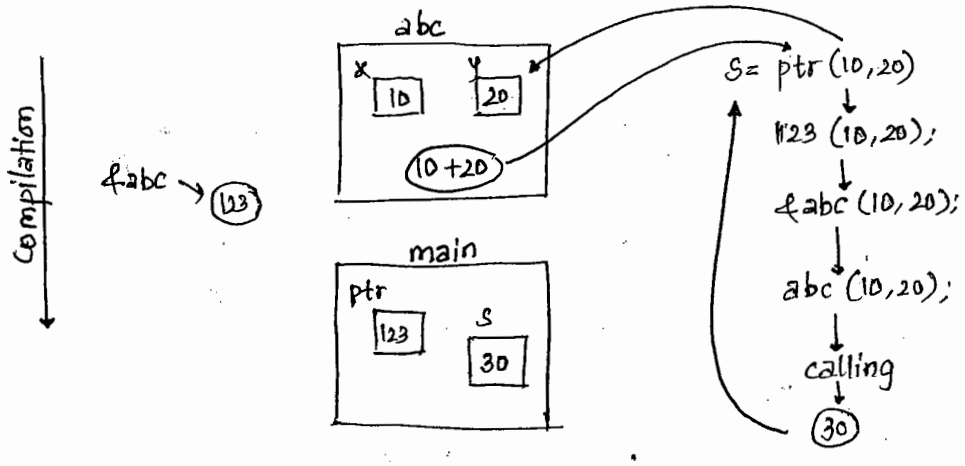
O/P : sum value is : 30



16th July 15

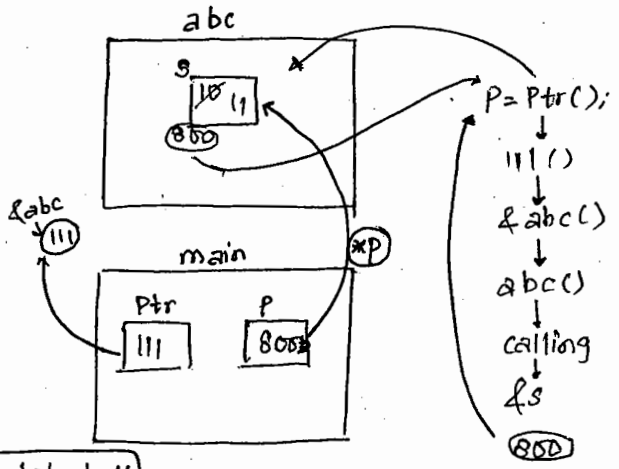
```
int abc (int x, int y)
{
 return (x+y);
}
void main()
{
 int (*ptr)(int, int);
   int s;
   ptr = &abc;
   s = ptr (10,20);    // s = abc(10,20);
   printf ("sum value is: %d", s);
}
```

O/P: Sum value is 30



→ int * abc()

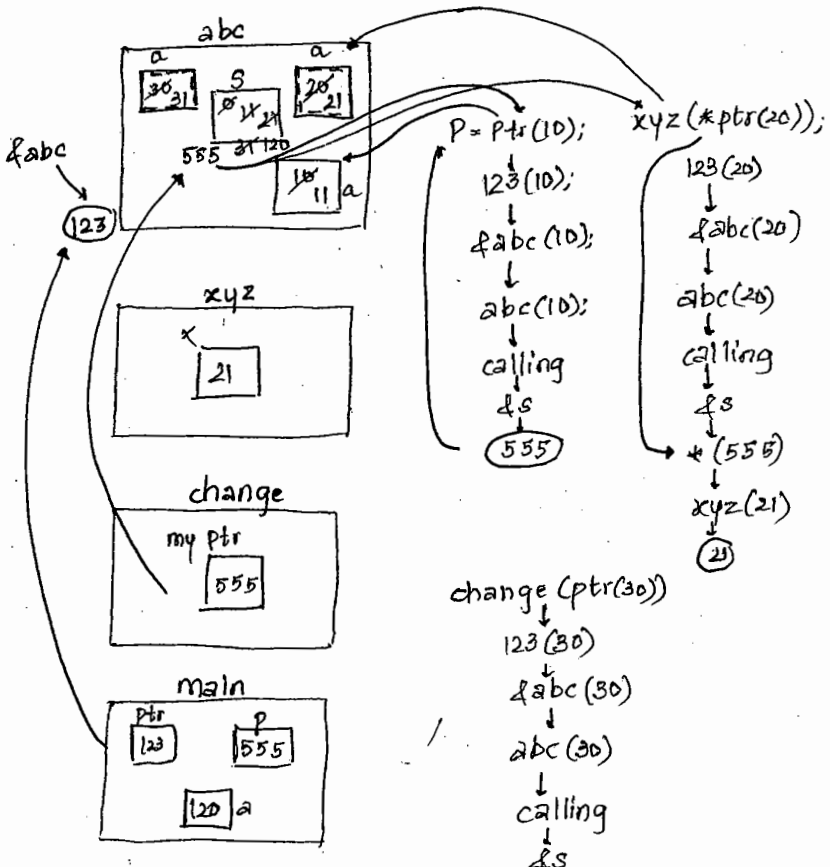
```
{
  static int s = 10;
   ++s;
   return &s;
}
void main()
{
  int *(*ptr)(); // pointer to fn
  int *p;      // pointer to integer
   ptr = &abc;
   p = ptr();
  printf("static data is : %d", *p);
}
```



O/P: static data is 11

→ int * abc(int a)

```
{
 static int s;
   s = ++a;
   return &s;
}
void xyz(int x)
{
  printf("\n static data
          in xyz : %d", x);
}
void change (int * myptr)
{
  *myptr = 120;
}
void main()
{
  int *(*ptr)(int);
  int *p;
  int a;
  ptr = &abc;
  p = ptr(10);
  xyz(*ptr(20));
  change (ptr(30));
  a = *p;
  printf("static data in
          main m/m : %d", a);
}
```



a = *p
= *555
a = 120

∴ a = 120

Recursion :- Function calling itself is called recursion.

→ The function in which control is present, if it calls itself again then it is called recursion process.

Advantages :-

1. By using recursion process only, function calling info. will be maintained in program.
2. By using recursion process stack evaluation takes places.
3. With the help of recursion only, infix, postfix, prefix notation are evaluated.
4. By using recursion process only trees and graphs are implemented.
5. By using recursion process we can develop tree traversing approach i.e inorder, preorder & postorder traversing process.
6. By using recursion only BFS, DFS are developed (Graphs traversing process)

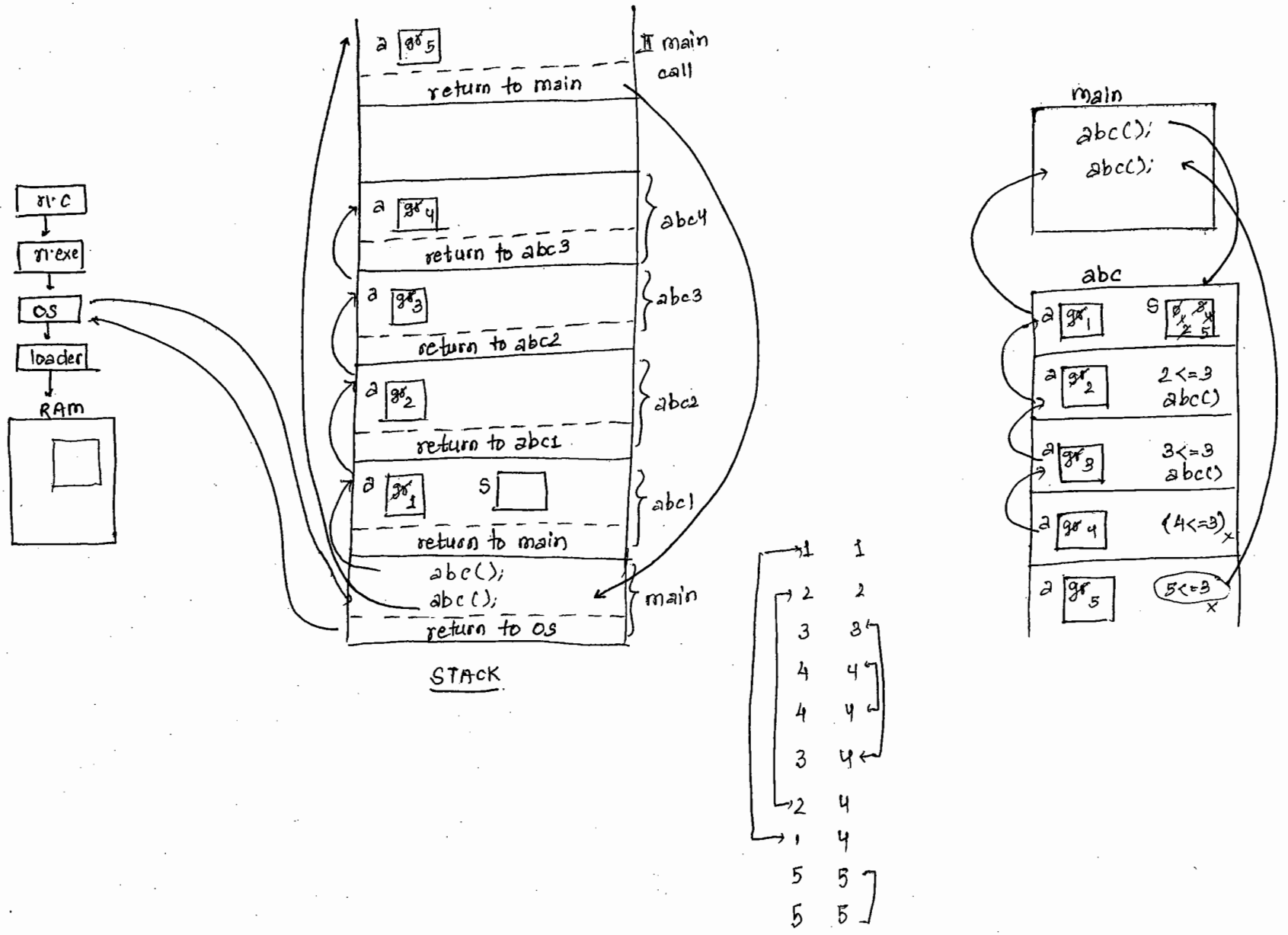
Drawbacks :-

1. It is a very slow process due to stack overlapping
2. Recursion based programs can create stack overflow
3. Recursion based functions can create ∞ loop.

⇒
```
void abc()
{
  int a;
  static int s;
  a = ++s;
  printf("\n%d %d", a, s);
  if (a<=3)
    abc();
  printf("\n %d %d", a, s);
}
void main()
{
  abc();
  abc();
}
```

O/P:

1	1
2	2
3	3
4	4
4	4
3	4
2	4
1	4
5	5
5	5


a.c
a.exe
OS
loader
RAM
a
return to main
II main call
a
return to abc3
abc4
a
return to abc2
abc3
a
return to abc1
abc2
a
S
return to main
abc1
abc();
abc();
return to OS
main
STACK
1 1
2 2
3 3
4 4
4 4
3 4
2 4
1 4
5 5
5 5
main
abc();
abc();
abc
a
S
2<=3
abc()
3<=3
abc()
(4<=3)
5<=3

→ void xyz()
```
{
 int a;
 Static int s = 5;
 a = s++;
 printf ("\n%d %d", a, s);
 if (a <= 7)
    xyz();
 printf ("\n %d %d", a, s);
}
Void main()
{
   xyz();
   xyz();
}
```

O/P:

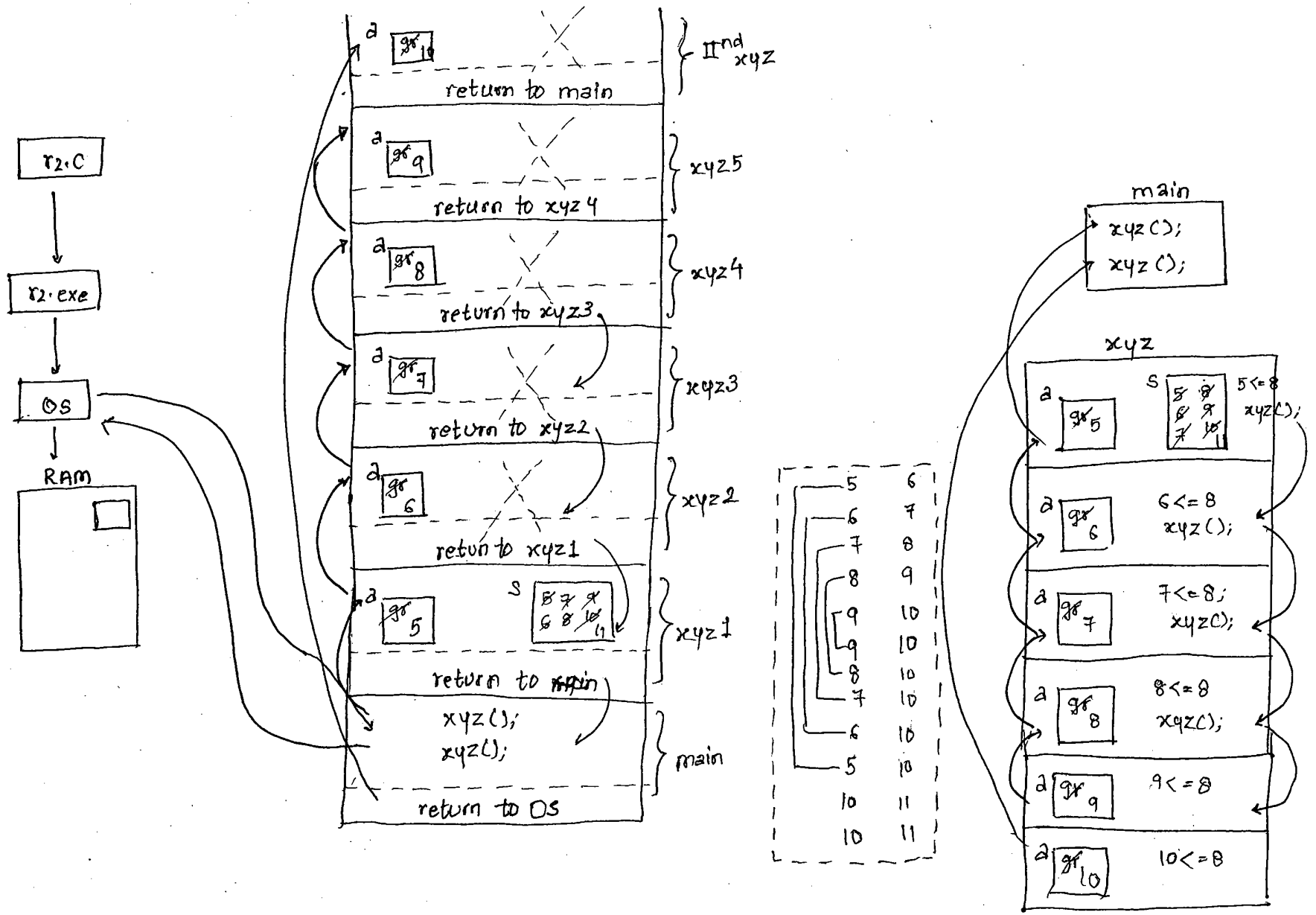
5	6
6	7
7	8
8	9
7	9
6	9
5	9
9	10
9	10

17|7|2015

→ void xyz()
```
{
 auto int a;
 static int s = 5;
 a = s++;
 printf ("\n %d", a, s);
 if (a <= 8)
 xyz();
 printf ("\n %d", a, s);
}
 void main()
 {
   xyz();
   xyz();
 }
```

O/P:

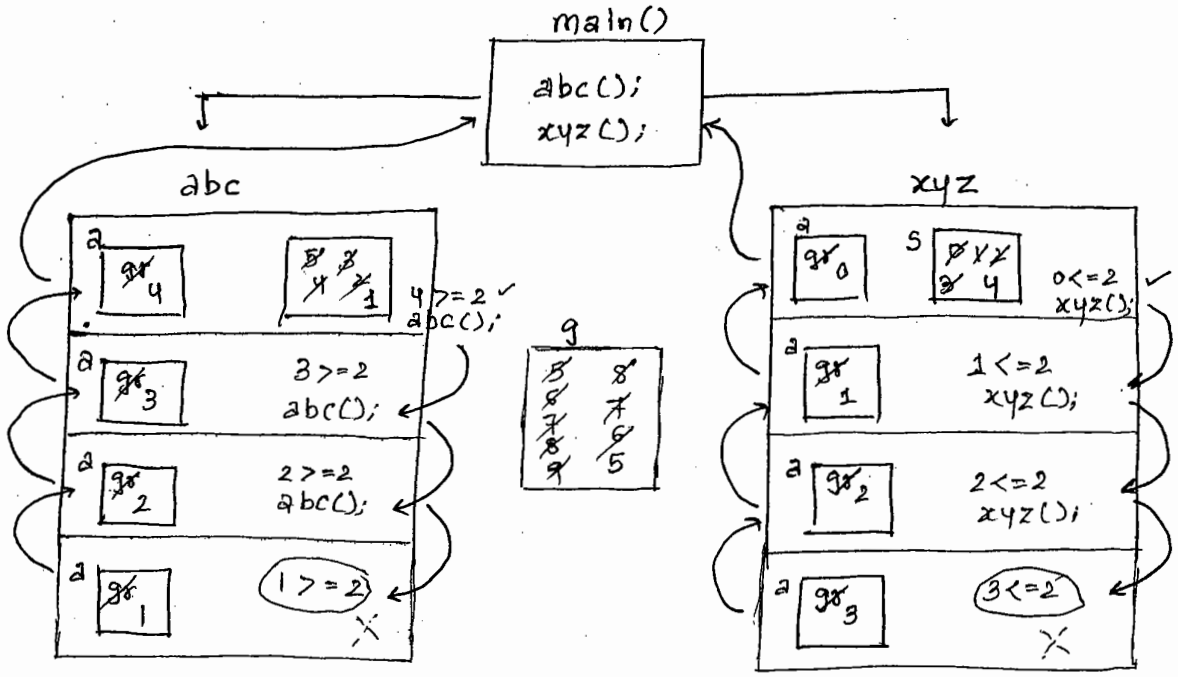
5	6
6	7
7	8
8	9
9	10
9	10
8	10
7	10
6	10
5	10
10	11
10	11


r2.c
r2.exe
OS
RAM
a
gr 10
return to main
II nd xyz
gr 9
return to xyz 4
xyz 5
gr 8
return to xyz3
xyz 4
gr 7
return to xyz2
xyz3
gr 6
return to xyz1
xyz 2
gr 5
S
return to main
xyz 1
xyz();
xyz();
main
return to OS
5 6
6 7
7 8
8 9
9 10
9 10
8 10
7 10
6 10
5 10
10 11
10 11
main
xyz();
xyz();
xyz
gr 5
5<=8
xyz();
gr 6
6<=8
xyz();
gr 7
7<=8;
xyz();
gr 8
8<=8
xyz();
gr 9
9<=8
gr 10
10<=8

```
extern int g;
void abc()
{
 int a;
 static int s = 5
 a = --s;
 ++g;
 printf("\n%d %d %d", a, s, g);
 if (a >= 2)
 abc();
 printf("\n%d %d %d", a, s, g);
}
void main()
{
  void xyz (void);
  abc();
  xyz();
}
void xyz()
{
  auto int a;
  static int s;
  a = s++;
  --g;
  printf("\n%d %d %d", a, s, g);
  if (a <= 2)
  xyz();
  printf("\n %d %d %d", a, s, g);
 }
 int g=5;
```

O/P:

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



- Recursions are classified into two types -
  1) Internal Recursive process
  2) External Recursive process

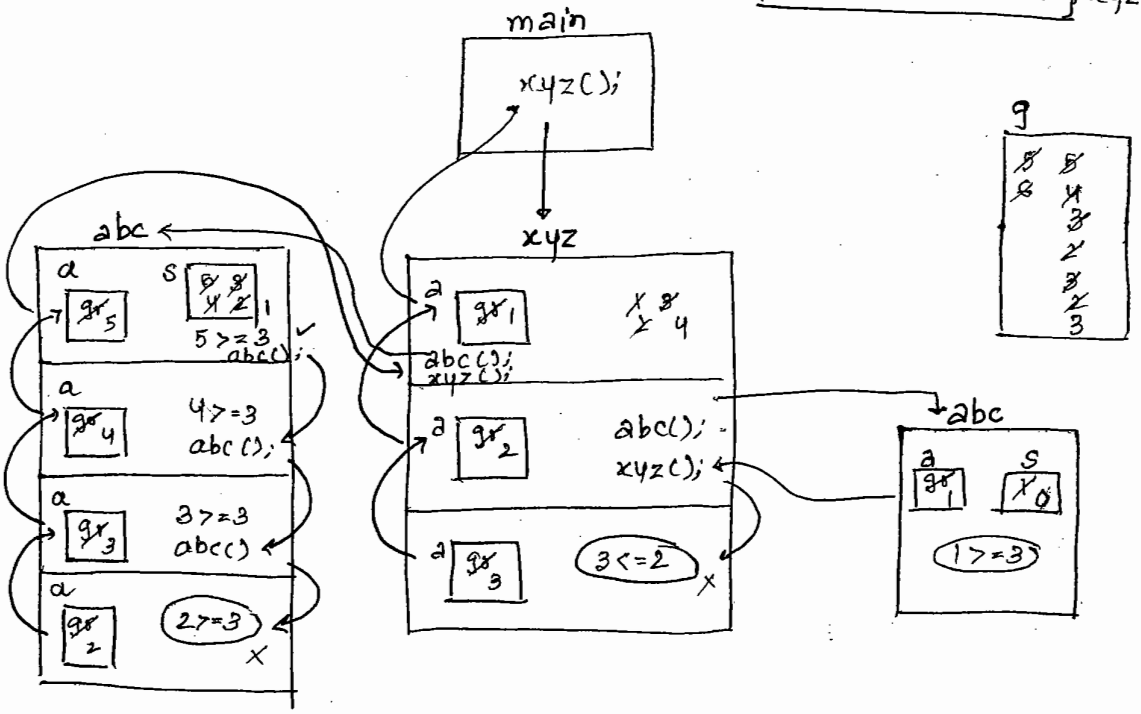
- When the function is calling itself then it is called Internal Recursive process. If Recursive function is calling another recursive function then it is called External Recursive process.

```
→ extern int g = 5;
  void abc()
  { auto int a;
    static int s = 5;
    a = s--;
    --g;
    printf("\n %d %d %d", a, s, g);
    if (a >= 3)
      abc();
    printf("\n %d %d %d", a, s, g);
  }
  void main()
  {
    void xyz(void);
    xyz();
  }
```

```
void xyz()
{
  int a;
  static int s = 1;
  a = s++;
  ++g;
  printf ("\n %d %d %d", a, s, g);
  if (a<=2)
  {
    abc();
    xyz();
  }
  printf ("\n%d %d %d", a, s, g);
}
```

O/P:

1	2	6	xyz1
5	4	5	abc1
4	3	4	abc2
3	2	3	abc3
2	1	2	abc4
2	1	2	abc4
3	1	2	abc3
4	1	2	abc2
5	1	2	abc1
2	3	3	xyz1
1	0	2	2nd abc
1	0	2	2nd abc
3	4	3	xyz3
3	4	3	xyz3
2	4	3	xyz2
1	4	3	xyz2



- It is possible to perform recursion of main function also
- By using autovariable if we are performing the recursion of main function then it became stack overflow but for every call, autovariable is reconstructed

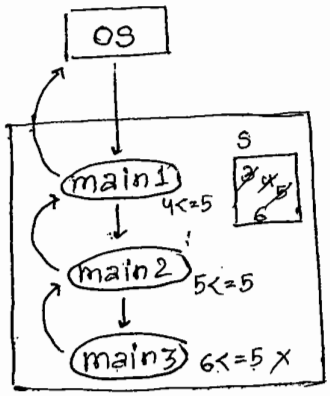
→ void main()
```
{
  int a = 5;
  --a;
  printf("%d", a);
  if (a >= 3)
  main();
  printf("%d", a);
}
```

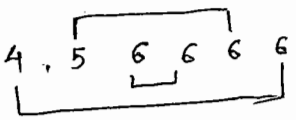
O/P: 4 4 4 4 - - - - stack overflow

∴ Loop will terminate automatically. Stack overflow because auto variable is there.

→ void main()
```
{ static int s = 3;
  ++s;
  printf ("%d", s);
  if (s<=5)
   main();
   printf ("%d", s);
}
```

O/P: 4 5 6 6 6 6





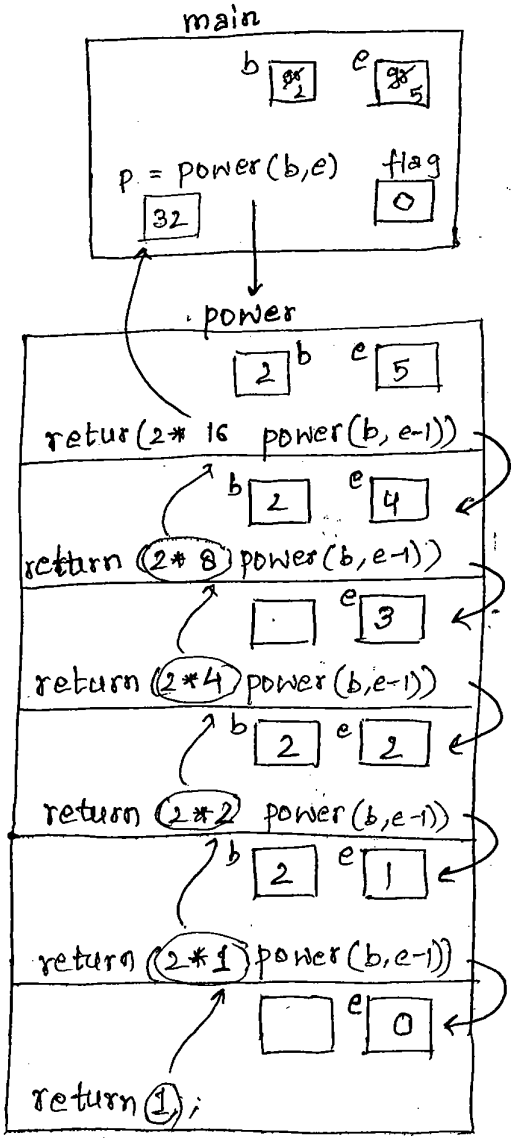
POWER PROGRAM (Recursion Approach)

```
float power (intb, inte)
{ if (e==0)
   return 1;
 else
   return (b * power (b, e-1));
}
void main()
{ int b, e, flag = 0;
  float p;
  clrscr();
  printf("Enter value of b: ");
```

```
scanf ("%d", &b);
printf ("Enter value of e : ");
scanf ("%d", &e);
if (e<0)
{
 flag = 1
 e = e*-1;
}
p = power (b,e);
 if (flag==0)
   printf ("\n %d ^ %d value is : %g", b, e, p);
 else
   printf ("\n %d ^ %d value is : %g", b, -e, 1/p);
 getch();
}
```

O/P:

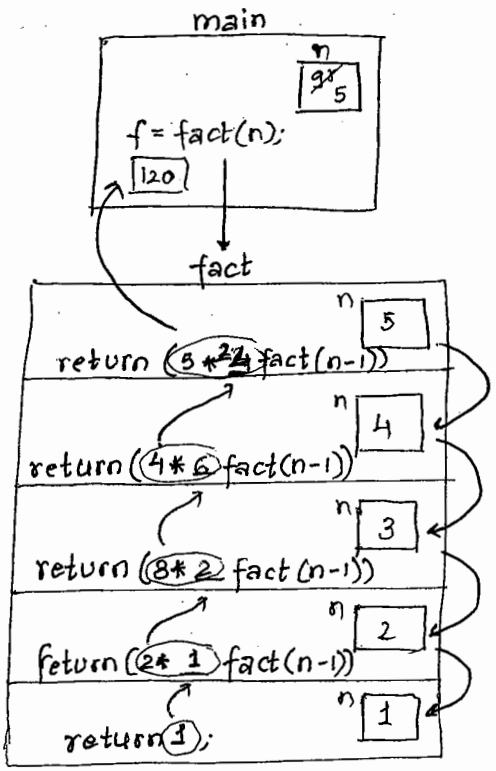
```
Enter value of b : 2
Enter value of e : 5
 32
```



FACTORIAL PROGRAM (Recursive Approach)

```
void main()
{ int n,f;
  int fact(int);
  clrscr();
  printf("Enter a value:");
  scanf("%d",&n);
  f=fact(n);
  printf("%d fact value is : %d", n,f);
  getch();
}
int fact(int n)
{
  if(n<0)
    return 0;
  else if(n<=1)
    return 1;
  else
    return (n*fact(n-1));
}
```

O/P:
Enter a value : 5
fact value : 120



- When the function is not returning any value then specify the return type as void
- Void means nothing i.e no return value

→
```
void abc()
{ printf("Hello abc\n");
}
void main()
{
  abc();
}
```

O/P: Hello abc

- If function return type is void then it is possible to place return statement also.
- From void function when we are placing empty return statement, then control will pass back to the calling place automatically without any values.

→ void abc()
```
{
  printf("Welcome abc\n");
  return;
}
void main()
{
  abc();
}
```
O/P: Welcome abc

- If the function return type is void, then it is possible to place return statement with value also
- from void function, when we are returning the value, then compiler will gives a warning message i.e void function may not return any values

---

→ void abc()
```
{
  int a = 10;
  ++a;
  printf("Hello abc : %d", ++a);
  return a;
}
void main()
{
  abc();
}
```
O/P: Hello abc : 12

- from void function, when we are trying to collect the value it gives an error i.e not an allowed type.

→ void abc()
```
{
  int a = 10;
  ++a;
  return a;
}
void main()
{
  int x;
  x = abc();
  printf("x = %d", x);
}
```
O/P: Error

→ void sum(int x, int y)
```
{
  printf("Sum value is : %d", x+y);
}
void main()
{
  Sum(10,20);
}
```
O/P :- Sum value is : 30

→
```
int sum (int x, int y);
{
  printf ("sum value is : %d", x+y);
}
void main ()
{
  sum (10,20);
}
```

O/P: sum value is : 30

→ When the function is returning the value then specifying the return statement is always optional.
In this case, compiler will give a warning message :- function should return a value.

→
```
int sum (int x, int y)
{
  printf ("sum value = %d", x+y);
  return x+y;
}
void main ()
{
  sum (10,20);
}
```

O/P: sum value = 30

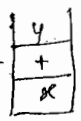
- When the function is returning the value then collecting the value is always optional.
In this case, compiler doesn't give any warning or error messages.

→
```
int sum (int x, int y)
{
  return (x+y);
}
void main ()
{
  int s;
  s = sum (10,20);
  printf ("sum value is : %d", s);
}
```

Sum value is : 30

- When we are returning simple format data, then it doesn't require to specify return statement within the parathesis, when we are returning expression format data, then it is recommended to specify the return statement within paranthesis.
bcz if paranthesis is not there then it inc. the burden on compiler

```
return x;
return (x);
```
} 100% same

```
return x+y;
```


6B

```
return (x+y);
```
↳ Sum → 2B

→
```
Float sum (int x, int y)
{
  return (x+y);
}
void main()
{
  int s;
  s = sum (12.8, 30);
  printf ("sum value is : %d", s);
}
```

O/P: sum value is : 42

int x, int y
12 30 12+30 = 42

- When the return type and parameter types are not matching at the time of execution then automatically type conversion will happen at the time of execution.

→
```
auto int g = 10;
void main ()
{
  ++g ;
  printf ("g= %d", g);
}
```

O/P: Error
Storage class 'auto' is not allowed here.

- By default any type of variable, storage class specifier is auto within the body, extern outside of the body

  In Global scope, it is not possible to place auto storage class specifier.

→
```
void abc (auto int a)
{
  ++a;
  printf ("a= %d", a);
}
void main ()
{
  abc (10);
}
```

O/P: Error
storage class 'auto' is not allowed here.

- For any type of parameters of a function, we can't specify storage class specifier because function m/m is temporary.
- for parameters, it is possible to specify register storage class specifier only.

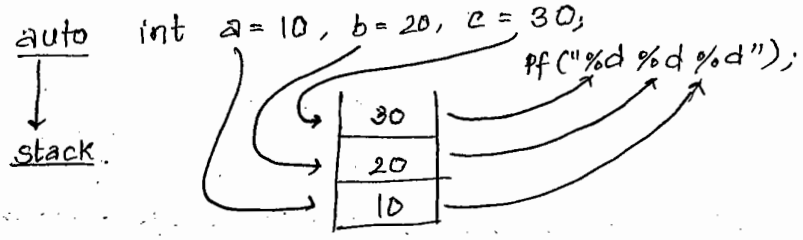
→
```
auto void abc
{
  printf ("Hello abc\n")
}
void main ()
{
  abc ();
}
```

O/P: Error
storage class 'auto' is not allowed here.

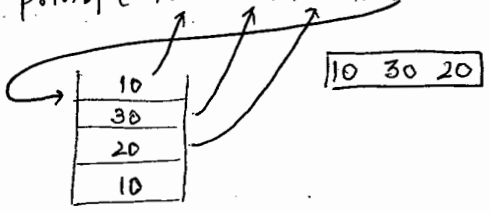
- By default scope of any function is extern i.e from anywhere in project we can call any function.
- By using auto storage class specifier, we can't restrict the scope of a function because it is self contained, independent block.
- for a function we can't apply auto, register storage class specifier.

→
```
void main()
{
  int a=10, b=20, c=30
  printf("%d %d %d");
}
```

O/P : 30 20 10

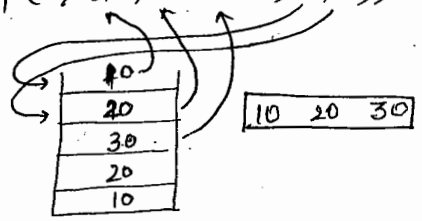


1. printf("%d %d %d", a);



10 30 20

2. printf("%d %d %d", a, b);



10 20 30

→
```
void main()
{
  auto static int a=10;
  ++a;
  printf("a=%d", a);
}
```

Error:
Too many storage classes in declaration.

- for any type of variable, only 1 storage class specifier is possible to be placed.

→
```
void abc()
{
  auto int a=10;
  static int s=a;
  ++a;
  ++s;
  printf("a=%d", a);
}
void main()
{
  abc();
}
```

O/P: Error Illegal initialization

- When we are working with static variables, always required to initialised with constant value only.

→
```
static int g = 5;
void abc()
{ static int s = 10;
++g;
++s;
printf ("\n s = %d  g = %d", s, g);
}
void main()
{ ++g;
abc();
}
```

O/P: s = 11 g = 7

- When we are declaring a static variable within the function then it is called internal static declaration.
- When we are declaring a static variable outside the function then it is called external static declaration.

➤ Basic difference b/w internal & external static declaration :
scope.
The scope of internal static declaration is restricted within the funcⁿ.
The scope of external static declaration is restricted within the file.

CASE 1:

project global variable

1.c
```
extern int g = 10;
void abc();
{
  ++g; ✓
}
void xyz();
{ ++g; ✓
}
```

main.c
```
void main()
{
  ++g; ✓
}
```

CASE 2:

file scope global variable

2.c
```
static int g = 10;
void abc()
{ ++g; ✓
}
void xyz()
{
  ++g; ✓
}
```

main.c
```
void main()
{
  ++g;
}
```
Error

- "C" programming lang. doesn't supports static functions
- Static function is a OOPL concept which is used to access static member of a class.
- In "C" programming lang., static functions are not possible but we can place static storage class specifier to a funcⁿ.
- When we are placing static storage class specifier to a function then scope of the function is limited to specific file only

CASE1:

default extern scope 1.c

```
Void abc()
{
}
Void xyz()
{
}
```

main.c

```
void main()
{
  abc(); ✓
  xyz(); ✓
}
```

CASE2:

1.c

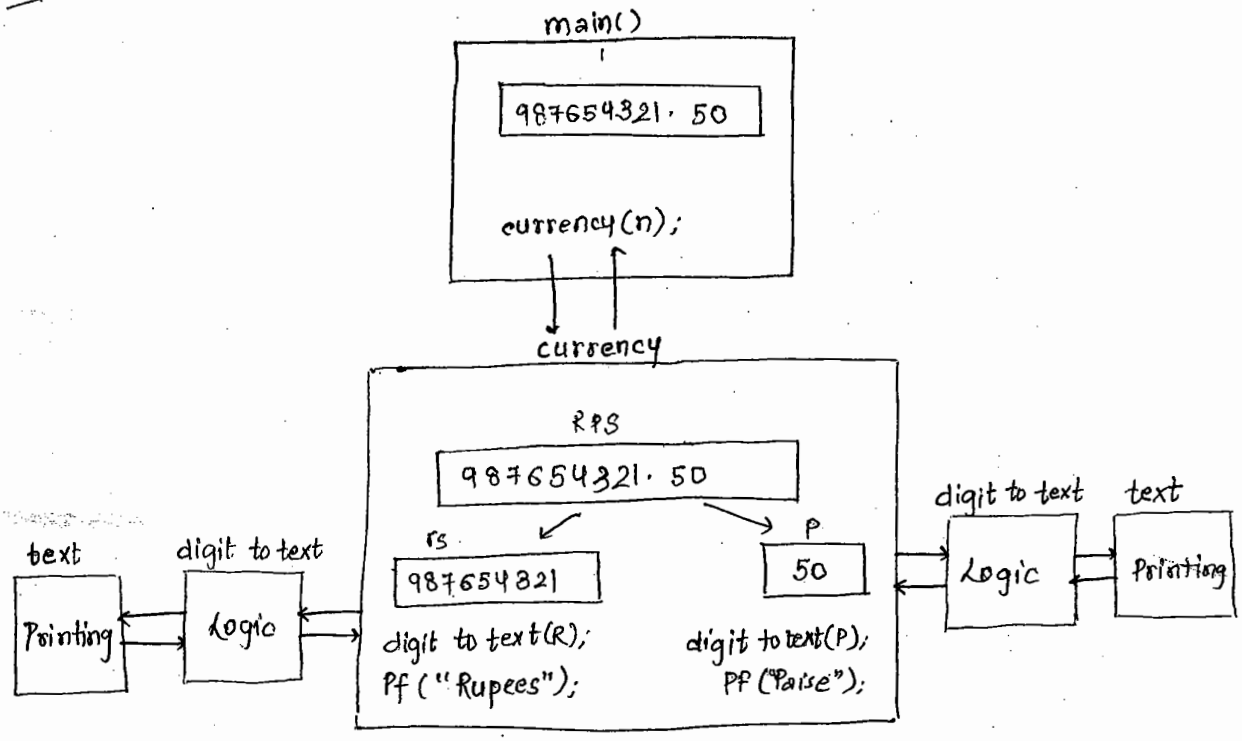
```
static void abc()
{
}
void xyz()
{
  abc(); ✓
}
```

main.c

```
void main()
{
  xyz(); ✓
  abc();
}
```

Error

21/7/2015.




```
Void text (long int no)
{
 switch (no)
   {
     case 1: printf ("ONE");
             break;
     case 2: printf ("TWO");
             break;
     case 3: printf ("THREE");
             break;
     case 4: printf ("FOUR");
             break;
     case 5: printf ("FIVE");
             break;
```

```
case 6 : printf ("SIX");
         break;
case 7 : printf ("SEVEN");
         break;
case 8 : printf ("EIGHT");
         break;
case 9 : printf ("NINE");
         break;
case 10 : printf ("TEN");
          break;
case 11 : printf ("ELEVEN");
          break;
case 12 : printf ("TWELVE");
          break;
case 13 : printf ("THIRTEEN");
          break;
case 15 : printf ("FOURTEEN");
          break;
case 16 : printf ("SIXTEEN");
          break;
case 17 : printf ("SEVENTEEN");
          break;
case 18 : printf ("EIGHTEEN");
          break;
case 19 : printf ("NINETEEN");
          break;
case 20 : printf ("TWENTY");
          break;
case 30 : printf (~~"FOURTEEN"~~); ("THIRTY");
          break;
case 40 : printf ("FORTY");
          break;
case 50 : printf ("FIFTY");
          break;
case 60 : printf ("SIXTY");
          break;
case 70 : printf ("SEVENTY");
          break;
case 80 : printf ("EIGHTY");
          break;
```

```
        case 90 : printf ("NINETY");
                  break;
        case 100: printf ("HUNDRED");
                  break;
        case 1000: printf ("THOUSAND");
                  break;
        case 100000: printf ("LAKH");
                  break;
        case 10000000 : printf ("CRORE");
                  break;
    }   case 32: // End of switch
} // End of text
Void digittotext ( long int n)
{
  long int t ;
    if (n == 0)
      printf ("ZERO");
    if (n >= 10000000)
     {
       t = n/10000000;
        if (t<= 20)
          text(t);
       else
         { text (t/10*10);
           text (t%10);
         }
        text (10000000);
         n = n % 10000000;
     }
     if (n>= 100000)
       {
          t = n/100000;
            if ( t<= 20)
                text (t);
     else
     {
        text ( t/10*10);
        text (t %10);
     }
      text (100000);
        n = n % 100000;
}
```

```
if (n >= 1000)
{
  t = n/1000;
   if (t <= 20)
     text(t);
  else
    {
     text (t/10* 10);
     text (t % 10);
    }
   text (1000);
    n = n % 1000;
 }
if (n >= 100)
 {
   t = n/100;
    text(t);
    text (100);
    n = n%100;
  }
  if (n <= 20)
    text (n);
  else
    {
     text (n/10 *10);
     text ( t %10);
    }
 }
 Void currency (double rps)
{
   long int r, p;
   r = (long int) rps;
   digittotxt (r);
   printf ("RUPEES");
   p (rps - r)* 100;
   digittotext (p);
   printf ("PAISE");
 }
 void main ()
 {
    double n ;
```

```
clrscr();
printf("Enter Amount:");
scanf("%lf",&n);
currency(n);
getch();
}
```

Calling & Declaration

```
int i1, i2;
int *ip1, *ip2;
int **ipp;
```

Calling

```
abc();
abc(i1, i2);
i1 = abc();
xyz(&i1, &i2);
abc(&i1, i2);
ip1 = abc(&i1, &i2);
swap(&i1, &ip1);
ipp = swap(&ip1, &ip2);
```

Declaration

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

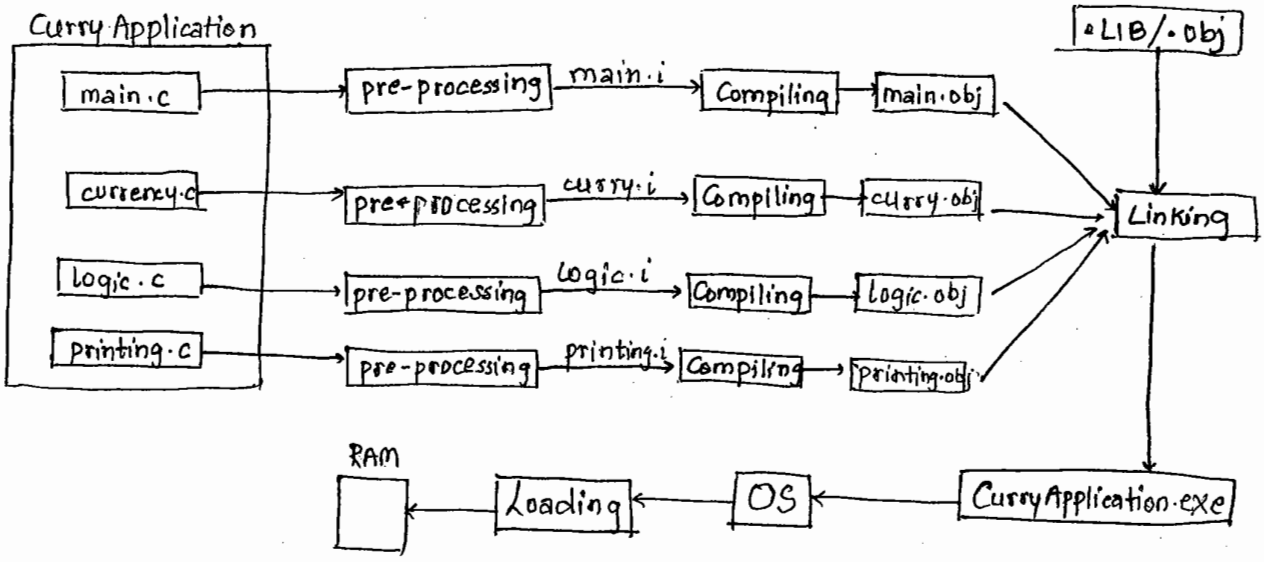
22/7/2015

PRE-PROCESSING

Pre-Processing

- Pre-Processing is a program which will be executed automatically b4 passing the source program to compiler.
- Pre-Processing is under control of Pre Processor directives.
- All pre-processor directives start with pound (#) symbol and should be not ended with semicolon (;)
- When we are working with Pre-Processor directives, it can be placed anywhere within the program but recommend to place on top of the prog. before defining first function
- In 'C' Programming lang. Preprocessor directives are classified into 4 types -

1. Macro substitution Directives
   Ex: # define
2. File Inclusion Directives
   Ex: # include
3. Conditional compilation Directives
   Ex: # if, # else, # endif, # elif
   # ifdef, # ifndef, # undef
4. Miscellaneous Directives
   Ex: # pragma, # error, # line



➤ When we are working with any kind of 'C' Application we are required to perform 4 steps

1) Editing
2) Compiling
3) Linking
4) Loading

1) Editing is a process of constructing the source program and saving with .c extension

→ To perform the editing process, we are required any kind of text editors like notepad, wordpad or any other C language related IDE.

2) Compiling is a process of converting high level programming lang. data in the machine readable data i.e object code or compiled code.

→ To perform compilation process, must be required, 'C' prog. language related IDE's only

3) Linking is a process of combining all obj files of current project along with standard Lib or obj files to construct an executable file

~~To construct an ex~~

→ When the linking process is successful then automatically executable file is generated with .exe extension

4) Loading is a process of carrying the application file from secondary storage area to primary memory.

→ Editing, Compiling and Linking is under control of IDE and loading is under control of OS

When we are working with any C application, it creates 5 types of files i.e

.c	.Bak	.exe
.I	.obj	

.c, .I, and .Bak contains user readable format data i.e source format

Generally .I file contains extended source code which is constructed after pre-processing.

.obj file contains compiled code which can be understandable to system only.

.exe file contains native code of OS.

1) MACRO SUBSTITUTION (#define)

- When we are working with # define at the time of pre-processing where an identifier is occurred, which is replaced with replacement text.
- Replacement text can be constructed using single or multiple tokens
- A Token can be Keyword, operator, seperator, constant or any other identifier

Syntax: `#define identifier replacement text`

- Acc. to syntax, atleast single space must be required between #define, identifier and identifier, replacement text.
- When we are working with #define, it can be placed anywhere in the program but recommended to place on top of the program before defining first function.
- By using #define, we can create symbolic constants which decreases the burden on programmer when we are working with array.

DESIGNING A C PROGRAM WITH DOS COMMANDS

→ For editing the program, we required to use edit command edit is a internal command which is available along with OS.

Syntax: `edit filename.c`

Ex: D:\C1100AM>edit p1.C

`code in p1.c`

```
# define A 15
 void main()
  {
    int x;
    x=A;
    printf("%d %d", x, A);
  }
// save p1.c (File ⟶ save)
// close p1.c (File ⟶ exit)
```

→ To perform the preprocessing we required to use cpp command. cpp is an external command which is available in C:\TC\BIN directory

Syntax: `CPP filename.c`

D:\C1100AM>CPP P1.C

NOTE: Preprocessing is a automated program which will be executed automatically before passing the source code to compiler.

→ If we required to create .i file explicitly then mandatory to perform.

code in p1.i

```
p1.c 1 :
p1.c 2 : void main()
p1.c 3 : {
p1.c 4 : int x;
p1.c 5 : x=15;
p1.c 6 : printf("%d %d", x,15);
p1.c 7 : }
p1.c 8 :
```

→ As per above observation, at the time of preprocessing where an identifier A is occurred, it is replaced with replacement text.

→ No any preprocessor related directives can be understandable to compiler, that's why all preprocessor related directives are removed from source code.

- .i file is called extended source code which is having actual source code which is passing to compiler.
- For compilation & linking process, we are required to use TCC command.
- TCC is a external command which is available in C:\tc\Bin directory.

Syntax: `TCC filename.c`

ex:- D:\C1100AM>TCC P1.C

- When we are working with TCC command compilation & linking both will be performed at a time.
- If compilation is success then we will get obj file, if linking is success then we will get .exe file.
- For loading or execution of program, we required to use application name or program name.exe

Syntax:

ex: D:\C1100AM>p1.exe
O/P: 15 15

ex: D:\C1100AM>P1
O/P: 15 15

```
# define size 120
  void main ()
{
  int x;
  x = ++size;
  printf ("x = %d", x);
}
```

O/P : Error L value req

- By using # define we can create symbolic constant value which is not possible to change at the time of execution.

```
# define A 2+3
# define B 4+5
 void main()
{ int c;
  C = A * B;
  printf ("c = %d", c);
}
```

O/P: C = 19

C = A * B
= 2+3 * 4+5
= 2 + 12 + 5
= 19

C = (A) * (B),
= (2+3) * (4+5);
= 5 * 9
= 45

- When we are creating the replacement text with multiple tokens then recommended to place replacement text in Parameters (c)

```
# define A (2+3)
# define B (4+5)
```

C = A * B
= (2+3) * (4+5)
= 5 * 9
C = 45

2) MACRO :-

- Simplified function is called macro
- When the function body contains 1 or 2 statements then it is called simplified function.
- When the simplified functions are occurred always recommended to go for macro

Advantages :-

- Macros are faster than normal functions.
- No any physical memory will be occupied when we are working with macros

- When we are working with macros, code substitution will happen in place of bouncing process

Drawbacks :-

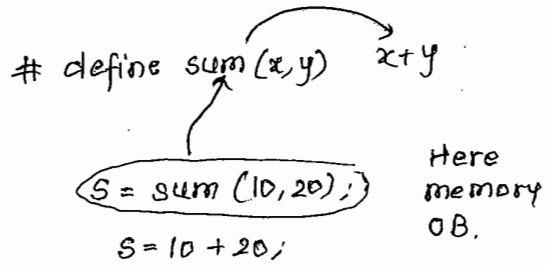
- No any syntactical problems can be considered at the time of pre-processing.
- Macros required to construct in single line only.
- No any type checking process is occurred, when we are working with macros (parameter checking process)
- No any control flow statements are allowed.

Prog1
```
int sum(int x, int y)
{
  return (x+y);
}
void main()
{
  int s;
  s = sum(10,20);
  printf("sum value is %d", s);
}
```
sum value is : 30

1) Parameters → 4B
2) Arguments → 4B
3) return value → 2B
4) func address → 2B/4B
   12B/14B

By using macros

```
# define sum(x, y) x+y
void main()
{
  int s;
  s = sum(10,20);
  printf("sum value is %d", s)
}
```



In previous program at the time of preprocessing when we are calling sum macro, automatically it is replaced with replacement text

Prog2.
```
int max(int x, int y)
{ if (x>y)
    return x;
  else return y;
}
void main()
{ int m;
  m = max(10,20);
  printf("Max value is : %d", m);
}
```

Using macro

```
# define max(x,y) x>y ? x:y

void main() {
 int m ;
  m = max (10,20);
  printf (" Max value is : %d", m);
}
```

O/P: Max value is : 20

Prog3

```
# define SQR(a) a*a
void main()
 {
  int i, j;
  i = SQR(2);
  j = SQR(2+3);
  printf (" i = %d  j = %d ", i, j);
 }
```

O/P: i = 4 j = 11

i = SQR (2);
= a * a;
= 2 * 2;
= 4

j = SQR(2+3);
= a * a;
= 2+3 * 2+3;
= 2+6+3;
= 11

```
# define SQR(a) (a)*(a)
```

i = SQR(2);
= (a)* (a);
= (2)*(2);
= 4

j = SQR(2+3);
= (a)* (a);
= (2+3)*(2+3);
= 5 * 5 ;
= 25 ;

Prog4 :

```
# define CUBE (a) (a)*(a)*(a)
void main()
 {
  int i, j ;
  i = CUBE (2);
  j = CUBE (2+3);
  printf (" i = %d  j = %d", i, j);
 }
```

O/P: i = 8 · j = 125

i = CUBE(a);
= (2)*(2)*(2)
= 8

j = CUBE (2+3)
= (a)*(a)*(a);
= (2+3)*(2+3)*(2+3)
= (5) * (5) * (5)
= 125

Prog5 : NESTED MACRO

```
# define SQR(a) (a)*(a)
# define CUBE(a) SQR(a)*(a)
void main()
 {
  int i ;
  i = CUBE (2+3);
  printf ("i = %d", i);
 }
```

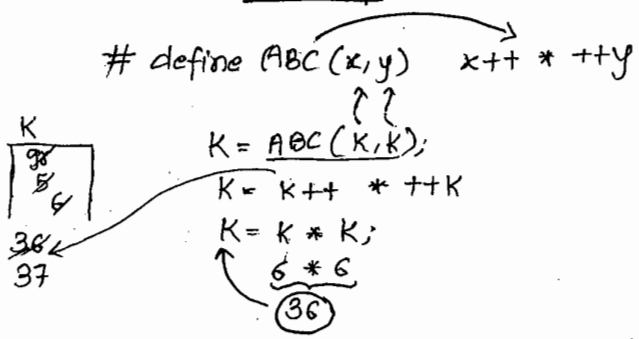
O/P : i = 125

i = CUBE (2+3);
= SQR (a)* (a)
= SQR (2+3) * (2+3);
= (a)* (a) * (5);
= (2+3)* (2+3) * 5,
= 5 * 5 * 5 = 125

Prog6:

```
#define ABC (x,y)   x++*++y
void main()
{
  int K;
  K=5;
  K=ABC(K,K);
  printf("K=%d ",K);
}
```

O/P: K=37



2) FILE INCLUSION PRE PROCESSOR (# include)

- By using this preprocessor we can include a file in another file. Generally by using this preprocessor, we are including Headerfiles.
- A Header file is a source file which contains forward declaration of predefined functions, global variables, constant values, predefined datatypes, predefined structures, predefined macros, inline functions.
- .h file doesn't provides any implementation part of predefined functions, it provides only forward declaration (Prototype).
- A 'C' program is a combination of predefined and userdefined functions.
- .C file contains implementation part of user defined functions and calling statement of predefined functions.
- If the function is userdefined or predefined, logic part must be required.
- project related .obj files provides implementation of user defined functions, .lib files provides implementation part of pre-defined functions which is loaded at the time of linking.
- As per function approach, when we are calling a function which is defined later for avoiding the compilation error, we are required to go for forward declaration i.e prototype is required.
- If the function is user-defined, we can provide forward declaration explicitly but if it is pre-defined func^n, we required to use header file.
- In 'C' programming language, .h files provides prototype of predefined function.
- As a programmer, it is possible to provide forward declaration of pre-defined func^n explicitly but when we are providing forward declaration then compiler thinks it is user-defined func^n so not recommended

➤ .h file doesn't pass for compilation process but .h file code is compiled. When we are including any header file at the time of pre-processing, that header file code will be substituted into current source code and along with current source code header file code also compile.

Syntax : `# include <filename.h>` (using angular body syntax)
or
`# include "filename.h"` (double quotation syntax)

- #include <filename.h> (when we are including header file.)

→ By using this syntax, then it will loaded from default directory i.e c:\tc\INCLUDE.

→ Generally by using this syntax we are including pre-defined header files.

→ When we are including user-defined header files by using this syntax then we need to place user-defined header file in predefined header directory i.e C:\TC\INCLUDE.

- #include "filename.h"

→ By using this syntax, when we are including header, then it is loaded from current working directory.

→ Generally by using this syntax we are including user-defined header files.

→ By using this syntax, when we are including pre-defined header files then first it will search in current project directory, if it is not available then loaded from default directory so it is time-taking process.

```
#include <stdio.h>
#include<conio.h>   console related
#include <stdlib.h>
#include <stdarg.h>
#include <ctype.h>
#include <limit.h>
#include <math.h>
#include <string.h>
#include <dos.h>
#include <dir.h>
#include <graphic.h>
#include <process.h>
```

3) CONDITIONAL COMPILATION PREPROCESSOR

→ When we are working with this preprocessor, then depends on the condition status, code will pass for compilation process

- If condition is true code is pass for compilation
- If condition is false corresponding code is removed from source.

→ the basic advantage of this preprocessor is reducing .exe file size because when source code is reduced then automatically object code is reduced, so exe file size also will be reduced.

```
void main()
{
 printf ("A");
#if 5<8!=1
   printf ("NIT");
   printf ("C");
#endif
   printf ("B");
}
```

O/P: AB

→ In this program, when the code is passing for preprocessing, condition becomes false.

→ That's why corresponding block of the code is removed from source at the time of pre-processing

```
* void main()
  { printf ("NIT");
  #if 2>5!=2<5
     printf ("A");
     Printf ("B");
  #else
     printf ("C");
     printf ("D");
  #endif
     printf ("Welcome");
  }
```

O/P: NITABWelcome

```
* void main()
  {
    printf ("Hello");
  #if 2>5!=0        not valid
     printf ("A");
     printf ("B");
  #elif 5<8         valid
     printf ("NIT");
     printf ("C");
  #else
     printf ("Hi");
     printf ("Bye");
  #endif
  }
```

O/P: HelloNITC

- #ifdef and #ifndef are called MACRO TESTING CONDITIONAL COMPILATION PRERROCESSOR
- When we are working with this pre-processor, depends on the condition only, code will pass for compilation process (depends on macro status)
- By using this preprocessor, we can avoid multiple substitution of header file code.

```
#define NIT
void main()
{
  printf ("Welcome");
  ifdef NIT
     printf ("Hello");
     printf ("NIT");
  #endif
}
```

O/P: WelcomeHelloNIT

- In previous prog. if NIT macro is not defined then corresponding block of the code is not passing for compilation process.
- In Previous prog., NIT is called null macro because it doesn't having any replacement text.

```
#ifndef TEST
  void main()
{ printf("NIT");
  #ifndef TEST
     printf("A");
     printf("B");
  #endif
     printf("Hello");
}
```

O/p: NITHello

undef :-

- By using this preprocessor, we can close the scope of an existing macro.
- Generally this macro is required, when we are redefining an existing macro
- After closing the scope of a macro, it is not possible to access until it is redefined.

```
#define A 11
  void main()
{ printf("%d", A);                    // 11
Error{ // A = 22      Here A is constant, it is already replaced with 11
Error{ // #define A22  Here A is already defined with 11, we cannot do this
  #undef A            first undef, then def.

  #define A22
    printf("%d", A);                  //22

  #undef A
  #define A 33
    printf("%d", A);                  //33

  #undef A
    printf("%d", A);                  //Error
}
```

O/p: Error

pragma

- It is a compiler dependent preprocessor i.e all the compilers doesn't support this preprocessor.
- A preprocessor directive that is not specified by ISO Standard.
- Pragmas offers control actions of the compiler and Linker.
- #pragma is a miscellaneous directive which is used to turn on or off certain features.
- It varies from compiler to compiler if the compiler is not recognized then it ignores it.
- #pragma startup and #pragma exit used to specify which function should be called upon startup (before main()) or program exit (just before program terminates)
- startup and exit functions should <u>not</u> receive or return any values.
- #pragma warn used to suppress (ignore) specific warning msg from compiler.

→ #pragma warn -rvl
return value warnings

→ #pragma warn -par
parameter not used warnings

→ #pragma warn -rch
unreachable code warnings

Prog-1:

```
#pragma warn -rvl
#pragma warn -par
#pragma warn -rch
int abc (int a)
  {
    print ("Hello abc");
  }
void main ()
  {
   abc (10);
    return ;
    getch();
  }
```

O/P: Hello abc

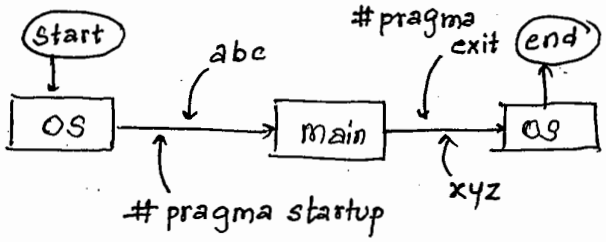
→ When this code is passed for compilation then we are not getting any return value, parameter never used and unreachable code warning messages.

Prog-2

```
#
void abc (void)
void xyz (void)
# pragma startup abc
# pragma exit xyz
void abc () {
    printf ("Hello abc\n");
}
```

O/P:
Hello abc
Hello main
Hello xyz

```
void xyz ()
{
  printf ("Hello xyz");
}
void main()
  {
   printf ("Hello main");
  }
```



- In previous program, abc function is loaded first before loading the main function and xyz function is loaded after loading main function.
- Between startup and exit automatically main function is executed.
- In Implementation when we are having more than 1 startup and exit function then according to the priority, we can execute those functions.

→ In #pragma startup, function which is having highest priority, it will be executed first and which is having least priority, it will be executed at last before main()

→ In #pragma startup, when equal priority occured then last specified funcⁿ will be executed first.

→ In #pragma exit, function which is having highest priority, it will be executed in end and which is having least priority, it will be executed first after main() only.

→ In #pragma exit, when equal priority occured, then last specified funcⁿ will be executed first.

```
void abc()
{ printf ("from abc\n");
}
void xyz()
{
  printf ("from xyz\n");
}
void close()
 {
  printf ("from close\n");
 }
void end()
  {
   printf ("from end\n");
  }
#pragma startup abc 2
#pragma startup xyz 1
#pragma exit close 1
#pragma exit end 2
 void main()
{
  printf ("from main\n");
}
```

O/P:
```
from xyz
from abc
from main
from end
from close
```

#error

- By using this preprocessor, we can create user defined error messags at the time of compilation.

```
#define NIT
void main()
{
 #ifndef NIT
  #error NIT MACRO NEED TO BE DEFINE
 #endif
 #ifdef NIT
   printf("Welcome");
   printf("NIT");
 #endif
}
```

O/P: WelcomeNIT

In previous program, if NIT MACRO is not defined, it gives the error at the time of compiling.

#line

- By using this preprocessor, we can create user defined line sequences in intermediate file i.e .I

```
void main()
{
    printf("A");
#if 5>2!=1
     printf("NIT");
     printf("B");
#endif
#line 4
   printf("Welcome");
   printf("C");
}
```

O/P: AWelcomeC

When the previous code is preprocessing, line sequence is reset to 4

11/7/2015

ARRAYS

- An array is a derived data type in 'C' which is constructed from fundamental data type of 'C' Prog. Language.
- An array is a collection of similar types of data elements in a single entity.
- In implementation when we require 'n' no. of values of same data type, then recommended to create an array.
- When we are working with arrays always **static** memory allocation will happen i.e compile time memory management.
- When we are working with arrays always memory is constructed in continuous memory location that's why possible to access the data randomly.
- When we are working with arrays all values will share same name with unique identification value called 'index').
- Always array index must be required to start with '0' & ends with (size-1).
- When we are working with arrays we required to use array subscript operator i.e [].
- Always array subscript operator require 1 argument of type unsigned integer constant, whose value is always '>0' only.

Syntax `Datatype arr [size];`

Properties of 1D Array

Size → no. of elements, Sizeof → no. of bytes

1. `int arr[5];`
   Size → 5
   sizeof(arr) → 10B (5 * 2 = 10B)

2. `int arr[4];`
   Size → 4
   Sizeof(arr) → 4B
   arr[0] → 1st element
   arr[1] → 2nd
   arr[2] → 3rd

3. int arr[]; error

4. int arr[0]; error

5. int arr[-5]; error

- In declaration of array size must be require to specify or else it gives an error i.e size is unknown.
- In declaration of array size must be unsigned integer constant, whose value is '> 0' only.

6. int arr[5] = {20, 10, 30, 40, 50};

   20 → arr[0];
   10 → arr[1];
   30 → arr[2];
   40 → arr[3];
   50 → arr[4];

7. int arr[5] = {10, 20, 30}

   arr[0] → 10
   arr[1] → 20
   arr[2] → 30
   arr[3] → 0
   arr[4] → 0

- In initialisation of array, if specified no. of elements are not initialised, the remaining all elements are automatically initialized with zero

8. int arr[2] = {10, 20, 30, 40, 50}; error
   In initialization of array we can't initially more than size of array elements, if we are initializing then it gives an error i.e too many initializations.

9. int arr[] = {10, 20, 30, 40, 50}; yes valid

   Size → 5;
   Size of (arr) → 10 B

→ • In initialisation of array specifying the size is optional, in this case how many elements are initialized that many variables are created automatically.

10. int arr[5];

    arr[0] = 10;
    arr[2] = 30;
    arr[4] = 50;

printf ("%d %d", arr[1], arr[3]);

O/P: gr → In declaration of the array by default all elements are having garbage values only bcz, by default it is autotype.

11. Static int arr[5];
arr[0] = 10;
arr[1]
arr[2]
printf ("%d %d", arr[3], arr[4]);

O/P: 0 0

12. int arr[2]
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
printf ("%d %d %d", arr[0], arr[1], arr[2]);

Valid

In C & CPP there is no any upper boundary checking process occurs, so when we are crossing the limit then depending upon OS, Security level anything can happen (segmentation fault).

13. float arr[5.8]; (error)

14. float arr[5];
size → 5
sizeof (arr) → 5X4 = 20B.

→ On DOS based Compiler at compile time we can create maximum of 64 KB data ie 65536B but in previous syntax we require 80,000B

16) long int arr[20000]; Error
20000 X 4 ⇒ 80,00B

17) char arr[40000] Valid

18) int size = 10;
int arr[size], error

19) Const int size = 10
int arr[size]; (error)

20) # define size 10
int arr[size]; (Valid)

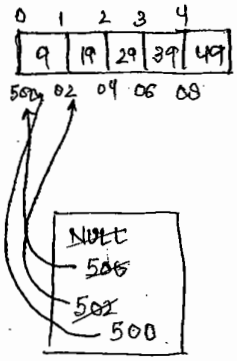
- In declaration of array size can't be variable or constant variable type.
- In declaration of array size can be symbolic constant value because at the time of preprocessing it is replaced with constant value.

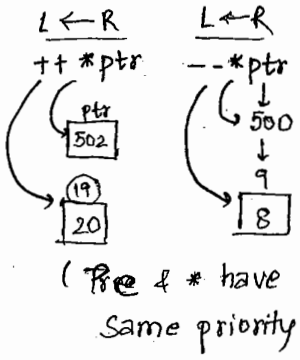
21) int arr [2+3]; Yes, valid
int arr [5];

22) int arr [2>5]; error // int arr [0];

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
int main()
{
  int arr [5] = {9, 19, 29, 39, 49}
  int near *ptr = (int near*) NULL ; // will take 2B
                  (equal priority for binary →  )
  ptr = &arr [0];
  ++ptr;
  ++ *ptr;
  --ptr;
  --*ptr;
  printf ("%d %d", arr [0], arr [1]);
  getch();
  return EXIT_SUCCESS
}
```

O/P: 8 20





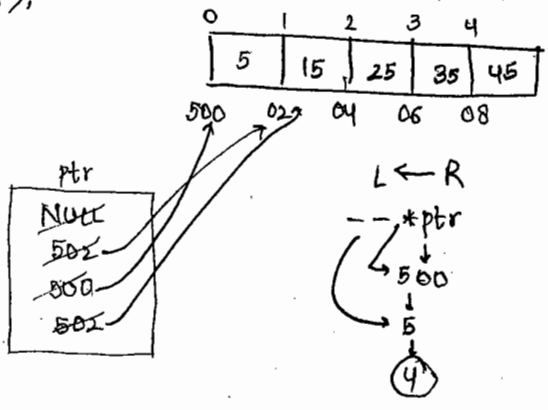
(Pre & * have same priority

* indirection operator & pre-operator both are having equal priority.
* when equal is occured, for unary operator it should be required to evaluate from Right to left only.
* Arithmatic operation of the pointers always data type dependent only.

```
# include <stdio.h>
# include <conio.h>
int main (void)
{
  int arr [5] = {5, 15, 25, 35, 45};
  int *ptr = (int *) NULL ; // ptr will take 4B (32 bit compiler)
  ptr = &arr [1];
  --ptr;
  --*ptr;
```

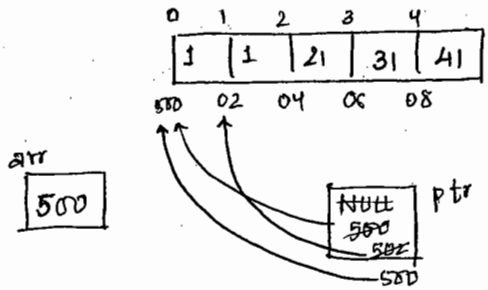
```
++ptr;
++*ptr;
printf("%d %d", arr[0], arr[1]);
getch();
return 0;
}
```

O/P: 4 16




```
#include <stdio.h>
#include <conio.h>
int main()
{
int arr[] = {1, 11, 21, 31, 41}
int *ptr = NULL;
ptr = arr;  // &arr[0]
++ptr;
--*ptr;
--ptr;
++*ptr;
Pf("%d %d %d", arr[0], arr[1], arr[2]);
getch();
return 0;
}
```

O/P: 2 10 21

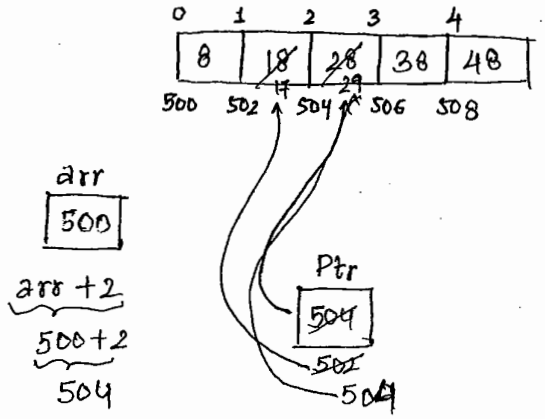


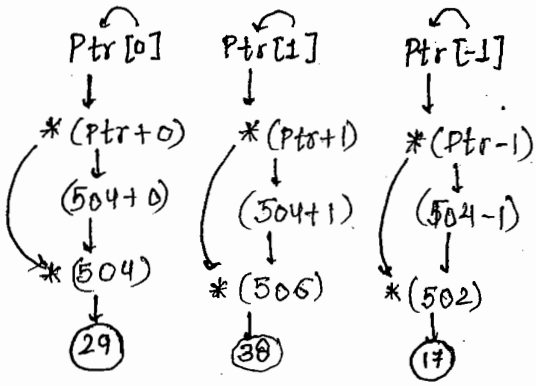
⇒ An array is an implicit pointer in C language which always maintain base address of an array.

⇒ Arr name always provides base address i.e &arr[0], arr+1 will provide next address of an array &arr[1]

```
# include <stdio.h>
# include <conio.h>
int main()
{
 int arr [] = {8, 18, 28, 38, 48}
 int *ptr = arr + 2;
 --ptr;
 --*ptr;
 ++ptr;
 ++*ptr;
 printf ("%d %d %d", ptr[0], ptr[1], ptr[-1]);
 getch();
 return 0;
}
```

O/P:- 29 38 17

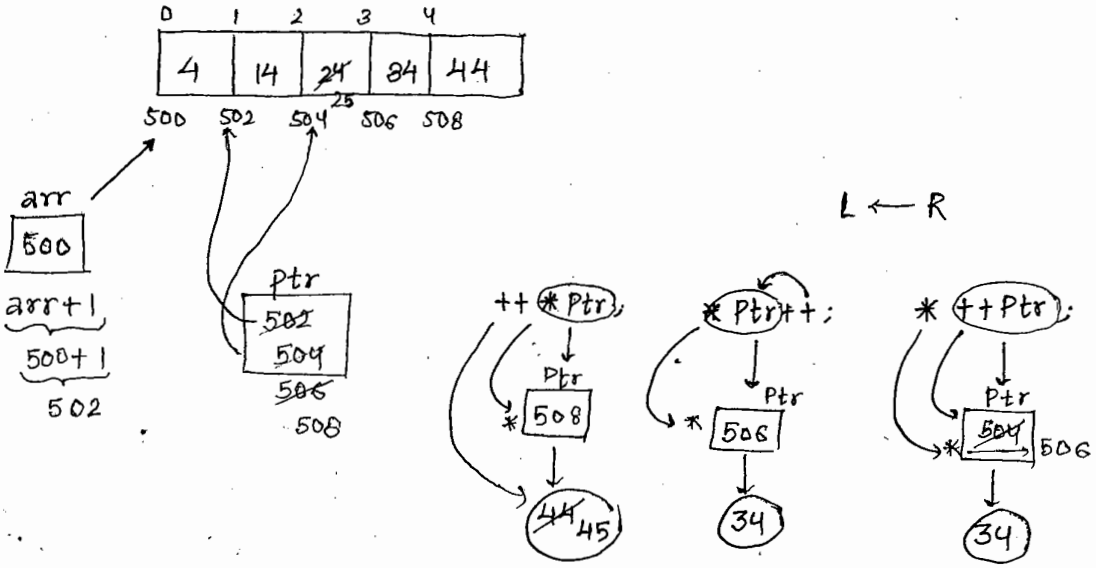




On pointer variable when we are applying subscript operator then index value will map with current value of the pointer, later it will access corresponding value.

```
# include <stdio.h>
# include <conio.h>
void int main()
{
 int arr [5] = {4, 14, 24, 34, 44};
 int * ptr = arr + 1;   // &arr [1]
 ++ptr;
 ++*ptr;
 printf ("%d %d %d", ++*ptr, *ptr++, *++ptr);
 getch();
 return 0;
}
```

O/P:- 45 34 34



Pointer Arithmetic Syntax :-

1. ++ptr;
2. ptr++;
3. --ptr;
4. ptr--;

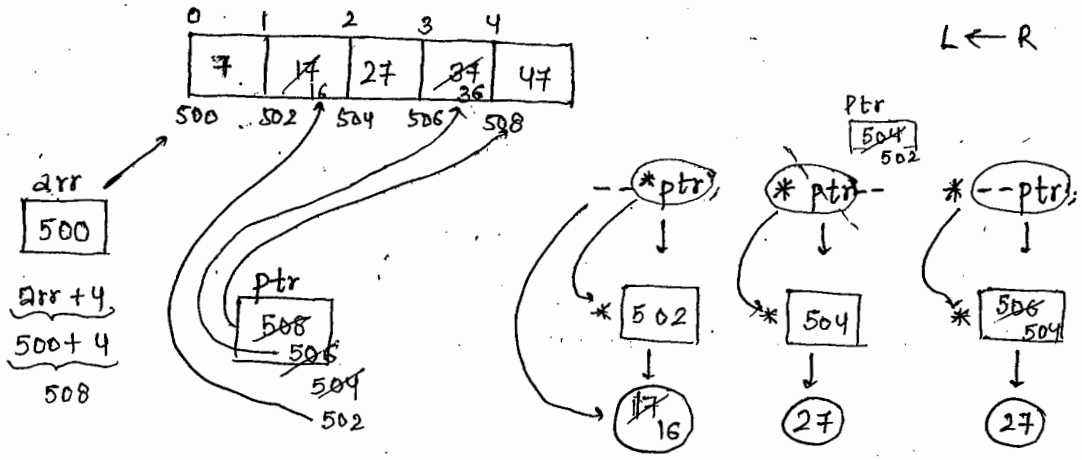
1. ++*ptr; (Pre incrementation of object)
2. (*ptr)++; (Post incrementation of object)
3. --*ptr; (Pre decrementation of object)
4. (*ptr)--; (Post decrementation of object)

1. *++ptr; (Pre incrementation of pointer & accessing the object
2. *ptr++; (Accessing the object & Post incrementation of pointer)
3. *--ptr; [Predecrementation of pointer & accessing the object]
4. *ptr--; [Accessing the object & post decrementation of pointer]

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int main (void)
{
  int arr[] = {7, 17, 27, 37, 47};
  int *ptr = arr + 4;
  --ptr;
```

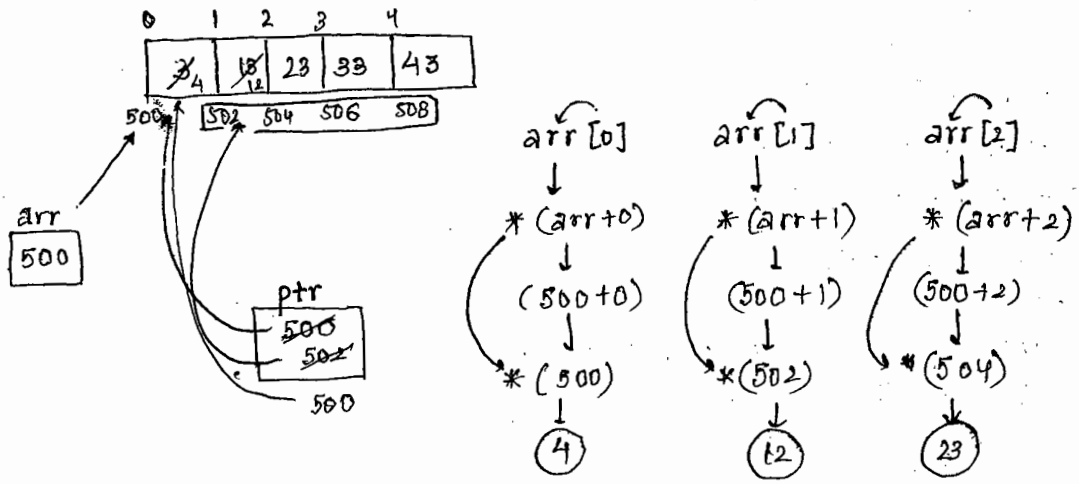
```
--*ptr;
printf("%d %d %d", --*ptr, *ptr--, *--ptr);
getch();
return EXIT_SUCCESS;
}
```

O/p :- | 16 | 27 | 27 |




```
#include <stdio.h>
#include <conio.h>
int main()
{
int arr[5] = {3, 13, 23, 33, 43};
int *ptr = arr;
++ptr;
--*ptr;
--ptr;
++*ptr;
printf("%d %d %d", arr[0], arr[1], arr[2]);
getch();
return 0;
}
```

O/p: 4 12 23

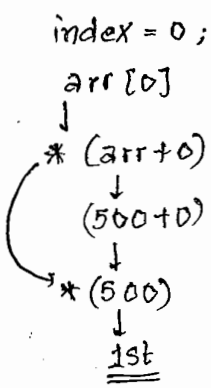


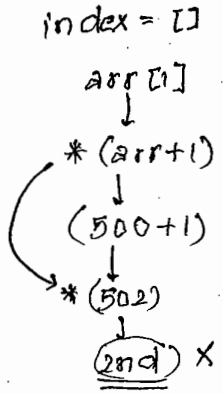
→ When we are working with arrays all element information doesn't maintain by program i.e compiler

→ When we are working with arrays, compiler will maintain only 1 element information that is called base address and remaining all elements are

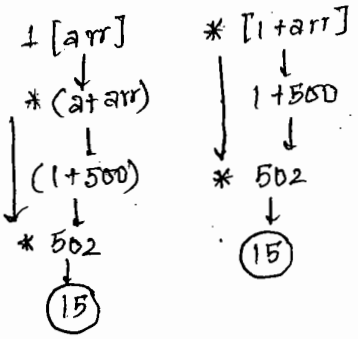
→ Acc. to index mapping mechanism, every element index value will map with base address of array to access corresponding element [Pointer Arithmetic Operation]

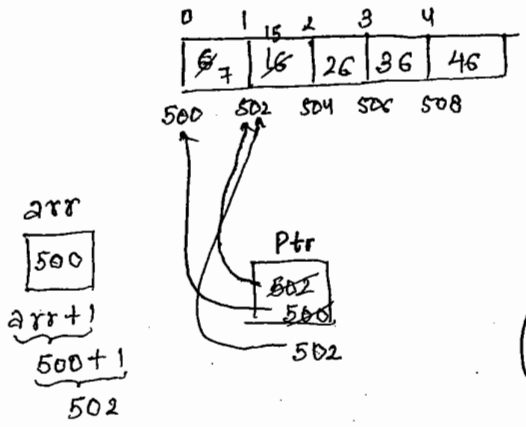
NOTE: Always Array index must be started with only 0, because according to mapping mechanism of index value is started from 1 then we can't access first element.

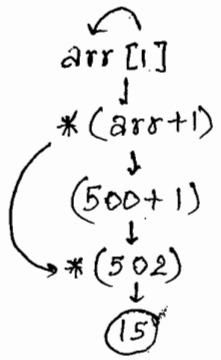


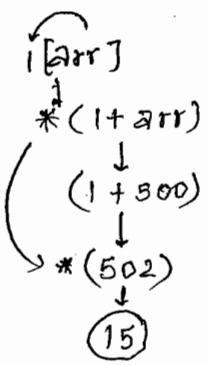



```
#include <stdio.h>
#include <conio.h>
int main()
{
  int arr[] = {6, 16, 26, 36, 46};
  int *ptr = arr+1;
  --ptr;
  ++*ptr;
  ++ptr;
  --*ptr;
  printf("%d %d %d %d", arr[1], *(arr+1), 1[arr], *(1+arr));
  getch();
  return 0;
}
```










```
# include <stdio.h>
# include <conio.h>
int main()
{
int ~~main()~~ arr[] = {12, 12, 22, 32, 42};
++arr;        //arr = arr+1;
++*arr;
--arr;        //arr = arr-1;
--*arr;
printf("%d %d", ~~*arr~~ arr[0], arr[1]);
getch();
return 0;
}
```

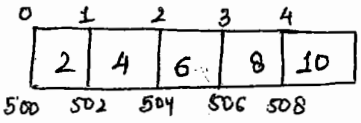
O/P: Error L value required

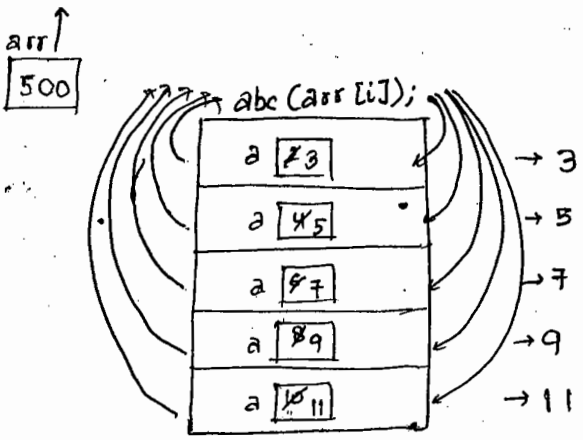
→ An array is a implicit constant pointer which always maintain base address and doesn't allows to modify it.

14th July 15

```
# include <stdio.h>
# ~~include~~ include <conio.h>
void abc (int a)
{
  ++a;
  printf("%d", a);
}
int main()
{
  int arr[5] = {2,4,6,8,10};
  int i;
  printf("\n Data in abc: ");
```

```
for(i=0; i<5; i++)
    abc(arr[i]);
printf("\n Arr Data List: ");
for(i=0; i<5; i++)
    printf("%d", arr[i]);
    getch();
    return 0;
}
```





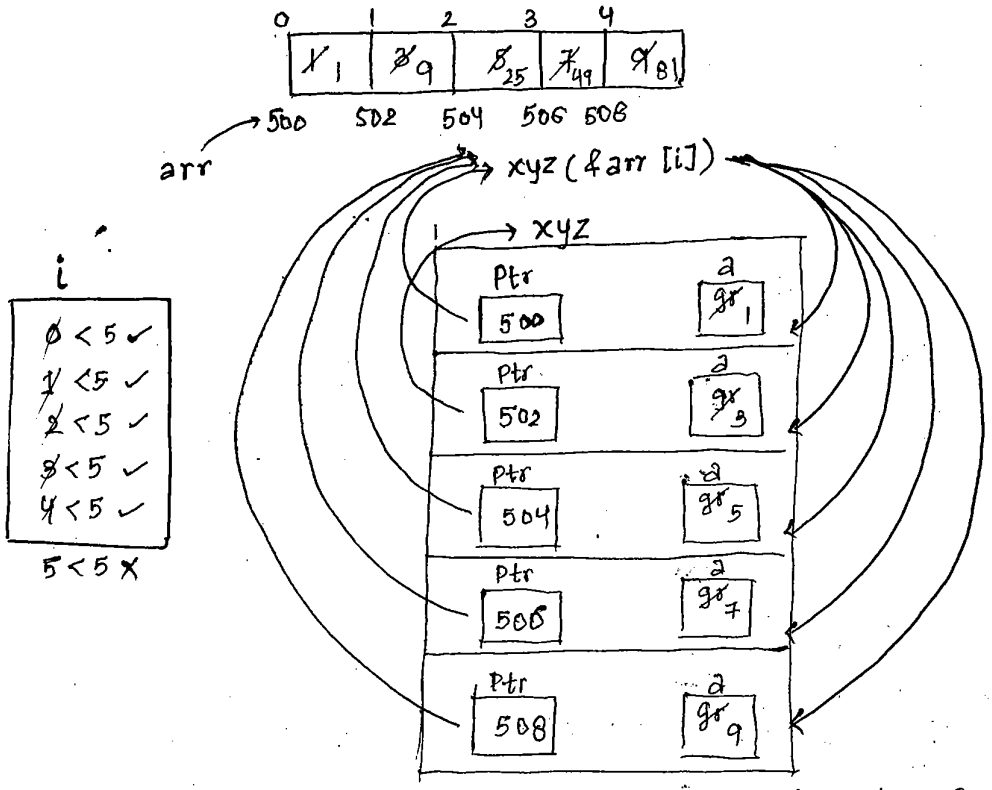
i
gr
0<5
1<5
2<5
3<5
4<5
5<5

O/P:
```
Data in abc: 3 5 7 9 11
Arr Data List: 2 4 6 8 10
```

- In the above program, array elements are passing by using call by value mechanism thats why no modification of abc() function is passing back to main()
- In implementation when we are expecting the modifications then it is required to go call by address mechanism.

→
```
#include <stdio.h>
#include <conio.h>
#include <math.h>
  void xyz(int *ptr)
{ int a;
  a = *ptr;
  *ptr = (int) pow(a,2);
  printf("%d", *ptr);
}
int main() {
    int arr[5] = {1,3,5,7,9}
    int i; clrscr();
    printf("\n Data in xyz :");
    for(i=0; i<5; i++)
      xyz(&arr[i]);
      printf("\n Arr Data List : ");
      for(i=0; i<5; i++)
      printf("%d", arr[i]);
      return 0;
    }
```

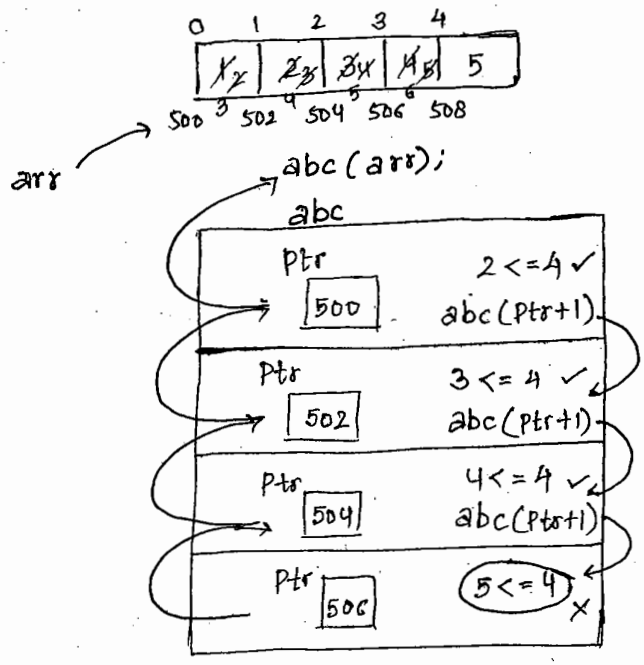
O/P:-
```
Data in xyz : 1 9 25 49 81
Arr Data List : 1 9 25 49 81
```



- In previous program, array elements are passing by using Call by address mechanism that's why all the modifications of xyz() function is passing back to main function.

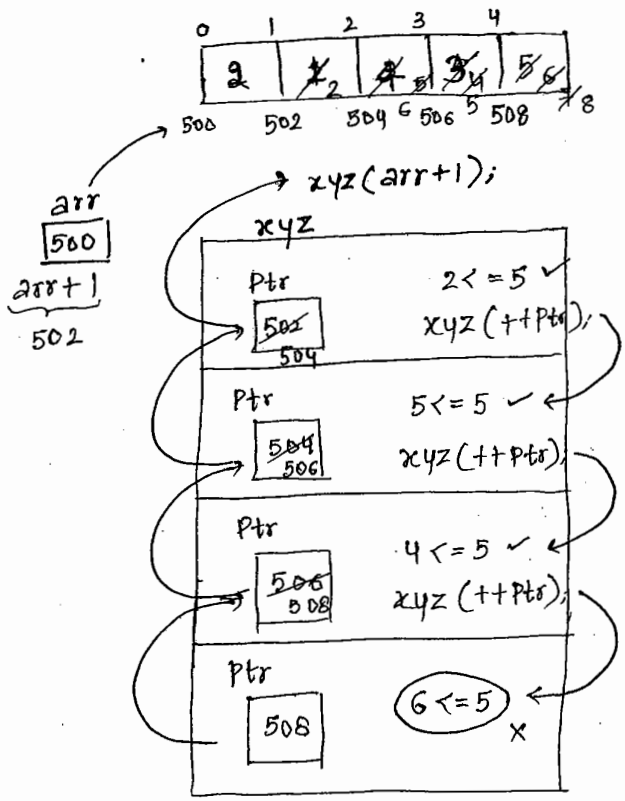
```
#include <stdio.h>
#include <conio.h>
void abc(int *ptr)
 {
   ++*ptr;
   if(*ptr<=4)
     abc(ptr+1);
   ++*ptr;
 }
int main()
 {
   int arr[5]={1,2,3,4,5};
   int i;
   clrscr();
   abc(arr);
   printf("\n Arr Data list: ");
   for(i=0; i<5; i++)
     printf("%d",arr[i]);
   getch();
   return 0;
 }
```

O/P: Arr data list : 3 4 5 6 5



Since it is a recursion when condition becomes false then second *ptr line is executed for each rollback & thus each index value is changed once again.

```
# include <stdio.h>
# include <conio.h>
  void xyz (int * ptr)
{
   ++ * ptr;
   if (* ptr <= 5)
     xyz (++ ptr);
    ++ * ptr;
 }
 int main ()
   {
    int arr [5] = {2, 1, 4, 3, 5};
    int i;
    clrscr();
    xyz (arr+1);
    printf ("\n Arr Data List:");
    for (i= 0; i< 5; i++)
    printf ("%d ", arr [i]);
    getch ();
    return 0;
 }
```



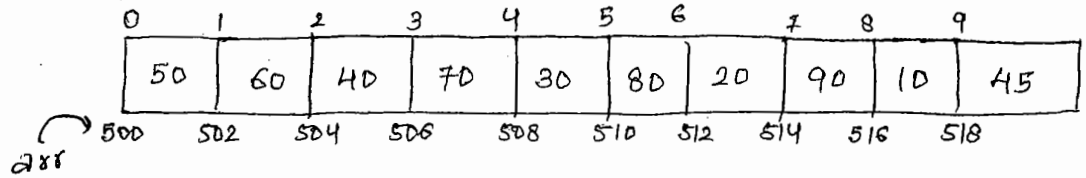
O/P:- Arr Data List : 2 2 6 5 8

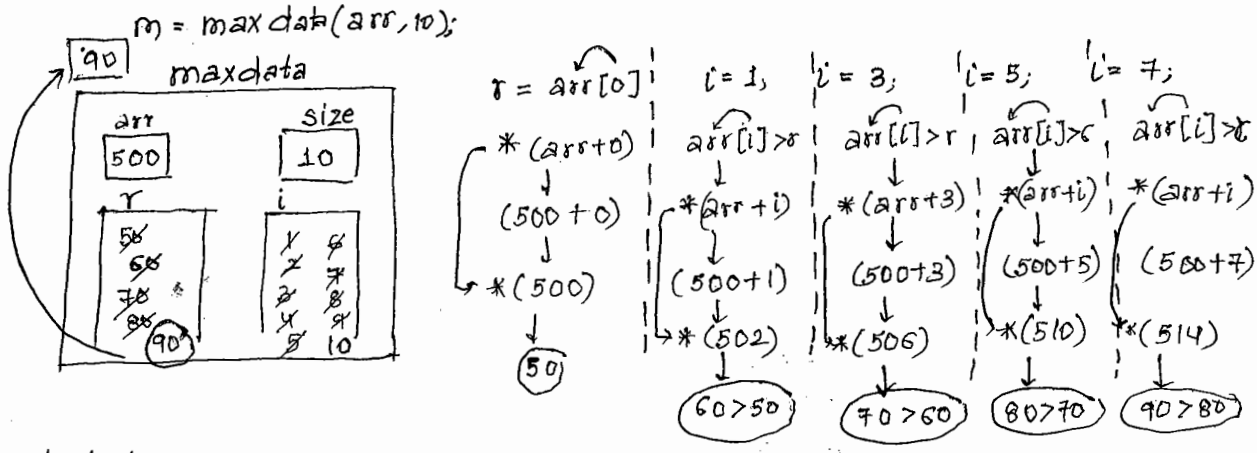
- It is not possible to pass complete array as a argument to the function. In implementation, when we required to pass complete array as a argument to the function but complete array we can access from outside of the function.
- In implementation when we required to access complete array from outside of the function then we required to pass base address of the array alongwith size.
- If we know the base address then along with the indexes, complete array we can access from outside of the function.
- In parameter location it is not possible to create an array.
- If array syntax is available, then it creates POINTER only.
- In parameter location, if `int arr[]` syntax is available then it creates a POINTER & it is indicating that hold 1D array address.

To find max no. from the data

```
# include <stdio.h>
# include <conio.h>
int maxdata (int arr[], int size)
 {
   int i, r;
   r = arr[0];   // r = *(arr+0);
   for(i=1; i<size; i++)
    {
      if (arr[i]>r)
       r = arr[i];
    }
 return r;
}
int main()
 {
   int arr[10];
   int i, m;
   clrscr();
   Printf ("Enter 10 values :");
   for (i=0; i<size; i++)
    scanf("%d", &arr[i]);
   m= maxdata(arr, 10);
   printf ("Max value of list: %d", m);
   getch();
   return 0;
```

O/P: Enter 10 values:
50 60 40 70 30 80 20 90 10 45





15| July-15

```
#include <stdio.h>
#include <conio.h>
#define size 10
int main()
{
 int arr[size];
 int mindata(int *);    // declaration
 int i, m;
 clrscr();
 printf("Enter %d values: ", size);
 for(i=0; i< size ; i++)
   scanf("%d", &arr[i]);
 m = mindata(arr);     //int mindata(int arr[])
 printf("Min data of list : %d", m);
 getch();
 return 0;
}
int mindata(int *ptr)  //int mindata(int arr[])
{ int r, i;
  r = *(ptr+0);   // r = ptr[0];
  for(i=1; i<size; i++)
  { if(*(ptr+i) < r)   // if(ptr[i] <r)
      r= *(ptr+i);   // r= ptr[i]
  }
  return r;
}
```

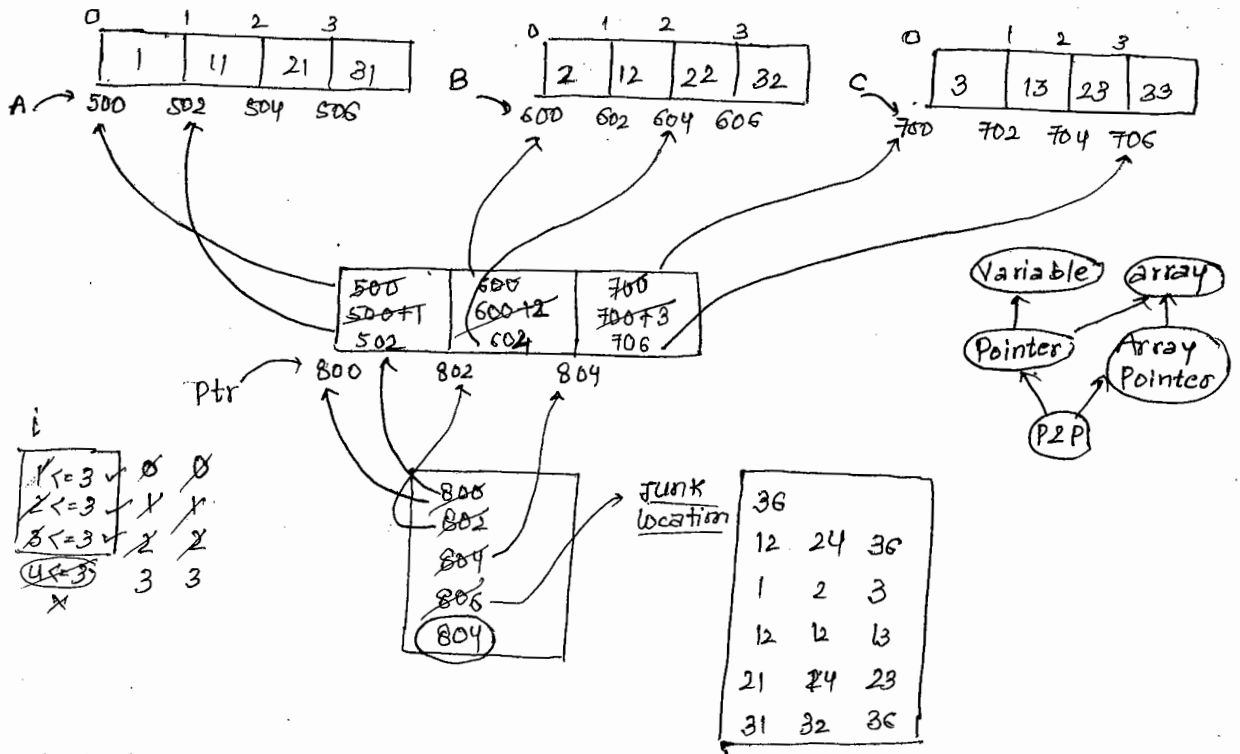
O/P:

```
Enter 10values : 50 60 70 40 30
                 80 20 80 10 45
Min data of list : 10
```

```
→ #Include <stdio.h>
  #include <conio.h>
   int main()
   {
     int A[] = {1, 11, 21, 31};
     int B[] = {2, 12, 22, 32};
     int C[] = {3, 13, 23, 33};
     int *ptr[3];      //Array Pointer
     int **pptr;       //Pointer to pointer
     int i;
     clrscr();
      ptr[0]=A;        // &A[0]
      ptr[1]=B;        // &B[0]
      ptr[2]=C;        // &C[0]
    pptr = ptr;        // &ptr[0]
    for(i=1; i<=3; i++)
      {
        *pptr++ = i;
       **pptr++ = i;
        ++ptr;
      }
    --pptr;
    printf("\n%d\n", **ptr);
    for(i=0; i<size; i++)
        printf("%d ", *ptr[i]);
   for(i=0; i<4; i++)
      printf("\n %d %d %d ", A[i], B[i], C[i]);
     getch();
     return 0;
   }
```

O/P :-

36			
12	24	36	
1	2	3	
12	12	13	
21	24	23	
31	32	36	



- Collection of similar types of pointer in a single variable is called Array pointer.
- When we required n no. of pointer variables of same data type then go for array pointer
- Array pointer name always gives base address of array pointer.
- By using pointer to pointer variable, we can maintain pointer address / array pointer address.

SORTING

- It is a procedure of arranging the elements in a particular order i.e ascending or descending order
- In 'C' prog. lang., we are having 8 types
  1) Bubble Sort
  2) Selection sort
  3) Insertion sort
  4) Merge sort
  5) Quick Sort
  6) Heap Sort
  7) Radix or Bucket
  8) shell sort

1) **BUBBLE SORT** :-

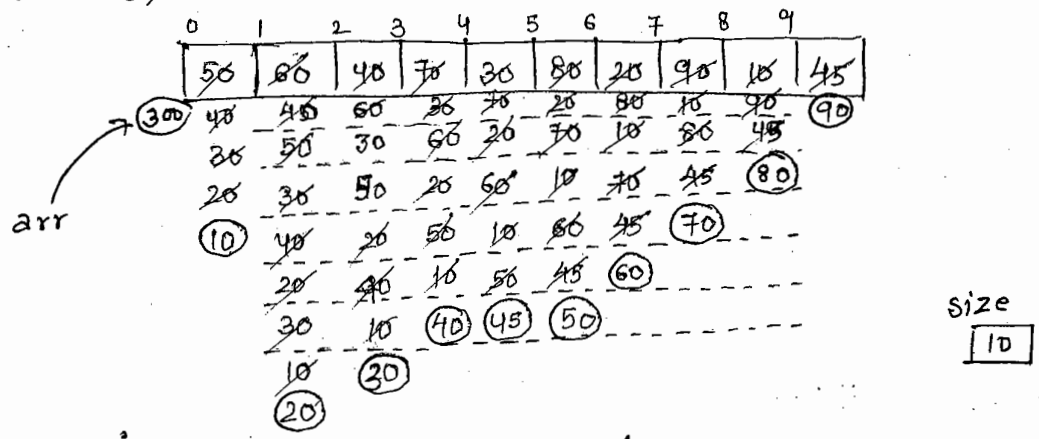
- When we are working with Bubble sort, adjacent elements are compared until last element will fix i.e max value of the list
- When we are working with Bubble Sort, if n no. of unsorted elements are available then n-1 comparision will take place.
- When we are working with Bubble sort, elements are arranged from max. to min i.e descending order but final result is as ascending order only.

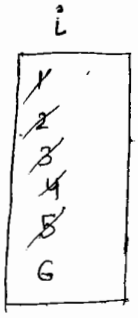
2) **SELECTION SORT** :-

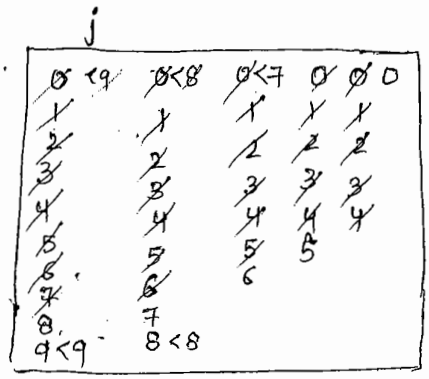
- When we are working with selection Sort, sequential comparision will take place until first value is fixed, i.e min. value of the list.
- If n no. of un-sorted elements are available then n no. of elements are available (n-1) comparison will take place.
- In this sort, elements are arranged in ascending order & final result also ascending order

```
# include <stdio.h>
# include <conio.h>
# define size 10
  int main()
{
  int arr[size];
  int t, i, j;
  clrscr();
  printf(" Enter %d values : ", size
  for (i=0; i< size ; i++)
    scanf ("%d", &arr[i])
  for (i=1; i< size; i++)
    {
      for(j = 0; j < size ; j++)
        { t = arr[j];
          arr [j] = arr [j+1];
          arr [j+1] = t;
        } //if
    } // inner
  } //outer
  printf ("\n sorted arr data list: ");
```

```
for (i= 0 ; i< size; i++);
   printf ("%d", arr[i])
   getch ();
   return 0;
```

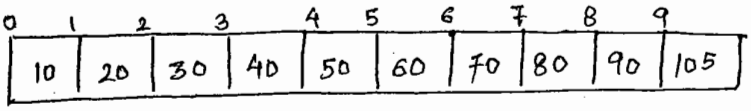







```
#include <stdio.h>
# include <conio.h>
# define size 10
int main()
{
  int arr [size]
   int sum =0, i;
   float avg;
   clrscr();
   printf (" Enter %d values : ", size);
   for (i = 0; i< size; i++)
    scanf ("%d", &arr [i]);
    for(i=0; i<size; i++)
    sum += arr[i];
    avg = (float)sum/size;
    printf ("\n Sum of list : %d", Sum);
    printf ("\n Avg of List : %.2f", avg);
    getch();
    return 0;
}
```

```
O/P :- Enter 10 values :
  10 20 30 40 50 60 70 80 90 105
  Sum of list : 555
  Avg of list : 55.50
```



sum

~~0+10~~ ~~150+60~~
~~10+20~~ ~~210+70~~
~~30+30~~ ~~280+80~~
~~60+40~~ ~~360+90~~
~~100+50~~ ~~450+105~~

555

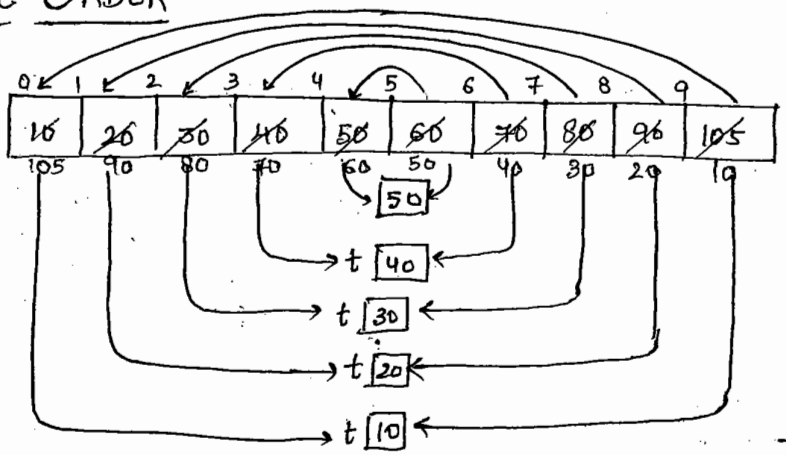
i

0<1 ✓
1<2 ✓
2<3 ✓
3<4 ✓
4<5 ✓
5<6 ✓
6<7 ✓
7<8 ✓
8<9 ✓
9<9 X → False so move outside of the loop.

Size
10

Avg
55.50

REVERSE ORDER



Size
10

i	size-i-1
0	9
1	8
2	7
3	6
4	5

```
# include <stdio.h>
# include <conio.h>
# define size 10
int main()
{
  int arr [size]
  int i,t;
  clrscr();
printf("Enter %d values:",size);
for(i=0; i<size; i++)
scanf("%d", &arr[i]);
for(i=0; i<size/2; i++)
{ t = arr[i]
  arr[i] = arr[size-i-1];
  arr[size-i-1] = t;
}
```

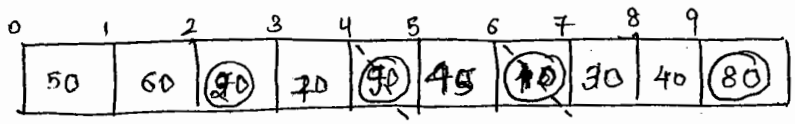
```
printf("\n Reverse arr data list: ");
for(i=0; i<size; i++)
printf("%d", arr[i]);
getch();
return 0;
}
```

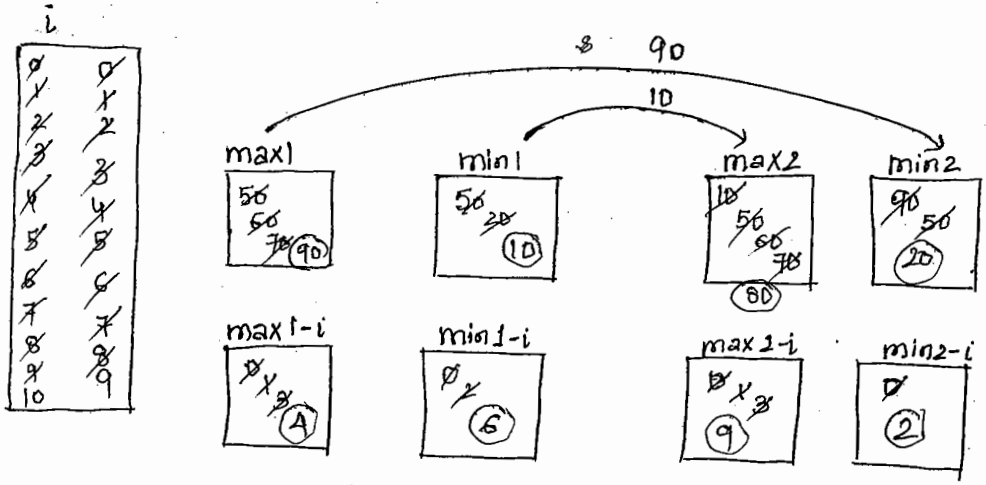
```
/*
  for(i=0; i<size/2; i++)
  { arr[i] = arr[i] + arr[size-i-1];   // a = a+b;
    arr[size-i-1] = arr[i] - arr[size-i-1];   // b = a-b;
    arr[i] = arr[i] - arr[size-i-1];   // a = a-b;
  }
*/
```

O/P:

```
Enter 10 values : 10 20 30 40 50 60 70 80 90 105
Reverse arr data list : 105 90 80 70 60 50 40 30 20 10
```

Program





For max 2, eliminate 90 we should not compare it because it is already value of max 1

Same for min 2, eliminate 10

```
#include <stdio.h>
#include <conio.h>
#define size 10
int main()
{
  int arr[size];
  int max1, min1, max2, min2;
  int max1_i, min1_i;
  int max2_i, min2_i;
  int i;
  clrscr();
  Printf("Enter %d values : ", size);
  for(i=0; i<size; i++)
   scanf("%d", &arr[i]);
  max1 = min1 = arr[0];
  max1_i = min1_i = 0;
  for(i=1; i<size; i++)
   {
    if(arr[i] > max1)
      {
       max1 = arr[i];
       max1_i = i;
      }
    if(arr[i] < min1)
      {
       min = arr[i];
       min1_i = i;
      }
   }
   max2 = min1;
   min2 = max1;
   for(i=0; i<size; i++)
    {
     if(arr[i] > max2 && arr[i] != max1)
       {
         max2 = arr[i];
         max2_i = i;
       }
     if(arr[i] < min2 && arr[i] != min1)
       {
         min2 = arr[i];
         min2_i = i;
       }
```

```
O/P :-
Enter 10 values : 50 60 20 70 90 45 10 30
                                    40 80
 max1: 90    index: 4
 min1: 10    index: 6
 max2: 80    index: 9
 min2: 20    index: 2
```

```
    }
    printf ("\n Max1 : %d  Index : %d", max1, max1_i);
    printf ("\n Min1 : %d  Index : %d", min1, min1_i);
    printf ("\n Max2 : %d  Index : %d", max2, max2_i);
    printf ("\n Min2 : %d  Index : %d", min2, min2_i);
    getch();
    return 0;
}
```

Two-dimensional Array

- In 2-d array, elements are arranged in rows, column format.
- When we are working with 2-d array, we required to use 2 subscript operators which indicates row size, column size
- The main m/m of 2d array is rows and elements are available in columns.
- On 2d array, when we are applying one subscript operator then it gives row name, raw name always provides corresponding row address
- From 2d array, when we required to access the element then 2 subscript operators required to use
- arr always provides main m/m, arr+1 will provides next memory of array

Syntax : `Datatype arr [RSIZE] [CSIZE];`

Properties of 2D Array.

1. int arr [3][4]
   size ---> 3*4
   sizeof (arr) ---> 24B
2. int arr [3][3]
   size ---> 3*3
   sizeof (arr) ---> 18B (3*3 = 9*2B = 18B)
   3 rows
   each row 3 columns
   9 int variables
3. int arr [][]; Error
4. int arr [3][]; Error
5. int arr [][3]; Error

In declaration of 2D array, it is mandatory to specify row and column sizes or else it gives an error

6. int arr[2][3] = {10, 20, 30, 40, 50, 60};

10 --> arr[0][0]
20 --> arr[0][1]
30 --> arr[0][2]
40 --> arr[1][0]
50 --> arr[1][1]
60 --> arr[1][2]

* By using above initialisation process, we required to initialise all elements in sequence only i.e selected no. of elements can't be initialised.

```
int arr[3][3] = {
                  {10},
                  {20,30},
                  {40,50,60}
                };
```

arr[0][0] --> 10
arr[0][1] --> 0
arr[0][2] --> 0
arr[1][0] --> 20
arr[1][1] --> 30
arr[1][2] --> 0
arr[2][0] --> 40
arr[2][1] --> 50
arr[2][2] --> 60

→ In initialisation of 2d array, if sepecific no. of elements are not initialised then remaining all elements are initialised with zero

8. int arr[][] = {

```
                  {10,20,30},
                  {40,50},
                  {60,70}
                };
```

Error

9. int arr[3][] = {

```
                  {10,20},
                  {30},
                  {40,50,60}
                };
```

Error

10. int arr[][3] = {

```
                  {10},
                  {20,30},
                  {40,50,60}
                };
```

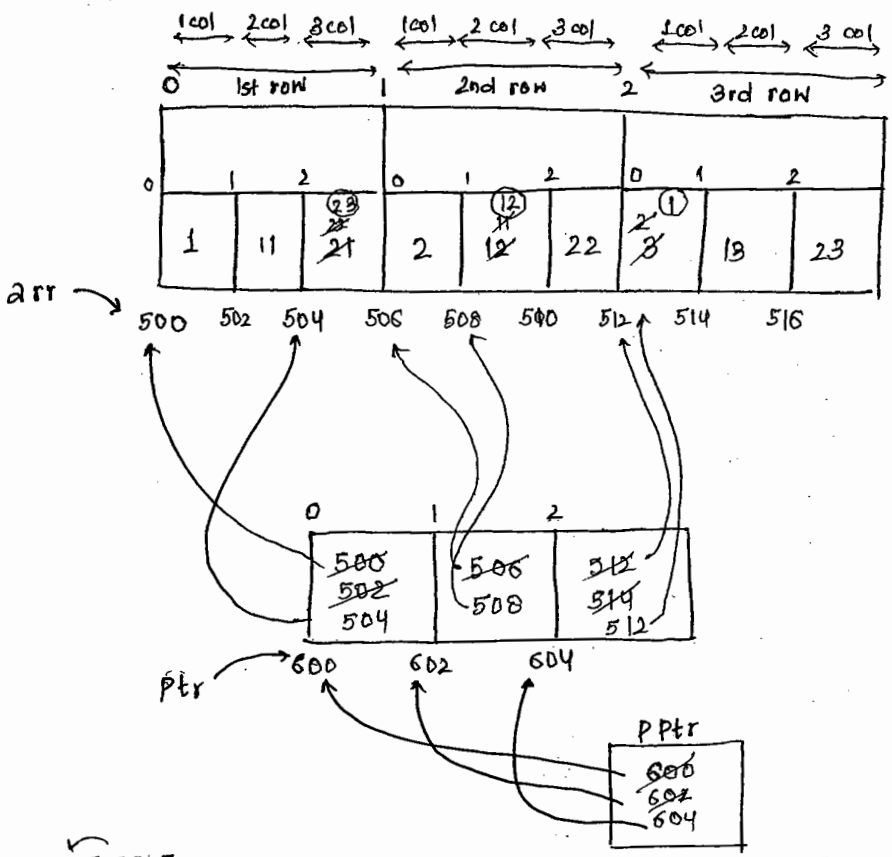
is it valid? Yes valid

→ In initialisation of 2d array, specifying the row size is optional but column size is mandatory.
But in declaration, row & column sizes are mandatory.

```
#include <stdio.h>
#include <conio.h>
int main()
{
  int arr[3][3] = {
                    {1, 11, 21},
                    {2, 12, 22},
                    {3, 13, 23}
                  };
  int *ptr[3];   // Array pointer
  int **pptr;    // pointer to pointer
  clrscr();
  ptr[0] = arr;  //&arr[0][0]
  ptr[1] = arr+1; //&arr[1][0]
  ptr[2] = arr+2; //&arr[2][0]
  pptr = ptr;

  ++*pptr;
  ++ptr[0];
  ++**pptr;
  ++**ptr[0];

  ++pptr;
  ++*pptr;
  --**pptr;
  ++*ptr[1];

  ++pptr;
  ++*pptr;
  --ptr[2];
  --**pptr;
  --*ptr[2];

  printf("\n%d %d %d", arr[0][2], arr[1][1], arr[2][0]);
  printf("\n%d %d %d", *(*(arr+0)+2), *(*(arr+1)+1), *(*(arr+2)+0));
  printf("\n%d %d %d", *ptr[0], *ptr[1], *ptr[2]);
  printf("\n %d %d %d", **(ptr+0), **(ptr+1), **(ptr+2));
  getch();
  return 0;
}
```

O/P :-

23	12	1
23	12	1
23	12	1
23	12	1



1 block is having 18 bytes and 1 row contains 6 bytes.

arr[0][2]
↓
*(arr+0)[2]
↓
((arr+0)+2)
↓
(500+0) ← main m/m
↓
*(500) ← row address
↓
(500+2) ← column address
↓
*(504)
↓
23

arr[1][1]
↓
*(arr+1)[1]
↓
((arr+1)+1)
↓
(500+1) ← main m/m
↓
*(506) ← row address
↓
(506+1) ← column address
↓
*(508)
↓
12

arr[2][0]
↓
*(arr+2)[0]
↓
((arr+2)+0)
↓
(500+2) ← main m/m
↓
*(512) ← row address
↓
(512+0) ← column address
↓
*(512)
↓
1

---

*Ptr[0]
↓
**(Ptr+0)
↓
(600+0)
↓
*(600)
↓
*(504)
↓
23

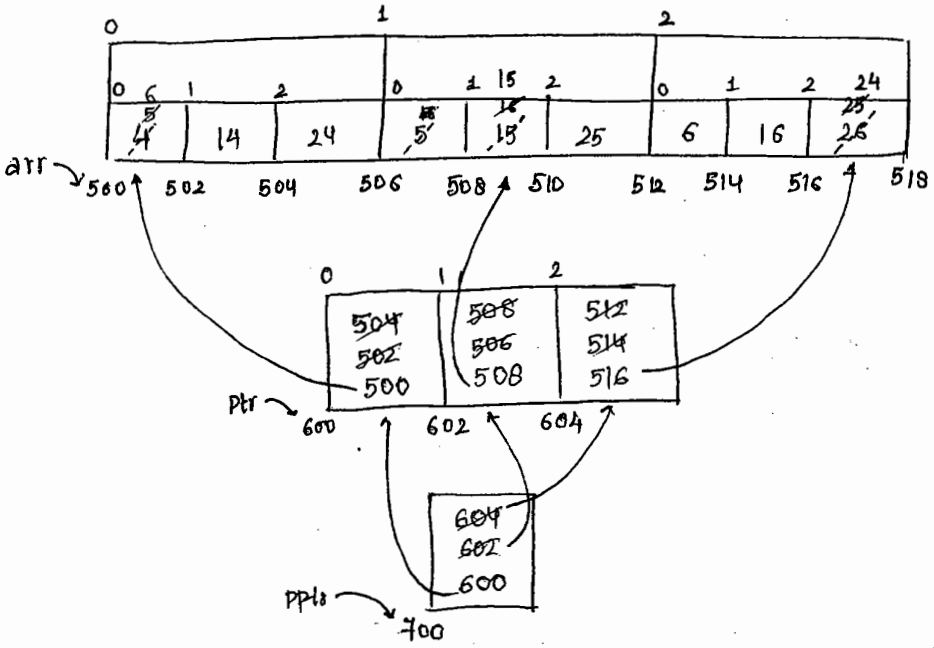
*Ptr[1]
↓
**(Ptr+1)
↓
(600+1)
↓
*(602)
↓
*(508)
↓
12

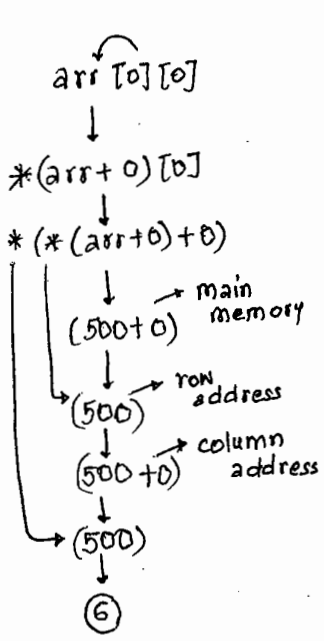
*Ptr[2]
↓
**(Ptr+2)
↓
(600+2)
↓
*(604)
↓
*(512)
↓
1

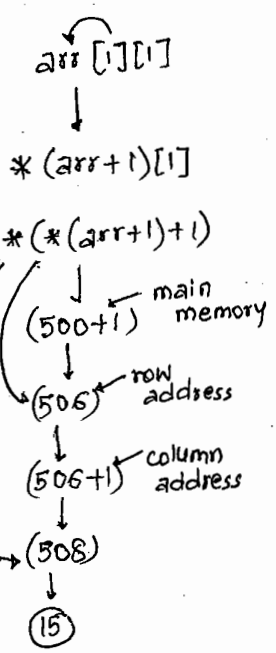
17/7/2015

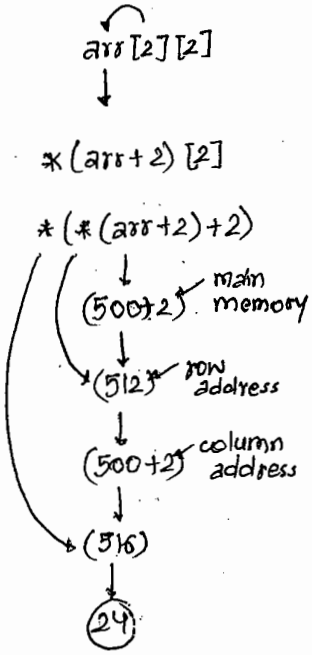
```
#include <stdio.h>
#include <conio.h>
int main()
{
  int arr[][3] = {
                   {4, 14, 24},
                   {5, 15, 25},
                   {6, 16, 26}
                 };
        int *ptr[3];        // array pointer
        int **pptr;         // pointer to pointer
        clrscr();
        ptr[0] = arr[0] + 2;    // &arr[0][2]
        ptr[1] = arr[1] + 1;    // &arr[1][1]
        ptr[2] = arr[2] + 0;    // &arr[2][0]
        pptr = ptr + 2;         // &ptr[2]

        ++*pptr;
        ++ptr[2];
        --**pptr;
        --*ptr[2];

        --pptr;
        --*pptr;
        ++ptr[1];
        ++**pptr;
        --*ptr[1];

        --pptr;
        --*pptr;
        ++**pptr;
        ++**ptr[0];

        printf("\n%d %d %d", arr[0][0], arr[1][1], arr[2][2]);
```

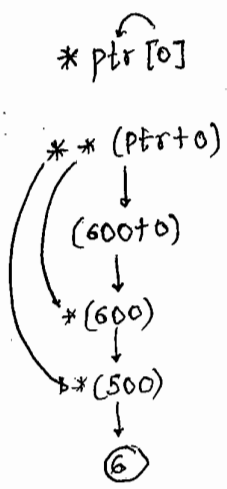
```
printf ("\n %d %d %d ", *ptr[0], *ptr[1], *ptr[2]);
printf ("\n %d %d %d ", **pptr, **pptr++, *pptr++);
printf ("\n %d %d %d", **pptr, **pptr--, **pptr--);

getch();
return 0;
}
```

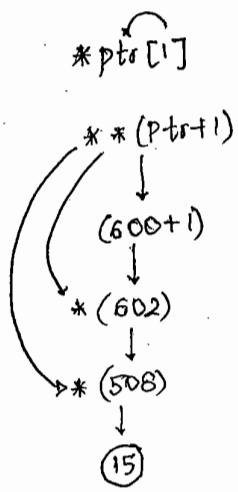


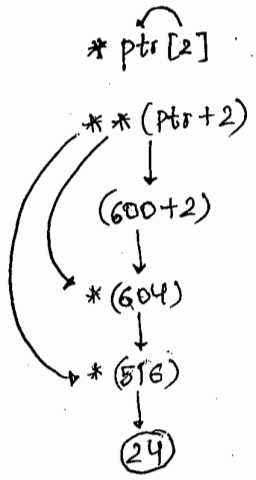


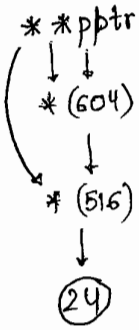


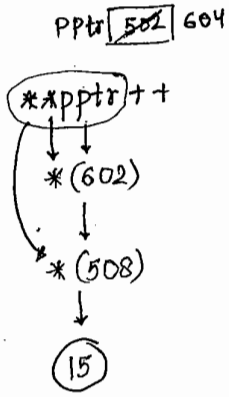


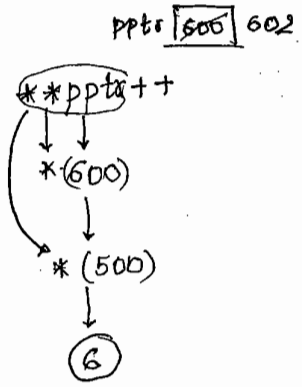

*ptr[0]
**(ptr+0)
(600+0)
*(600)
**(500)
6


*ptr[1]
**(ptr+1)
(600+1)
*(602)
**(508)
15

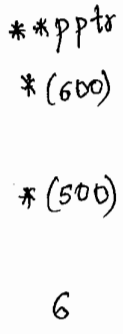

*ptr[2]
**(ptr+2)
(600+2)
*(604)
**(516)
24

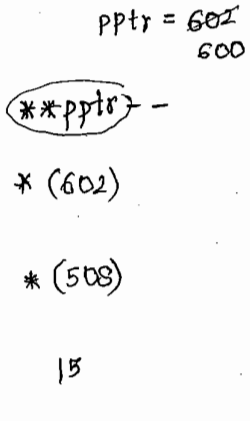

**pptr
*(604)
*(516)
24

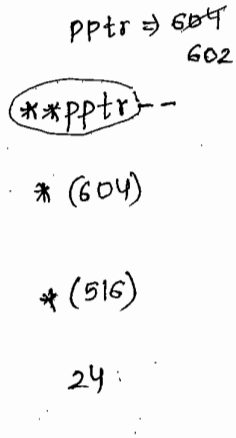

pptr 502 604
**pptr++
*(602)
*(508)
15


pptr 600 602
**pptr++
*(600)
*(500)
6


pptr
600
602
604


**pptr
*(600)
*(500)
6

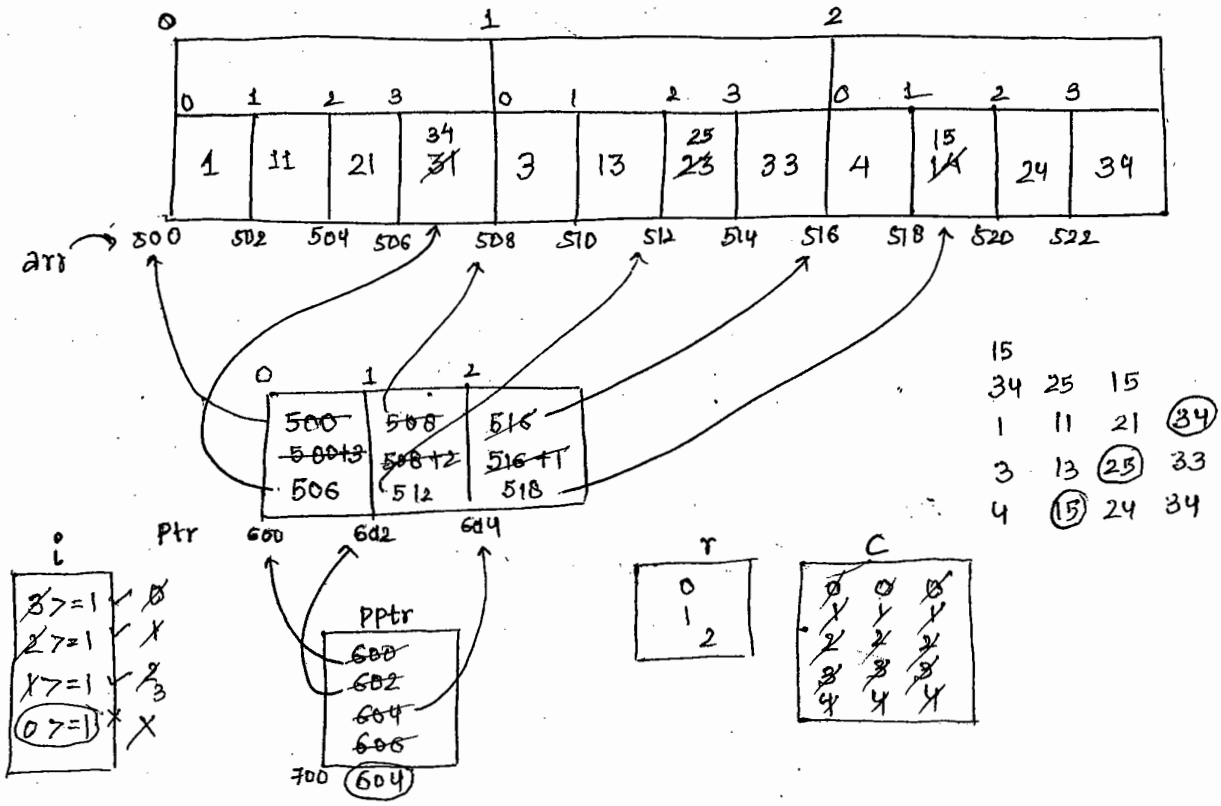

pptr = 602
600
**pptr--
*(602)
*(508)
15


pptr ⇒ 604
602
**pptr--
*(604)
*(516)
24

Left ← Right

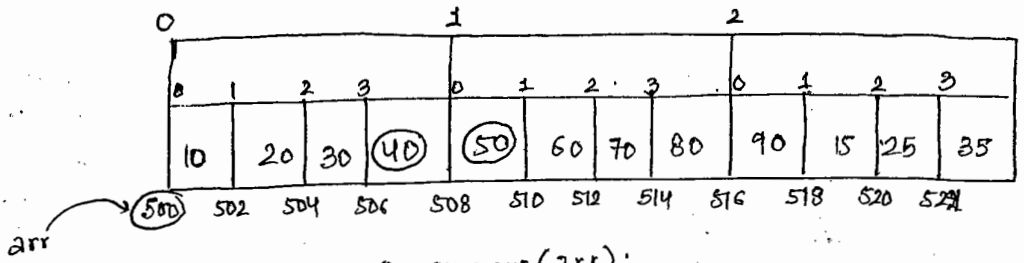
20/7/2015

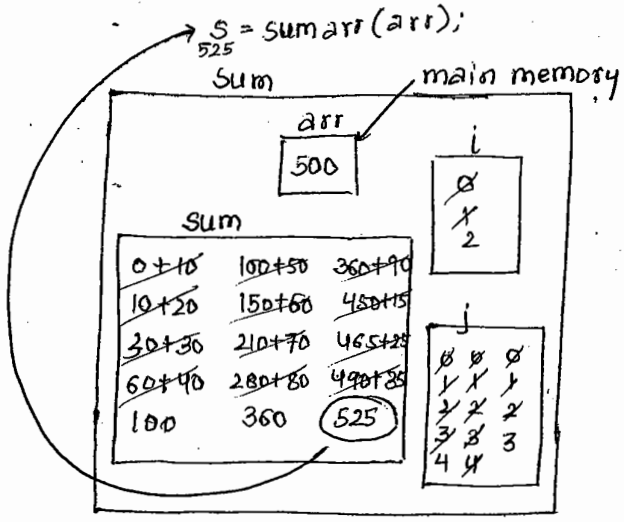
```
#include <stdio.h>
#include <conio.h>
int main()
{
  int arr[][4] = {
                   {1, 11, 21, 31},
                   {8, 18, 23, 33},
                   {4, 14, 24, 34}
                  };
 int *ptr[3];
 int**pptr;
 int i, r, c;
 clrscr();
 ptr[0] = &arr[0][0];   //arr;      //arr[0]+0;
 ptr[1] = &arr[1][0];   //arr+1;    //arr[1]+0;
 ptr[2] = &arr[2][0];   //arr+2;    //arr[2]+0;
 pptr = ptr;            //&ptr[0]
for(i=3; i>=1; i--)
{
  *pptr++ = i    //*pptr = *pptr+i;
 **pptr++ = i    //**pptr = **pptr+i;
 ++pptr;
}
--pptr;
 printf("%d\n", **pptr);
 for(i=0; i<3; i++)
  printf(" %d", *ptr[i]);
 //printf("%d", **(ptr+i));
 for(r=0; r<3; r++)
 {
  printf("\n");
  for(c=0; c<4; c++)
   printf("%3d", arr[r][c]);
 //printf("%3d", *(*(arr+r)+c));
}
getch();
 return 0;
```




```
#include <stdio.h>
#include <conio.h>
#define rsize 3
#define csize 4
int sumarr(int arr[][csize])
{
 int i, j, sum=0;
 for(i=0; i<rsize; i++)
  {for(j=0; j<csize; j++)
    {sum += arr[i][j];    //sum = sum+*(*(arr+i)+j);
    }
  }
 return sum;
}
int main()
 {
   int arr[rsize][csize];
   int r, c, s;
   clrscr();
   printf("Enter %d*%d values: ", rsize, csize);
```

```
for (r=0; r<rsize; r++)
for (c=0; c<csize; c++)
scanf ("%d", &arr[r][c]);
s = sumarr(arr);
printf ("Sum value of list: %d", s);
getch();
return 0;
}
```





i=0; j=0
arr[i][j]
((arr+i)+j)
(500+0)
*(500)
(500+0)
*(500)
10

i=0; j=3;
arr[i][j]
((arr+i)+j)
(500+0)
*(500)
(500+3)
*(506)
40

i=1; j=0;
arr[i][j]
((arr+i)+j)
(500+1)
*(508)
508+0
*(508)
50

i=1; j=3;
arr[i][j]
((arr+i)+j
(500+1)
*(508)
(508+3)
*(514)
80

i=2; j=3;
arr[i][j]
((arr+i)+j
(500+2)
*(516)
(516+3)
*(522)
35

MULTIPLICATION OF TWO MATRICES

```
# include <stdio.h>
# include <conio.h>
# define rsize 10
# define csize 10
int main()
{
  int m1[rsize][csize],
  int m2[rsize][csize],
  int m3[rsize][csize];
  int r1, c1, r2, c2, r3, c3, c, r, t;
  clrscr();
  printf("\n ENTER MAT1 DETAILS: ");
  printf("\n ENTER NO OF ROWS: ");
  scanf("%d", &r1);
  printf("\n ENTER NO. OF COLUMNS:");
  scanf("%d", &c1);
  printf("\n ENTER MAT2 DETAILS: ");
  printf("\n ENTER NO. OF ROWS: ");
  Scanf("%d", &r2);
  printf(" ENTER NO. OF COLUMNS: ");
  scanf("%d", &c2);
  If (c1 != r2)
  {
   printf("\n Error: Input data");
   getch();
   return 1;
  }
printf("\n ENTER MAT1 %d*%d ELEMENTS: ", r1, c1);
 for(r=0; r< r1 ; r++)
 for (c = 0; c <c1; c++)
  scanf("%d", &m1[r][c]);
 printf("\n ENTER MAT2 %d*%d ELEMENTS:", r2, c2);
  for(r=0; r<r2 ; r++)
  for(c=0; c<c2; c++)
  scanf("%d", &m2[r][c]);
printf("\n THE RESULT\n");
r3 = r1;
c3 = c2;
```

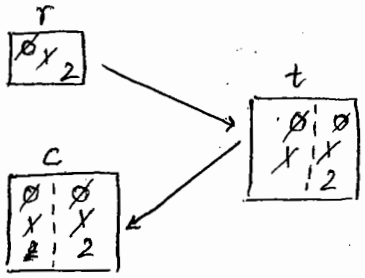
O/P:-

```
ENTER MAT1 DETAILS:
ENTER NO OF ROWS: 2

ENTER NO OF COLUMNS: 2
&
ENTER MAT2 DETAILS:
ENTER NO. OF ROWS: 2
ENTER NO. OF COLUMNS: 2

ENTER MAT1 2*2 ELEMENTS:
 2  3
 4  1

ENTER MAT 2*2 ELEMENTS:
 4  1
 2  3

THE RESULT
 14  11
 18  7
```

```
for (r=0; r<r3; r++)
{
 printf ("\n");
 for(c=0; c<c3; c++)
  {
   m3[r][c] = 0;
   for (t=0; t<c1; t++)
    m3[r][c] += m1[r][t] * m2[t][c];
    printf("%3d", m3[r][c]);
  }
}
getch();
return 0;
}
```

r1	c1	r2	c2	r3	c3
2 ✓	2	2	2 ✓	2	2

$$m1\begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix} * m2\begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix} = m3\begin{bmatrix} gr & gr \\ 0+8+6 \ (14) & 0+2+9 \ (11) \\ gr & gr \\ 0+16+2 \ (18) & 0+4+3 \ (7) \end{bmatrix}$$



21st July 15

3D ARRAY

- In 3D array elements are arranged in blocks, rows & column format.
- When we are working with 3D array we required to use three subscript operator which indicates block size, row size & column sizes.
- The main memory of 3D array is blocks, sub main memory is rows & elements are available in columns.

- On 3d array when we are applying 2 subscript operators then it gives rowname, row name always provides corresponding row address.
- On 3d array when we are using one subscript operator then it gives blockname, blockname always provides corresponding block address.

- from 3d array, when we required to access the data then 3 subscript operators required to use
- The Main memory of 3d array are blocks, that's why arrname always provides main memory address, arr+1 will provide next main m/m of array.

Syntax :- `Datatype arrName [BSize][Rsize][Csize]`

Properties of 3d array

1. int arr[2][3][4];
   size → 2*3*4
   sizeof(arr) → 48B

2. int arr[2][2][2];
   size → 2*2*2
   sizeof(arr) → 16B
   2 blocks
   each block → 2 rows
   each row → 2 columns
   8 int variable

arr[0][0][0]	→ 1	arr[1][0][0]	→ 5
arr[0][0][1]	→ 2	arr[1][0][1]	→ 6
arr[0][1][0]	→ 3	arr[1][1][0]	→ 7
arr[0][1][1]	→ 4	arr[1][1][1]	→ 8

3. int arr[][][]; Error
4. int arr[][][3]; Error
5. int arr[][2][3]; Error

- In declaration of 3d array, mandatory to specify block size, row size and column sizes.

6. int arr[2][2][2] = {10, 20, 30, 40, 50, 60, 70, 80};

arr[0][0][0]	→ 10	arr[1][0][0]	→ 50
arr[0][0][1]	→ 20	arr[1][0][1]	→ 60
arr[0][1][0]	→ 30	arr[1][1][0]	→ 70
arr[0][1][1]	→ 40	arr[1][1][1]	→ 80

- By using above initialization process, we required to initialize all elements in sequence only i.e selected no. of elements we can't initialize.

7. int arr[2][3][4] = {

```
{
    {10,20},
    {30},
    {40,50,60}
},
{
    {70,80,90,15},
    {25,35},
    {10}
}
};
```

- In initialization of 3d array, if specific no. of elements are not initialized then remaining all elements are automatically initialised with zero.

8. int arr[][][] = {

```
{
  {10,20},
  {30},
  {40,50,60}
},
{
  {70},
  {80,90}
}
};
```

Error

9. int arr[][][3] = {

```
{
  {10},
  {20,30},
},
{
  {40,50,60},
  {70},
}
};
```

Error

10. int arr[][2][3] = {

```
{
  {10,20,30}
  {40,50}
},
{60},
{70,80}
}
};
```

Valid

- In initialisation of 3d array, specifying the block size is optional. but row and column sizes are mandatory.

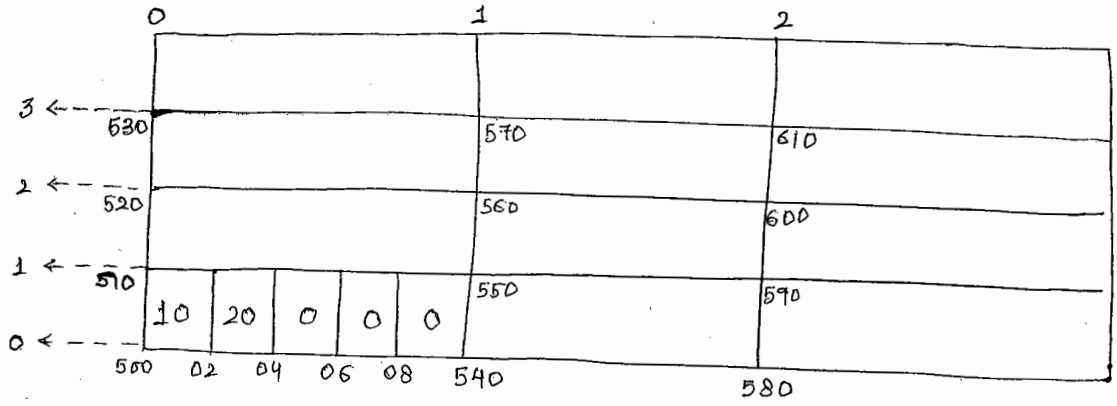
4D ARRAY

- In 4D Array, elements are arranged in sets, blocks, rows and column format.
- When we are working with 4D array, we required to use 4 subscript operators which indicates set size, block size, row size & column size.
- The Main memory of 4d array is sets, next main memory is blocks, sub-main memory is rows & elements are available in columns.
- On 4D array when we are using 1 subscript operator, then it gives set name, set name always provides corresponding set address
- On 4D array, when we are using 2 subscript operators, then it gives block name, block name always provides corresponding block address.
- On 4D array, when we are using 3 subscript operators, then it gives row address, 4 subscript operator provides elements.
- In declaration of 4D array, mandatory to specify all information, in initialization, specifying the block address is optional

syntax: `Datatype arr[SSIZE][BSIZE][RSIZE][CSIZE];`

Memory Management in 3D Array or Multidimensional Array

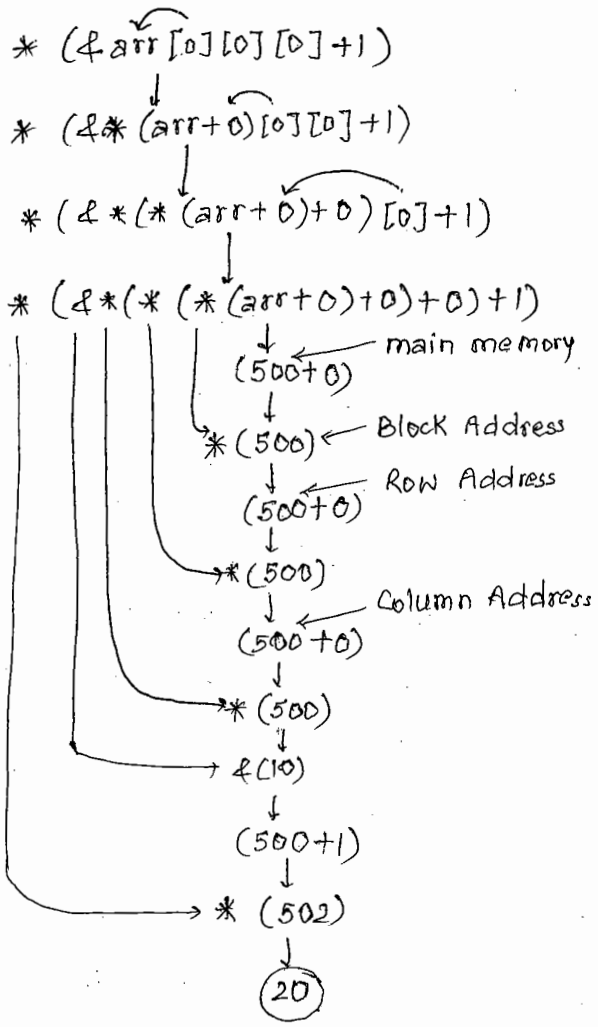
* `int arr[3][4][5] = {10, 20};`



1) arr → 500
2) arr[0] → 500
3) arr[0]+1 → 510
4) arr[0][0] → 500
5) arr[0][0]+1 → 502
6) arr[0][0][0] → 10

1) arr+1 → 540
2) arr[1] → 540
3) arr[1]+1 → 550
4) arr[1][1] → 550
5) arr[1][1]+1 → 552
6) arr[0][0][0]+1 → 11

1) arr+2 → 580
2) arr[2] → 580
3) arr[2]+1 → 590
4) arr[2][2] → 600
5) arr[2][2]+2 → 604
6) &arr[0][0][0]+1 → 502



CHARACTER OPERATIONS IN 'C'..

- In 'C' programming lang, we are having 256 characters
- This all 256 characters are represented with the help of an integer value called Ascii value.
- The range of Ascii value is −128 to +127
- When we are working with characters, the representation of character must be within the single quotes only
- Within ' ' any content is called character constant
- The size of character constant is 2 bytes because by default it returns an integer value i.e Ascii value of a character constant

SPECIAL CHARACTERS IN C

	Ascii value
'A'	A 65
'a'	a 97

'Z'	90	Z
'z'	122	z
'0'	48	0
'9'	57	9
'\b'	8	backspace
'\r'	13	carriage return
' '	space 32	
'\t'	9	tab
'\n'	10	new line
'\\'	\ (92)	
'\''	' (39)	
'\"'	" (34)	
'\a'	beep 7	
"%%"	% 37	
'\%'	% 37	
';'	; 59	
':'	: 58	

1. printf ("Welcome"); Welcome
2. printf ("\"Welcome\""); "Welcome"
3. printf ("%d %d", 'A', 'a'); 65 97
4. printf ("%d %d", 'Z', 'z'); 90 122
5. printf ("%c %c", 68, 100); D d
6. printf ("%c %c", 68, 100); D d
7. printf ("%d %c", 'd', 100); 100 d
8. printf ("%d %d", 'A'+32, 'a'-32); 97 65
9. printf ("%c %c", 'A'+32, 'a'-32); a A
10. printf ("\'Welcome\'"); Welcome
11. printf ("abc\xyz"); abc
xyz
12. printf ("abc\\xyz"); abc\xyz

13. printf ("abc\\\\xyz"); abc\\
xyz

14. printf ("abc %xyz", 10); abc
xyz

15. printf ("abc %d xyz.", \n); abc 10 xyz

16. printf ("abc\t xyz"); abc xyz

17. printf ("abc %xyz", '\t'); abc 9xyz

18. printf ("abc %c xyz", a); abc xyz

19. printf ("Hello \b"); Hello

20. printf ("Hello\b\babc"); Helabc

21. printf ("Hello\b\b\b\b\b abc"); abcLo

22. printf ("Welcome\r"); Welcome

23. printf ("Welcome\r Hello"); Hellome

24. printf ("Hello\r Welcome"); Welcome

25. printf ("%d %d %d" 10); %d

26. printf ("% %d = %d", 10); %d = 10

27. printf ("Hello \a"); Hello beep (sound)

Buffer Concept :-

- Temporary storage area is called buffer.
- All standard I/O devices contain temporary memory those are called standard I/O buffer.
- In Implementation when we are passing more than required number of elements then automatically remaining all values are stored in standard input buffer and these values automatically will pass to next input functionality.

```
# include <stdio.h>
# include <conio.h>
int main()
{
   int v1, v2;
```

```
clrscr();
printf ("\n Enter value of V1 : ");
scanf (" %d ", &v1);
printf ("\n Enter value of V2: ");
scanf ("%d ", &V2);
printf ("\n value 1 : %d  Value 2 : %d", V1, V2);
getch ();
return 0;
}
```

O/P:

```
Enter Value of V1 : 10
Enter Value of V2 : 20
Value 1 : 10   value 2 : 20
```

O/P:

```
Enter value of V1 : 10 20
Enter value of V2 :
Value 1 : 10    Value 2 : 20
```

➤ In implementation when we required to remove the data from standard input buffer then recommended to go for flushall() or flush() function.

flushall():- It is a predefined function which is declared in stdio.h, By using this function, we can delete the data from standard input buffer.

→ flushall() function doesn't required any parameters and doesn't return any value also.

Syntax: `void flushall (void);`

Fflush():- It is a predefined function which is declared in stdio.h, by using this function we can clear the data from standard input buffer.

→ Fflush() function requires 1 argument of type FILE* and returns void type data

Syntax: `void fflush (FILE *stream);`

```
# include <stdio.h>
# include <conio.h>
int main ()
{
    int V1, V2 ;
     clrscr ();
     printf ("\n Enter the value of V1 : ");
     scanf ("%d", &V1);
     printf ("\n Enter the value of V2 : ");
     flushall ();          // fflush (stdin);
     printf scanf ("%d ", &V2);
```

O/P:

```
Enter the Value of V1 : 10 20 30
Enter the value of V2 : 40
Value 1 :   Value 2 : 40
```

```
    printf ("\n Value1: %d   Value2 : %d", V1, V2);
    getch();
    return 0;
}
```

Prog -
```
#include <stdio.h>
#include <conio.h>
int main()
{
    char ch1, ch2;
    clrscr();
    printf ("\n Enter char1: ");
    //scanf("%c", &ch1);
    printf ("\n Enter char2: ");
    //fflush (stdin)
    //scanf ("%c", &ch2);
    // ch2 = getchar();
    ch2 = getch();
    printf ("\n char1 : %c   char2 : %c", ch1, ch2);
    getch();
    return 0;
}
```

OUTPUT:

```
Enter char1: A
Enter char2:
char1: A     char2: B
```

→ by using scanf function we can't read multiple characters properly bcoz it reads the data from standard input buffer.

→ In implementation when we are working with multiple characters then always recommended go for getche(), or getch() function.

getche():-

- It is a predefined unformatted function which is declared in conio.h, by using this function we can read a character directly from keyboard. getche() function returns an integer value i.e Ascii value of input character.

getch():-

- It is a predefined unformatted function which is declared in conio.h, by using this function we can read a character directly from Keyboard

getch() function returns an integer value i.e Ascii value of input character.

Syntax:-

```
int getch (void);
```

NOTE:- The basic difference between getche() and getch() function is

When we are reading the character by using getche() function then it displays what character will pass but if we are using getch() function then it will not display

getchar(), putchar():-

→ getchar() is a predefined macro which is defined in stdio.h.

→ By using getchar() macro we can read a character from standard input buffer.

→ getchar() macro returns an integer value i.e Ascii value of a input character

Syntax:

```
int getchar(void);
```

- putchar() is a predefined macro which is defined in stdio.h by using this macro we can print a character on console.
- putchar() macro required 1 argument of type integer i.e Ascii value of a printing character.

Syntax:

```
int putchar(int ch);
```

- When we are working with putchar() macro what character will print same character Ascii value is return back.

```
#include <stdio.h>
#include <conio.h>
int main()
{
char ch1, ch2;
clrscr();
printf("Enter a char: ");
```

```
    ch1 = getchar();
    ch2 = putchar(ch1);
    printf ("\nchar1 : %c    char2 : %c", ch1, ch2);
    getch();
    return 0;
}
```

Output:

```
Enter a char : A
A <------ ch2 = putchar(ch1);
char1 : A   char2 : A
```

Numeric Value to Integer :-

```
# include <stdio.h>
# include <conio.h>
  int main()
  {
     char ch  int value;
     clrscr();
     printf ("Enter a value (0-9)");
     ch= getchar();
     if (ch>='0' && ch<='9'),
        {
          value = ch - '0';
          printf ("input value is : %d", value);
        }
      else
          printf ("invalid input data");
          getch();
          return 0;
      }
```

O/P:-
```
Enter a value (0-9): 5
Input value is : 5
```

value = ch - '0'
= '5' - '0'
= 53 - 48
= 5

Read Only Variables -

- Constant variables are called Read-only variables
- When we are applying const modifier to a variable which enables read only property.
- When we are applying read only property on a variable then it doesn't allows to modify the value by using variable name.

Ex:-

```
#include <stdio.h>
#include <conio.h>
int main()
{
    int a = 10;
    ++a;
    printf("a = %d", a);
    return 0;
}
```

O/P: a = 11

```c
#include <stdio.h>
int main()
{
  int a = 10;
  ++a;
  printf("a = %d", a);
  return 0;
}
```

O/P: a = 11

```c
#include <stdio.h>
int main()
{
  const int a = 10;
  ++a;
  printf("a = %d", a);
  return 0;
}
```

O/P: Error

→ const int a ;

'a' is a variable of type constant integer.

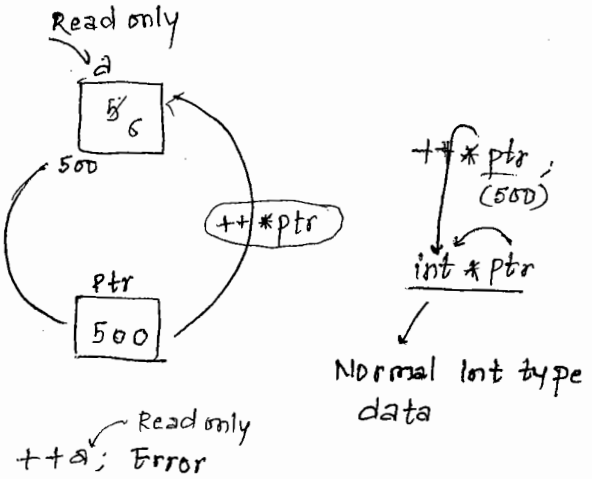
→ int const a;

'a' is a variable of type integer constant.

- If the variable is constant integer or integer constant both are having read only properties so doesn't allows to modify the data by using Variable name.

```c
#include <stdio.h>
int main()
{
  int const a = 5;
  int *ptr;
  ptr = &a;
  //++a; Error
  ++*ptr;
  printf("a = %d", a);
  return 0;
}
```

O/P: a = 6



* `int *ptr;`

- Ptr is called pointer to integer
- When we are working with pointer to integer then it allows to modifies normal and read only variable data also.
- In implementation, when we are working with read only variables then recommend to go for pointer to constant integer or pointer to integer constant variables.

* `const int *ptr;`

ptr is called pointer to constant integer

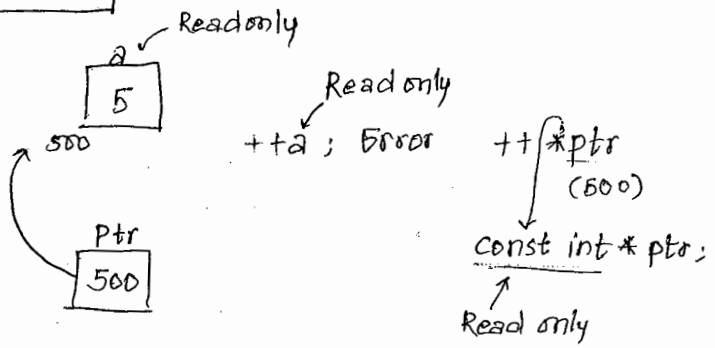
* `int const *ptr;`

ptr is called pointer to integer constant.

➤ When we are working with pointer to constant integer or pointer to integer constant then it doesn't allows to modify the data even though it is normal or read only.

```
#include <stdio.h>
int main()
{
  int const a = 5;
  const int *ptr;
  ptr = &a;
  ++a;     Error
  ++*ptr;  Error
  printf("a = %d", a);
  return 0;
}
```

O/P: Error



```
#include <stdio.h>
int main()
{
  int a = 5;
  int const *ptr;
  ptr = &a;
  //++a; yes valid
  ++*ptr
  printf("%d", a);
  return 0;
}
```

O/P: Error

```
#include <stdio.h>
#include <conio.h>
int main()
{
  const int arr[2] = {5,15};
  int *ptr;
  ptr = &arr[0];
++ptr;
  ++*ptr;
  --ptr;
  --*ptr;
  printf("%d %d", arr[0], arr[1]);
  getch()
  return 0;
}
```

O/P: 4 16

When we are working with pointer to integer variable then it doesn't having any limitation i.e it is possible to modify pointer value and object value also.

→
```
#include <stdio.h>
#include <conio.h>
int main()
{
  int const arr[2] = {10, 20}
  const int *ptr;   → here object is constant
                      pointer to integer constant
  ptr = &arr[1];
  --ptr;        yes
  --*ptr;       Error
  ++ptr;        yes
  ++*ptr        Error
  printf("%d %d", arr[0], arr[1]);
  getch();
  return 0;
}
```

- When we are working with pointer to constant integer or pointer to integer constant then object modification is restricted but pointer modification is allowed.
- In implementation when pointer modification is required to be restricted then go for constant pointer type.
- When we are working with constant pointer type, then always recommended to initialise pointer variable.

`int *const ptr;`

→ ptr is called constant pointer to integer.

```
# include <stdio.h>
# include <conio.h>
 int main()
{
 const int arr[2] = {2, 12};
 int *const  ptr = &arr[0]
 ++ptr;        Error
 ++*ptr;       yes
 --ptr;        Error
 --*ptr        yes
 printf ("%d %d", arr[0] ,arr[1]);
 getch()
 return 0;
}
```

- When we are working with constant pointer to integer, then it doesn't allows to modify pointer value but it is possible to change object value
- In implementation, when we are required to be restricted, pointer modification and object modification then go for constant pointer to constant integer or constant pointer to integer constant.

* `const int *const ptr;`

  ptr is called constant pointer to constant integer

* `int const *const ptr;`

  ptr is called constant pointer to integer constant.

- By using this type of pointers, we can't perform any kind of operations xcept accessing the data

```
# include <stdio.h>
# include <conio.h>
 int main()
 {
   const int arr[2] = {10, 20};
   const int *const ptr = &arr[0];
   ++ptr;      error
   ++*ptr      error
   --ptr;      error
   --*ptr      error
  printf ("%d %d", arr[0], arr[1]);
  getch();
}
```

STRINGS

- Character array or group of characters or collection of characters are called Strings.
- In implementation when we are manipulating multiple characters, then recommended to go for strings.
- Within the ' ' any content is called character constant, within the " " any content is called string constant.
- character constant always returns an integer value i.e ASCII value of a character
- String constant always returns base address of a string
  When we are working with string constant, always ends with nul('\0')
- The representation of null character is nul('\0') and Ascii value is 0

Syntax:

```
char str [SIZE];
```

NOTE: NULL is a global constant value which is defined in <stdio.h>

- NULL is a macro which is having the replacement data as 0 or (void *)0

  Ex: int x = NULL ; int *ptr = NULL

- nul(\0) is a ASCII character data which is having Ascii value as 0

PROPERTIES OF STRINGS

1. char str [5] ;
   size ---> 5
   sizeof (str) ---> 5B
2. char str[4];
   size ---> 4
   size of (str) ---> 4B

   4 char variables
   str [0] → 1
   str [1] → 2
   str [2] → 3
   str [3] → 4
3. char str[0] ; Error
4. char str [-5]; Error
5. char str [] ; Error

- In declaration of string size must be unsigned integer constant whose value is greater than zero only.

6. `char Str [5] = {a,b,c,d,e};` Error

7. `char str [5] = {'A','B','C','D','E'};`

A ---> str[0]
B ---> str[1]
C ---> str[2]
D ---> str[3]
E ---> str[4]

8. `char str [5] = {'A','P','P'};`

str[0] ---> A
str[1] ---> P
str[2] ---> P
str[3], str[4] ---> \0(nul)

- In initialization of the string, specific characters are not initialized then remaining all elements are automatically initialised with \0(nul).

9. `char str [3] = {'A','P','P','L','E'};` Error

In initialization of the string, it is not possible to initialize more than size of string elements

10. `char str [5] = {100, 65, 97, 48, 57};`

str[0] ---> d (100)
str[1] ---> A (65)
str[2] ---> a (97)
str[3] ---> 0 (48)
str[4] ---> 9 (57)

- In initialisation of the string, if we are assigning numeric value, then according to Ascii value, corresponding data will be stored.

11. `char str [] = {'H','E','L','L','O'};` valid

size ---> 5
sizeof(str) ---> 5B

- In initialisation of the string, specifying the size is optional, in this case, how many characters are initialised that many variables are created.
- When we are working with strings, always recommended to initialise the data in double quotes only.

12. char str[10] = "Hello";

13. char str[] = "Hello";

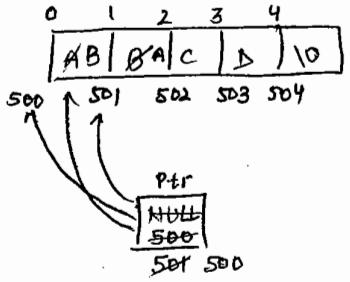
size --> 5

sizeof (str) --> 6B

- When we are working with string constant, always it ends with \0 (nul) character thats why one extra byte m/m is required but if we are working with character array then it doesn't require one extra byte memory.

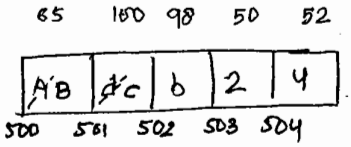
```
# include<stdio.h>
# include<conio.h>
 int main()
 {
  char str[5] = {'A', 'B', 'C', 'D', '\0'};
  char near *ptr = (char near*) NULL;
  ptr = &str[0];
  ++ptr;
  --*ptr;
  ++*ptr;
  printf("%c %c%c", str[0], str[2], str[1]);
  getch();
  return 0;
 }
```

O/P: BCA



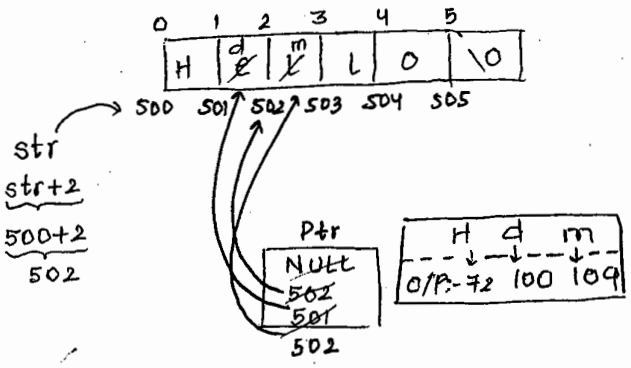
O/P: B C A

→
```
# include<stdio.h>
# include<conio.h>
 int main()
 {
  char str[5] = {'A', 100, 'b', 50, '4'}
  char * ptr = (char*) NULL;
  ptr = &str[1];
  --ptr;
  ++*ptr;
  ++ptr;
  --*ptr;
  printf("%c %c%d", str[0], str[1], str[2]);
  getch();
  return 0;
 }
```



Prog3

```
# include <stdio.h>
# include <conio.h>
int main()
{ char str[] = "Hello";
  char *ptr = NULL;
  ptr = str + 2;
  --ptr;
  --*ptr;
  ++ptr;
  ++*ptr;
  printf ("%d %d %d", str[0], str[1], str[2]);
  getch();
  return 0;
}
```



Prog4

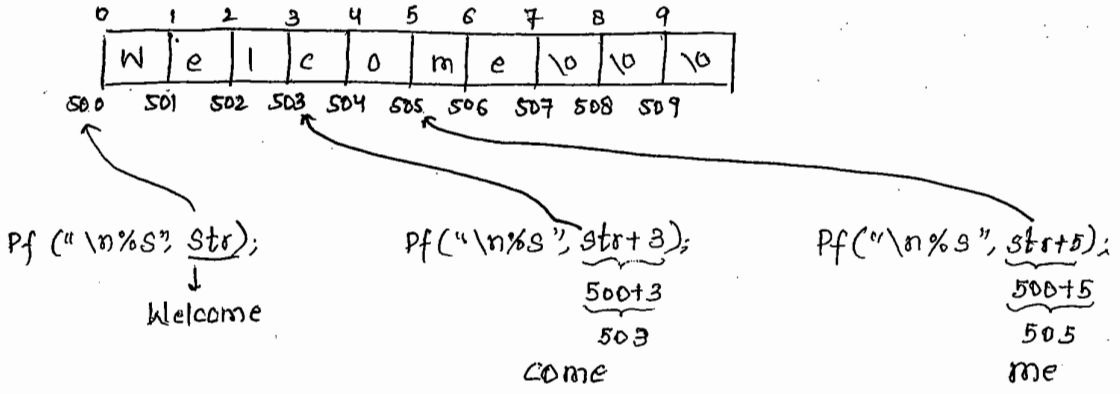
```
# include <stdio.h>
# Include <conio.h>
int main ()
{
  char str [] = "Hello"
  printf ("%c %c %c %c %c", str[0], str[1], str[2], str[3], str[4]);
  getch();
  return 0;
}
```

O/P : Hello

- When we are working with character operations recommended to go for %c format specifier.
- When we are working with string operations recommended to go for %s format specifier
- When we are working with %s format specifier then we required to pass an address of a string, from given address upto null, entire content will print on console

```
# Include <stdio.h>
# Include <conio.h>
int main()
{
  char str [10] = "Welcome";
  printf ("\n %s", str);
  printf ("\n %s", str+3);
  printf ("\n %s", str+5);
  getch();
  return 0;
}
```

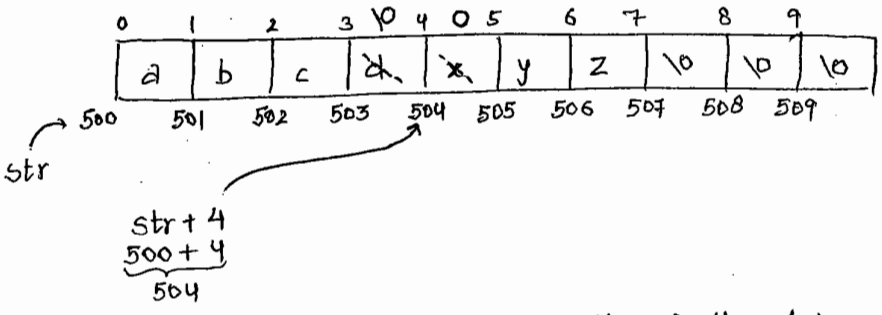
O/P : Welcome
come
me




```
# include <stdio.h>
# include <conio.h>
int main()
{ char str[10] = "abcdxyz";
  printf("\n%s", str);
  printf("\n%s", str+4);

  str[3] = 0;   → character value is \0
                  str[3] = \0
  printf("\n%s", str);
  printf("\n%s", str+4);

  str[3] = '0';     //str[3] = 48 (Ascii value)
  str[4] = '\0';
  printf("\n%s", str);
  printf("\n%s", str+5);
  getch();
  return 0;
}
```

O/P:-

```
abcdxyz
xyz
abc
xyz
abc0
yz
```



- When null character is occurred in the middle of the string, then we are not able to print complete data bcz null character indicates termination of string.

```c
# include <stdio.h>
# include <conio.h>
int main()
{
 char str[] = "Welcome";
 puts(str);
 printf("%s", str);
 return 0;
}
```

O/P:
```
Welcome
Welcome
```

puts() method have inbuilt new line character (JAVA)

puts() – It is a predefined unformatted function, which is declared in stdio.h

- By using this function, we can print string data on console.
- puts() function required 1 argument of type char* and returns an integer value.
- When we are working with puts function, automatically it's prints new line character after printing string data.

Syntax:

```
int puts(char *str);
```

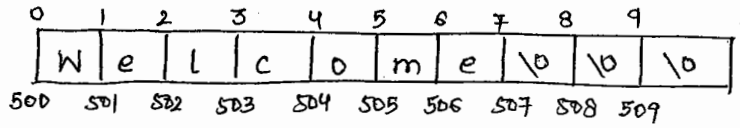
NOTE: Functions that works with the help of format specifier and which can be applied for any datatype, these are called formatted function Eg :- printf(), scanf(), fprintf(), fscanf(), cprintf(), cscanf();

→ Functions that doesn't require any format specifier and which is required for specific datatype are called Unformatted function
Eg :- puts(), gets(), getche(), cputs(), cgets()

```c
# include <stdio.h>
# include <conio.h>
int main()
{
 char str[10] = "Welcome";
 int i;
 clrscr();
 for(i=0; str[i] != '\0'; i++)
  puts(str+i);
 getch();
 return 0;
}
```

O/P:
```
Welcome
elcome
lcome
come
ome
me
e
-------
```
1 Blank line

at i = 7 condition becomes false

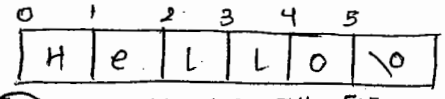


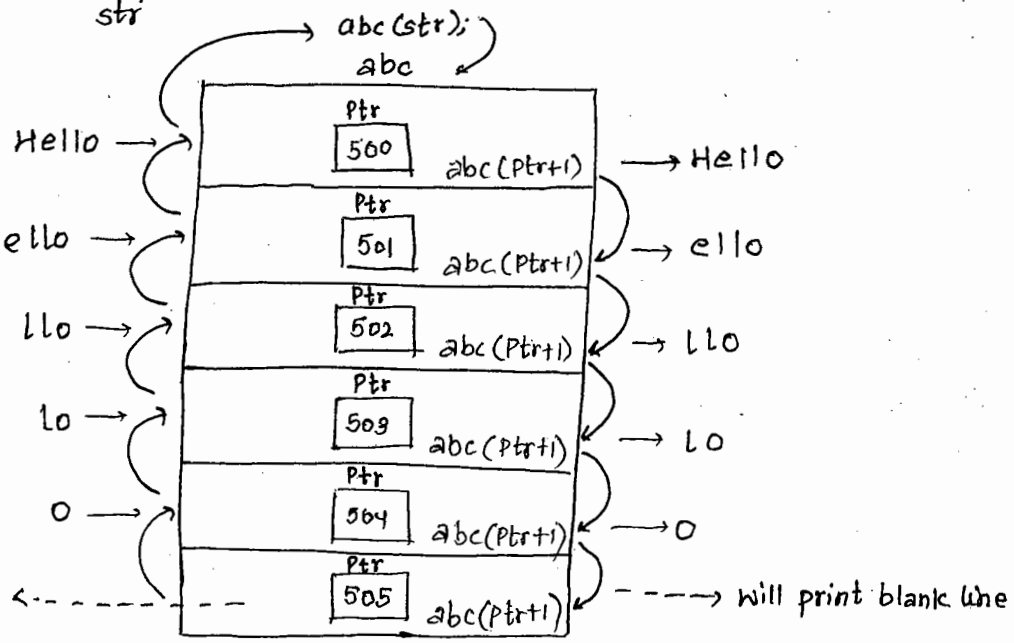
i=0;	i=1;	i=2;	i=3;	i=4;	i=5	i=6
puts(str+i)	puts(str+1)	puts(str+1)	puts(str+i)	puts(str+i)	puts(str+1)	puts(str+i)
↓ (500+0)	↓ (500+1)	↓ (500+2)	↓ (500+3)	↓ (500+4)	↓ (500+5)	↓ (500+6)
puts(500)	puts(501)	puts(502)	puts(503)	puts(504)	puts(505)	put(506)
↓	↓	↓	↓	↓	↓	↓
Welcome	elcome	Lcome	come	ome	me	e

```
# include <stdio.h>
# include <conio.h>
 void abc (char*ptr)
{ puts (ptr);
  if (*ptr)
   abc (ptr+1);
   puts (ptr);
}
int main ()
 {
   char str[] = "Hello";
   clrscr ();
   abc (str);
   getch ();
   return 0;
}
```

O/p:

```
He l lo
e l l o
l l o
l o
o
---
o
l o
l l o
e l l o
H e l L o
```





```c
#include <stdio.h>
int main()
{
char str[] = "Rama Rao";
puts(str);
printf("%s", str);
return 0;
}
```

O/P:
```
Rama Rao
Rama Rao
```

```c
#include <stdio.h>
#include <conio.h>
int main()
{
char str[10];
clrscr();
printf("Enter a string: ");
scanf("%s", str);
//gets(str);
printf("input string:%s", str);
getch();
return 0;
}
```

O/P:
```
Enter a string : Naresh IT
input string : Naresh
```

- By using scanf function, we can't read the string data properly when we have multiple works because in scanf function space, tab and newline characters are treated like seperators so when the seperator is present, it is replaced with \0 character
- In scanf function, when we are using %[^\n]s format specifier, then it indicates that read the string data upto newline character occurrence.

gets():-

- It is a predefined unformatted function which is declared in stdio.h.
- By using this function we can use read the string data properly, when we are having get even multiple words.

gets() function requires one argument of type (char*) & returns (char*) only
In gets() function only newline character is treated as seperator.

Syntax: `char* gets(char *str);`

```
#include <stdio.h>
#include <conio.h>
int main()
{
 char s1 [10] = "Hello";
 char s2 [10] = "Welcome";
 clrscr();
 puts (s1);
 puts (s2);
 s2 = s1;        // Address = Address;
 puts (s1);
 puts (s2);
 getch();
 return 0;
}
```

O/P: Error L value required

- Any kind of string manipulations, we can't perform directly by using operators.
- In implementation when we required to perform any kind of string operations then recommended to go for any string handling functions or go for user defined function logic.
- String related pre-defined functions are declared in string.h

1. strcpy()
2. strlen()
3. strrev()
4. strcat()
5. strupr()
6. strlwr()
7. strcmp()
8. stricmp()
9. strstr()

1) strcpy()- By using this predefined function, we can copy a string to another string.

→ It requires 2 arguments of type (char*) and returns (char*) only.

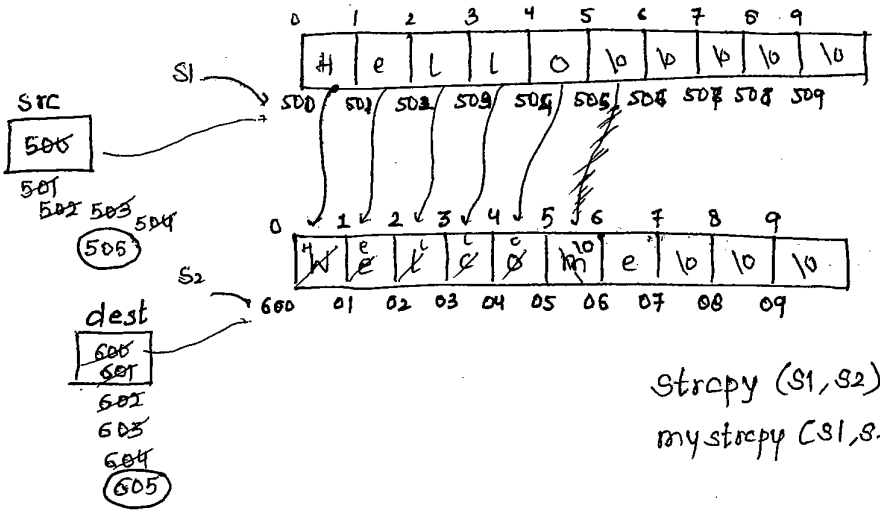
→ When we are working with strcpy() from given source address upto \0 entire content will be copied to destination string

Syntax: `char * strcpy (char * dest, const char * src);`

Prog-
```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{
  char s1[10] = "Hello";
  char s2[10] = "Welcome";
  clrscr();
  puts(s1);
  puts(s2);
  strcpy(s2, s1);
  puts(s1);
  puts(s2);
  getch();
  return 0;
}
```

O/p: 1. strcpy(s2, s1);
```
Hello
Welcome
Hello
Hello
```
2. strcpy(s2, s1+2);
```
Hello
Welcome
Hello
WeHello
```
3. strcpy(s2+2, s1);
```
Hello
Welcome
Hello
WeHello
```
4. strcpy(s2+2, s1+2);
```
Hello
Welcome
Hello
Wello
```

Copying a string to another string without using strcpy



strcpy(S1, S2);
mystrcpy(S1, S2);

```
#include <stdio.h>
#include <conio.h>
void mystrcpy(char *dest, const char *src)
{
  while(*src != '\0')
  {
    *dest = *src;
    ++src;
    ++dest;
  }
  *dest = '\0';
```

```
int main()
{
    char s1[10] = "Hello";
    char s2[10] = "Welcome";
    clrscr();
    puts(s1);
    puts(s2);
    mystrcpy(s2, s1);    // strcpy(s2, s1);
    puts(s1);
    puts(s2);
    getch();
    return 0;
}
```

2) strlen() – By using this predefined function, we can find length of string

→ strlen function requires 1 argument of type (const char*) and returns an int type.

→ When we are working with strlen() from given address upto ~~nul~~ \0, Entire character count value will return.

→ Length of the string means total no. of characters excluding \0 character

→ Size of the string means total no. of characters including \0 character

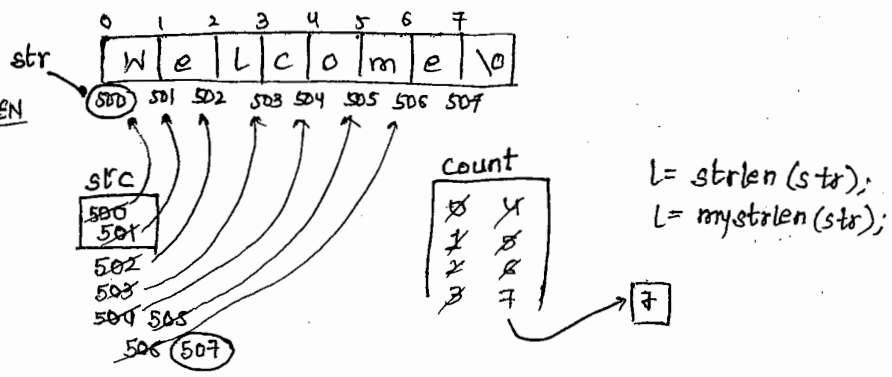
Syntax: `int strlen(const char* str);`

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
{
 char str[] = "Welcome";
 int l, s;
 clrscr();
 l = strlen(str);
 s = sizeof(str);
 printf("\n Length of string : %d", l);
 printf("\n size of string : %d", s);
 getch();
 return 0;
}
```

O/P:
```
Length of string : 7
size of string : 8
```

1. strlen(str); O/P: 7
2. strlen(str+3); O/P: 4
3. strlen(str+5); O/P: 2
4. strlen(str+7); O/P: 0

WITHOUT USING STRLEN FINDING LENGTH



USER-DEFINED CODE.

```
# include <stdio.h>
# include <conio.h>
# include <string.h>
 int mystrlen (const char * src)
{
 int count = 0
  while (* src != '\0')
  {
     ++ count;
     ++ src;
  }
   return count ;
}
 int main ()
  {
   char str [10];
   int s, l;
   clrscr ();
    printf ("\n Enter a string : ");
    gets (str);
    s = sizeof (str);
    l = mystrlen (str);
    printf ("\n size of string : %d", s);
    printf ("\n length of string : %d", l);
    getch ();
    return 0;
   }
```

O/P:
```
Enter a string : Welcome
size of string : 10
length of string : 7
```

3) strrev():- By using this predefined function, we can make string
→ strrev() requires 1 argument of type (char *) and returns (char *)
→ When we are working with strrev() from given address upto null entire string data will make reverse except null characters

Syntax: `char * strrev (char * str);`

```
# Include <stdio.h>
# include <conio.h>
# include <string.h>
  int main ()
  { char str [10] = "Welcome";
    clrscr ();
```

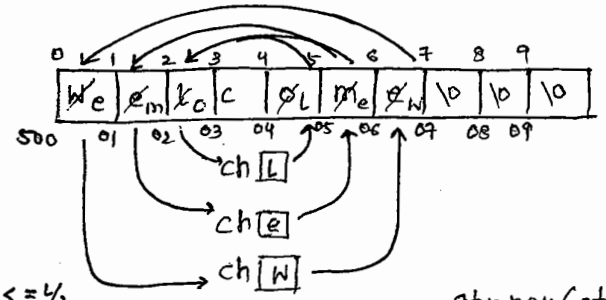
```
puts (str);
strrev(str);
printf ("Reverse string : %s", str);
getch();
return 0;
}
```

O/P:
```
Welcome
Reverse string : emocleW
```

1. strrev(str); emocleW
2. strrev(str+3); Welemoc
3. strrev(str+5); Welcoem
4. strrev(str+7); Welcome

FINDING REVERSE WITHOUT USING PRE-DEFINED FUNCTION strrev():-

strrev() is a dependent function.
1 more function is required i.e. strlen()



src
500

L
7

i<=l/2
0<3 ✓
1<3 ✓
2<3 ✓
3<3 ✗

```
strrev(str);
mystrrev (str);
```

USER DEFINED CODE

```
# include <stdio.h>
# include <conio.h>
# include <string.h>
Void mystrrev (char * src)
 {
    int i, l;
    char ch;
    l = strlen(src);
    for (i=0; i< l/2; i++)
    {  ch = src [i];
       src [i] = src [l-i-1];
       src [l-i-1] =ch;
    }
 }
 int main (void)
 {
   char str[10];
   clrscr();
printf ("\n Enter a string: ");
 gets (str);
 mystrrev (str);   // strrev (str)
 printf ("\n Reverse String : %s", str);
 getch();
 return 0;
 }
```

O/P:-
```
Enter a string: Welcome
Reverse string: emocleW
```

4) strcat() :- By using this predefined function, we can concatenate a string to another string

→ concatenation means copying data from end of the string i.e appending process.

→ strcat() requires 2 argument of type(char*) and returns(char*) only

Syntax : `char* strcat (char* dest, const char *src);`

→ When we are working with strcat() function, always appending will take place at end of the destination only.

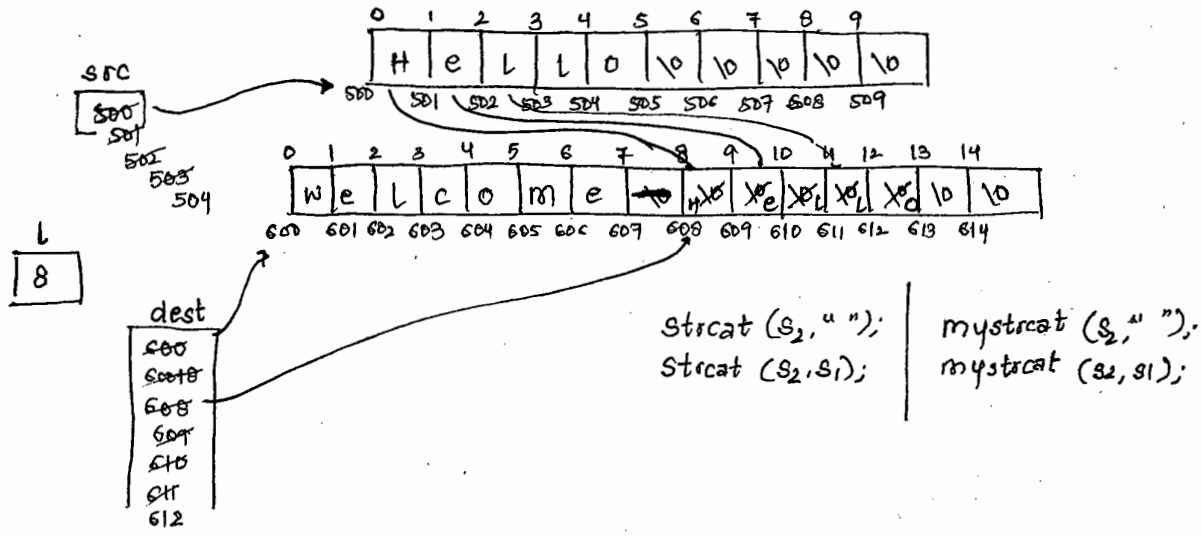
```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{
 char s1 [10] = "Hello";
 char s2 [15] = "Welcome";
 clrscr();
 puts (s1);
 puts (s2);
 strcat (s2, " ");
 strcat (s2, s1 );
 puts (s1);
 puts (s2);
 getch();
 return 0;
}
```

1. `strcat (s2, " ");` `strcat (s2 , s1);` — `strcat(s2, " ");` `strcat (s2+3, s1);`

```
Hello
Welcome
Hello
Welcome Hello
```

2. `strcat ( s2, " ");` `strcat (s2, s1+2);` — `strcat (s2, " ");` `Strcat (s2+3, s1+2);`

```
Hello
Welcome
Hello
Welcome llo
```



USER DEFINED CODE

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void mystrcat(char *dest, const char* src)
{
  int l;
  l = strlen(dest);
  dest = dest + l;
  while (*src != '\0')
  {
    *dest = *src;
    dest++;
    src++;
  }
  *dest = '\0';      // while (*dest++ = *src++) { }
}
int main()
{
  char s1[15];
  char s2[10];
  clrscr();
  printf("\n Enter str1 : ");
  gets(s1);
  printf("\n Enter str2: ");
  gets(s2);
  mystrcat(s1," ");
  mystrcat(s1,s2);
  puts(s1);
  puts(s2);
  getch();
  return 0;
}
```

5) strupr()

By using this predefined function, we can convert a string into upper case.
strupr() function requires 1 argument of type (char*) and returns (char*)
When we are working with strupr() function from given address upto null all lower case characters are converted into uppercase.

Syntax: `char*strupr(char*str);`

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
int main()
{
 char str[10];
 clrscr();
 printf("Enter a string: ");
 gets(str);
 strupr(str);
 printf("Upper case String: %s", str);
 getch();
 return 0;
}
```

O/p: Enter a string: welcome
Upper case String: WELCOME

```
1. strupr(str);      WELCOME
2. strupr(str+2);    weLCOME
3. strupr(str+5);    welcoME
4. strupr(str+7);    welcome
```

USER-DEFINED CODE

```
#include <stdio.h>
#include <conio.h>
void mystrupr(char*str)
{
 int i;
 for(i=0; str[i]!='\0'; i++)
 {
  if(str[i]>='a' && str[i]<='z')
    str[i]=str[i]-32;
 }
}
int main(void)
{
 char str[10]="welcome";
 clrscr();
 puts(str);
 mystrupr(str);
```

O/p: Welcome
Uppercase String: WELCOME

```
printf("Upper Case String : %s", str);
getch();
return 0;
}
```

6) strlwr() :- By using this predefined function, we can convert a string to lowercase.

- strlwr() function requires 1 argument of type (char*) & returns (char*)
- When we are working with strlwr(), from given address upto null all upper case characters are converted into lower case.

Syntax: `char* strlwr(char* str);`

```
# include <stdio.h>
# include <conio.h>
# include <string.h>
void mystrlwr(char* str)
{
  while(*str != '\0')
  {
    if(*str >= 'A' && *str <= 'Z')
    {
      *str = *str + 32;
      ++str;
    }
  }
int main()
{
  char str[15];
  clrscr();
  printf("Enter a string: ");
  gets(str);
  mystrlwr(str);          //strlwr(str);
  printf("Lower Case String: %s", str);
  getch();
  return 0;
}
```

O/P:

```
Enter a string : WELCOME
Lower case string : welcome
```

7) strcmp() - By using this predefined function, we can compare

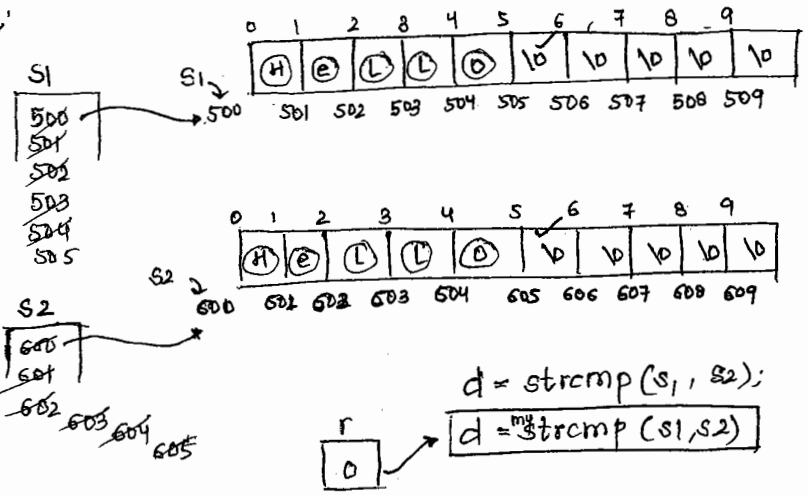
- strcmp() requires 2 arguments of type (const char*) & returns an integer value.
- When we are working with strcmp(), then character by character comparision takeplace until 1st unpaired character set is occurred.
- When 1st unpaired character set is occured then it returns ASCII value difference.

➤ At the time of comparision if there is no any diff., then by default it returns 0.

Syntax: `int strcmp (const char* s1, const char *s2);`

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{
char s1[10] = "Hello";
char s2[10] = "Hello";
int d;
clrscr();
puts(s1);
puts(s2);
d = strcmp(s1,s2);
printf("ASCII value DIFF: %d",d);
getch();
return 0;
}
```

O/P:
```
Hello
Hello
ASCII value DIFF: 0
```



USER - DEFINED CODE

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int mystrcmp (const char *s1, const char * s2)
{ int r=0;
  while (*s1 != '\0' || *s2 != '\0')
  {
    if (*s1 != *s2)
    {
      r= *s1 - *s2
      return r; // return back to main function
    }
    ++s1;
    ++s2;
  }
```

```
int main()
{
 char s1[10];
 char s2[10];
 int d;
 clrscr();
 printf("\nEnter str1: ");
 gets(s1);
 printf("\n Enter str2: ");
 gets(s2);
 d = mystremp(s1,s2);        // d = strcmp(s1,s2);
 printf("\n ASCII value diff: %d", d);
 getch();
 return 0;
 }
```

6) stricmp() :- By using this predefined function, we can compare the strings without any case i.e uppercase and lowercase contents both are treated like same.

→ When we are working with stremp() function it works with the help of case i.e uppercase and lowercase content both are different.

→ striemp() requires 2 arguments of type (const char*) & returns an int value

syntax: `int striemp(const char *s1, const char *s2)`

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{
 char s2[10] = "hello";
 char s2[10] = "HELLO";
 int d;
 clrscr();
 puts(s1);
 puts(s2);
 d = striemp(s1, s2);
 printf("ASCII VALUE DIFF : %d", d);
 getch();
 return 0;
}
```

O/P:
```
hello
HELLO
ASCII VALUE DIFF : 0
```

1. S1 → ABC, S2 → ABC } 0
S1 → abc, S2 → abc } 0 } 0
S1 → ABC, S2 → abc } -32
S1 → abc, S2 → ABC } 32

2) S1 → ABC A, S2 → bca B } -1 (-32)

3) S1 → bca B (-32), S2 → ABC A } +1

4) a) BCA → B, ABC → A } +1
b) bca → b, abc → a } +1

5) S1 → abc A (65), S2 → nul 0 65

6) S1 → nul, S2 → abc } -97

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int mystricmp(const char *s1, const char *s2)
{
 int d1, t1, t2, d2;
 while(*s1 != '\0' || *s2 != '\0')
  {
   if(*s1 - *s2 == 32 || *s1 - *s2 == -32 || *s1 - *s2 == 0)
     {
       s1++;
       s2++;  // case 1
     }
     else
       {
         t1 = *s1;
         t2 = *s2;
        if(t1 >= 'a' && t1 <= 'z' || t2 >= 'a' && t2 <= 'z')
          {
            t1 -= 32;    // case 3,5
            return (t1 - t2);
          }
       else
          if(t2 >= 'a' && t2 <= 'z' && !(t1 >= 'a' && t1 <= 'z') && !t1 == 0)
          {
            t2 -= 32;   // case 2
            return (t1 - t2);
          }
        else return (t1 - t2); // case 6, 4(b)
       }
       else
           return (t1 - t2);  // case 4(a)
      }
```

```
    return 0;
}
int main()
{
    char s1[10];
    char s2[10];
    int d1, d2;
    printf("Enter str1 : ");
    gets(s1);
    printf("Enter str2 : ");
    gets(s2);
    d1 = stricmp(s1, s2);
    d2 = mystricmp(s1, s2);
    printf("\n%d %d", d1, d2);
    getch();
    return 0;
}
```

9) strstr() :- By using this predefined function, we can find substring of a string.

- strstr() func^n requires 2 arguments of type const char* and returns char*

Syntax : `char *strstr(const char *str, const char* sub);`

- If searching substring is available, then strstr() returns base address of substring else returns null.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{
    char str[50];
    char sub[10];
    char *ptr;
    clrscr();
    printf("Enter a string.");
    gets(str);
    printf("Enter a substring : ");
    flushall();
    gets(sub);
    ptr = strstr(str, sub);
```

O/P:

```
Enter a string : Hello Naresh IT
Enter a substring : Naresh
Naresh IT
```

```
if (ptr! = NULL)
   printf ("\n %s", ptr);
else
   printf ("\n substring not found");
  getch ();
  return 0;
}
```

28|7|15.

Prog :-
```
# include <stdio.h>
# include <conio.h>
# include <string.h>
  int alphacount (char str[])
 {
   int count = 0, i ;
    for (i=0; str [i] ! = '\0'; i++)
   {
     if (str [i] > = 'A' && str [i] <= 'z' || str[i] >= 'a' && str [i] <= 'z')
      ++ count;
    }
    return count;
    }
    int main ()
     {
       char str [50];
       int c ;
       clrscr();
       printf (" Enter a string: ");
       gets (str);
       c = alphacount (str);
       printf (" Total count value = %d", c);
       getch();
       return 0;
     }
```

OUTPUT :

```
Enter a string : Welcome! @ # * % Hello 123456
Total count value : 12
```

```
#include <stdio.h>
#include <conio.h>
void changecase (char *str)
{
 int i;
 for(i=0; str[i]!='\0'; i++)
  {
   if (str[i]>='A' && str[i]<='Z')
      str[i]=str[i]+32;
    else if (str[i]>='a' && str[i]<='z')
      str[i]=str[i]-32;
    else;
  }
}
int main()
{
  char str[50];
  clrscr();
  printf("Enter a string:");
  gets(str);
  changecase(str);
  printf("change case string: %s", str);
  getch();
  return 0;
}
```

O/P: Enter a string: Hello Naresh IT
change case string: hELLO nARESH it

Program: COUNTING NO. OF VOWELS

```
#include <stdio.h>
#include <conio.h>
 int main()
{
 char str[50];
 int i, count=0;
 clrscr();
 printf("Enter a string:");
 gets(str);
 for(i=0; str[i]!='\0'; i++)
  {
    switch(str[i])
      {
        case 'A':
        case 'a': ++count;
                  break;
```

```
case 'E':
case 'e' : ++count;
           break;
case 'I':
case 'i' : ++count;
           break;

case 'O':
case 'o' : ++count;
           break;

case 'U' :
case 'u' : ++count;
           break;
}
}
printf ("Total count value = %d", count);
getch();
return 0;
}
```

O/p : Enter a string : Welcome Naresh IT Hello
Total Count value : 8

Here it is taking duplicate values too.

for unique occurrence

```
#include <stdio.h>
#include <conio.h>
int main()
{
char str[50];
char temp[] = "AEIOU";
int i, k, count = 0;
clrscr();
printf ("Enter a string:");
gets (str);
for (k=0; temp[k] != '\0'; k++)
{
  for (i=0; str[i] != '\0'; i++)
    { if (temp[k] == str[i] || temp[k] == str[i] - 32)
      {
        ++count;
        break;
      }
    }
}
printf ("Total count value = %d", count);
getch();
return 0;
}
```

O/p: Enter a string : Welcome Naresh IT Hello
Total Count Value : 4

PALINDROME PROGRAM :-

```
#include <stdio.h>
# include <conio.h>
# include<string.h>
int main()
{
   char str [15];
   char temp [15];
   clrscr ();
   printf ("Enter a string: ");
   gets (str);
   strcpy (temp, str);
   strev (temp);
   if (strcmp (str, temp)==0)
      printf ("\n %s PALINDROME", str);
  else printf ("\n %s NOT PALINDROME", str);
  getch ();
  return 0;
}
```

O/P:
```
Enter a string: MadAm
MadAm  PALLINDROME
```

To print total no. of occurrence of a digit into '*'

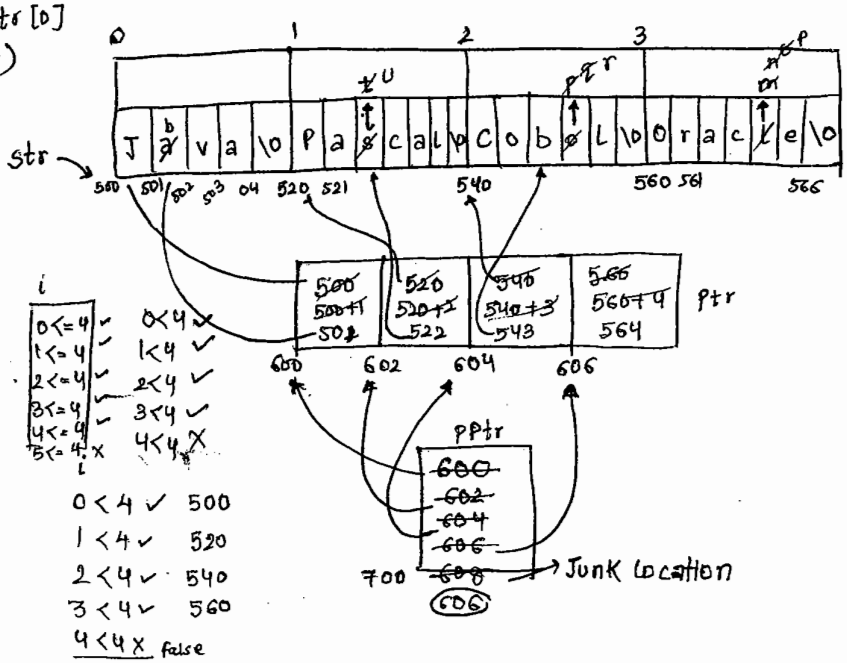
```
# include <stdio.h>
# include <conio.h>
# include <string.h>
int main()
{
  char str [50];
  char ch ;
  int i, count;
  clrscr();
  printf ("Enter a string: ");
  gets (str);
  for (ch='0' ; ch<='9'; ch++)     //for(ch = 'A'; ch<='Z'; ch++)
  {
    count = 0;
    for(i = 0; str[i]! = '\0'; i++)
    {
      if (ch==str [i])   // if (ch == str [i] || ch == str [i] - 32)
        ++ count ;
    }
   if (count > 0)
   {
```

```
        printf("*");
        --count;
      }
    }
}
 getch();
 return 0;
}
```

O/P:

```
Enter a string: 34569525
2: *
3: *
4: **
5: * * *
6: *
9: *
```

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
 { char str[4][20] = {
                      "Java",
                      "Pascal",
                      "Cobol",
                      "Oracle"
                     };
char *ptr[4]
char **pptr
int i;
clrscr();
ptr[0] = str;      //str[0];      //&str[0][0];
ptr[1] = str+1;    //str[1];      //&str[1][0];
ptr[2] = str+2;    //str[2];      //&str[2][0];
ptr[3] = str+3;    //str[3];      //&str[3][0];
pptr = ptr;        //&ptr[0]
for(i=1; i<=4; i++)
{
   *pptr++ = i;
   **pptr++ = i;
   ++pptr;
 }
 --pptr;
 puts(*pptr);
 [for(i=0; i<4; i++)
   puts(ptr[i]);]
 [for(i=0; i<4; i++)
   puts(str+i);]
 getch();
 return 0;
}
```



i	
0<=4 ✓	0<4 ✓
1<=4 ✓	1<4 ✓
2<=4 ✓	2<4 ✓
3<=4 ✓	3<4 ✓
4<=4 ✓	4<4 ✗
5<=4 ✗	

i

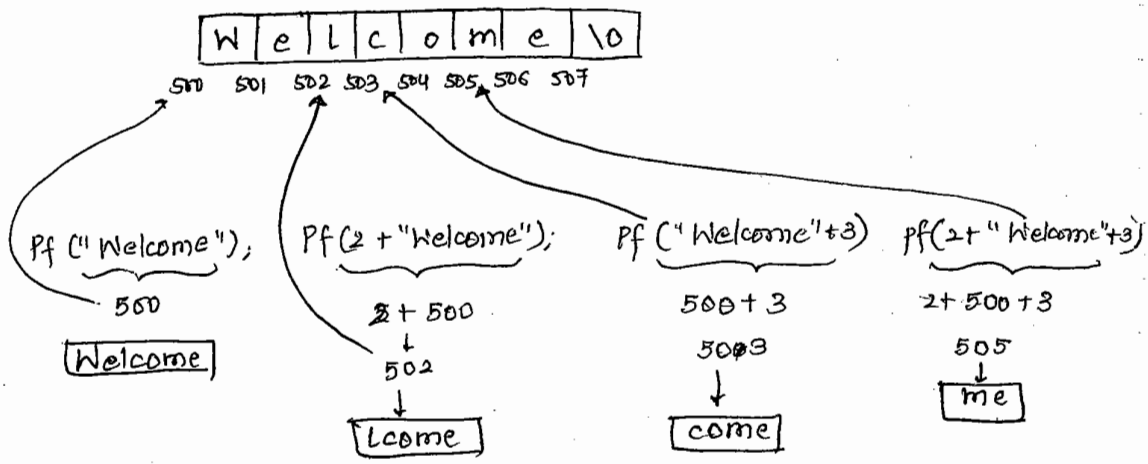
0 < 4 ✓ 500
1 < 4 ✓ 520
2 < 4 ✓ 540
3 < 4 ✓ 560
4 < 4 ✗ false

INTERVIEW QUESTIONS

```
#include <stdio.h>
#include <conio.h>
int main()
{ printf ("Welcome");
  return 0;
}
```

O/P: Welcome

1. printf (" Welcome"); Welcome
2. printf (2+"Welcome"); Lcome
3. printf ("Welcome"+3); come
4. printf (2+ Welcome"+3); me




```
#include <stdio.h>
int main()
{ char str [] = "Welcome";
  printf ("str");
  return 0;
}
```

1. printf (str); Welcome
2. printf (2+str); Lcome
3. printf (str+3); come
4. printf (2+str+3); me

```
# include <stdio.h>
  int main()
{
  puts ("Welcome");
  return 0;
}
```

1. puts ("Welcome"); — Welcome
2. puts (2 + "Welcome"); — lcome
3. puts ("Welcome" + 3); — come
4. puts (2 + "Welcome" + 3); — me

```
# include <stdio.h>
  int main()
{  char str[] = "%d Hello %d";
   printf (str);
   return 0;
}
```

O/P: gr Hello gr

1. printf (str); — gr Hello gr → printf is formated
2. printf (str, 10, 20) — 10 Hello 20
3. puts (str); — %d Hello %d → puts is unformatted ∴ doesn't understand format specifiers
4. puts (str, 10, 20); — Error
   → puts takes only one argument

```
# include <stdio.h>
  int main()
{  printf ("Hai" "Bye");
   return 0;
}
```

O/P: Hai Bye — Printf internally concates 2 strings seperated with space

1. printf ("Hai" "Bye"); — HaiBye
2. printf ("Hai", "Bye"); — Hai — Here also we have 2 strings but ',' ∴ will treat as 1o ∴ only 1 string is printed
3. printf ("Hai %s", "Bye"); — Hai bye
   1st %s (in 1st argument) treated as format specifier
4. printf ("Hai %s", "Bye %s", "Hello"); — Hai Bye %s
   In 2nd argument it is not treated as format specifier.
5. printf ("Hai %s %s", "Bye", "Hello"); — HaiByeHello;
   → The correct way

```
#include <stdio.h>
int main()
{
  printf("\n%s", "Hello");        // Hello
  printf("\n%-3s", "Hello");      // Hel
  printf("\n%.3s", 2+"Hello");    // llo
}
```

```
#include <stdio.h>
int main()
{
  char str[]="Hello";
  printf("\n%-5s", str);          // Hello
  printf("\n%.3s", str);          // Hel
  printf("\n%.3s", 2+str);        // llo
  return 0;
}
```

```
#include <stdio.h>
int main()
{
  char str1[]
  char str2[]
  char temp[10];
  sprintf(temp, "%s%.3s", str1+3, str2);
  printf("\n%s", temp);
  return 0;
}
```

O/P: comeHel

STRINGZING OPERATOR (#)

This operator is introduced in NCC version. By using this operator, we can convert the text in the form of string i.e replacement in " ".

```
#include <stdio.h>
#define ABC(xy) printf(#xy "=%d", xy);
int main()
{
  int a, b;
  a = 10; b= 20;
  ABC(a+b)
  return 0;
}
```

O/P: a+b = 30

```
# define ABC(xy)   printf (#xy " = %d", xy);

ABC(a+b)

printf (# a+b  " = %d", a+b);
printf ("a+b" " = %d ", a+b);
printf ("a+b = %d", a+b);  → concatenation
```

O/P :- a+b = 30

TOKEN PASTE OPERATOR (##)

- Ncc Programming language supports this operator.
- By using this operator, We can concatenate multiple tokens.

```
# include <stdio.h>
# define ABC(x,y) printf("%d", x##y)  isliye yahan nahi hai
void main()
{
   int var12 = 120;
   ABC(var,12);
   return 0;
}
```

```
# define ABC(x,y)   printf("%d", x##y)

ABC(var, 12);

printf("%d", var##12);
printf("%d", var12);
```

O/P : 120

```
# include <stdio.h>
# define START m##a##i##n
void START()
{  printf(" With out main");
}
```

```
void START()        start is replaced with
  m##a##i##n
  ma##i##n
  mai##n
  main
```

O/P :- Without main

21/7/2015.

MEMORY MANAGEMENT IN 'C'

In C Programming lang., we are having 2 types of m/m management

1. Static Memory management
2. Dynamic Memory Management

1) Static Memory Management

- When we are creating the memory at the time of compilation then it is called Static memory allocation or Compile-time m/m.
- Static memory Allocation is under control of compiler.
- When we are working with static memory Allocation, it is not possible to extend the memory at the time of execution, if it is not sufficient
- When we are working with static memory Allocation, we need to go for preallocation of memory i.e how many bytes of data need required to be created is needed to be decided using coding only

```
Ex: int a;              // 2B
    float f;            // 4B
    int arr[10];        // 20B
    char str[50];       // 50B
```

2) Dynamic Memory Management

- It is a procedure of allocating or deallocating the memory at run time i.e dynamically
- By using DMA, we can utilize the memory more efficienty according to the requirement
- By using DMA, whenever we want, which type we want & how much we want, that time, type and that much we can create dynamically.
- DMA related, all predefined functions are declared in <malloc.h>
  <alloc.h>
  <stdlib.h>

DMA related predefined functions are :-

1. malloc()
2. calloc()
3. realloc()
4. free()
5. farmalloc()
6. faralloc()
7. farrealloc()
8. farfree()

1) malloc():- By using this predefined function, we can create the memory dynamically at initial stage.

- malloc function require 1 argument of type size_type i.e, datatype size
- malloc() creates memory in bytes format and initial value is garbage.

Syntax: `void *malloc (size_type);`

NOTE: DMA related functions can be applied for any datatype thats why functions returns void* i.e. generic type.

- When we are working with DMA related functions, we are required to perform Type casting because functions returns void*.

```
Ex:- int *ptr;
     ptr = (int*) malloc (sizeof (int));          //2B

     float *ptr;
     ptr = (float*) malloc (sizeof (float));      //4B

     int *arr;
     arr = (int*) malloc (sizeof(int)*10);        //20B

     char *str;
     str = (char*) malloc (sizeof (char)*50);     //50B
```

2) free():- By using this predefined function, we can deallocate dynamically allocated memory.

- When we are working with DMA related memory, it stores in heap area of data segment and it is permanent memory, if we are not deallocating
- When we are working with DMA related programs, at end of the program, recommended to deallocate memory by using free() function.
- free() function requires 1 argument of type (void*) & returns void type

Syntax: `void free (void *ptr);`

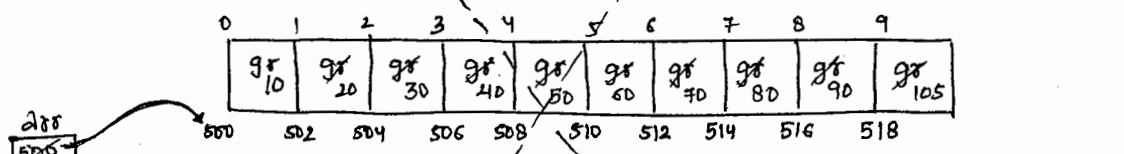
```
#include<stdio.h>
#include <conio.h>
#include <malloc.h>
int main()
{
    int *arr;
    int sum =0, i, size;
    float avg;
    clrscr();
    printf("\n Enter array size");
```

```
scanf("%d", &size);
~~int~~ arr = (int *) malloc (sizeof(int) * size);
printf("\n Default values: ");
for(i=0; i<size, i++)
printf("%d", arr[i]);

printf("\n Enter %d values : ", size);
for(i=0 ; i<size ; i++)
{
  scanf("%d", &arr[i]
  sum += arr[i]      //sum = sum + *(arr+i);
}
  avg = (float) sum/size ;
  printf("\n Sum of list : %d", sum);
  printf("\n Avg of list : ~~%d~~ %.2f", avg);
  getch();
  return 0;
}
```

```
free(arr);
arr = NULL;
```

O/P:

```
Enter array size : 10
Default values : gr gr gr gr gr ...
Enter 10 values : 10 20 30 40 50 60 70 80 90 105
Sum of list :  555
Avg of list :  555.50
```



i

0<10 ✓
1<10 ✓
2<10 ✓
3<10 ✓
4<10 ✓
5<10 ✓
6<10 ✓
7<10 ✓
8<10 ✓
9<10 ✓
10<10 X

Sum

~~0+10~~ ~~150+60~~
~~10+20~~ ~~210+70~~
~~30+30~~ ~~280+90~~
~~60+40~~ ~~360+90~~
~~100+50~~ ~~450+105~~
150 555

avg

555.50

size

10

3) calloc() By using this predefined function, we can create the memory dynamically at initial stage.

→ calloc() requires 2 argument of type (count, size_type)

→ count will provide no. of elements, size_type is datatype size

→ When we are working with calloc() function, it creates the memory in block format & initial value is zero

Syntax: `void* calloc(count, size_type)`

Eg:

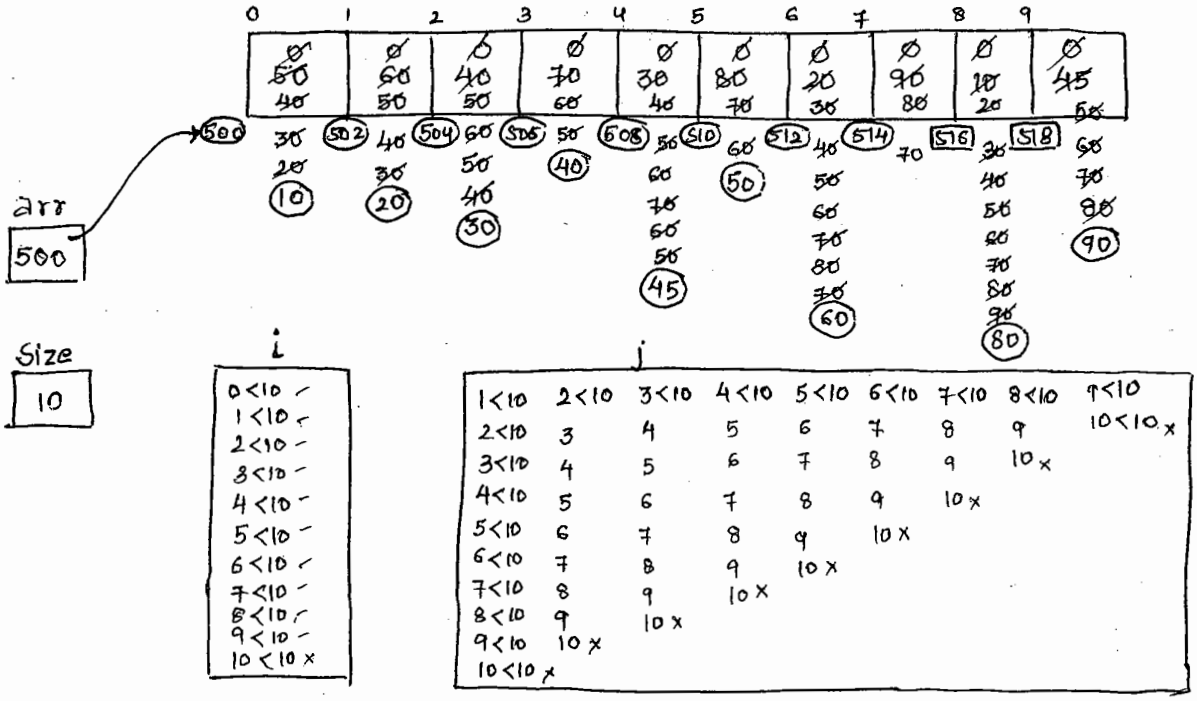
```
int *arr;
arr = (int*) calloc (10, sizeof (int));     //20B

char *str;
str = (char*) calloc (50, size of char));   //50B
```

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
int main()
{
  int *arr;
  int size, i, j, t;
  clrscr();
  printf("Enter array size: ");
  scanf("%d", size);
  arr = (int*)calloc (size, sizeof (int));
  printf("\n Default values : ");
  for (i=0; i<size; i++)
  printf ("%d", arr [i]);

  printf("\n Enter %d values: ", size);
  for (i=0; i<size; i++)
  printf scanf (" %d", &arr [i]);

  for (i=0; i<size; i++)
   {
    for (j= i+1; j<size; j++)
     {
      if (arr [j] < arr [i])
       {
        t = arr [i];
        arr [i] = arr [j];
       } arr [j] = t;
     }
```

```
printf("\n Sorted arr List:");
for(i=0; i<size, i++);
 { printf("%d", arr[i]);
  free(arr);
  arr = NULL;
  getch();
  return 0;
}
```

O/P:

```
Enter array size: 10
Default values: 0 0 0 0 0 0 0 0 0 0
Enter 10 values: 50 60 40 70 30 80 20 90 10 45
Sorted arr List:
10 20 30 40 45 50 60 70 80 90
```



4) realloc() :- By using this predefined function, we can create the memory dynamically at middle stage of the program.

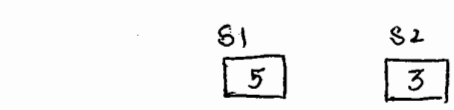
- Generally this function is required to use when we are reallocating memory.
- realloc() requires 2 argument of type void*, size_type
- void* indicates previous block base address, size_type is datatype size
- When we are working with realloc() function, it creates the memory in bytes format and initial value is garbage.

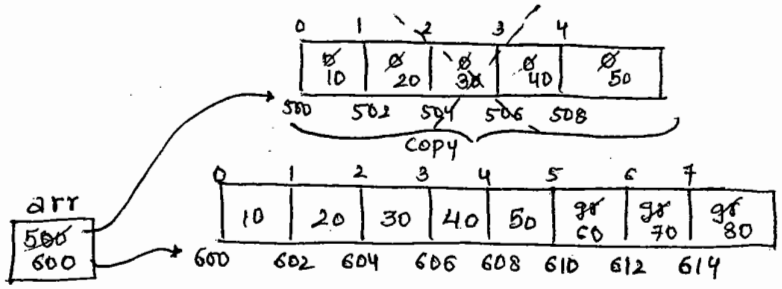
Syntax: `void * realloc(void*, size_type)`

Eg:-

```
int * arr;
arr = (int*) calloc (5, sizeof(int));     //10B
----
----
arr = (int*) realloc (arr, sizeof(int)* 10);   //20B
```

```c
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
int main()
{
  int *arr;
  int s1, s2, i;
  clrscr();
  printf("Enter array size1: ");
  scanf("%d", &s1);
  arr = (int *) calloc (s1, sizeof(int));
  printf("\n Enter %d values : ", s1);
  for(i=0; i<s1; i++)
  scanf("%d ", &arr[i]);
  printf("\n Enter array size2: ");
  scanf("%d", &s2);
  arr = (int *) realloc (arr, sizeof(int)*(s1+s2));
  printf("\n Enter %d values", s2);
  for( i=s1; i<s1+s2; i++)
  scanf("%d", &arr[i]);
  printf("\n Arr Data list: ");
  for(i=0; i<s1+s2; i++)
  printf("%d", arr[i]);
  free(arr);
  arr = NULL;
  getch();
  return 0;
}
```





O/P:

```
Enter array size1: 5
Enter 5 values: 10 20 30 40 50
Enter array size2: 3
Enter 3 values: 60 70 80
Arr Data List: 10 20 30 40 50 60 70 80
```

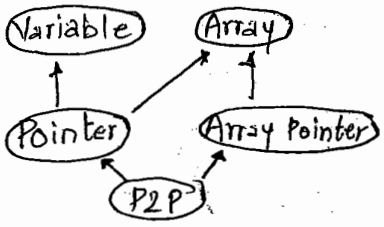
2D Array Dynamic Creation

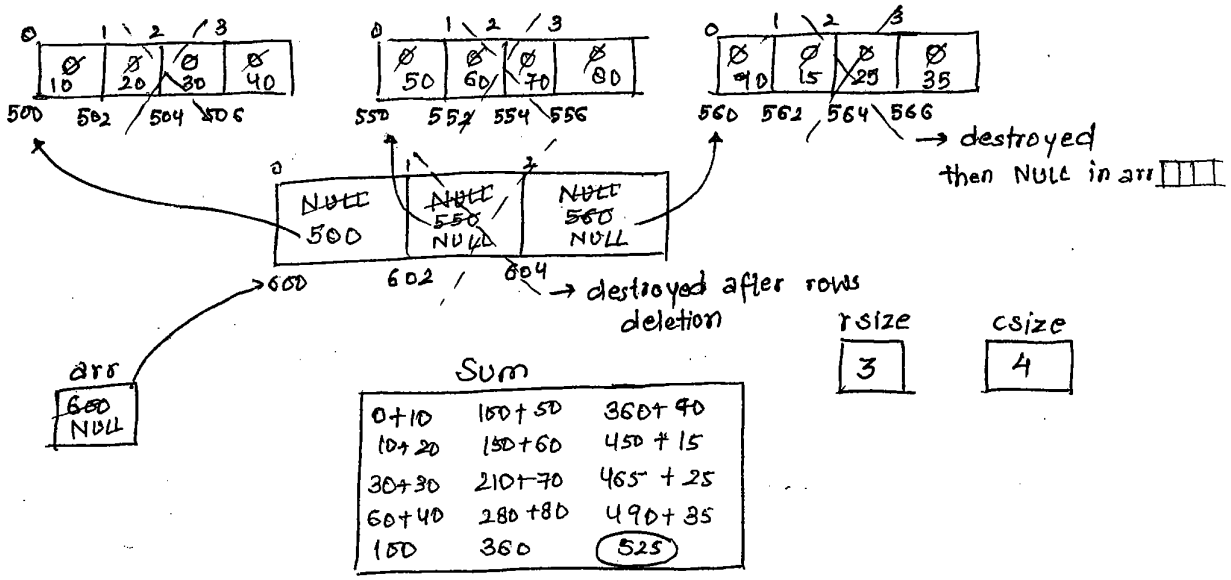
- To develop 2D Array dynamically, we are required to take pointer to pointer variable then 1 array is required to create, to manage multiple rows.

```
#include <stdio.h>
# Include <conio.h>
# Include <malloc.h>
int main()
{
   int **arr;
   int rsize, csize;
   int sum = 0, r,c;
   clrscr();
   printf("Enter row size: ");
   Scanf("%d", &rsize);
   arr = (int**) calloc (rsize, sizeof (int*));
   printf("Enter column size: ");
   Scanf("%d", &csize);
  for (r=0; r < rsize; r++)
  arr[r] = (int*) calloc (csize, sizeof (int));
  Printf(" Enter %d * %d values: ", rsize, csize);
  for (r = 0; r< rsize ; r++)
  for (c = 0 ; c < csize ; c++)
  Scanf ("%d ", &arr [r] [c]);

  for (r=0; r <rsize ; r++)
   {
    for (c= 0 ; c < csize; c++)
      {
       sum + = arr [r][c];      // sum = sum+*(*(arr+r)+c),
      }
   }
  printf("\nsum value is : %d", sum),
  for (r=0; r< rsize ; r++)
   {
     free (arr[r];
     arr [r] = NULL;
   }
     free (arr);
     arr = NULL;
     getch();
   } return 0;
```

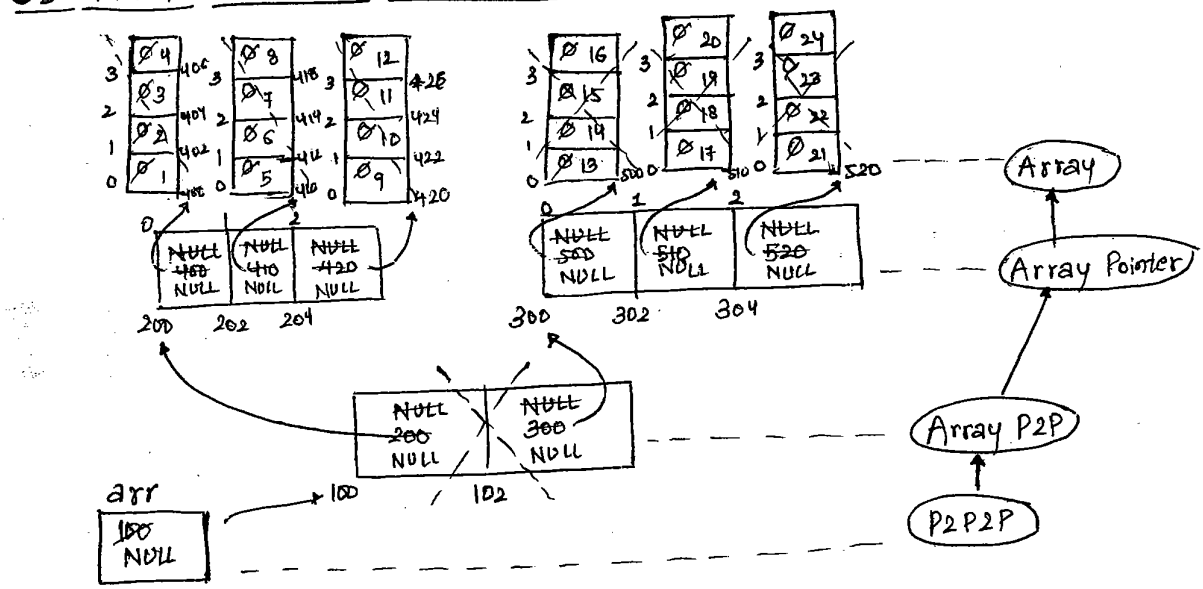
O/P :-

```
Enter row size : 3
Enter column size: 4
Enter 3 * 4 values :
   10  20  30  40
   50  60  70  80
   90  12  25  35
Sum value is : 522
```



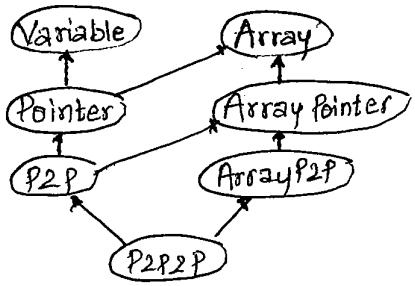


3D ARRAY DYNAMIC CREATION.



Destroyed in reverse order

Array → Array Pointer → Array P2P → P2P2P



bsize
2
rsize
3
csize
4

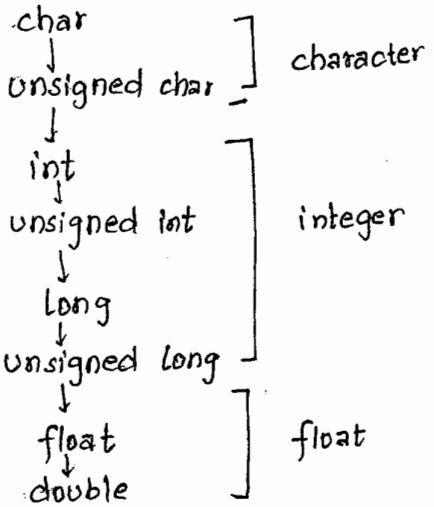
```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
int main()
{
  int ***arr;
  int bsize, rsize, csize;
  int b,r,c;
  clrscr();
  printf("Enter block size");
  scanf("%d", bsize);
  arr = (int ***)calloc(bsize, sizeof(int**));
  printf("Enter row size:");
  scanf("%d", &rsize);
  for(b=0; b< bsize; b++)
  arr[b] = (int **) calloc (rsize, sizeof(int*));
  printf("Enter column size:");
  scanf("%d", &csize);
  for(b=0; b< bsize, b++)
  for(r=0; r<rsize; r++)
  arr[b][r] = (int *) calloc(csize, sizeof(int));
  printf("Enter %d*%d*%d values:", bsize, rsize, csize);
  for(b=0; b<bsize; b++)
  for(r=0; r<rsize; r++)
  for(c=0; c<csize; c++)
  scanf("%d", &arr[b][r][c]);

  printf("\n Arr Data List:");
  for(b=0; b<bsize; b++)
  {
    printf("\n Block: %d", b+1);
    for(r=0; r<rsize; r++)
    {
      printf("\n");
      for(c=0; c<csize; c++)
      printf("%3d", arr[b][r][c]);
                  // *(*(*(arr+b)+r)+c);
    }
}
```

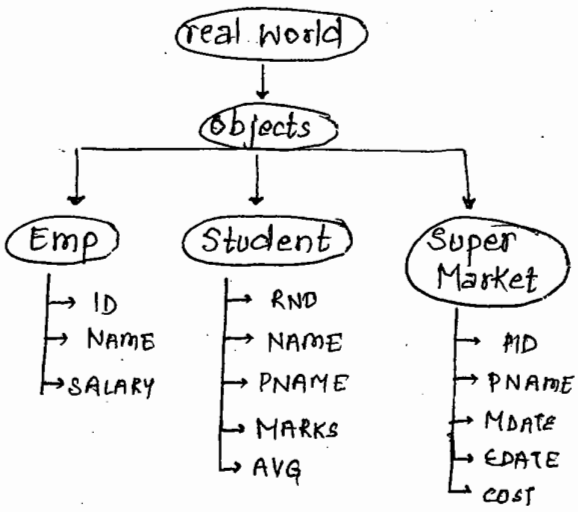
```
    for (b=0; b<bsize; b++)
     {
      for (r=0; r<rsize; r++)
        {
          free (arr [b][r]);
           arr[b] [r] = NULL;
        }
          free (arr [b])
          arr [b] = NULL;
      }
    free(arr);
     arr = NULL;
     getch ();
     return 0;
}
```

```
O/P:
 Enter block size: 2
 Enter row size: 3
 Enter column size: 4
 Enter 2*3*4 values:
 1 2 3 4 5 6 7 8 9 10 11 12
 13 14 15 16 17 18 19 20 21 22 23 24

 Arr Data list:
 Block: 1
       1  2  3  4
       5  6  7  8
       9  10 11 12
 Block: 2
       13 14 15 16
       17 18 19 20
       21 22 23 24
```

STRUCTURES

- In 'C' programming language, we are having 3 types of datatypes
  1. Primitive Datatype
  2. Derived Datatype
  3. User-defined Datatype
- All primitive datatypes are used to manipulate basic data types i.e char, int, float
- All derived datatypes works for primitive datatypes.
- In real world, every information will be there in the form of objects.
- Every object having their own properties and behaviour.
- No any primitive or derived datatypes support real time object information.
- When the primitive or derived datatypes are not supporting user requirement then go for user-defined datatypes.





- A structure is a collection of different types of data elements in a single entity.
- A structure is a combination of primitive and derived datatype variables.
- By using structures we can create user defined datatypes
- Sizeof the structure is sum of all member variable sizes.
- Least size of structure is 1B
- In 'C' programming language, it is not possible to create empty structure.

- C language structure contain data members only but in C++, data members and member functions.

- Syntax:

```
struct tagname
{
 Datatype1  mem1;
 Datatype2  mem2;
 Datatype3  mem3;
 ----
 ----
};
```

- According to syntax of the structure, semicolon must be required at end of the structure body.

Ex1:
```
struct emp
{
 int id;
 char name[36];
 int sal;
};
```
sizeof(struct emp) -----> 40B (2+36+2)

Ex2:
```
struct student
{
 int rno;
 char sname[20];
 char fname[20];
 char sname[20];
 int marks[6];
 int tmarks;
 float avg;
};
```
sizeof(struct student) ----> 80B (2+20+20+20+12+6)

Syntax to create Structure variable

```
struct tagname variable;
```

Ex1:
```
struct emp
{
 int id;
 char name[36];
 int sal;
};
```

```
void main()
{
  emp e1;
  struct emp e1;
}
```

⇒
```
struct emp
{
  int id;
  char name [36];
  int sal;
}
  e1, e2;
void main()
{
  struct emp e3, e4;
}
```

sizeof (struct emp) ---→ 40B
sizeof (e1) ⟶ 40B
sizeof (e2) ⟶ 40B

- When we are creating the structure variable at end of the structure body, then it becomes global variable i.e. e1, e2
- When we are creating the structure variable within the body of the function, then it is auto variable which is local to specific function i.e e3, e4.

Syntax to create structure type pointer :-

```
Struct tagname *ptr;
```

Eg :- on next page

Ex:
```
struct emp
{
  int id;
  char name[36];
  int sal;
} e1, *ptr1;
void main()
{
 struct emp e2;
 struct emp *ptr2;
  ptr1 = &e1;
  ptr2 = &e2;
}
 // size of(e1) ---> 40B
 // size of (ptr1) --> 2B
```

Size of user defined pointer is 2B only because it holds address.

Syntax to create structure type array :-

```
struct tagname arr[SIZE];
```

Ex-
```
struct emp
 {
  int id;
  char name[36];
  int sal;
 };
void main()
 { struct emp arr[10];
// size ---> 10
// sizeof (arr) ---> 400B(10 * 0B = 400B)
}
```

CREATING THE ARRAY DYNAMICALLY

```
struct emp
{ int id;
  char name[36];
  int sal;
};
 void main ()
{ struct emp * arr;
  arr = (struct emp *) calloc (10, sizeof(struct emp));
```

```
    // size 10
    // size of (arr) ⟶ 28
    free(arr);
}
```

Syntax to Initialize Structure Variable

```
struct tagname variable = {value1, value2, value3, ...};
```

Ex-
```
struct emp
{
 int id;
 char name[36];
 int sal;
} e1 = {101, "Raj", 125000}, ~~125000~~ e2 = {102, "Teja"};
void main()
{
  struct emp e3 = {103};
}
```

- In initialisation of structure variable, if specific no of members are not initialized, then remaining all members are initialised with 0 or nul.
- If value type member is not initialised, then it becomes 0, if string type data is not initialised, then it becomes null.

Syntax to access Structure Members :-

By using following operaters we can access structure members :-

1. struct to member
2. pointer to member

→ If variable is normal operator struct to member operator.
→ If variable is pointer type, then go for pointer to member operator.

Syntax :-

```
struct tagname variable;
Variable.member = value;
        (or)
variable.member;

struct tagname variable;
struct tagname * ptr;
ptr = &variable;
ptr → member = value;
        or
ptr → member;
```

```
struct emp.
{
  int id;
  char name [36];
  int sal;
};
 e1 = {101, "Raj", 12500}, e2 = {102, "Teja"};
void main()
 { struct emp e3, e4, e5, e6;
   e3.id = 108;
   e3.name = Rajesh;      Error
   e3.name = "Rajesh";    Error   ∴ left side string, right side string
correct                             it is not possible to assign 1 string
 way  strcpy (e3.name, "Rajesh");   value to other string
   e3.sal = 14000;
   e4 = e3 + 1;           Error
   e4 = e3.id+1;          Error
   e4.id = e3.id+1;       yes

   e4.name = e3.name;     Error
   strcpy(e4.name, e3.name);

   e4.sal = e1 + e2;      Error
   e4.sal = e1.sal + e2.sal;  yes

   e5 = e4;  ✓
   We can assign 1 structure variable to another structure
   variable of same type

   e4 == e5;      Error
   e4.id = e5.id;     yes
   e3.name > e4.name  Error
   strcmp(e3.name, e4.name);
   e3.sal < e2.sal    // yes
}
```

→ Any kind of manipulations can be performed on structure members.
→ Except the assignment, no any other operations can be performed on structure variables.
→ When 2 variables are same structure type then it is is possible to assign 1 variable data to another variable.

```
struct emp;
{
  int id;
  char name[36];
  int sal;
};
  void main()
  {
   struct emp e1;
   struct emp* ptr;
   ptr = &e1;

  // e1.id = 101;
   ptr → id = 101;

   strcpy (ptr → name, "Rajesh");
  //strcpy (e1.name, "Rajesh");

   //e1.sal = 12500;
   ptr → sal = 12500;
  }
```

typedef :-

→ It is a keyword, by using this keyword, we can create user-defined name for existing data type.

→ Generally typedef keyword used to create an alias name for existing datatype.

Syntax: `typedef Datatype user_defined_name`

```
# include <stdio.h>
# include <conio.h>
type int myint;  // myint → user defined name . ∴ alias name of integer
int main()                                      data type
{
    int x;
    myint y;
    typedef myint smallint;  // smallint is an alias name to myint
    smallint z;
    clrscr();
    printf("Enter 2 values:");
    scanf("%d%d", &x, &y);
    z = x+y;
    printf("Sum value is : %d", z);
    getch();
    return 0;
}
```

O/P:

```
Enter 2 values : 10 20
Sum value is : 30
```

```
#include <stdio.h>
#include <conio.h>
#define MYCHAR char    // MYCHAR is identifier which is replaced with char
typedef char BYTE;
int main()
{
  char ch1 = 'A';
  MYCHAR ch2 = 'b';    // char ch2 = 'b';
  BYTE ch3;
  ch3 = ch2 - ch1 + 20;    // ch3 = 98 - 65 + 20;  53 (it corressponding data is 5)
  printf("char1: %c  char2: %c  char3:%c", ch1, ch2, ch3);
  getch();
  return 0;
}
```

O/P: char1: A char2: b char3: 5

- By using #define, we can't create alias name because at the time of preprocessing, identifier is replaced with replacement text
- #define is under control of preprocessor, typedef is under control of compiler

INTERVIEW QUESTIONS

1)
```
struct
{
  int id;
  char name[36];
  int sal;
};
```
Valid

- When we are working with structures, mentioning the tagname is optional, if tagname is not given, then compiler creates nameless structure.
- When we are working with nameless structures, it is not possible to create structures variable whether the body of the function, i.e global variables are possible to create.

2)
```
struct
{ int id;
  char name[36];
  int sal;
} e1, e2, e3;
Void main()
{ struct tagname   Yes valid
}
```

* type def struct

```
{
 int id;
 char name [10]
  int sal;
} EMP;
 void main()
{ Emp. e1;          Valid
}
```

In previous syntax, 1st compiler creates a nameless structure, then it gives an alias name called EMP

* typedef struct emp

```
{  int id;
   char name [36];
   int sal;
} EMP;
Void main()
{ struct emp e1;
    Emp e2;          Valid
}
```

In previous syntax, 1st compiles creates a user-defined datatype called struct emp, then it gives an alias name called EMP

→ struct emp

```
{
   int id;
   char name[36];
   int sal;
} e1, e2, e3;
```

What is e1, e2, e3 ? Global variables

→ typedef struct

```
{
  int id;
  char name [36];
  int sal;
} EMP e1, e2, e3;
```

Error

typedef struct emp

```
{
  int id;
  char name [36];
  int sal;
} EMP e1, e2, e3;
```

Error

- When the structure body is started with typedef keyword, then it is not possible to create structure variable at end of the body i.e global variables are not possible to create.

```
typedef struct
{ int id;
  char name[36];
  int sal;
} Emp, e1, e2, e3;
```

Yes valid, Emp e1, e2, e3, alias name.

→
```
typedef struct emp
{ int id;
  char name[36];
  int sal;
} EMP, e1, e2, e3;
```

What is Emp, e1, e2, e3 ? Alias names

→
```
struct emp
{
};   Not valid, Error
```

→ In 'C' programming language it is not possible to create empty structure because least size of structure is 1 byte.

→
```
struct emp
{
  int id = 101;
  char name[36] = "Payal";
  int sal = 12500;
};   Not Valid, Error
```

- For creation of structure, it doesn't occupy any physical memory.
- When we are working with the structure, physical m/m will occupy.
- When we are creating variable but for initialization of member we required physical m/m.

---

* Self referential structure - placing a structure type pointer has a member to same structure is called self-referential structure. By using left referential structure, we can handle any type of data structure.

Ex-
```
struct emp
{ int id
  char name[36];
  int sal;
  struct emp *ptr;
};
  // Size of (struct emp) → 42B
```

→
```
struct emp
{
  int id;
  char name[36];
  int sal;
};
void abc()
{
  struct emp e1, e2;
}
void main()
{
  struct emp e1, e2;
}
```
Valid

When we are creating the structure in global scope, then it is possible to access within the program in any function.

→
```
void main()
{
  struct emp
  {
    int id;
    char name[36];
    int sal;
  };
  struct emp e1, e2;
}
void abc()
{
  struct emp e1, e2;
}
```
Is it valid?
No, Error

When the structure is created within the body of the function, then that structure need to be accessed in same function only.

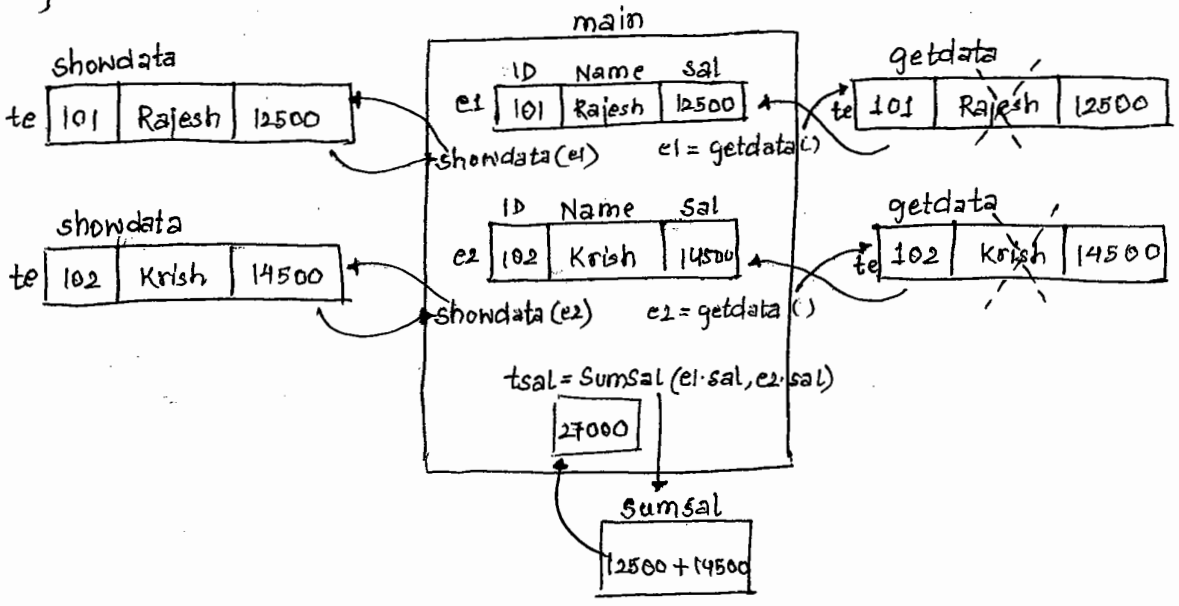
PROG :-

```
#include <stdio.h>
#include <conio.h>
typedef struct
{
  int id;
  char name[36];
  int sal;
} EMP;
EMP getdata()
{
```

```
    EMP te;
    printf ("\n ENTER EMP ID:");
    scanf ("%d, &te.id);
    printf ("\n ENTER EMP NAME: ");
    fflush (stdin);
    gets (te.name);
    printf ("\nENTER EMP SALARY:");
    scanf ("%d", &te.sal);
    return te;
}
void showdata (EMP te)
{
    printf ("\n ID: %d  NAME: %s  SALARY = %d", te.id, te.name, te.sal);
}
int sumsal (int s1, int s2)
{
    return (s1+s2);
}
int main()
{
    EMP e1, e2;
    int tsal;
    e1 = getdata();
    e2 = getdata();
    showdata (e1);
    showdata (e2);
    tsal = sumsal (e1.sal, e2.sal);
    printf ("\n Sum Salary = %d", tsal);
    return EXIT_SUCCESS;
}
```

O/P:

```
ENTER EMP ID : 101
ENTER EMP NAME: Rajesh
ENTER EMP SALARY : 12500
ENTER EMP ID: 101
ENTER EMP NAME: Krish
ENTER EMP SALARY: 14500
ID: 101 Name: Rajesh sal: 12500
ID: 102 Name: Krish  Sal: 14500
Sum Salary : 27000
```



PROG :- DISPLAYING DATA IN ALPHABETICAL ORDER

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SIZE 5
struct emp
{
  int id;
  char name[36];
  int sal;
};
struct emp getdata()
{ struct emp te;
  printf("\n ENTER ID:");
  scanf("%d", &te.id);
  printf("\n ENTER NAME:");
  fflush(stdin);
  gets(te.name);
  printf("\n ENTER SAL:");
  scanf("%d", &te.sal);
  return te;
}
void showdata(struct emp te)
{
  printf("\n Id: %3d  Name: %5s  SAL: %3d", te.id, te.name, te.sal);
}
int main()
{ struct emp arr[SIZE];
  struct emp te;
  int i, j;
  printf("\n Enter %d emp's data:", SIZE);
  for(i=0; i<SIZE; i++)
     arr[i] = getdata();
  for(i=0; i<SIZE; i++)
  {
    for(j=i+1; j<SIZE; j++)
    {
      if(stricmp(arr[i].name, arr[j].name)>0)
        // if(arr[i].sal > arr[j].sal) Ascending
        // if(arr[j].id > arr[j].id) Descending
```

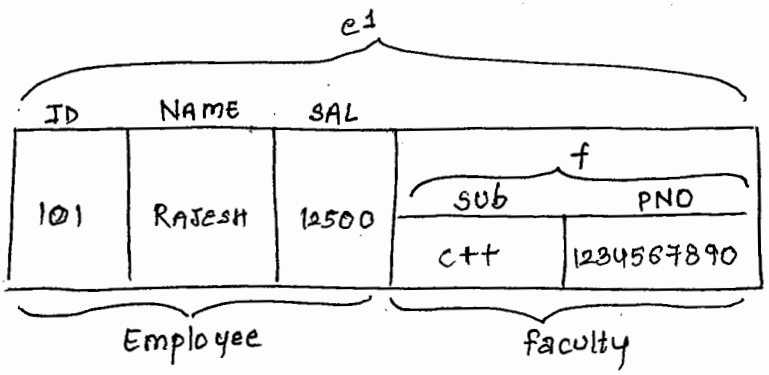
```
        {
          te = arr[i];
          arr[i] = arr[j];
          arr[j] = te;
        }
      }
    }
        printf("\n sorted emp data:");
        for (i=0; i< SIZE ; i++).
            showdata (arr [i]);
            return EXIT_SUCCESS;
    }
```

NESTED STRUCTURE

- It is a procedure of placing a structure within an existing structure body.
- When we are working with nested structure, size of the structure is sum of inner structure properties and outer structure properties is to be calculated.
- When we are working with nested structure, it is not possible to access inner structure members directly by using outer structure variable.
- In order to access inner structure ~~variable~~ members by using outer structure variable we are required to create inner structure variable within the body only.
- Inner structure variable doesnot have to access outer structure members directly or indirectly

Ex-
```
#include <stdio.h>
#include <conio.h>
#include <string.h>
struct emp
{ int id;
  char name[30];
  int sal;
    struct faculty
    {
      char sub [20];
      char pno [10];
    }
};
void main ()
{
  struct emp e1;        //sizeof(e1) ----> 70 B (40+30)
```

```
struct faculty f1;        // sizeof(f1) ----> 30B
clrscr();
e1.id = 101;
strcpy(e1.name, "Rajesh");
e1.sal = 12500;
//strcpy(e1.sub, "c++");  Error
strcpy(e1.f.sub, "c++");
strcpy(e1.f.pno, "1234567890");
printf("\n ID: %d NAME: %s SAL: %d SUB: %s PNO: %s", e1.id, e1.name,
        e1.sal, e1.f.sub, e1.f.pno);
getch();
return 0;
}
```



CONTAINERSHIP OR COMPOSITION :-

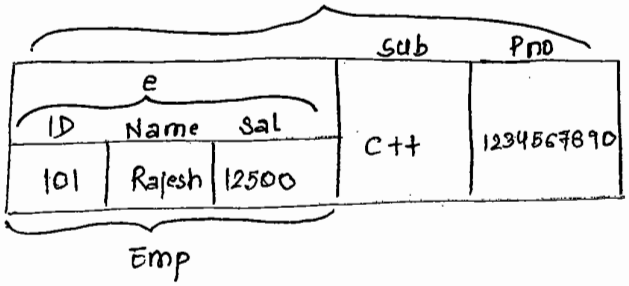
→ It is a procedure of placing a structure variable as a member to another structure.

→ Containership will occupy less memory when we are comparing it with nested structure.

→ Containership looks like inheritance in OOP language.

Eg:-
```
#include <stdio.h>
#include <conio.h>
#include <string.h>
struct emp
 {
  int id;
  char name[30];
  int sal;
 };
struct faculty
 {
  struct emp e
  char sub[20];
  char pno[10];
 };
```

```
void main()
{
 struct emp e1;        // size of (e1) ----> 40B
 struct faculty f1;    // size of (f1) ----> 70B (40+30)
 clrscr();
 f1.e.id = 101;
 strcpy (f1.e.name, "Rajesh");
 f1.e.sal = 12500;

 strcpy (f1.sub, "C++");
 strcpy (f1.pno, "1234567890");
 printf ("\n ID: %d NAME: %s SAL: %d SUB: %s PNO: %s", f
        f1.e.id, f1.e.name, f1.e.sal, f1.sub, f1.pno);
 getch();
 return 0;
}
```



2/8/2015.

UNION

→ A Union is a collection of different types of dataelements in a single entity.

→ It is a combination of primitive and derived datatype variables.

→ By using union, we can create user-defined data type elements.

→ Size of a union is max. size of a member variable.

→ In implementation, for manipulation of the data, if we are using only one member then it is recommended to go for union.

→ When we are working with unions, all member variables will share same m/m location.

→ By using union, when we are manipulating multiple members then actual data is lost.

Syntax:
```
union tagname
{
  Datatype1 mem1;
  Datatype2 mem2;
  Datatype3 mem3;
  - - - -
  - - - -
};
```

Ex-1
```
union abc
{
  int i;
  float f;
  char c;
};
// sizeof (Union abc)
```
↳ 4B

Ex2:
```
union emp
{
  int id;
  char name [36],
  int sal;
};
sizeof (union emp)
```
↳ 36B

- All the properties of structures are applicable to union also like variable creation, pointer creation, array creation, typedef approach,
- The basic difference b/w structure and union is

1. Size - structure size is sum of all member variable size
   Union size is max. size of a member variable

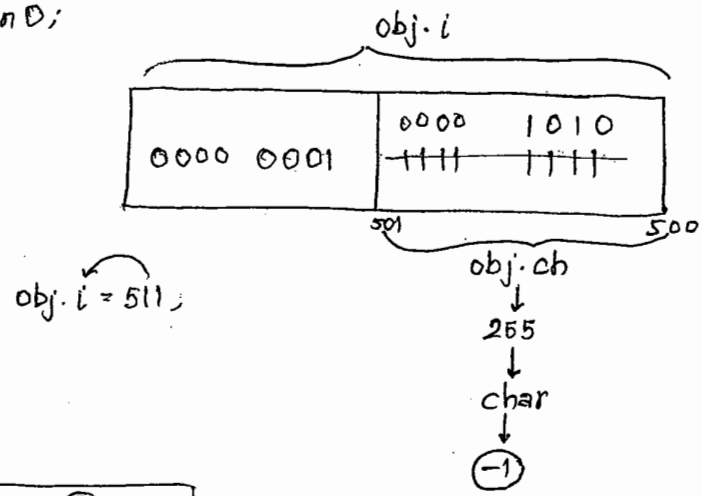
2. Memory Allocation - In structure, memory will be allocated for all the members but
   In Union, memory will be allocated for one member which occupies max. size.

3. Data Manipulation - In structure, data can be manipulated on multiple members properly without losing the content but
   - In union if we are manipulating multiple members then actual data is lost.

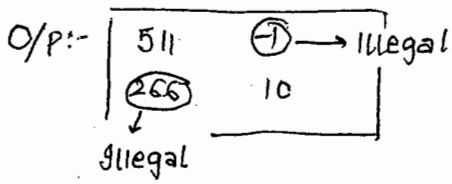
4. Initialisation - When we are working with structures, multiple members can be initialised but
   - In union only 1 member required to be initialised.

Ex-
```
#include <stdio.h>
#include <conio.h>
union abc
{
  int i;
  char ch;
};
int main()
{
  union abc obj;
  clrscr();
  obj.i = 511;
  printf("\n %d %d", obj.i, obj.ch);
  obj.ch = 10;
  printf("\n%d %d", obj.i, obj.ch);
  getch();
  return 0;
}
```

union abc obj;
↓
2B



O/P:-



ENUM

- enum is a Keyword, by using this Keyword we can create sequence of integer constant value.
- Generally by using enum, we can create userdefined datatype of integer
- Size of enumerator datatype is 2B & the range from -32768 to 32767
- It is possible to change enum related variable value but it is not possible to change enum constant data

Syntax :- `enum tagname {const1 = value, const2 = value, const3 = value};`

- Acc. to syntax, if const1 value is not initialised, then by default sequence will start from 0 and next generated value is prev. const value + 1.

→
```
#include<stdio.h>
enum ABC {A,B,C};
int main()
{
  enum ABC x;
    x = A+B+C;    // x = 0+1+2;
  printf("\n x = %d",x);
  printf("\n %d %d %d", A,B,C);
  return 0;
}
```

O/p :
```
x = 3
0 1 2
```

→
```
#include <stdio.h>
#include <conio.h>
enum month {Jan=1, feb, Mar};
int main()
{
  enum month Apr;
  Apr = Mar+1;
  printf("\n %d %d", feb, Apr);
  getch();
  return 0;
}
```

O/p: 2 4

++ Feb; Error

++ Apr; Yes

→ In previous program, enum month is an user defined data type of int, Apr is variable of type enum month

→ enum related constant values are not allows to modify, enum related variables are possible to modify.

```
#include <stdio.h>
#include <conio.h>
typedef enum ABC {A=40, B, X=50, Y} XYZ;
int main()
{
 enum ABC c;
  XYZ z;
 c = B+1;  // c = 41+1;
 z = Y+1;  // z = 51+1
 printf("\nc= %d  z=%d ", c,z);
 printf("\n B = %d  Y=%d", B,Y);
 getch();
 return 0;
}
```

↳ alias name of ABC

O/P:

C = 42	Z = 52
B = 41	Y = 51

In previous program enum ABC is an user-defined data type of integer, XYZ is an alias name to enum ABC

FILE OPERATIONS

- A file is a name of physical memory location in secondary storage area.
- File contains sequence of bytes of data in secondary storage area in the form of unstructured manner.
- In implementation, when we required to interact with secondary storage area, then recommended to go for file Operations.
- By using files, primary m/m related data can be send to secondary storage area & secondary storage area info can be loaded to primary memory.
- In 'C' programming lang., IO Operations are classified into 2 types -
  1. standard I/O operations
  2. secondary I/O operations.
- When we are interacting with primary I/O devices, then it is called standard I/O operations.
- When we are interacting with secondary I/O devices, then it is called secondary I/O operations
- Standard I/O related and Secondary I/O related, all predefined functions are declared in stdio.h only.
- File operations related specific functions are -

<STDIO.H>

Function			
fopen	fclose	fprintf	fputs
fscanf	fgetc	fgets	fgetchar
fread	fwrite	flushall	ftell
feof	fseek	remove	rename
rewind	fflush		

Constant	datatypes	global variables
EOF	FILE	stdin
NULL		stdout
SEEK_CUR		stderr
SEEK_END		stdprn
SEEK_SET		

```
→ #include <stdio.h>
  # include<dos.h>
  # include <stdlib.h>
  int main()
  {
    FILE *fp;
    //system("CLS"); //clrscr();
  fp=fopen("E:\\1.txt","W");
    if (fp==NULL)
     {
       printf("\n UNABLE TO CREATE FILE");
       // system("PAUSE"); // getch();
       return EXIT_FAILURE;
     }
    fprintf(fp,"Welcome");
    fprintf(fp,"\n%d NARESH IT%f", 100, 12.50);
    fclose(fp);
    return EXIT_SUCCESS;
  }
```

When we are opening file & parameters must be passed 1. path 2. mode

W → Write

→ Exception rised
file may open or may not open
if open then it returns ADDRESS
if not then it returns NULL.

O/P:
```
Welcome
100 Naresh 12.50
```

Prog2:- for creating ASCII text file

```
  # include <stdio.h>
  # include <stdlib.h>
  int main()
  {
     FILE *fp;
     char path[50];
     int i;
     printf("Enter a file path:");
     gets(path);
     fp = fopen(path,"W");
     if (fp == NULL)
    {
       printf("\n UNABLE TO CREATE FILE");
       return EXIT_FAILURE;
     }
    for(i=-128; i<=127; i++)
    fprintf(fp, "%d=%c\t", i, i);
    fclose(fp);
    return EXIT_SUCCESS;
  }
```

Static path E:\\1.txt
dynamic path E:\1.txt

⟶ Range of ASCII -128 to 127

- stdio.h provides standard I/O related predefined function prototype.
- conio.h provide console related predefined function prototype.
- FILE is a predefined structure which is available in stdio.h
  By using FILE structure, we can handle file properties.
  Size of FILE structure is 16 bytes.
- fp is a variable of type FILE*, which maintain address of FILE.
  Size of fp is 2 bytes because it holds address.

fopen() :- It is a predefined function, which is declared in stdio.h,
- → by using this function we can open a file in specific path with specific mode.
- → It requires 2 arguments of type const char*.
- → On success, fopen() returns FILE*, On failure returns NULL
- → Generally fopen() is failed to open FILE in following cases
  1. path is incorrect
  2. mode is incorrect
  3. Permissions are not available
  4. Memory is not available.

Syntax:

```
FILE *fopen (const char *path, const char *mode);
```

fprintf() :-
- ▸ By using this predefined function, we can write the content in file.
- ▸ fprintf() can take any no of arguments but 1st argument must be FILE* and remaining arguments are of

Syntax :-

```
int fprintf (FILE *stream, const char *path, ---);
```

fclose() :-
- ▸ By using this predefined function, we can close the file after saving data.
- ▸ fclose() requires 1 argument of type FILE* and returns an int value

Syntax :-

```
int fclose (FILE *stream)
```

FILE MODES :-

→ Always FILE modes will indicate that for what purpose file need to be open or create.

FILE modes are classified into 3 types -

1. write
2. read
3. append

Depending on operations, file modes are classified into 6 types -

1) write (w) - Create a file for writing, if file already exists, then it will override (old file is deleted, new file created)

- In w mode, file exists or not, always new file is constructed.

2) read (r) - Open an existing file for reading, if file not exists then fopen() returns NULL

- When we are working with r mode, if file doesn't exist, new file is not constructed.

3) append (a) :- Open an existing file for appending (write the data at end of file) or create a new file for writing if it doesn't exist

- When we are working with "a", if file doesn't exist, then only new file is constructed

4) w+ (write and read) - Create a file for update i.e write & read, if file already exists then it will override.

- In w+ mode, if file is available or not, always new file is constructed.

5) r+ (read and write) - Open an existing file for update i.e read & write.

- Generally r+ mode is required, when we need to update existing information.
- In r+ mode, if file doesn't exist, new file is not constructed

6) a+ (w+ and r+) - Open an existing file for update or create a new file for update.

- By using a+ mode, we can perform random operations.

Files are classified into two types -

1. text files
2. Binary files

→ In Text Files, data is represented with the help of Ascii values
•txt, •c, •cpp

→ In binary files, data is represented with the help of byte
•exe, •mp3, •mp4, •jpeg

1) To specify that a given file is being opened or create in "text mode" then append "t" to the string mode.
eg:- rt, wt, at, rt+, wt+, at+

2) To specify binary mode then append "b" to end of the string mode.
eg:- rb, wb, ab, r+b, w+b, a+b.

3) "fopen" and "fsopen" also allow that "t" or "b" to be inserted between the letter and the '+' character in the string
Eg:- rt+ is equivalent to r+t.

4) If "t" or "b" is not giving in the string the mode is governed by "f" mode, if "f" mode is set to O_BINARY, files are opened in BINARY Mode

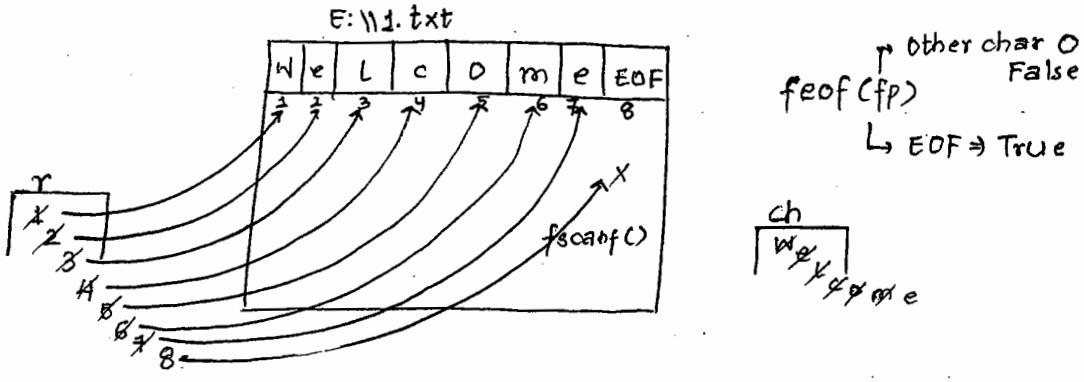
5) If "f" mode is set to O_TEXT, they are opened in text mode. These constants are defined in FCNTL.H

for reading data from files :-

```
# include<stdio.h>
# include <stdlib.h>
 int main()
{
  FILE *fp;
  char ch;
  fp = fopen ("E:\\1.txt", "r");
  if (fp == NULL)
  { printf ("\n FILE not FOUND");
    return EXIT_FAILURE;
  }while (1)
  {
    fscanf(fp, "%c", &ch);
    if (feof (fp))    //if (ch == EOF)
     break;
     printf ("%c", ch);
  }
```

```
fclose(fp);
return EXIT_SUCCESS;
}
```

O/p: Welcome



fscanf():- It is a predefined function which is declared in stdio.h, by using this function, we can read the data from file.

- fscanf() can takes any no. of arguments but 1st argument must be and remaining arguments should be scanf() function format.
- When we are working with fscanf() function, it can read entire content of file except

Syntax :- `int fscanf(FILE * stream, const char* format, ...);`

feof():- By using this function, we can find eof character position.

- It requires 1 argument of type FILE* and returns an int value.
- When file pointer is pointing to EOF character then it returns non-zero value, if it is pointing to other than EOF character then it returns zero.

Syntax :- `int feof (FILE * Stream);`

fgetc():- It is a predefined unformatted function which is declared in stdio.h, by using this function we can read the data from file

* including EOF character also

➤ It returns an int value i.e Ascii value of a character

Syntax: `int getc (FILE * stream);`

Reading 3 lines at a time & displaying from a file :-

Turbo C

```
#include <stdio.h>
#include <conio.h>
#include <dos.h> → for using delay function
int main()
{
 FILE *fp;
 char path[30];
 char ch;
 int count=0;
 clrscr();
 printf("Enter a file path: ");
 gets(path);
 fp=fopen(path,"rt");
 if(fp==NULL)
 {
  printf("\n FILE NOT FOUND");
  getch();
  return1;
 }
  while(1)
  {
   ch=fgetc(fp);      //fscanf(fp,"%c",ch);
   if(ch==EOF)          //if(feof(fp))
   break;
   printf("%c",ch);
   if(ch=='\n')
    ++count;
    if(count==3)
     {
      delay(1000);
      count=0;
     }
   }
   fclose(fp);
   getch();
   return 0;
  }
```

3/8/2015.

```
#include <stdio.h>
#include <conio.h>
int main()
{
  FILE *fp;
  char path[50];
  char str[50];
  clrscr();
  //printf("Enter a file path:");
  fprintf(stdout,"Enter a file path:");  stdout → it is the standard o/p buffer
  gets(path);
  fp = fopen(path,"a");   → append
  if(fp==NULL)   ⇒ When because of some reason file is unable to open
  {
   fprintf(stderr,"UNABLE TO CREATE FILE");
   //printf("UNABLE TO CREATE FILE");
   getch();
   return 1
  }
 printf("Enter a string:");
 fflush(stdin);
 gets(sto);
 fputs(str,fp);
 //fprintf(fp,"%s",sto);
 fclose(fp);
 return 0;
}
```

delay :- It is a predefined function which is declared in dos.h

- By using this function, we can suspend the program from execution.
- delay() function requires 1 argument of type unsigned integer i.e milliseconds value

Syntax :- `void delay(unsigned milliseconds);`

- By using delay(), we can suspend the program for max. of 1 min 5 sec

Sleep() :- It is a predefind function which is declared in dos.h

By using this function, we can suspend the program from execution

Sleep() function requires 1 argument of type unsigned integer seconds format data

Syntax :- `void sleep(unsigned seconds);`

- stdout :- It is a global pointer variable which is defined in stdio.h
  → By using this global pointer, we can handle standard output buffer.

- stdin :- By using this global pointer, we can handle standard input buffer.

- stderr :- By using this global pointer, we can handle standard I/O related error.
  When we are working with stderr, it will redirect the data back to stdout.

- stdprn :- By using this global pointer, we can handle printer

- fseek() :- By using this predefined function, we can create the movement in file pointer.
  → fseek() requires 3 arguments of type FILE*, long integer & integer type.

  Syntax :- `int fseek(FILE* stream, long offset, int whence);`

  stream will provides file information
  offset is no. of bytes
  whence value is file pointer location

whence value can be recognized by using following constant values.

1. SEEK_SET :- This constant will pass the file pointer to beginning of the file.
2. SEEK_CUR :- This constant will provide constant position of file pointer
3. SEEK_END :- This constant value will send the file pointer to end of the file.

These all constant values can be recognised using INTEGER VALUES also.

SEEK_SET value is 0

SEEK_CUR value is 1

SEEK_END value is 2

rewind() :- By using this predefined function we can send the control to the beginning of the file

rewind() requires 1 argument of type FILE*

Syntax : `void rewind(FILE* STREAM)`

The behaviour of rewind() is similar to -

```
fseek(FILE*, 0, SEEK_SET);
```

ftell():- By using this predefined function, we can find the size of the file.

- ftell() requires 1 argument of type FILE* and returns long integer value.
- Generally ftell() returns file pointer posn, so if file pointer is pointing to eof character then it is equal to size of the file.

Syntax:- `long ftell(FILE *stream);`

remove():- By using this predefined function, we can delete a file permanently from Harddisk.

- remove() requires 1 argument of type const char* & returns int value

Syntax: `int remove (const char *filename);`

rename():- By using this predefined function, we can change the name of an existing file.

- rename() funcn requires 2 arguments of type const char* & returns int value.

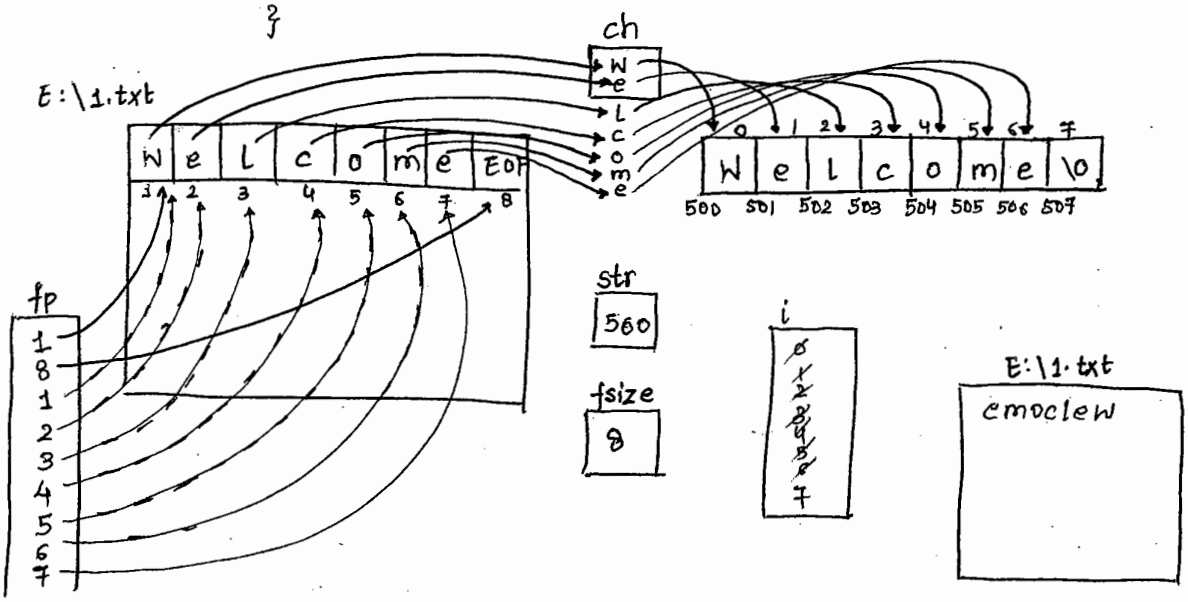
Syntax: `int rename (const char *oldname, const char *newname);`

---

prog: To reverse the string data present in a file

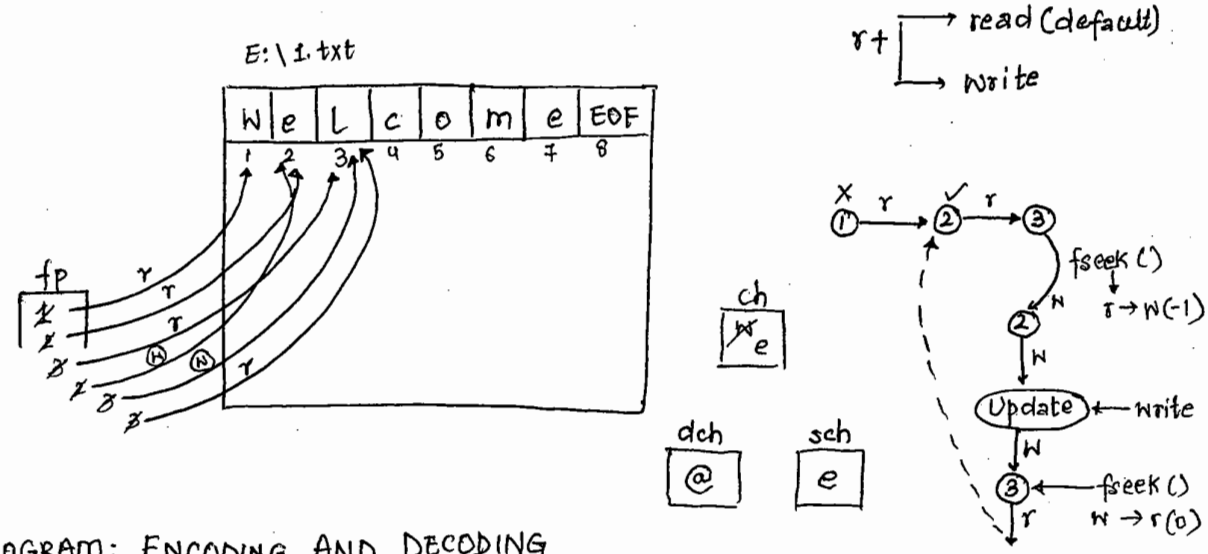
```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
int main()
{
  FILE *fp;
  char path[30];
  long int fsize, i=0;
  char *str;
  char ch;
  printf("Enter a file path:");
  gets(path);

  fp = fopen(path,"rt");
  if (fp == NULL)
   {
     printf("\n FILE NOT FOUND:");
     return 1;
   }
  fseek(fp, 0, SEEK_END);  // file pointer pointing to end of the file posn.
```

```
fsize = ftell (fp);  // ftell finds the no. of bytes holded by fp and that
                        is assigned to fsize.
fseek (fp, 0, SEEK_SET);  // rewind (fp)  fp points to the starting position
                                          of the file
str = (char*) calloc (fsize, sizeof (char));  // dynamically creating a string of
while(1)    // entering ∞ loop                   size fsize for all the data of file
{
   ch = fgetc (fp);  → getting each character from existing file.
   if (feof (fp))  → if gets end of file then break
     break;
   str[i++] = ch;  → Add each character to str[] & increment i
}
   str[i] = '\0';  → In string, we don't have EOF, we have \0 only
   fclose (fp);  → Save the file & close.
   remove (path);  → delete the existing file.
   strrev (str);  → reversing the string characters
   fp = fopen (path, "wt");  → then opening the same file in write mode
   fputs (str, fp);  → Writing character from str into fp pointing file
   // fprintf (fp, "%s", str);
   fclose (fp);
   free (str);  → freeing the m/m hold by str. string
   str = NULL;
   return 0;
}
```



PROGRAM:- FIND AND UPDATE (Replace) A CHARACTER

```
# include <stdio.h>
# include <stdlib.h>
int main()
{
   FILE *fp;
   char path[50];
   char ch, sch, dch;
   printf("Enter a file path: ");
   gets(path);
   fp = fopen(path, "rt+");
   if (fp == NULL)
    {
     printf("\n FILE NOT FOUND:");
     return 1;
    }
    printf("\n ENTER a char (s): "); // Enter source character to be replaced
    fflush(stdin);
    sch = getchar();          // Get it from user
    printf("\n Enter a char (D): "); // Enter dest character to be replaced with
    fflush(stdin);
    dch = getchar();
    while(1)
    {  ch=fgetc(fp); // The character read by the file pointer
       if(ch == EOF)
        break;
       if(ch==sch)
       {
          fseek(fp, -1, SEEK_CUR); //r ---> w(-1)   The current position pointed by the file pointer is shifted 1 Byte back & mode changed from read to write automatically.

          fprintf(fp, "%c", dch); #

          fseek(fp, 0, SEEK_CUR); // w -----> r(0)
       }
    }
    fclose(fp);
    return 0;
}
```

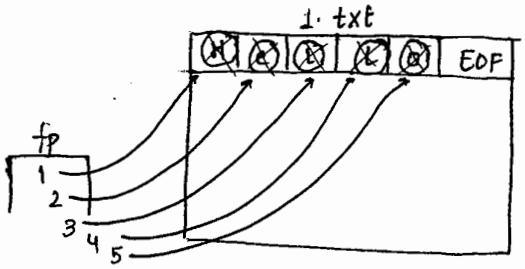


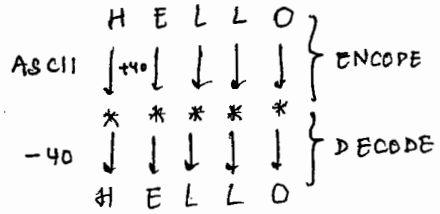
PROGRAM: ENCODING AND DECODING

```
#include <stdio.h>
#include <conio.h>
int main()
{
   FILE * fp
   int flag, code = 0;
   char path [30];
   char ch;
   clrscr();
   printf ("Enter a file path: ");
   gets(path);
   fp = fopen (path, "r+");
   if (fp == NULL)
  {
     printf ("\n FILE NOT FOUND");
     getch();
     return 1;
   }
   do
     {
       printf ("\n1 FOR ENCODE:");
       printf ("\n2 FOR DECODE:");
       scanf ("%d", &flag);
     }
     while (flag != 1 && flag != 2);   // When we enters i/p other then 1 & 2
     if (flag == 1)                    then do while repeats to take value
        code = 40;                     again by user can enter value other
     else                              than 1 & 2
        code = -40;
     while (1)
        {
```

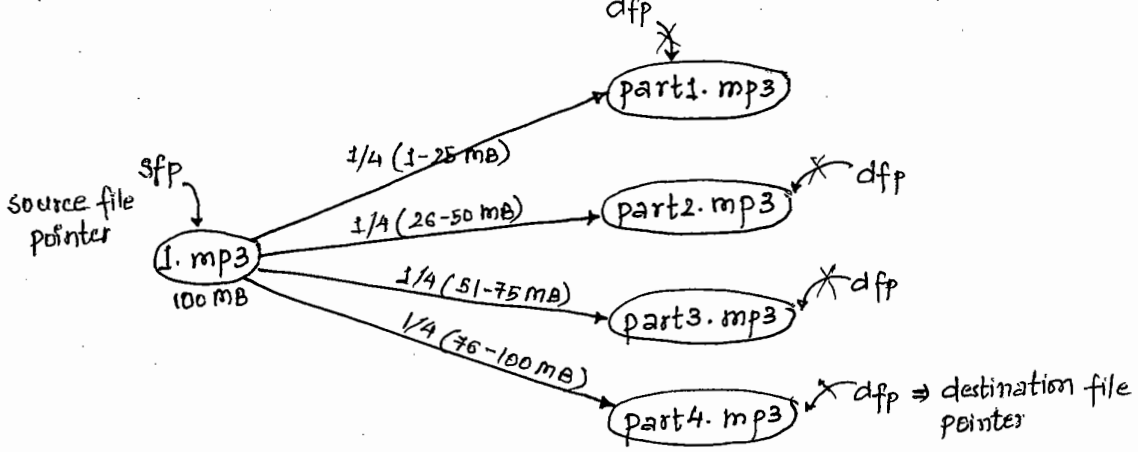
```
        ch = fgetc(fp);
        if (ch == EOF)
            break;
        if (ch != '\n' && ch != '\r')   → For encoding & decoding, we have to perform all other operations instead of these 2.
        {
            fseek(fp, -1, SEEK_CUR);        //r ---> H(-1)
            fprintf(fp, "%c", ch + code);   //update
            fseek(fp, 0, SEEK_CUR);         //w ---> r(0)
        }
    }
    fclose(fp);
    return 0;
}
```

Enter a file path: E:\1.txt





PROGRAM: OPERATIONS ON AUDIO FILE (Binary File)




```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define f1 "E:\\part1.mp3"
#define f2 "E:\\part2.mp3"
#define f3 "E:\\part3.mp3"
#define f4 "E:\\part4.mp3"
int main(void)
{
```

```
FILE * sfp;
FILE * dfp;
char spath [30];
char dpath [4][30] = {f1, f2, f3, f4};
long int fsize = 0, nb = 0;        nb → no. of bytes
int i = 0;
char ch
printf ("\n Enter sfp path:");
gets (spath);
sfp = fopen(spath, "rb");          rb → read binary file
if sfp == NULL)
{
  printf ("\n %s NOT FOUND", spath);     // This means file is already opened,
  return EXIT_FAILURE;                      now we need to find path of file
}
fseek (sfp 0, SEEK_END); // If the file is open then find the file size
 fsize = ftell (sfp);                    // gives filer path
  rewind (sfp);     //fseek(sfp, 0, SEEK_SET);
  dfp = fopen (dpath [i], "wb");  // To open 1st part
  if (dfp == NULL)  // If this condition is false then it means 1st part is created
{
  fclose (sfp);
   printf ("\n%s UNABLE TO CREATE");
   return EXIT_FAILURE;
}
  while (1)
  {
    ch = fgetc (sfp);
    if (feof (sfp))
    break;                                  } reading 1st part data
    fprintf (dfp, "%c", ch);
    ++nb;      ⇒ increments no. of bytes
     if (nb = fsize/4 && i = 3)  ⇒ we have to close only 3 initial parts within
                                   the loop [0,1,2] & not the 4th part, otherwise source
     {                             file will be terminated
       fclose (dfp);
        nb = 0;
        ++i;
        dfp = fopen (dpath [i], "wb");
        if (dfp == NULL)
      {
        printf ("\n %s UNABLE TO CREATE ", dpath [i]);
        fclose (sfp);                  \\part1 closed    // part2 closed
        return EXIT_FAILURE;
      {
```

```
        }
    }
    fclose(sfp);
    fclose(dfp);
    return EXIT_SUCCESS;
}
```

PROGRAM: TO COMBINE 2 OR MORE AUDIO FILES TOGETHER :

```
#include <stdio.h>
#include <stdlib.h>
#define f1 "E:\\part1.mp3"
#define f2 "E:\\part2.mp3"
#define f3 "E:\\part3.mp3"
#define f4 "E:\\part4.mp3"
int main(void)
{
    FILE *sfp;
    FILE *dfp;
    char dpath[30];
    char spath[4][30] = {f1, f2, f3, f4};
    int i = 0;
    char ch;
    printf("\nEnter dfp path: ");
    gets(dpath);
    dfp = fopen(dpath, "wb");
    if (dfp == NULL)
    {
        printf("\n%s UNABLE TO CREATE", dpath);
        return EXIT_SUCCESS
    }
    for(i = 0; i<4; i++)
    {
        sfp = fopen(spath[i], "rb");
        if (sfp == NULL)
        {
            fclose(dfp);
            printf("\n%s NOT FOUND", spath[i]);
            return EXIT_FAILURE;
        }
        while(1)
        {
```

```
    {
      ch = fgetc (sfp);
          if (feof (sfp))
            break;
       fprintf (dfp, "%c", ch);
    }
    fclose (sfp);
  }
  fclose (dfp);
  return EXIT_SUCCESS;
}
```

4/8/2015

COMMAND-LINE ARGUMENTS

- It is a procedure of passing the arguments to the main function from command prompt.
- By using command line arguments, we can create user-defined commands.
- In implementation, when we required to develop an application for dos operating system then recommended to go for command line arguments.
- DOS is a character based or command based OS i.e if we require to perform any task, we need to use specific command only.
- When we are developing the program in command line arguments, then main function will take 2 arguments i.e argc & argv
- argc is a variable of type an integer, it maintains total no. of arguments count value.
- argv is a variable of type char*, which maintains actual argument values which is passed to main().
- In C and C++, by default total no. of arguments are 1, i.e programname.exe

Ex-
```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;
    printf("\nTotal No. of Arguments: %d", argc);
    for(i=0; i<argc; i++)
    printf("\n %d. Argument: %s", i+1, argv[i]);
    return 0;
}
//save the file as mytest.c
// compile mytest.c
// Build or rebuild mytest.c
```

Ex

- Load the command prompt & change the directory to specific location where application file is available.
- To execute the program, just we required to use program name or program name.exe
- CommandLine Arguments related programs not recommended to execute by IDE because we can't pass the arguments to main-function.

```
O/P: D:\C400PM\4>mytest 10 Hello 20
     Total No of Arguments : 4
  1. Argument: mytest
  2. Argument: 10
  3. Argument: Hello
  4. Argument: 20
```

→ argc & argv are local variables to main() so command Prompt related data we can't access outside of the main()

→ In implementation, when we required to access command prompt related data outside of main(), then recommended to go for _argc & _argv variables, which is defined in dos.h.

```
#include <stdio.h>
#include <dos.h>
   void abc()
   {
      int i;
      printf("\n Data in abc:");
      printf("\n Total No of arguments : %d", _argc);
      for(i=0; i<_argc; i++)
      printf("\n %d. Argument : %s", i+1, _argv[i]);
   }
   int main(int argc, const char*argv[])
     {
        int i;
        printf("\n Data in main :");
        printf(
        for(i=0; i<argc; i++)
            printf("\n %d Arguments: %s", i+1, argv[i]);
        abc();
        return 0;
        }
          // Save as mycmd.c
          // Compile mycmd.c
          // Build are Rebuild mycmd.e
```

```
O/P : D:\C400PM> mycmd Hello 10 Welcome

Data in main :
  Total No of Arguments : 4
  1. Argument : D:\C400PM\MYCMD.EXE
  2. Argument : Hello
  3. Argument : 10
  4. Argument : Welcome

Data in abc :
  Total No of Arguments : 4
  1. Argument : D:\C400PM\MYCMD.EXE
  2. Argument : Hello
  3. Argument : 10
  4. Argument : Welcome
```

Due to local variables

Due to global variables - argc & argv

- By using command Line Arguments, it is possible to pass any type of data but all are treated like string only.
- When we are working with Command line Arguments, always recommended to convert string type to numeric type properly before execution of the logic
- Conversion related all predefined functions are declared in stdlib
- stdlib related predefined functions are -

<stdlib.h>

atoi()	atol()	atof()	_atold()	ltoa()
ltoa()	abort()	exit()	_exit()	_c_exit()
atexit()	raise()	signal()		
min()	max()	random()	randomize()	
rand()	srand()	abs()	labs()	
strtod()	strtol	strtold()	strtoul()	
wctomb()	wcstombs()	fcvt()	gcvt()	
lfind()	lsearch()	_lrotl()	_lrotr()	div() ldiv()
Constants	EXIT_SUCCESS	EXIT_FAILURE		

1) atoi() - By using this, we can convert a string into integer type
   - atoi() function requires 1 argument of type char* & returns int type

Syntax :- `int atoi(const char *str);`

2) atol() :- By using this, we can convert a string into long integer type.

Syntax: `long atol (const char* str);`

3) atof() :- By using this predefined function, we can convert a string into double datatype.

Syntax: `double atof (const char* str);`

4) _atold() :- By using this predefined funcn, we can convert a string into long double type.

Syntax: `long double _atold (const char*str);`

By using fcvt(), gcvt(), ecvt(), we can convert float to string type.

USER-DEFINED FUNCTION for atoi()

```
#include<stdio.h>
#include<stdlib.h>
int myatoi(const char*str)
{
  int r=0; i=0; flag=0;
  if(str[0]=='+'||str[0]=='-')
    {
      i=1;
      if(str[0]=='-')
        flag=1;
    }
  for(;str[i]!='\0';i++)
    {
      if(str[i]>='0'&&str[i]<='9')
        r=r*10+str[i]-'0';
      else
        {
          if(flag==0)
            return r;
          else
            return(-r);
        }
    }
  if(flag==0)
    return r;
  else
    return(-r);
}
```

```
int main(int argc, const char *argv[])
{
  int i, n;
   if (argc != 2)
   {
    printf ("\n invalid command syntax");
     return EXIT_FAILURE;
   }
   // n = atoi(argv[1]);
     n = myatoi(argv[1]);
     for(i=1; i<=10; i++)
         printf ("\n%d* %2d = %2d", n, i, n*i);
      return EXIT_SUCCESS;
}
   //save as mytab.c
   // compile mytab.c
   // Link or Build mytab.c
   // Run mycmd.exe using command prompt
```

Explaination

1) my tab 25
   mytab +25
   mytab 25ABC
   mytab +25ABC
   } 25

3) mytab abc
   mytab
   mytab
   mytab
   mytab
   } 0

2) mytab -25
   mytab -25ABC
   } -25

<ctype.h>

- character type related specific functions are available in ctype.h
- ctype.h related predefined functions or MACROs are-
  1. isalpha(); - checking alphabet or not
  2. isascii(); - checking I/P data within ascii range or not.
  3. iscntrl(); - If control button from keyboard
  4. isspace(); - If space button from keyboard
  5. isupper(); - required to use from A to Z
  6. isdigit(); - checking whether numeric type or not
  7. islower(); - To check if it is a-z
  8. tolower(); - convert upper to lower
  9. toupper(); - convert lower to upper
  10. toascii(); - converts to ascii value

→ When the function prefix 'is' available, then it returns boolean type i.e true or false

→ If 'to' is available, then it returns integer value

```
# include <stdio.h>
# include <ctype.h>
  int main ()
{
   char ch;
   int flag;
   printf ("Enter a char: ");
   ch = getchar();
   flag = isdigit (ch);
   if (flag != 0)
      printf ("\n %c IS DIGIT DATA", ch);
   else
      printf ("\n %c IS NOT DIGIT DATA", ch);
      return 0;
   }
```

O/P:

```
Enter a char: 9
9 IS DIGIT DATA
```

```
# include <stdio.h>
# include <ctype.h>
 int main()
 {
    char ch1, ch2;
    printf ("Enter a char: ");
    ch1 = getchar();
    ch2 = tolower (ch1);
    printf("\n% Lower case char is : %c", ch1, ch2);
     return 0;
    }
```

O/P:

```
Enter a char: A
A Lower case char is : a
```

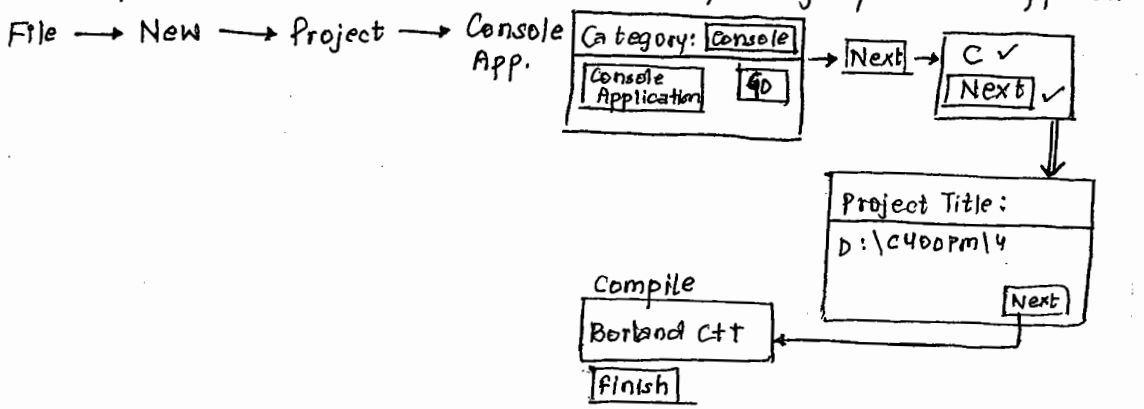
PROJECT-1

Q Create a console based applicated using code Block IDE with the name of Employee Management System with following modules.

1) Login Module
2) Add new record
3) Display records
4) Search specific record
5) Update existing records
6) Delete existing records
7) Display backup data

CONFIGURATION OF PROJECT :-

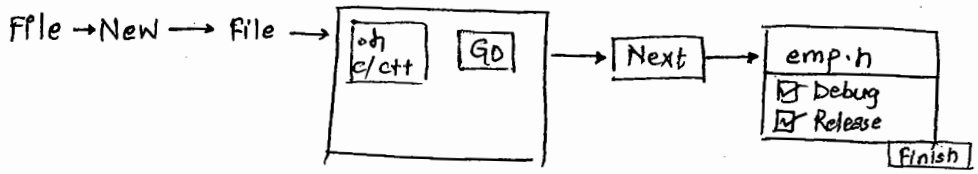
Open CODE BLODE ; create new project of category console Applicatin



⇒ To maintain employee information, we require a structure that's why it is recommended to create user defined datatype by using structures.

⇒ Always recommended to create a structure in header files only. [user-defined header files]

⇒ To create user defined header files, it is recommended to following steps :-

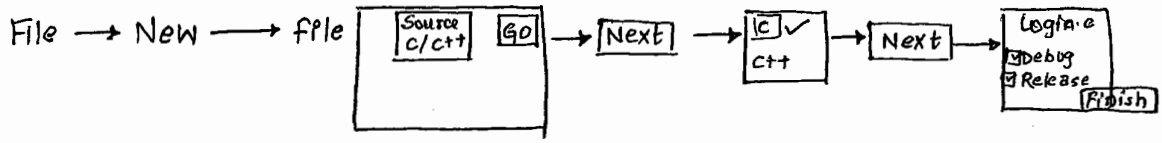


Code in emp.h

```
#ifndef EMP_H_INCLUDED
#define EMP_H_INCLUDED
 typedef struct
  {
    char ID [10];
    char NAME [36];
```

```
    char EMAIL [20];
    unsigned int SALARY;
    float PF;
} EMP;
# endif // EMP_H_INCLUDED
```

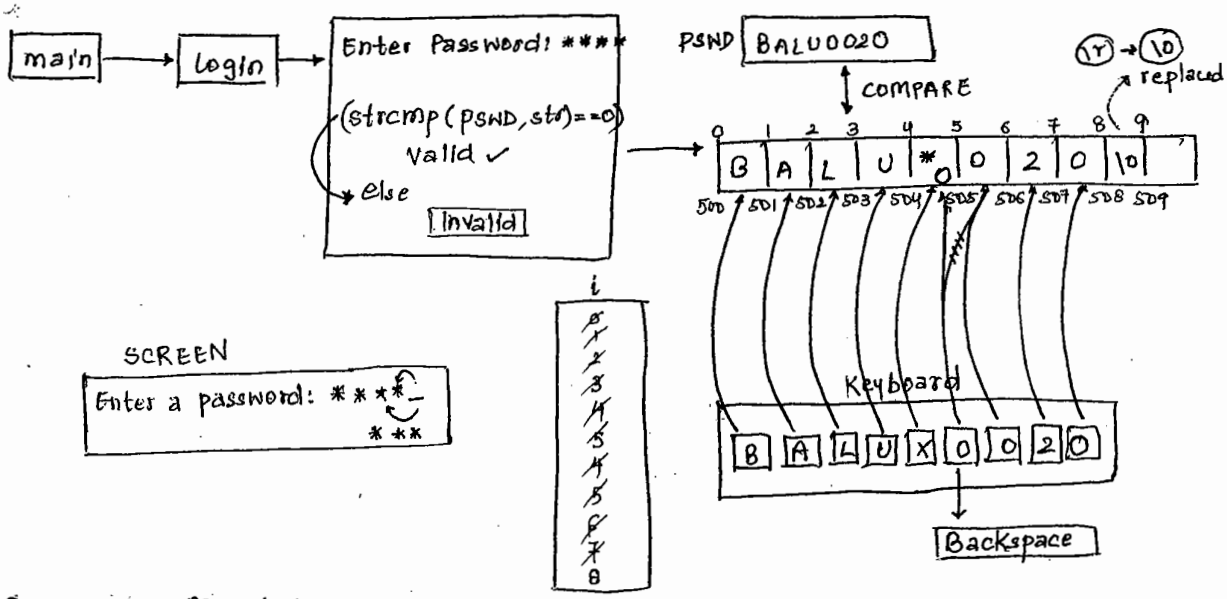
- Create a new module file with a name login.c



CODE IN LOGIN.C

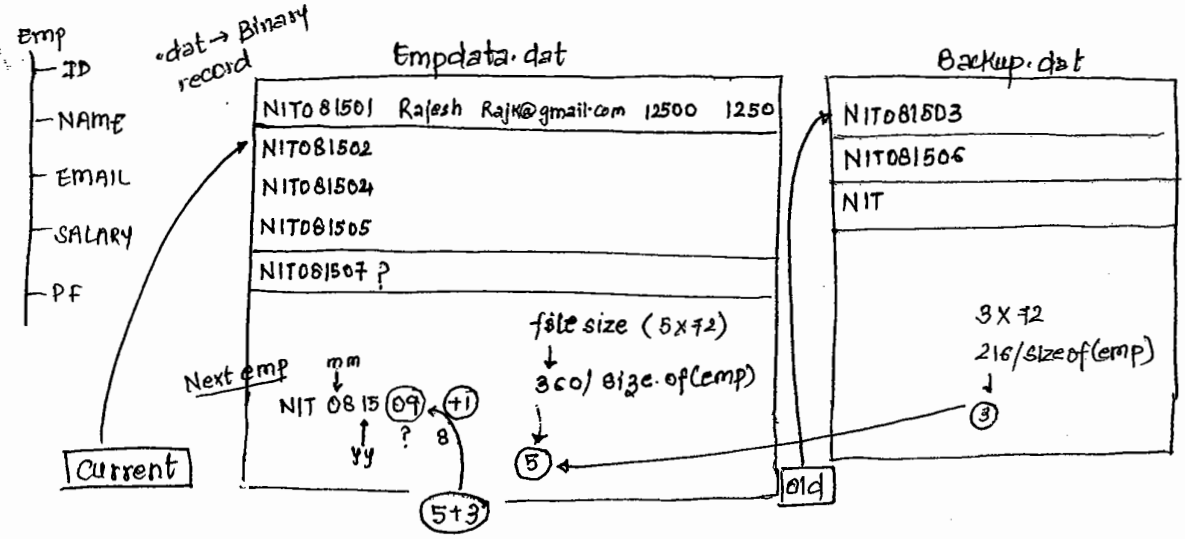
```
# Include <stdio.h>
# include <conio.h>
# include <string.h>
# include <process.h>
Void password()
{
  char pswd[10] = "BAL00020";
  char str[10];
  int i = 0;
  void mainmenu(void);
  printf(" Enter password");
While(1)
{
   str[i] = getch();
   if (str[i] == '\r')  // Enter button action
    {
      str[i] = '\0';
      break;
    }
   else if (str[i] == 0)  // backspace action
   {
      if (i>0)        // Min limit of backspace
      {
        printf ("\b");  // backspace
        printf(" ");  // remove * replace with 'space'
        printf("\b");  // backspace
        --i;
        str[i] = '0';
      }
   }
```

```
else
   {
     printf ("* ");
      ++i;
   }
 }
if (strcmp (pswd, str) == 0)
  {
    mainmenu();    // for next view of project
  }
else
   {
    printf ("\n invalid password: ");
    exit (1);
   }
 }
```



CODE IN MAIN.C

```
# include <stdio.h>
# include <stdlib.h>
int main()
{  void password(void)
   password();
   return 0;
}
```



CREATE A NEW MODULE FILE WITH THE NAME empcount.c

CODE IN empcount.c

```
#include <stdio.h>
#include <emp.h>
/* Application global variables */
    FILE *cur;
    FILE *old;
    EMP e;
    int ne;
int curcount()
{
   int n;
   cur = fopen ("EMPDATA.dat", "rb");
   if(cur==NULL)   // When the database file docsnot exist (EMPDATA)
        return 0;   // No Records
   fseek(cur,0, SEEK_END); // Will reach end of the file & will give total no. of bytes
   n = ftell(cur) / sizeof(EMP); // n will give total no. of current records
   fclose(cur);
    cur = NULL;
   return n;       // n returns total no. of records
}
  int oldcount()
  {
     int n;
      old = fopen ("BACKUP.dat", "rb");
      if (old == NULL)
           return 0;
      fseek (old, 0, SEEK_END);
       n = ftell (old)/ sizeof (EMP);
       fclose (old);
```

```
    old = NULL;
    return n;
}
```

SQLite Database

→ It is a light weighted server less open source database engine.

→ When we are working with this database it doesnot require any pre configuration and it doesnot require server also.

→ Generally this database is used in mobile application.

→ When we are developing stand alone applications then recommended to go for SQLite database.

→ Standalone application means, the system in which application is running in same system database is available.

→ The complete reference of SQLite database is available in www.sqlite.org.

→ Document reference is available in www.sqlite.org/docs.html

→ In 'C' language when we are developing standalone application with database they recommended to go for SQLite in place of using FILE system.

→ In 'C' language we doesnot have any kind of predefined functions to interact with SQLite database.

→ In implementation when we are interacting with SQLite database engine then recommended to go for

API'S which is provided by SQLite database.

→ SQLite related all API's are developed in native 'C' language.

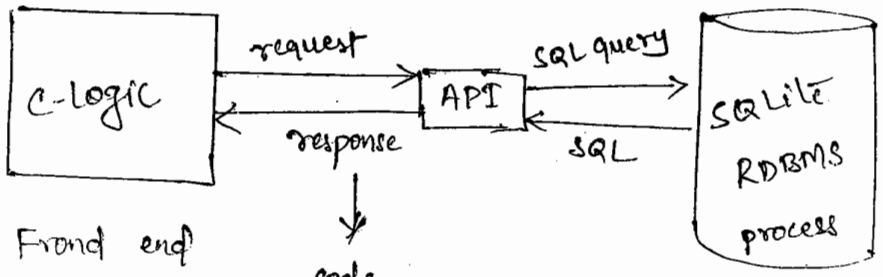
→ Complete API reference we can get from www.sqlite.org/cintro.html

→ Function means when we are performing the task within the program.

→ API means b/w the application when we need to perform.

→ In 'C' application when we are performing the subtask they we'll go for the function, if 'C' application is interacting with any other application they go for API. (Application programming Interface).

C-Application



Front end

Back end

code

Success (0)

(or)

failure (non-zero)

Configuration of SQLite database with Dev C++ IDE.

→ When we are working with Dev C++ IDE it provides multiple packages for different type of application development like 2D or 3D Graphics, Cryptography, Image manipulation, Networking, Database, multithreading and many more.

→ When we are working with any kind of packages first we require to config that specific package.

→ Goto devpaks.org website then select database category.

→ From list of libraries recommended to select sqlite3 ~~vers~~ library version: 3.7.4 (up to now it is latest)

→ After selecting specific database click on download url and wait until download is finished.

→ Once download is finished run package installer file. Then automatically configuration is completed.

Configuration of SQLite database with current project

→ Create a new project directory with the name SQLite_emp.

→ Go to www.sqlite.org website then click on download tab.

→ Under source code section download sqlite-amalgamation-xxxxxx.zip file.

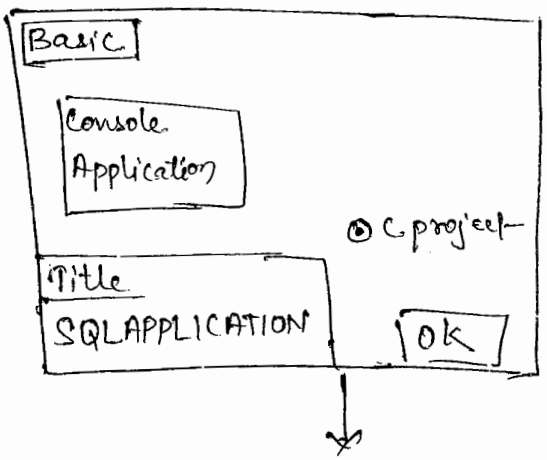
→ Extract the downloaded zip file which is having following files

- shell.c
- sqlite3.c
- sqlite3.h
- sqlite3ext.h

→ Copy sqlite.c and sqlite3.h into newly constructed folder i.e SQLiteemp.

→ Open Dev C++ IDE and create new console application type project with the name SQL APPLICATION and save this project template into SQLiteemp folder which was created earlier.

File → New → Project



Save project template & main.c in SQLiteemp directory only.

→ Go to project menu and select Add to project.

→ Select- Sqlite3.c & sqlite.h files. then click on open button.

25/08/2015

Syntax to open and close the database

→ For openning the database we have an API called sqlite3_open(),

for closing the database we have an API called sqlite3_close().

→ For handling the complete application we are creating three application variables' i.e!-

→ sqlite3 *db; // database pointer

→ char* errmsg; // char type pointer

→ int * errcode; // global int variable

→ sqlite3 is a predefined structure which is available in "sqlite3.h", Using this structure we can handle sqlite database.

2015

→ sqlite3_open():-

→ Using this API we can open SQLite database

→ This API requires two arguments of type const char* and sqlite3**.

→ On success this API returns zero, On failure it returns non-zero value.

```
rcode = sqlite3_open("EMPDATABASE.db",&db);
if (rcode==0)  // success
{
   // go for next step
}
else
printf("\n Error in opening database : %s",
        sqlite3_errmsg(db));
```

sqlite3_errmsg() :-

→ Using this API we can find any kind of error which is occurred at the time of working with SQLite data base.

→ This API requires one argument of type sqlite3 *** and returns char*.

⇒ sqlite3_close() :-

→ Using this API we can close a database which is opened by sqlite3_open() API.

→ sqlite3_close() requires one argument of type sqlite3*.

Syntax to create table

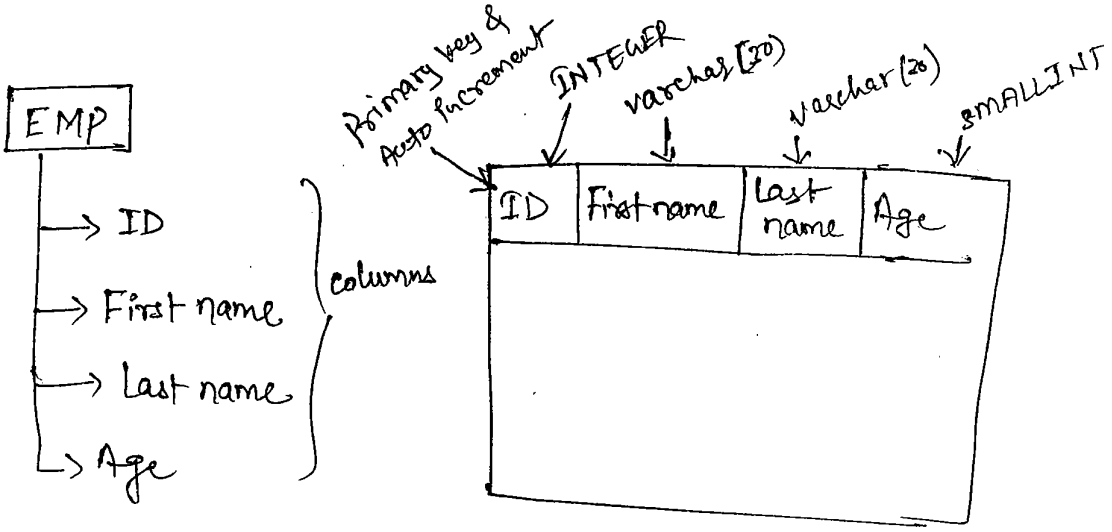
→ A table is a basic element of any kind of database.

→ Table maintains the information in the form of rows and columns with the help of RDBMS rules.

→ When we require to create any kind of table then we need

to use create table statement.

→ When we are creating table initially we need to decide ~~skeleton~~ skeleton or structure of table.



Syntax:-

Create table <tableName> (column1 Datatype, column2 Datatype, column3 Datatype, ...)

Ex:-

```
create table if not exists EMP (
ID INTEGER PRIMARY KEY AUTOINCREMENT,
FirstName varchar(30), Last-Name varchar(30),
Age smallint)
```

→ The last query is available in front end only and it is not understandable to 'c' compiler that's why this query is needed to be passed to back end using API.

sqlite3_exec()

→ Using this API we can execute any type of SQL query.

→ When we are using this API we need to pass database pointer, Query string, attributes and character pointer

→ If the query execution is success then we will get return value SQLITE_OK i.e. 0 else it returns positive value.

⇒ char* sql_stmt = " create table if not..... ";

```
if (sqlite3_exec(db, sql_stmt, NULL, NULL, &errmsg)
        != SQLITE_OK)
{
 //failure stop
}
else    // success
fprintf (stdout, "\n Table is created");
```

sqlite3_free():-

→ Using this API we can release dynamically created memory ~~for~~ by SQLite database.

Adding rows in table

→ When we are adding the data in table we need to use insert statement.

↔ Using insert statement we can add all columns data or we can pass specific column data also

Syntax 1:-

```
insert into tableName values (value1, value2, value3, ...);
```

Ex:- ✗

```
Insert into EMP values (1, 'Rajesh', 'Kumar', 26);
```

✗ Not suitable because ID has auto incrementation.

Syntax 2:-

```
insert into tableName (column1, column2, ...) values(
value1, value2, value3, ...);
```

✓

Ex:-

```
insert into EMP (FirstName, LastName, Age) values(
'Rajesh', 'kumar', 26);
```

✓

EMP

ID	First name	Last name	Age
1	Rajesh	Kumar	26

Back end

C Application

Enter First name: Rajesh
Enter last name: kumar
Enter Age: 26

Front end

sqlite3_exec();

fname: Rajesh
lname: Kumar
Age: 26

sprintf (sql_strat, " insert into EMP (Firstname, Lastname, age) values ('%s', '%s', %u", fname, lname, age);

copy

insert into EMP (Firstname, Lastname, Age) values ('Rajesh', 'Kumar', 26).

ID
1
2
3

* sqlite3_prepare_v2()'-

→ Using this API we can prepare the database which can take huge memory from front end application.

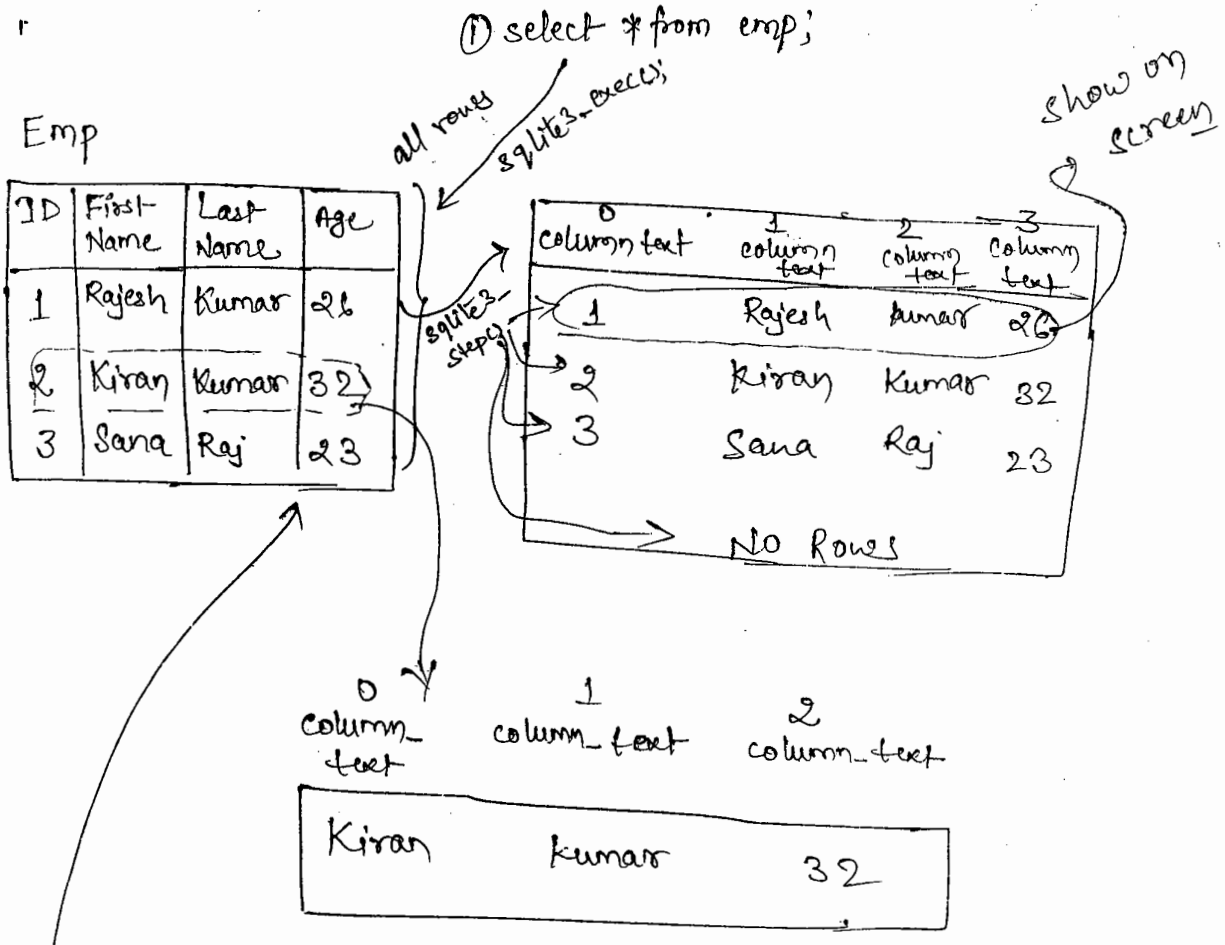
Procedure to retrieve the data from table

→ When we require to retrieve the data from table then we need to use select statement.

→ By using select statement we can retrieve all rows or any specific row also possible.

Emp

ID	First Name	Last name	Age
1	Rajesh	Kumar	26
2	Kiran	Kumar	32
3	Sana	Raj	23



② Select (FirstName, LastName, Age) from Emp where ID == 2;

sprintf(sql_stmt, "select (Firstname, lastname, Age) from Emp where ID == %d", id);

sqlite_exec()

C Application

Enter Emp ID: 2

* sqlite3_step() :-

→ Using this API we can retrieve a single row at a time from memory.

→ When row is available then this API returns ~~sqlite~~ SQLITE_ROW, when all rows are finished then it returns SQLITE_DONE.

* sqlite3_column_text() :-

→ Using this API we can extract the content based on ~~ID~~ Index.

Procedure to update the Record.

→ Using update statement we can update the data in table.

→ When we are working with update query we require to use set keyword along with where condition.

Emp

ID	First Name	Last Name	Age
1.	Rajesh	Kumar	26
2.	Kiran	Kumar	32
3.	~~Saha~~ Raj	~~Raj~~ Sana	~~25~~ 32

sqlite_exec();

C-Application

Enter ID: 3
Enter new First name: Raj
Enter new Last name: Sana
Enter new age: 32

fname [Raj]
lname [Sana]
age [32]
id [3]

update emp set Firstname='Raj', Lastname='Sana', age=32 where ID==3

ID
1.
2.
~~3.~~

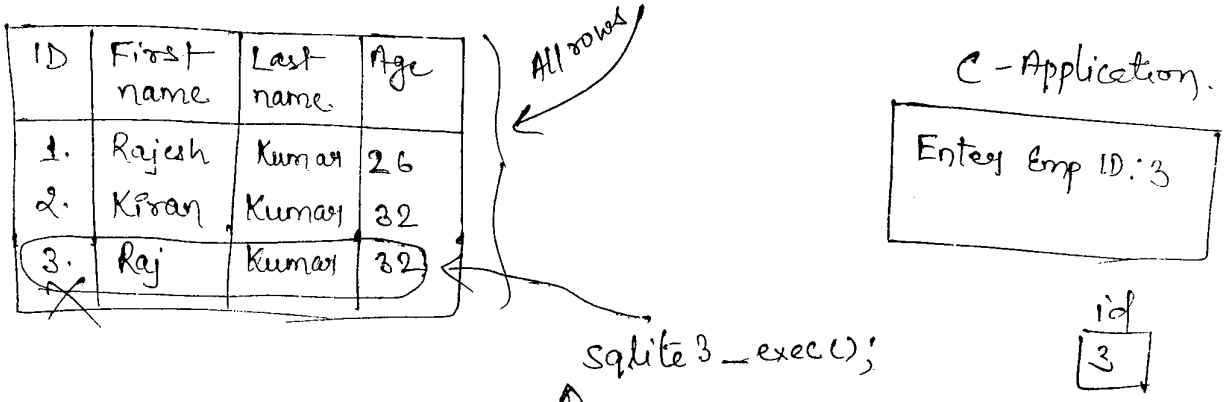
Sprintf(sql_stmt, "update Emp set FirstName=\'%s\', LastName=\'%s\', Age=%u" when ID==%u", fname, lname, age, id);

Deleting the data from table

→ Using delete statement we can delete the data from table.

→ Using delete statement we can delete all the row or any specific row also can be deleted.

① delete from Emp;



ID	First name	Last name	Age
1.	Rajesh	Kumar	26
2.	Kiran	Kumar	32
3.	Raj	Kumar	32

delete from Emp where ID == 3

sprintf (sql_stmt, "delete from Emp where ID == %d", id);

Deleting the table

→ Using drop statement we can delete table

→ `drop table emp;`

Code in main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include "sqlite3.h"

/* run this program using the console pauser or add your own getch,
system("pause") or input loop */

sqlite3*db;

int main(int argc, char *argv[])
{
        int option,ercode;
        ercode=sqlite3_open("test.db", &db);
        if(ercode)
        {
         //failed
         fprintf(stderr,"\nCan't open database: %s\n", sqlite3_errmsg(db));
         return 0;
   }
   else
   {
                // success
                while(1)
                {
                        system("CLS");
                        printf("\nFor Create Table---------->1: ");
                        printf("\nFor Insert Data----------->2: ");
                        printf("\nFOr Display Data---------->3: ");
                        printf("\nFor Update Table Data----->4: ");
                        printf("\nFor Delete Data----------->5: ");
                        printf("\nFor Delete Table---------->6: ");
                        printf("\nFor Exit Application------>7: ");
                        scanf("%d",&option);
                        switch(option)
                        {
```

```
                    case 1:create_table();
                            break;
                    case 2:insertdata();
                            break;
                    case 3:display();
                            break;
                    case 4:update_table();
                            break;
                    case 5:deletedata();
                            break;
                    case 6:droptable();
                            break;
                    case 7:
                            sqlite3_close(db);
                            return 0;
                default: printf("Invalid Option: \n");
                    system("PAUSE");

                                    break;
                    }
            }
        }
}
```

Code in createtable.c

```
#include <stdio.h>
#include <process.h>
#include "sqlite3.h"
void create_table()
{
  char*zErrMsg;
  extern sqlite3*db;
  char *sql_stmt="create table if not exists myTable (ID INTEGER PRIMARY
KEY AUTOINCREMENT,FirstName varchar(30), LastName varchar(30), Age
smallint)";
  if(sqlite3_exec(db,sql_stmt, NULL, NULL, &zErrMsg) != SQLITE_OK)
```

```
  {
   fprintf(stderr,"Failed to create table:%s\n",sqlite3_errmsg(db));
   sqlite3_close(db);
   sqlite3_free(zErrMsg);
        system("PAUSE");
        exit(1);
  }
  else
        fprintf(stdout,"Table is created\n");
  system("PAUSE");
  return;
}
```

Code in display.c

```
#include <stdio.h>
#include <malloc.h>
#include "sqlite3.h"
void displayallrow()
{
        char*query_stmt="select * from myTable";
        extern sqlite3*db;
        char*zErrMsg;
        sqlite3_stmt *statement;
        sqlite3_prepare_v2(db,query_stmt, -1, &statement, NULL);
   if (sqlite3_exec(db, query_stmt,0, 0, &zErrMsg) == SQLITE_OK)
        {
     printf("\n==================================\n");
          while(sqlite3_step(statement)== SQLITE_ROW)
     {
      printf("\n%5s %10s %3s %5s",sqlite3_column_text(statement,
0),sqlite3_column_text(statement, 1),sqlite3_column_text(statement,
2),sqlite3_column_text(statement, 3));
      }

     printf("\n==================================\n");
     sqlite3_finalize(statement);
        }
```

```
        else
        {
         fprintf(stdout,"\nError:%s",sqlite3_errmsg(db));
         sqlite3_free(zErrMsg);
        }
   getch();
        return;
}
void  displaysinglerow()
{
        unsigned int id;
        sqlite3_stmt*statement;
        extern sqlite3*db;
        char*zErrMsg;

   char *query_stmt=(char*)calloc(200,sizeof(char));
        printf("\nEnter ID: ");
        scanf("%u",&id);
        sprintf(query_stmt,"SELECT FirstName,LastName,Age FROM myTable
WHERE ID ==%u",id);;
   sqlite3_prepare_v2(db,query_stmt, -1, &statement, NULL);
   if (sqlite3_exec(db, query_stmt,0, 0, &zErrMsg) == SQLITE_OK)
        {
         if(sqlite3_step(statement) == SQLITE_ROW)
    {
      printf("\n==============================\n");
          //display here selected row data
     printf("\n%5s %3s %2s",sqlite3_column_text(statement,
0),sqlite3_column_text(statement, 1),sqlite3_column_text(statement, 2));
      fprintf(stdout,"\n1 Row is selected");
     printf("\n==============================\n");

    }
    else
    {
     fprintf(stderr,"\nNo Rows are selected");
    }
    sqlite3_finalize(statement);
```

```
    }
        else
        {
         fprintf(stdout,"\nError:%s",sqlite3_errmsg(db));
         sqlite3_free(zErrMsg);
        }
    free(query_stmt);
        getch();
        return;
}
void display()
{
        int option;
        printf("\nDisplay All Rows.....1: ");
        printf("\nDisplay single Row...2: ");
        scanf("%d",&option);
        if(option==1)
     displayallrow();
        else if(option==2)
          displaysinglerow();
        else
                fprintf(stderr,"\nInvalid Option: ");
    getch();
        return;
}
```

Code in updatedata.c

```
#include <stdio.h>
#include <malloc.h>
#include "sqlite3.h"
#define TRUE 1
#define FALSE 0
int isValidid(int tempid)
{
        sqlite3_stmt*statement;
        extern sqlite3*db;
```

```
        char*zErrMsg;
        char *query_stmt=(char*)calloc(200,sizeof(char));
        sprintf(query_stmt,"SELECT FirstName,LastName,Age FROM myTable
WHERE ID ==%u",tempid);;
   sqlite3_prepare_v2(db,query_stmt, -1, &statement, NULL);
   if (sqlite3_exec(db, query_stmt,0, 0, &zErrMsg) == SQLITE_OK)
        {
         if(sqlite3_step(statement) == SQLITE_ROW)
         {
          sqlite3_finalize(statement);
     return TRUE;   //tempid id valid
     }
     else
     {
       return FALSE;
       sqlite3_finalize(statement);
     }

   }
        else
        {
         fprintf(stdout,"\nError:%s",sqlite3_errmsg(db));
         sqlite3_free(zErrMsg);
         return FALSE;
        }
}
void update_table()
{
        char *fname;
        char *lname;
        unsigned int age;
        unsigned int tempid;
        sqlite3_stmt*statement;
   char *query_stmt;
   extern sqlite3*db;
   char*zErrMsg;
        printf("\nEnter ID: ");
        scanf("%u",&tempid);
```

```
        if(isValidid(tempid)==TRUE)
        {
              query_stmt=(char*)calloc(200,sizeof(char));
           fname=(char*)calloc(30,sizeof(char));
           lname=(char*)calloc(30,sizeof(char));
           fprintf(stdout,"Enter New First Name: ");
           fflush(stdin);
           gets(fname);
           fprintf(stdout,"Enter New Last Name: ");
           fflush(stdin);
           gets(lname);
           fprintf(stdout,"Enter New Age: ");
           fscanf(stdin,"%u",&age);
           sprintf(query_stmt,"update myTable set
FirstName=\'%s\',LastName=\'%s\',Age=%u WHERE ID
==%u",fname,lname,age,tempid);
           sqlite3_prepare_v2(db,query_stmt, -1, &statement, NULL);
      if (sqlite3_exec(db, query_stmt,0, 0, &zErrMsg) == SQLITE_OK)
        fprintf(stdout,"\nRow is Updated");
        }
        else
        {
              fprintf(stderr,"RECORD NOT FOUND");
        }
        free(query_stmt);
        free(fname);
        free(lname);
        getch();
        return;
}
```

Code in insertdata.c

```
#include <stdio.h>
#include <malloc.h>
#include "sqlite3.h"
void insertdata()
{
```

```
        char *fname;
        char *lname;
        unsigned int age;
        char ch;
        extern sqlite3*db;
        char*zErrMsg;
        sqlite3_stmt *statement;
        fname=(char*)calloc(30,sizeof(char));
        lname=(char*)calloc(30,sizeof(char));
        char *sql_stmt=(char*)calloc(200,sizeof(char));
        do
        {
         fprintf(stdout,"Enter First Name: ");
         fflush(stdin);
         gets(fname);
         fprintf(stdout,"Enter Last Name: ");
         fflush(stdin);
         gets(lname);
         fprintf(stdout,"Enter Age: ");
         fscanf(stdin,"%u",&age);

        sprintf(sql_stmt,"insert into myTable (FirstName, LastName, Age) values
(\'%s\', \'%s\',%u)",fname,lname,age);
        sqlite3_prepare_v2(db,sql_stmt, -1, &statement, NULL);
    if (sqlite3_exec(db, sql_stmt,0, 0, &zErrMsg) == SQLITE_OK)
    {
                fprintf(stdout,"\nNew Record inserted");
                fprintf(stdout,"\nDo you want to continue Y?: ");
                fflush(stdin);
                ch=getchar();
        }
        else
        {
          fprintf(stdout,"\nError:%s",sqlite3_errmsg(db));
          sqlite3_free(zErrMsg);
          ch=getchar();
    }
        }while(ch=='y'||ch=='Y');
```

```
        free(fname);
        free(lname);
        free(sql_stmt);
        return;
}
```

Code in deletedata.c

```
#include <stdio.h>
#include <malloc.h>
#include "sqlite3.h"
#define TRUE 1
#define FALSE 0
void deleteallrow()
{
        //delete all record from table
   extern sqlite3*db;
        char*zErrMsg;
        sqlite3_stmt*statement;
        const char *query_stmt="delete from myTable";
        char *sql_stmt=(char*)calloc(200,sizeof(char));
        sqlite3_prepare_v2(db,query_stmt, -1, &statement,NULL);
        if(sqlite3_exec(db, query_stmt,0,0,&zErrMsg)== SQLITE_OK)
        {
          if (sqlite3_step(statement) == SQLITE_ROW)
            fprintf(stdout,"\nrecords are deleted");
     else
          fprintf(stderr,"\nNO Rows are selected");
    sqlite3_finalize(statement);
 }
 else
        fprintf(stdout,"\nError:%s",sqlite3_errmsg(db));
   sqlite3_free(zErrMsg);
 getch();
 return;
}
void deletesinglerow()
{
```

```
 int isValidid(int);
 unsigned int tempid;
 extern sqlite3*db;
 sqlite3_stmt*statement;
 char*zErrMsg;
 char *query_stmt=(char*)calloc(200,sizeof(char));
 printf("\nEnter ID: ");
 scanf("%u",&tempid);
 if(isValidid(tempid)==TRUE)
 {
       sprintf(query_stmt,"DELETE FROM myTable WHERE ID ==
%u",tempid);
       sqlite3_prepare_v2(db,query_stmt, -1, &statement, NULL);
       if(sqlite3_exec(db, query_stmt,0, 0, &zErrMsg) == SQLITE_OK)
       {
             printf("\nRecord is Deleted");
             getch();
             return;
  }
 }
 fprintf(stdout,"\nRecord not Found");
       getch();
       return;
}
void deletedata()
{
       int option;
       printf("\nDelete All Rows.....1: ");
       printf("\nDelete single Row...2: ");
       scanf("%d",&option);
       if(option==1)
   deleteallrow();
       else if(option==2)
         deletesinglerow();
       else
             fprintf(stderr,"\nInvalid Option: ");
  getch();
       return;
```

```
}
```

Code in droptable.c.c

```
#include <stdio.h>
#include "sqlite3.h"
void droptable()
{
   extern sqlite3*db;
   char*zErrMsg;
        char *sql_stmt="drop table myTable";
        if(sqlite3_exec(db,sql_stmt, NULL, NULL, &zErrMsg)==SQLITE_OK)
        {
                fprintf(stdout,"\ndeleted table");
               sqlite3_free(zErrMsg);
                getchar();
                return;
         }
        else
        {
                fprintf(stderr,"\nError:%s",sqlite3_errmsg(db));
                sqlite3_free(zErrMsg);
                getchar();
                return;
         }
}
```

26/08/2015

Multithreading

→ Multithreading means simultaneously running multiple processes at a time.

→ Multithreading is an ability of an operating system which can allow multiple processes simultaneously.

→ Multithreading is possible in multiprocess ~~based~~ architecture based Operating System only.

→ Using multithreading we can develop parallel applications.

→ When we are working with DOS O.S. it doesnot support multithreading.

→ DOS is a single user, single task based Operating System. So doesnot allow to run multiple processes.

Process :-

→ When a compuley program is loaded into the memory for execution then it is called process.

Thread :-

→ A thread is a sequence of instructions within a process which can be executed independently from other code.

→ In C & C++ when we require to develop multiprocessor based application then go for pThread (POSIX)

Library.

→ ~~Pox~~ POSIX means Portable Interface of Unix operating system.

→ for Unix and Linux based compilers this library is built In.

→ When we are working with Windows based compiler then only explicitly we require, to config.

Integration of pThread Library with DevC++ IDE.

→ Open devpaks.org website and select POSIX category.

→ Click on pthread_w32 library version.

→ Wait until download is finished
-then run package installer file. They automatically pthread library is installed.

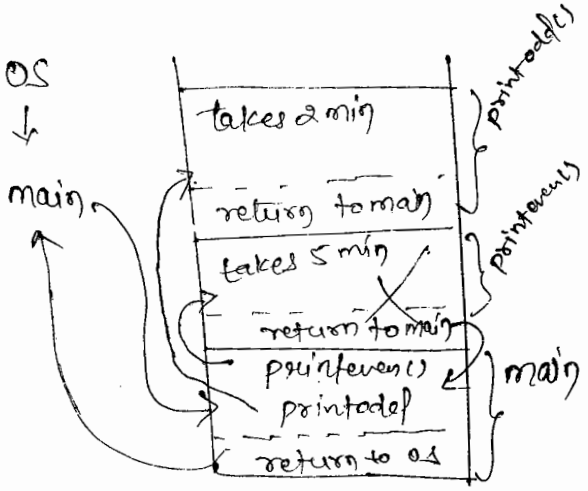
Single process based C application

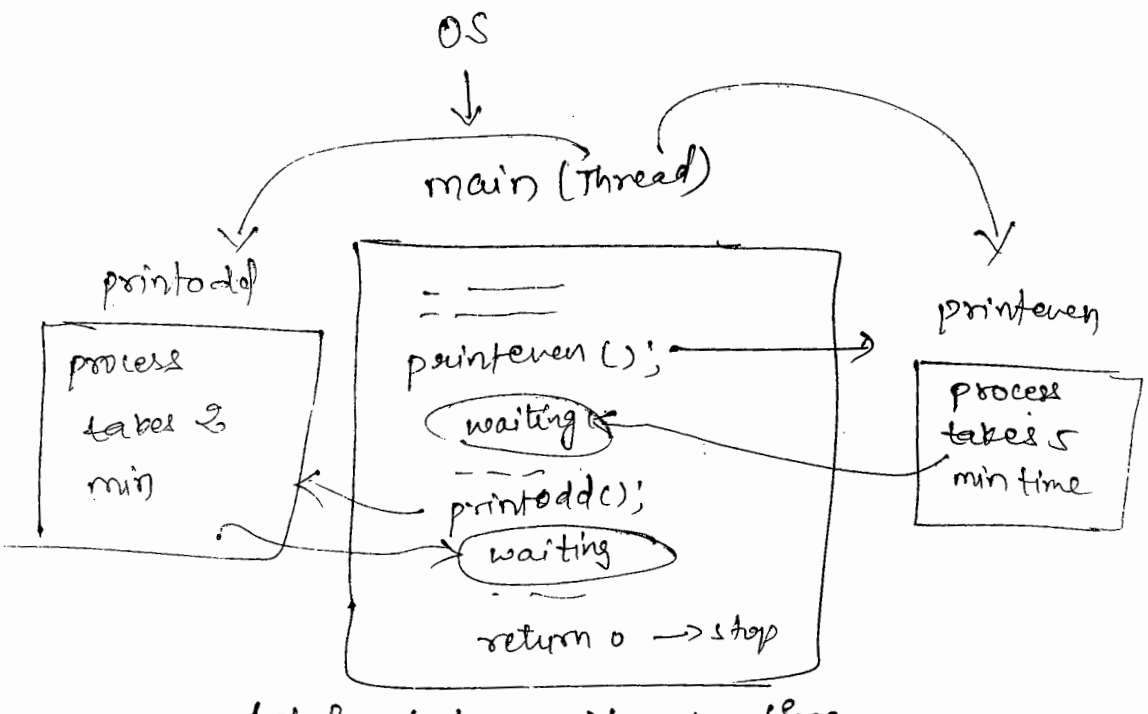
→ In normal C applications at a time we can execute one process only. or one thread can only be evaluated.

→ By default application is started from main() and as per the requirement main() can be call any other functions.

⇒
```
# include <stdio.h>
void printeven()
{ // printing all even no from 2 to 123456789
}
void printodd()
{ // printing all odd numbers from 1 to 123456
```

```
}
int main()
{
  ----
  printeven();
  ----
  printfodd();
  ----
  return 0;
}
```





total takes 7 min time

→ As per above observation application is started from main() and it is handled by thread.

→ As per the requirement main() is calling printeven() so thread is replaced with printeven() code so automatically main() goes in waiting status (Idle state) for 5 min. of time

→ When the main() is in waiting status no any instructions of main() can be executed.

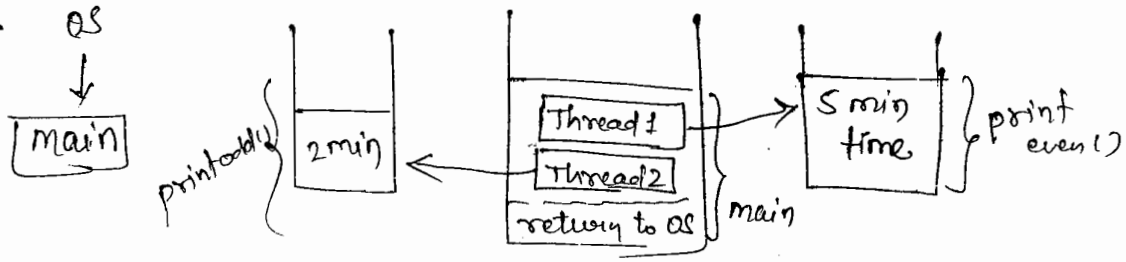
→ After 5 minutes of time thread is shifted back to main() and again it is replaced with printodd().

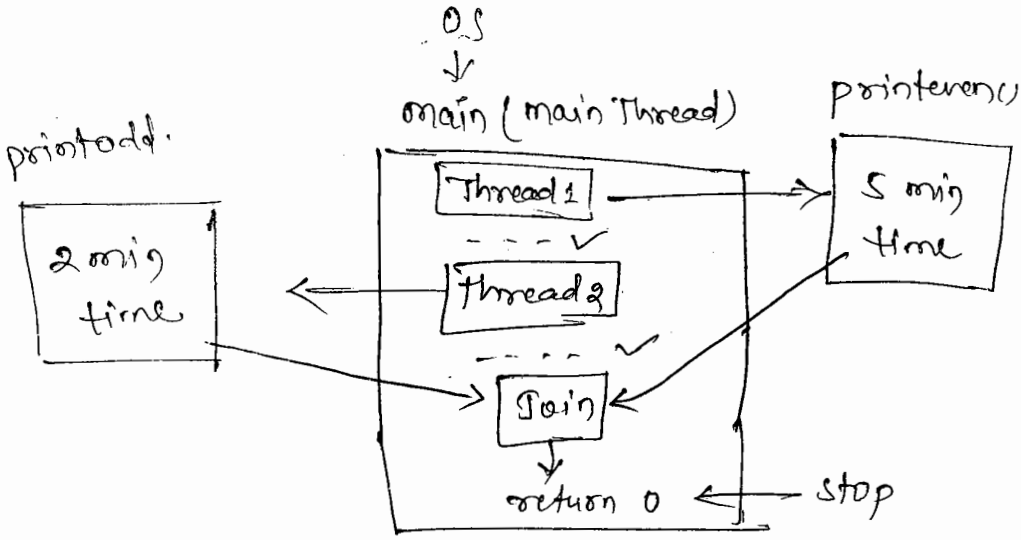
→ Where print-odd() is in the execution then main() will be in waiting status for 2 minutes of time.

→ To complete the process completely it takes 7 minutes to which is possible to finish in 5 minutes with the help of multiprocessing.

Multiprocess based C Application

→





2.

⇒

→ Open Dev C++ IDE and create a new console based C project with the name pthreadEx.

→ Save the project template into any specific drive.

→ Go to project menu and select project option, click on parameters tab then select Add library or object button.

→ Select libpthreadGC2.a file from C:\Program Files\Dev-Cpp\Lib

=>
```
#include <stdio.h>
#include <conio.h>
#include <pthread.h>

void * printeven (void * msg)
{
  int i;
  char* data = (char*) msg;
  for (i = 2; i<=1234; i+=2)
  printf ("\n%s=%d", data, i);
}

void * printodd (void *msg)
{
   int i;
   char *data = (char*) msg;
   for (i=1; i<=1234; i+=2)
   printf ("\n%s:%d", data, i);
}
int main (int argc, char *argv[])
{
   pthread_t thread1, thread2;
   int thread1_ercode, thread2_ercode;
```

```
thread1_ercode = pthread_create(&thread1, NULL,
    printeven, (void*)" from even");
    if (thread1_ercode != 0)
    {
        printf("\n Error in thread 1");
        return 1;
    }
    thread2_ercode = pthread_create(&thread2,
    NULL, printodd, (void*)" from odd");
    if (thread2_ercode != 0)
    {
        pthread_cancel(thread1);
        printf("\n Error in thread 2");
        return 1;
    }
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return 0;
}
```

Graphics

→ When we are applying visual affects in DOS based application then it is called Graphics.

→ Graphics related C applications doesnot works on High end processors because of resolution problem. (XP is supported)

→ When we are working with Graphics related applications then recommended to go for <graphics.h> header file.

→ <graphics.h> provides all the predefined functions prototypes which are related to graphics.

→ When we are working with graphic related functions mandory to find out EGAVGA.BGI file location.

→ Generally this file is available in C:\TC\BGI directory.

→ EGAVGA.BGI file provides graphics related all resources.

→ When we are working with graphics related applications then initially we require to convert Dos mode into graphics mode.

→ After completion of the program at end of the application we require to convert graphics mode into Dos mode.

initgraph() :-

→ Using this predefine function we can initialize the graphics.

→ At the time of initialization we will get initialization related errors also.

→ If initialization is done properly then we will get the return value grOk. or else negative return value is returned.

→ graphresult():-

→ Using this predefine function we can find initialization code i.e. success or failure.

⇒

⇒ grapherrormsg():-

→ Using this predefined function we can find graphics initialization related error messages.

→ This function requires one argument of type integer i.e. error code value.

⇒ cleardevice():-

→ Using this predefined function we can clear the data from console or graphics mode.

⇒ closegraph():-

→ Using this predefined function we can close the graphics i.e converts graphics mode to DOS mode.

setcolor()

→ Using this predefined function we can change the color in graphics mode.

— setbkcolor()

⇒ Using this predefined function we can change background color.

⇒
```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
#include <stdlib.h>
#include <dos.h>
#include <process.h>
int main()
{
 int gdriver = DETECT, gmode, errcode;
 int radius = 200, i=0, flag=0;
 int midx, midy;
 clrscr();
```

```
initgraph(&gdriver, &gmode, "C:\\TC\\BGI");
ercode = graphresult();
if (ercode != grOk)
 {
  printf("Error : %s", grapherrormsg(ercode));
  getch();
  return 1;
 }
  midx = getmaxx()/2;
  midy = getmaxy()/2;
  while (!kbhit())
  {
     setcolor(i);
     setbkcolor(15);
    if (flag==0)
        circle (midx, midy, radius--);
    if (flag==1)
        circle (midx, midy, radius++);
    delay (100);
    if (radius==0)
     {
       flag = 1;
```

```
            cleardevice();
        }
        if (radius == 200)
        {
            flag = 0;
            cleardevice();
        }
        ++i;
        if(i>15)
        i=0;
    }
    closegraph();
    return 0;
}
```

printing without any printing functions.

```
int main()
{
    char far *ptr = (char far *)0XB8000000;

    char far *ptr = (char far *)0XB8000000;

    *(ptr+0) = 'B';
    *(ptr+1) = 1;
    *(ptr+2) = 'A';
    *(ptr+3) = 2;
    *(ptr+4) = 'L';
    *(ptr+5) = 3;
    *(ptr+6) = 'U';

    return 0;
}
```