Rule 6

we can find size of an pointer using 'sizeof' operator.

16/10/12

Function

✓ self contained block of one or more statements or a subprogram, which is designed for a particular task is called a function.

Advantages of function :-

- By using function we can design an application in module format.
- when we are designing the application in module format, then easily we can debug the program.
- By using functions, we can reduce the coding part of the application / program.
- By using functions, we can keep track of what they are doing.

✓ The basic purpose of function is code reuse.

✓ A 'C' program is a collection of functions.

✓ From any function, we can invoke any another functions

✓ Always compilation process will happen from top to bottom.

✓ Execution process starts from main and ends with main only.

✓ On implementation when we are calling function, which is defined later for avoiding the compilation error, we require to go for forward declaration.

✓ Declaration of function means need to mention return type, name of the function and parameter type information.

✓ On function defination, first line is called function declarator or function header.

✓ Always function declaration should match with function declarator.

✓ On implementation whenever a function is not returning a value, then specify the return type as void.

✓ 'void' means nothing, i.e. no return values.

✓ On implementation, whenever a function returns other than 'void', then specify the return type as return value type; i.e. what type of value is returning, same type of return statement should be mentioned.

✓ Default return type of any function is an int.

✓ Default parameters of any function is void.

```
return_type function_name (parameters)
{
    statement-block;
    return statement;
}
```

- Return type, parameters and return statements are optional.
- On implementation, whenever a function is returning the value, then specifying the return statement is optional. On this case, compiler gives a warning message i.e. "function should return a value".
- When the function is returning the value, then collecting the value also optional.
- On C-programming language, functions are classified into two types :-
  1. Library functions / built-in functions / predefined functions
  2. User defined functions

① Library functions :-

- These all are setup pre-implemented functions, which are available along with compiler.
- The implementation part of the predefined functions are available in .LIB or .OBJ files in machine readable format.
- .LIB or .OBJ files contain precompiled code
- When we are working with predefined function, to avoid the compilation errors, we require to go for header files.
- Header file (.h files) does not provide any implementation part of predefined functions, it provides only forward declaration.

Limitations

- All predefined functions contain limited task only, i.e. for what purpose function is designed, for same purpose we require to use.
- As a programmer, we don't have any control on predefined functions, because coding part is there in precompiled format.
- As a programmer, we can't alter or modify the behaviour of any predefined function, ex:- printf(), pow(), sqrt(), atoi(), strcpy().
- In implementation, whenever a function is not supporting user requirements, then go for user defined functions.

(2) User defined functions :-

- As per the project requirement or as per the client requirement whatever the functions we are implementing, those are called user defined functions.
- As a programmer, we are having full control on user-defined functions, because coding part is available.
- As a programmer, when we require to alter or modify the behaviour of any user-defined function, then we can modify according to the requirement.
- User-defined functions are classified into 4 types :-

(i) function with no argument and no return type

(ii) function with argument and no return type

(iii) function with no argument and one return value.

(iv) function with arguments and return type.

Note:
- All predefined functions are user defined functions only, somewhere else another programmer has implemented these functions, so it became user-defined functions to that programmer and we are using in precompiled format, so it became predefined for us.

About main() :-

- 'main()' is an identifier in the program, which indicates startup point of an application.
- 'main()' is a userdefined function, with predefined signature for linker.
- Without using main function we can implement the program, but we can't execute. In this case, compilation is success, but execution is failure.
- Any application should be required only one function with the name called 'main()', if more than one 'main()' are implementation, then compiler gives an error; i.e. "multiple declarations for main()".
- If it is required to change the name of 'main()' function, then it is possible; but one more function is required to design with the name called 'main()'.
- Generally, main() function doesn't returns any values, that's why return type of 'main()' function is void.

- On implementation, when we require to provide the exit status of an application back to the O.S, then specify the main() function return type as an int.

- void main() doesnot provide any exit status back to the operating system.

- int main() provides the exit status back to the operating system; i.e. success or failure.

- When the main() function return type is int, then it is possible to return the values from -32768 to +32767, but those all are meaningless exit status, except '0' and '1'.

- When the exit status need to be informed as a success, then return value should be '0'; i.e. "return 0;".

- When we need to inform the exit-status as failure, then return value as '1' i.e. "return 1;".

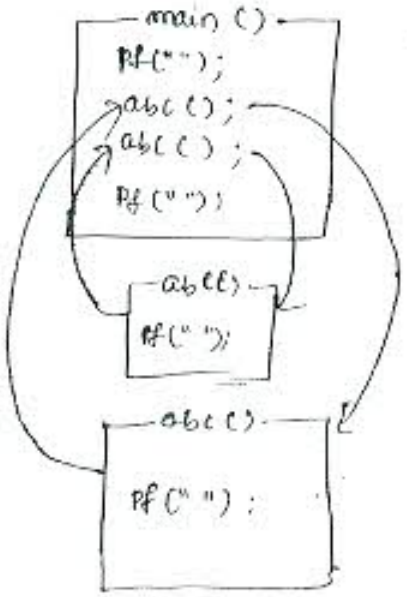
- When user is terminating the program explicitly then exit code will be '-1', i.e. "return -1".

11/01/12

(ex1)
```
void abc() // &abc
{
  Printf("\n Hello abc");
}
void main() // &main
{
  Printf("\n Hello main1 ");
  abc(); // calling, &abc()
  abc(); // calling, &abc()
  Printf("\n Hello main2");
}
```

(op)
```
Hello main1
Hello abc
Hello abc
Hello main2
```

↳ Compilation from top to bottom
↳ Execution from main() method.

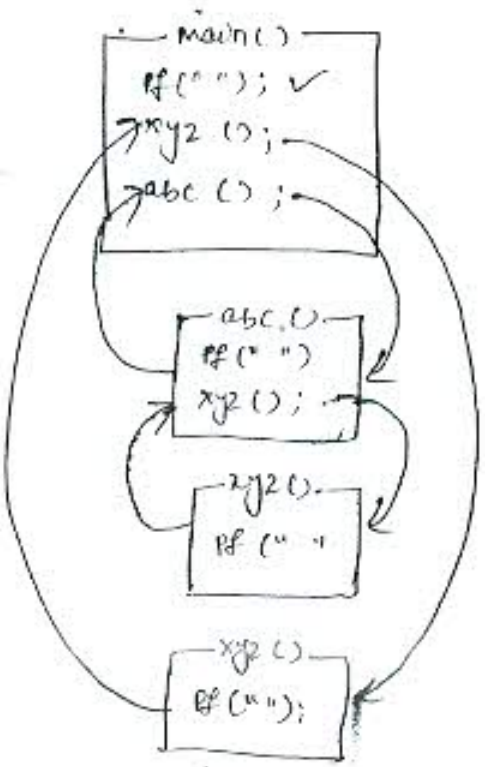


- Always compilation process takes place from top to bottom.
- Inorder to compile the code, if any functions are occured, then with that particular function name, one unique identification value will be created, called function address.
- Whenever we are calling any function, that calling statement will be substituted with corresponding function address, so this process is called binding procedure.
- With the help of binding process only, compiler recognises that which function need to be called at the time of execution.

(ex2)
```
void xyz()  // &xyz
{
    printf("\n welcome xyz()");
}
void abc()  // &abc
{
    printf("\n welcome abc()");
    xyz();  // calling, &xyz()
}
void main()  // &main()
{
    printf("\n welcome main()");
    xyz();  // calling, &xyz()
    abc();  // calling, &abc()
}
```

(op)

welcome main()
welcome xyz()
welcome abc()
welcome xyz()



- When we are working with the functions, it can be implemented randomly; i.e. in any sequence functions can be implemented.
- From any function, we can call any other functions.
- After execution of any function, control should be passed back to the "calling location" automatically; i.e. from which function, it is called, to same function it will be passed back.

e.g.
```
void main()
{
   printf("\n welcome main");
   abc();
}
void abc()
{
   printf("\n Hello abc");
}
```
O/P: error: type mismatches in redeclaration of 'abc'.

①
```
void abc()
{
}
void main()
{
   abc();
}
```
invalid

②
```
void main()
{
   abc();
}
void abc()
{
}
```
abc(); → ① address ② type information
- return type (void)
- Parameter (void)
- no. of parameters (0)

mismatch → invalid

✓ at the time of compilation:-
creates ① Address of function
② type information
- return type (void)
- parameters (void)
- no. of parameters (0)

③
```
void main()
{
    printf("\n main()");
    abc();
}
int abc()
{
    pf("\n abc()");
}
```

→ ① address
② type info
→ return type (int)
→ parameter (void)
→ no. of parameter (0)

→ no error : o/p :- main()
abc()

- whenever any function is compiled, alongwith the function address type information will be maintained by compiler.
- Type information means return type of the function, parameter type, and no. of parameters.
- If the function is compiled before calling, then all type informations are available, but when the function is compiled after calling, then automatically default informations will be maintained by compiler.
- Default type information is :-
  - return type → int,
  - parameter type → void, &
  - no. of parameters → '0'.
- In above Program (2), due to default type information compiler is expecting an integer as a return value, but actual return type is void, so mismatch is occurred.
- In implementation, when we are calling a function, which is defined later, then for avoiding the compilation error, we require to go for forward declaration
- Declaration of function means, need to mention return type, name of function, and parameter type information.
- forward declaration of a function provides type information explicitly.

(2nd)
```
void main()
{
    void abc(void); // declaration
    // void abc();
    printf("\n welcome again");
    abc();
}
void abc()
{
  printf("\n welcome abc");
}
```

(o/p)

```
welcome main
welcome abc
```

(3rd)
```
void main()
{
    printf("\n welcome abc");
    xyz();
}
void main()
{
    printf("\n Hello main");
    xyz();
    abc();
}
void xyz()
{
  printf("\n welcome xyz");
}
```

(o/p) :- Error :

Type mismatch in redeclaration of 'xyz'.

- In order to call the 'xyz()' function in abc() and main(), we required to go for forward declaration of 'xyz()' in 'abc()' and 'main()' also.
- In implementation, when we are declaring a function more than once, then recommended to go for global declaration of the function in place of declaring multiple times.
- whenever we are declaring a function top of the program before defining a function, then it is called global declaration.
- when the global declaration is available, then it's not required to go for local declaration, if the local declaration also available, then global declaration will be ignored.

(Exc)

```
void xyz();//global declaration
void abc()
{
  //void xyz(); // local declaration
   Printf("\n welcome abc");
   xyz();
}
void main()
{
    // void xyz(); // local declaration
    printf("\n welcome main");
    xyz();
    abc();
}
void xyz()
{
   Printf("\n welcome xyz");
}
```

o/p

welcome main
welcome xyz
welcome abc
welcome xyz

STORAGE CLASSES OF C' :-

- Storage classes of C' provides following information to compiler, i.e:-
  1. Storage area of a variable
  2. Scope of a variable, i.e in which block the variable is visible.
  3. lifetime of a variable, i.e how long the variable will be there in active mode.
  4. Default value of a variable, if it is not initialized.
- Depending on the storage area and behaviour, storage classes are classified into two types :-
  (i) Automatic Storage class
  (ii) Static Storage class.

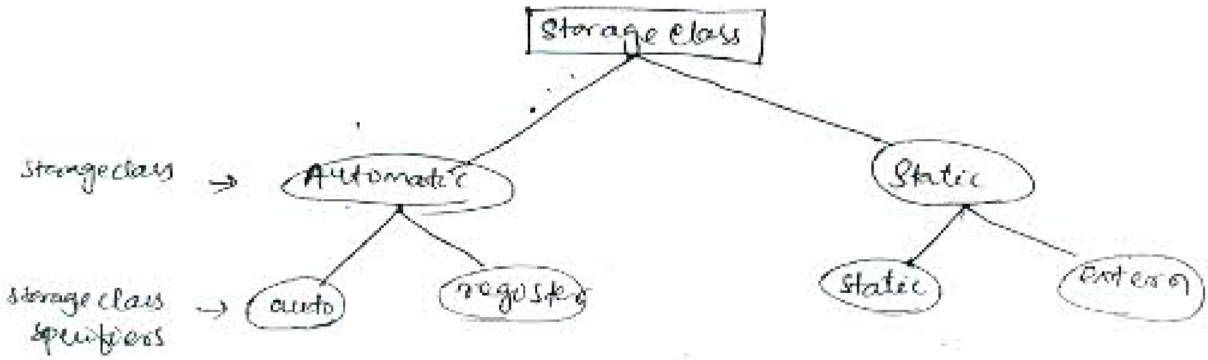
(i) Automatic Storage class :-

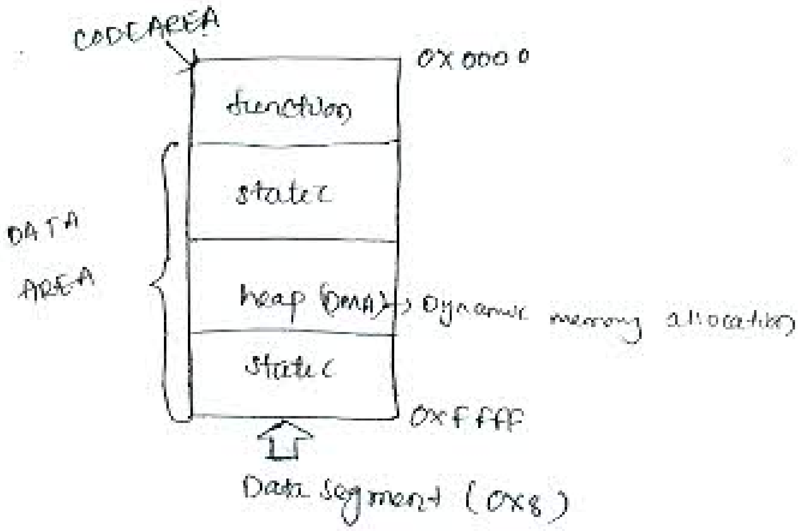
- Automatic storage class variables will be created automatically & destroyed automatically.
- Automatic storage class variables are stored in stack area of data segment or cpu registers.
- Under Automatic storage class, we are having two types of storage class specifiers, such as :-
  (a) auto
  (b) register.

(ii) Static storage class :-

- Static storage class variables are created only once and throughout the program, these variables will be there in active mode only.
- Static storage class variables are stored in static area of data segment.
- Under static storage class, we are having two type of storage class specifiers, such as:-

(a) static
(b) 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

default → auto

- In C- prog. language, there are four types of scopes are available;
  i.e. ① body scope,
  ② function scope,
  ③ program scope, and
  ④ file scope.

- By default, any kind of variable, storage class specifier is auto within the body.

18/10/12

① register variables :-

↳ Register variables are special kind of variables, which are stored in cpu registers.

- The advantage of register variable is faster than normal auto variable.

Limitations :-

- Register variables are limited, i.e. depending on the cpu capacity, 4 to 6 are maximum variables.
- When we are working with register variables, we can't access address.
- Pointer or pointer related concepts can't be applied for register variables.
- When we are working with register variable, if the memory is available in register, then only it holds the data in cpu register or else it is stored in stack area of data segment.

ex)
```
void main()
{
    register int a = 5;
    ++a;
    printf("\n value of a : %d", a);
    printf("\n Enter a value : ");
    scanf("%d", &a); // error : ------>
    ++a;
    printf("value of a :", a);
}
// input value is 100
```

if
```
scanf("%d", a);
```
then o/p :-
```
value of a : 6
value of a : 7
```

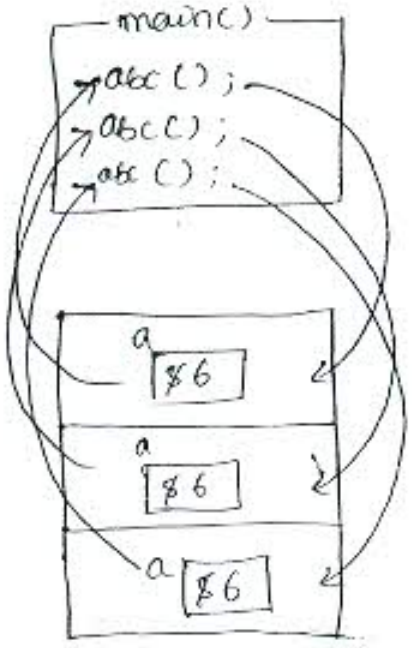
o/p → error; must take address of memory location.

✓ In implementation, when we are using a variable throughout the program in no. of times, then go for register variable.

(2nd)
```
void abc()
{
    auto int a=5;
    ++a;
    printf("\n a=%d", a);
}
void main()
{
    abc();
    abc();
    abc();
}
```

op :-

```
a = 6
a = 6
a = 6
```

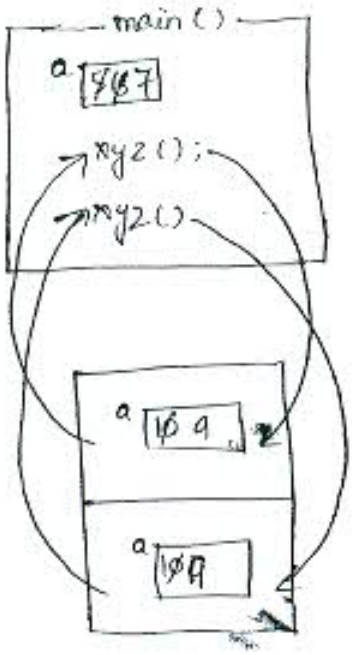


- According to storage class of C', the lifetime of auto variable is restricted within the body, that's why how many times we are calling the abc() function, those many times auto variable will be created.

(3rd)
```
void xyz()
{
    int a=10;
    --a;
    printf("\n a=%d", a);
}
void main()
{
    auto int a=5;
    ++a;
    xyz();
    xyz();
    ++a;
    printf("\n a=%d", a);
}
```

op)

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



→ By default, any kind of variable storage class specifies is auto

(Eg)
```
void abc()
{
  auto int a = 5;
  ++a;
  Printf("\n a= %d",a);
}
void main()
{
    abc();
    abc();
    printf("\n a= %d",a); // 'a' is not accessible in main()
}
```
O/P
error : undefined symbol 'a'

- According to Storage classes of C, the scope of the auto variable is restricted within the body.
- In the above program, variable 'a' is created in "abc()" method/function and we are trying to access in main function, so it gives an error.

(x5)
```
void main()
{
    int a = 10;
    clrscr(); // function call not allowed before variable declaration.
    int b=20;
    ++a;
    ++b;
    printf("%d %d", a, b);
}
```

o/p: error: declaration not allowed here.

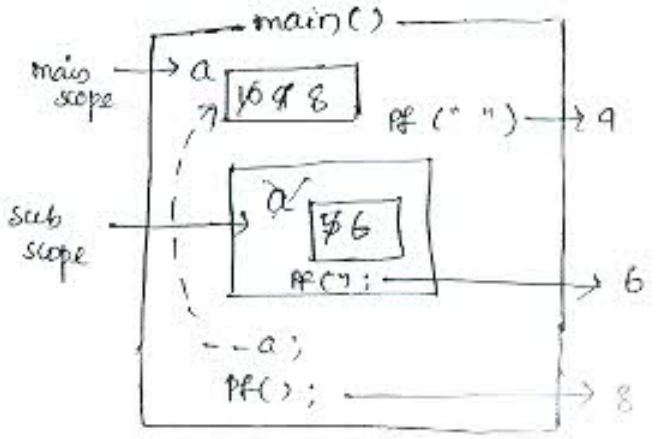
- In C- programming language, variable declaration must exist at the top of the program, before writing the first statement only.

(ex6)
```
void main()
{
    auto int a = 10;
    int a = 20; // auto, same name not allowed.
    ++a;
    printf("a = %d", a);
}
```

error: multiple declaration for 'a'

- In any scope, only one identifier can be constructed with a unique name.
- When we are declaring more than one variable in same block with same name, then compiler gives an error, i.e "multiple declaration for the variable".

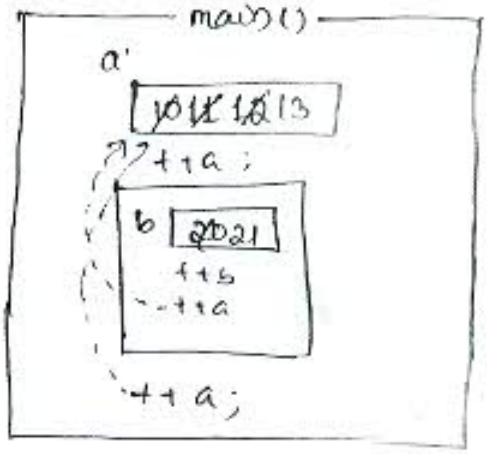
(ex7)
```
void main()
{
    int a=10;
    --a;
    printf("\n a = %d", a);
    {
        auto int a=5;
        ++a;
        printf("a = %d", a);
    }
    --a;
    printf("\n a = %d", a);
}
```



o/p:-

a = 9
a = 6
a = 8

ex8)
```
void main()
{
    auto int a=10;
    ++a;
    printf("\n a= %d", a);
    {
       int b=20;
       ++b;
       ++a;
     printf("\n b %d %d", a, b);
    }
    ++a;
    printf("\n %d", a);
}
```



o/p
```
a= 11
21 12
13
```

two standards :-
- → K & R-C standard
- → ANSI-C standard

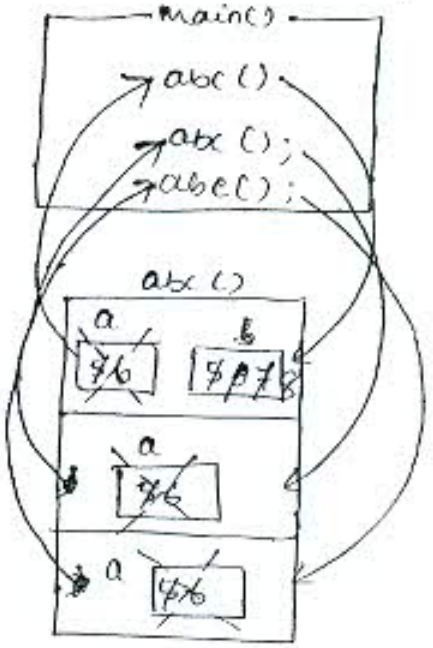
- As per K and R-C Standard, the above program gives an error, because all variable declarations should exist top of the program within the main scope only.
- According to ANSI-C Standard, the above program is valid, because whenever we are opening the body, those itself variables can be created
- whenever, we are creating a variable in main scope, it can be accessed automatically within the subscope, when subscope doesnot contain the same identifier

ex9)
```
void main()
{
    int a=1;
    ++a;
    printf("\n a= %d", a);
    {
       int a= 2;
       auto int b=3;
       ++a;
       ++b;
       printf("\n a= %d  b= %d", a, b);
    }
    ++a;
    ++b; // 'b' is not accessed outside the subscope.
    printf("\n a= %d  b= %d", a, b);
}
```
error: undefined symbol 'b';

- Whenever, we are creating a variable directly to subscope, then we can't access that variable within the main scope.

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

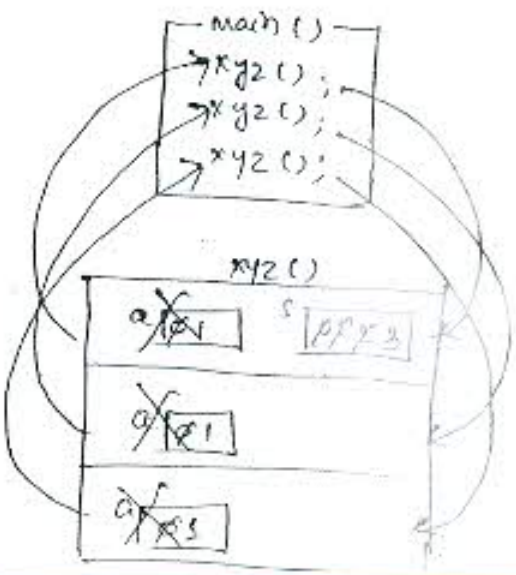
o/p:
a=6	s=6
a=6	s=7
a=6	s=8



- When we are working with the static variable, it can be created only once, when we are calling the function first time and throughout the program that data is available, because lifetime of the static variable is entire the program.

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

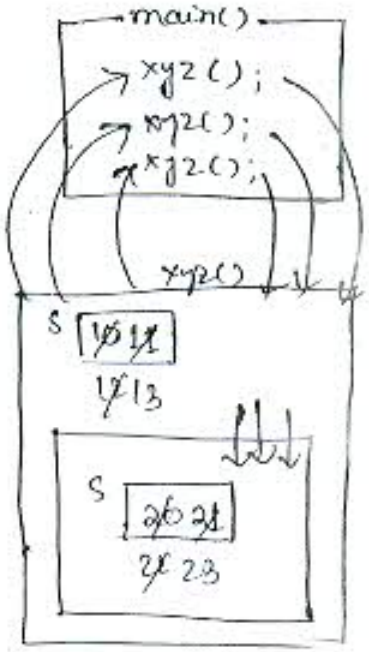
o/p:
a=1	s=1
a=1	s=2
a=1	s=3



- By default, value of the static variable is '0', if we are not initializing it.

Ex12)

```
void xyz()
{
    static int s=10;
    ++s;
    printf("\n static1 : %d", s);
    {
        static int s = 20;
        ++s;
        printf("\n static2 : %d", s);
    }
}
void main()
{
    xyz();
    xyz();
    xyz();
}
```



Static 1 : 11
Static 2 : 21
Static 1 : 12
Static 2 : 22
Static 1 : 13
Static 2 : 23

o/p

```
Static 1 : 11
Static 2 : 21
Static 1 : 12
Static 2 : 22
Static 1 : 13
Static 2 : 23
```

Ex13)

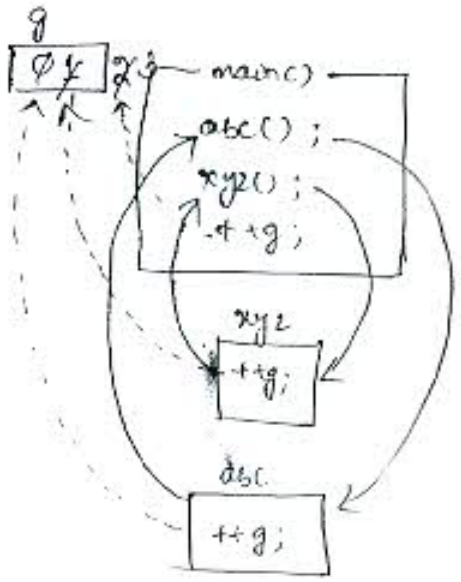
```
void abc()
{
    static int s = 10;
    ++s;
    printf("\n s = %d", s);
}
void main()
{
    abc();
    abc();
    printf("s=%d", s); // scope of 's' is inside abc()
}
```

error: undefined symbol 's'

- According to the storage classes of C, the scope of the static variable is restricted within the function only, that's why abc() function related static data, we can't access in main() function.

- When we are working with auto variables, scope and lifetime both are restricted within the body.
- When we are working with the static variable, scope is restricted within the function but lifetime is not restricted.
- In implementation, when we need to access a variable in more than one function, then go for global variables, i.e. extern variables are required.
- When we are declaring a variable outside of the function, then it is called global variable.
- The scope & lifetime of the global variable is entire the program.

ex)
```
int g; //global variable
void abc()
{
  ++g;
}
void xyz()
{
  ++g;
}
void main()
{
  abc();
  xyz();
  ++g;
  printf("g=%d", g); //3
}
```



ex)
```
void abc()
{
  ++g;
}
int g;
void xyz()
{
  ++g;
}
void main()
{
  ++g;
  abc();
  xyz();
```
→
```
if void abc()
{
=
}
```
→ then output: g = 2

o/p:- error : undefined symbol 'g'

- Due to top to bottom compilation process. When we are compiling '++g' statement in abc() function, then it gives an error. Because physical memory is not available
- When we are working with global variables, it can be constructed top of the program or middle of the program or bottom part of the program also.
- On implementation, when we are accessing a global variable before declaration, then for avoiding the compilation error, we requires to go for ~~global~~ forward declaration of global variable.
- By using 'extern' keyword, we can provide forward declaration of global variable.

(Ex)
```
void abc()
{
   extern int g; // declaration
   ++g;
}
int g; // defining
void xyz()
{
   ++g;
}
void main()
{
   ++g;
   abc();
   xyz();
   printf("g = %d", g); //3
}
```

- The basic difference between the declaration of the global variable & defination of a global variable is :-

  When we are declaring the global variable, it does not occupy any physical memory, just it avoids compilation error.
- When we are defining a global variable, then it occupies actual physical memory of a variable.

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

op:- Error : undefined symbol 'g'

- Inorder to access the global variable in abc(), xyz() and main() functions we require to go for forward declaration of global variable in abc(), xyz() and main() functions also.

- On implementation, when we need to go declare a global variable more than once, then recommended to go for global declaration of global variable.

- When we are declaring a global variable top of the program, before defining the first function, then it is called global declaration.

- If the global declaration is provided, then it is not required to for local declaration, if local declaration also available, then it is ignored about global declaration.

(ccc)
```
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
  ++g;
  abc(); xyz();
  printf("g = %d", g);
}
```

(op)
g = 8

(ex6)
```
extern int g;
void abc()
{
    ++g;
}
void main()
{
    ++g;
    abc();
    printf("g:%d", g);
}
```

- no error in compilation

linking error: undefined symbol _g in module

- when we are working with global variable, if declaration statement is not available, then we will get the error at the time of compilation.
- If the defination statement is not available, then we'll get the error at the time of linking.

(ex7)
```
extern int g=5;
void abc()
{
    --g;
}
void main()
{
    --g;
    abc();
    printf("g:%d", g);
}
int g; //ignored by compiler
```

o/p:- g=3

(ex8)
```
extern int g=10;
void xyz()
{
    ++g;
}
void main()
{
    ++g;
    xyz();
    printf("g:%d", g);
}
```

o/p:- g=12

- when the declaration part contains initialization, then it is not required to go for defination, because physical memory is created in declaration statement, so defination is not required.
- Initialization of the global variable can be placed in declaration or defination also.

(Ex)
```
extern int g = 10;
void abc()
{
  ++g;
}
void main()
{
  ++g;
  abc();
  printf("g : %d", g);
}
int g = 5; //error : variable 'g' is initialized more than once.
```

Formal Arguments and Actual Arguments :-

- In function calling statement, whatever the data we are passing, those are called actual arguments or arguments.
- In function declaration, whatever variables we are creating, those are called formal arguments or parameters.
- In order to call any function, if it is required specific no. of parameters, then it can't invoke the function with less than or more than required arguments.
- Where the implementation part of the function is placing, it is called function definition.
- In function definition, first line is called function header or function declaration.
- Whenever we are providing the type information of the function explicitly, then it is called declaration or forward declaration.
- Declaration of the function doesnot contains body, definition contains body of the function.

Calling convension :-

- In implementation whenever a function requires parameter, then as a programmer we require to mention parameter sequence also.
- calling convension indicates that in which sequence, parameters can be created. i.e. left to right or right to left.
- In C- prog. language, there are two types of calling conventions are available. i.e.
  1. cdecl (_cdecl)
  2. pascal (_pascal)

① cdecl calling conversion :-

- In cdecl calling conversion, parameters will be passed towards from right to left.
- By default, any function calling conversion is cdecl.
- Whenever we are working with cdecl calling conversion, then recommended to place the function_name in lowercase.

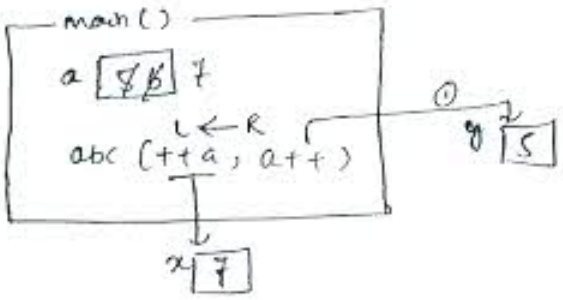
② pascal calling conversion :-

- In pascal calling conversion, arguments will be passed towards from left to right.
- When we are working with pascal calling conversion then recommended to place the function_name is uppercase.

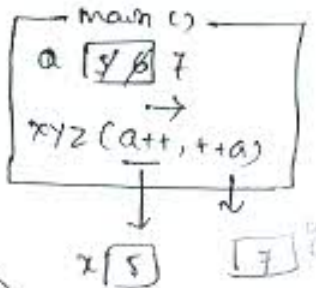
(ex.)
```
void cdecl abc (int x, int y)
{
    printf("\n x=%d y=%d", x, y);
}
void main()
{
    int a = 5;
    abc (++a, a++);
    printf("\n a=%d", a);
}
```

o/p :-

x = 7	y = 5
a = 7	



(ex.)
```
void pascal XYZ(int x, int y)
{
    printf("\n x=%d y=%d", x, y);
}
void main()
{
    int a;
    a=5;
    XYZ(a++, ++a);
    printf("a=%d", a);
}
```



o/p

x = 5	y = 7
a = 7	

parameter passing techniques :-

In C-programming language, there are 2 types of parameter passing techniques are available. i.e.

{ ① call by value / pass by value
② call by address / pass by address.

① Call by value :-

- Calling a function by sending the value type data or passing the value type data to the function is called call by value.
- In call by value, actual arguments and formal arguments, both are value type data.
- In call by value, if any modifications are occurred in formal arguments, then those changes will not be affected on actual arguments.

ex:- printf(), pow(), sqrt(), cos(), fmod().

② Call by address :-

- Calling a function by sending an address type data or passing the address type data to a function is called call by address.
- In call by address, actual arguments are address type, formal arguments are pointer types.
- In call by address, if any modifications are occurred in formal arguments, those changes will be carry back to actual arguments.

ex:- scanf(), strcpy(), memset(), setdate()

Note: C-prog. language doesnot support call by reference.

∵ call by reference is a OOPL concept which works with the help of reference variables. So C-doesnot support the reference variables, that's why call by reference is not possible.

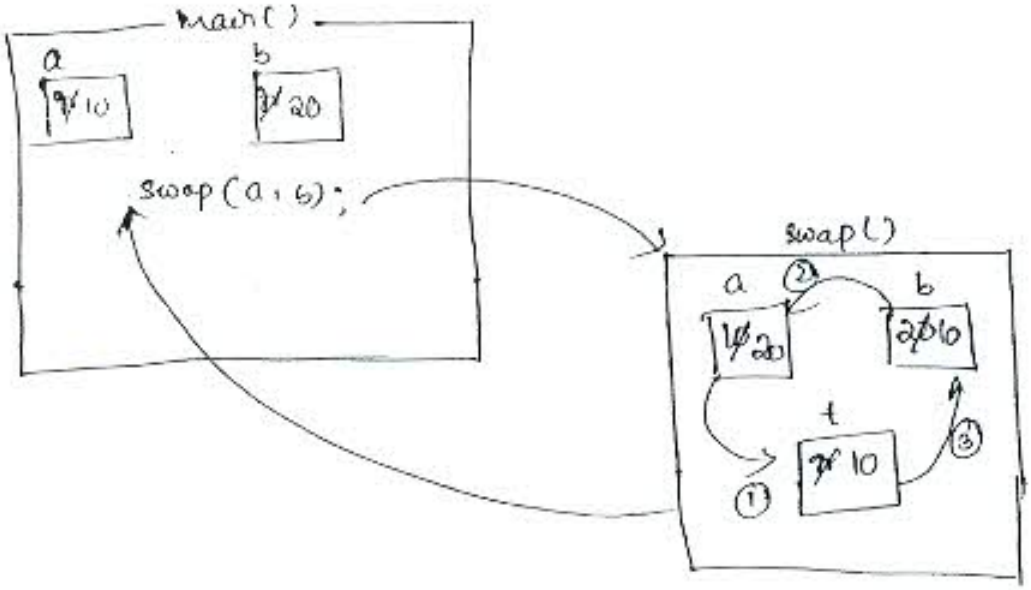
(ex1)
```
void swap()
{
    int t;
    t=a;
    a=b;
    b=t;
    printf("\n data is swap a=%d  b=%d", a, b);
}
void main()
{
    int a,b;
    a=10; b=20;
    swap(a,b);
    printf("\n data in main  a=%d  b=%d", a, b);
}
```

o/p :-
```
error:
undefined symbol 'a', 'b'
```

- According to the storage classes of C, by default any kind of variable storage class specifier is auto, so we can't extend the auto variable scope to "swap()" function.

- In the above program, in order to call the 'swap()' function, it doesnot require any parameter, but we are calling the function with two arguments, so it is not possible.

(ex2)
```
void swap(int a, int b)
{
    int t;
    t=a;
    a=b;
    b=t;
    printf("\n in swap a=%d  b=%d", a, b);
}
void main()
{
    int a, b;
    a=10; b=20;
    swap(a, b);
    printf("\n in main: a=%d  b=%d", a, b);
}
```



o/p :-

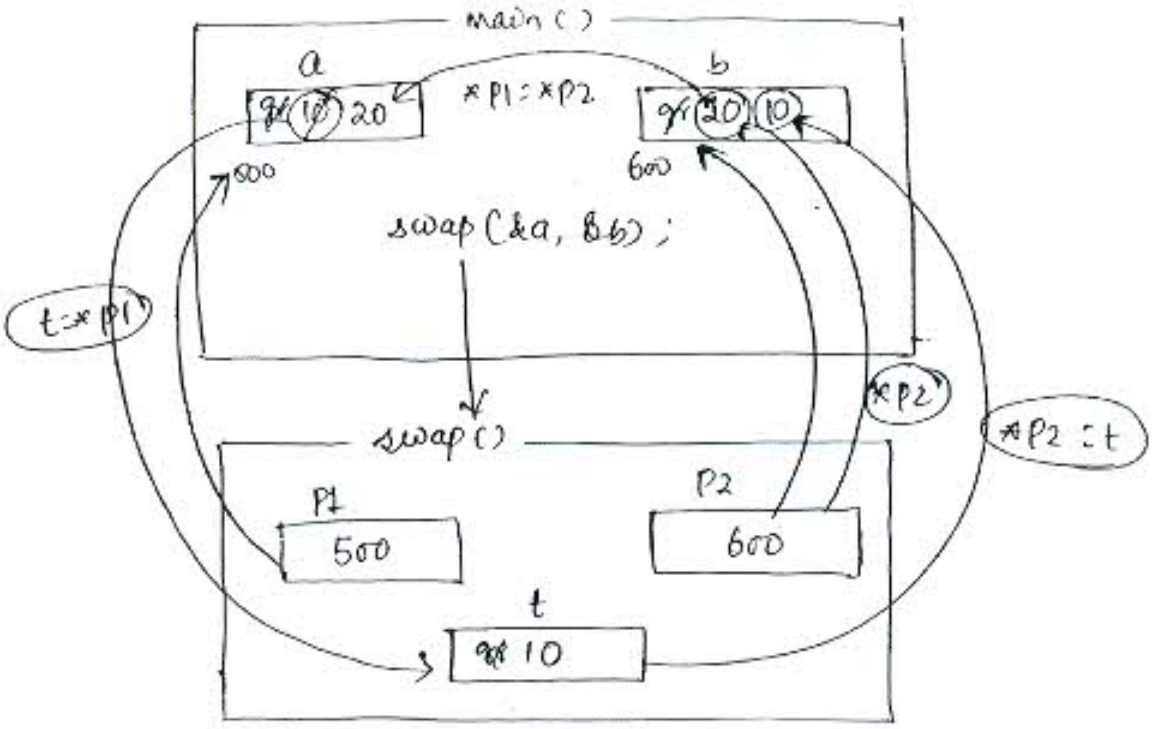
in swap : a = 20 b = 10
in main : a = 10 b = 20

- In the above program, swap() function works with the help of call by value mechanism, that's why no any modification of the swap() function will carry back to main() function.
- In implementation, when we are expecting the modification from the function, then always recommended to go for call by address.

(21)
```
void swap (int *p1, int *p2)
{
    int t;
    t = *p1 ; // t = a;
    *p1 = *p2 ; // a = b;
    *p2 = t ; // b = t;
} Pf ("\n in swap : a = %d  b = %d", *p1, *p2);
void main()
{
    int a , b;
    a = 10; b = 20;
    swap(&a, &b);
    printf ("\n in main: a = %d  b = %d", a, b);
}
```

o/p :-

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



- On the above program, swap() function works with the help of call by address mechanism, that's why all the modification of swap() function will be carried back to main() function.

(Q1) int a = 10, b = 20;

a = a + b;

b = a - b;

a = a - b;

o/p :- a = 20 b = 10

(Q2) 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

(Q3) 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);
```

...

2/10/12

(Q4)

```
Enter value of a : 2
Enter value of b : 5
2^5 value is : 32
```

```
void main()
{
  int a, b, p;
  clrscr();
  pf("Enter value of a :");
  sf("%d", &a);
  pf("\nEnter value of b :");
  sf("%d", &b);
  p = pow(a,b);
  pf("\n %d ^ %d value is : %d", a, b, p);
}
```

Note

```
Enter a value of a : 2
Enter b value of b : 5
Pow domain problem
2^5 value is 33
```

⇒ add #include <math.h> at the above of the program.

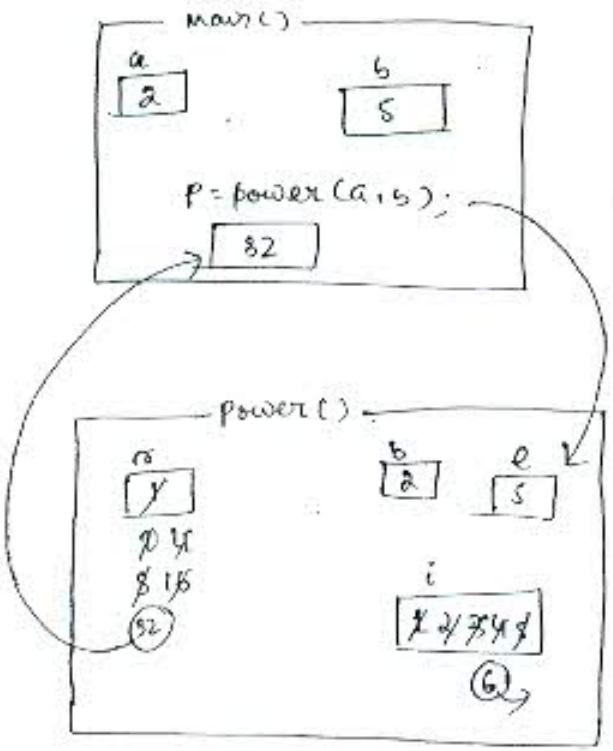
```
int power(int b, int e)
{
  int r=1, i;
  for(i=1; i<=e; i++)
      r = r*b;
  return r;
}
void main()
{
  int a, b, p;
  clrscr();
```

```
printf("Enter value of a: ");
scanf("%d", &a);
printf("\n Enter value of b: ");
scanf("%d", &b);
// p= poco(a,b);
p= power(a,b);
printf("\n %d ^ %d value is: %d", a, b, p);
getch();
}
```

```
Enter value of a: 2
Enter value of b: 5
2^5 value is : 32
```

```
Enter value of a: 2
Enter value of b: 16
2^16 value is : 0
```

$2^{16} = 65536$
$= 0$



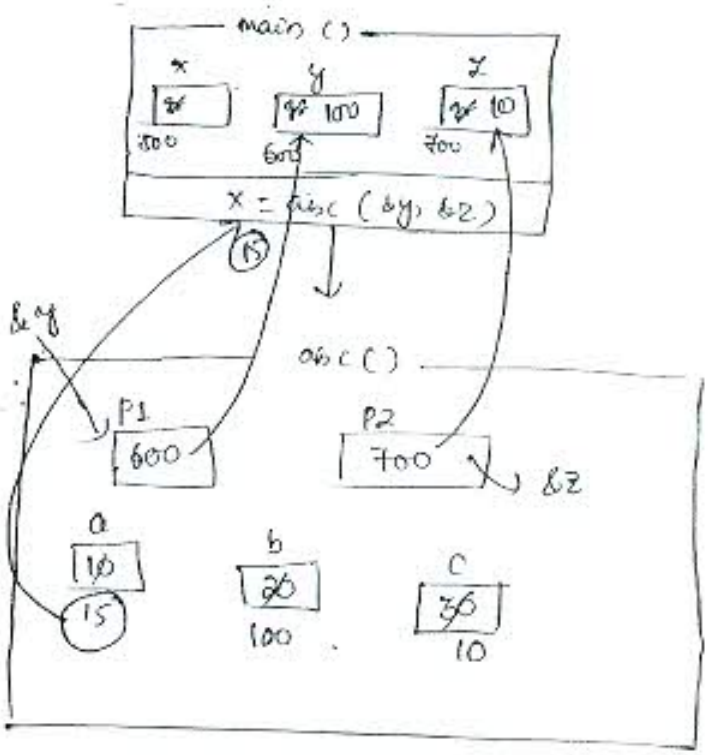
- 'return' is a keyword, by using 'return' keyword, we can pass the control back to the calling location with arguments or without arguments.
- By using function return type, it is possible to return only one value.
- Return stmt can return only one value from the function; even by using function return type, it is not possible to return more than one value.
- In implementation, when we need to return more than one value from the function, then go for call by address.
- By using call by address, it's not possible to return more than one value, but we can collect more than one value using pointers.

- In a function, we can place any no. of return statements, but at given any point of time, only one return stmt. can be executed.

(x5)
```
int max(int x, int y)
{
    if (x>y)
        return x;
    else
        return y; // return (x>y? x : y);
}
void main()
{
    int m;
    m = max(10,20);
    printf("\n max value = %d", m);
}
```

o/p:
```
max value : 20
```

(x6)
```
int abc (int *P1, int *P2)
{
    int a=10, b=20, c=30;
    a+=5;
    b*= a+3;
    c/=3;
    *P1 = b;   // y = b
    *P2 = c;   // x = c
    return a;
}
void main()
{
    int x, y, z;
    x = abc(&y, &z);
    printf("\n x=%d y=%d z=%d", a, b, c);
    getch();
}
```

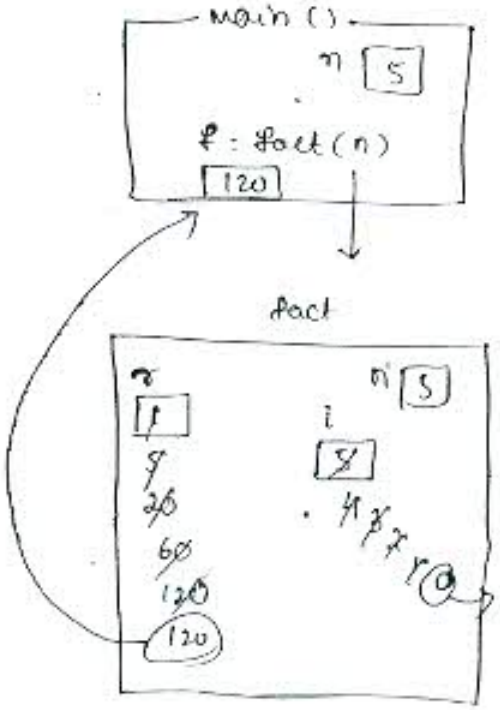


o/p
```
x = 15  y = 100  z = 10
```

Ex 7

```
Enter a value : 5
5 factorial value is : 120
```

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



- As per the ANSI-C standard, the above program is error, because when ever we are calling a function, which is defined later, for avoiding the compilation error, we are required to go for forward declaration.

- As per K & RC Standard, whenever a function is returning an integer type and parameter type is other than float, then it is not required to go for forward declaration.

RECURSION :-

- Function calling itself is called Recursion.

Purpose

- By using recursion, function calling information / statement we are maintaining.
- By using recursion, we can evaluate stack expressions.
- By using recursions only; infix, prefix, & postfix notations work.

Drawbacks

- It a very slow process due to stack overlapping.
- Recursive programs can create stack overflow.
- Recursive functions will create infinite loops also.

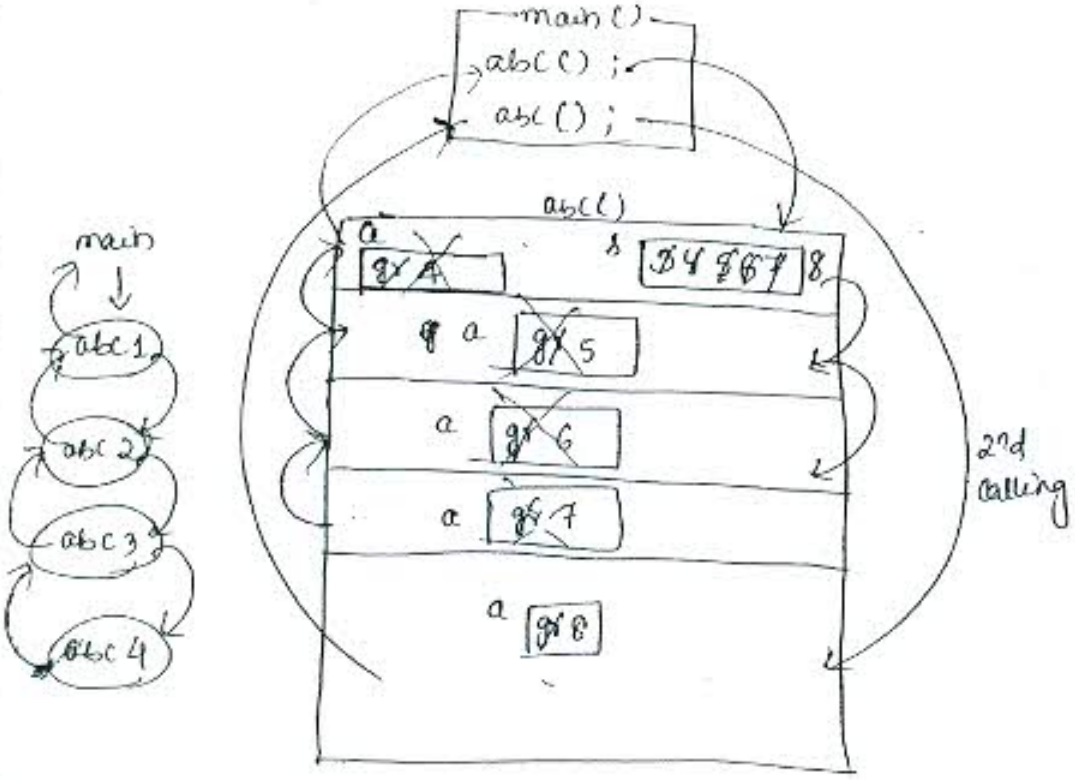
10/12

Ex)

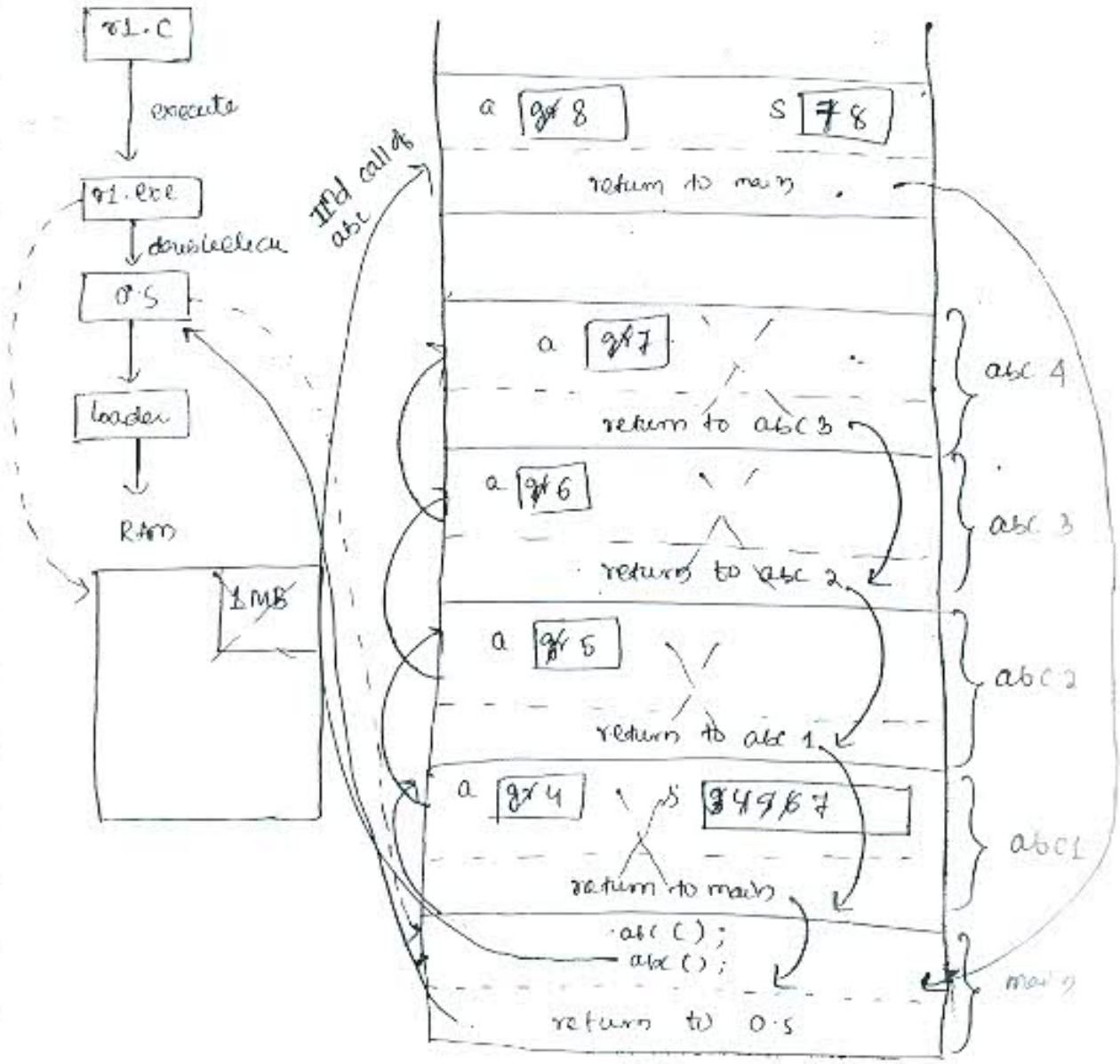
```
void abc()
{
    auto int a;
    static int s = 3;
    a = ++s;
    printf("\n %d %d", a, s);
    if (a <= 6)
        abc();
    printf("\n %d %d", a, s);
}
void main()
{
    abc();
    abc();
}
```

o/p

4	4
5	5
6	6
7	7
7	7
6	7
5	7
4	7
8	8
8	8

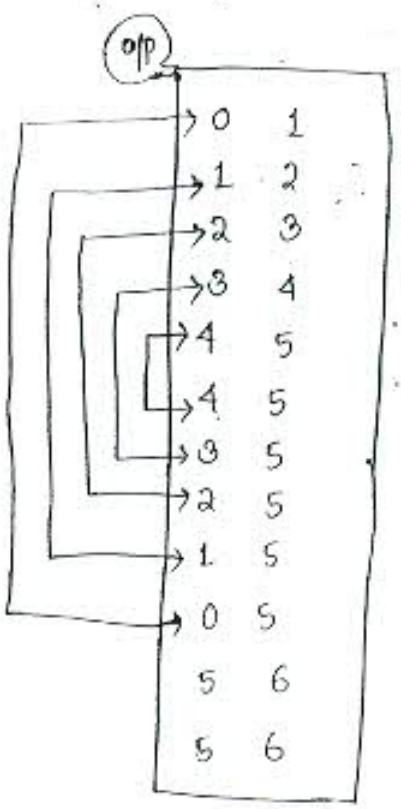


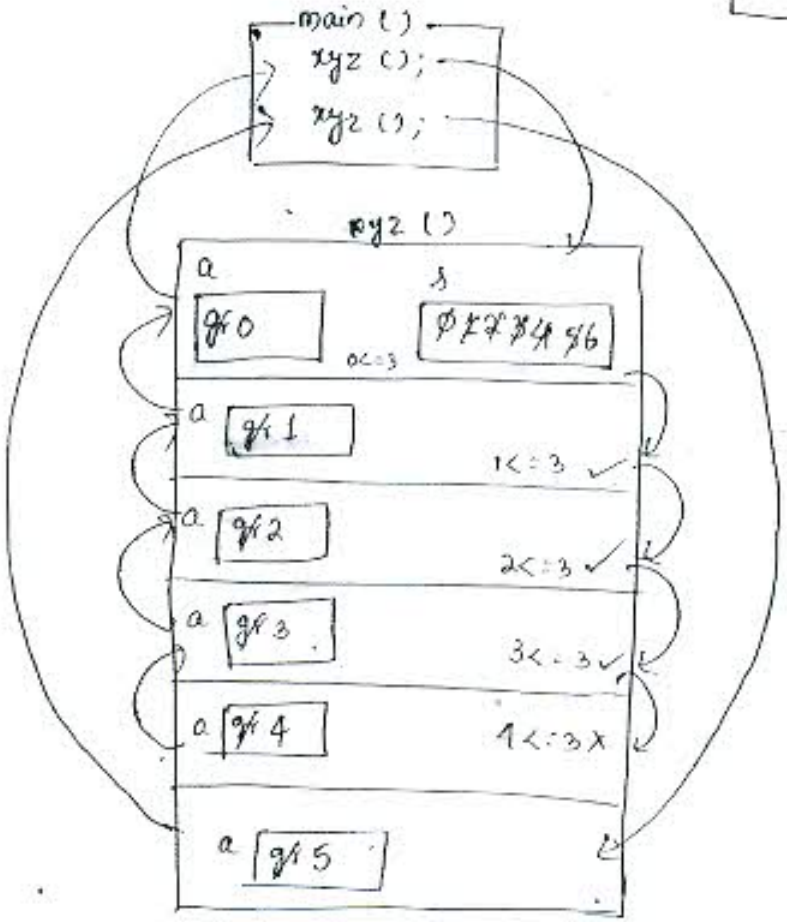
4 4
5 5
6 6
7 7
7 7
6 7
5 7
4 7
8 8
8 8



(Ex2)

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





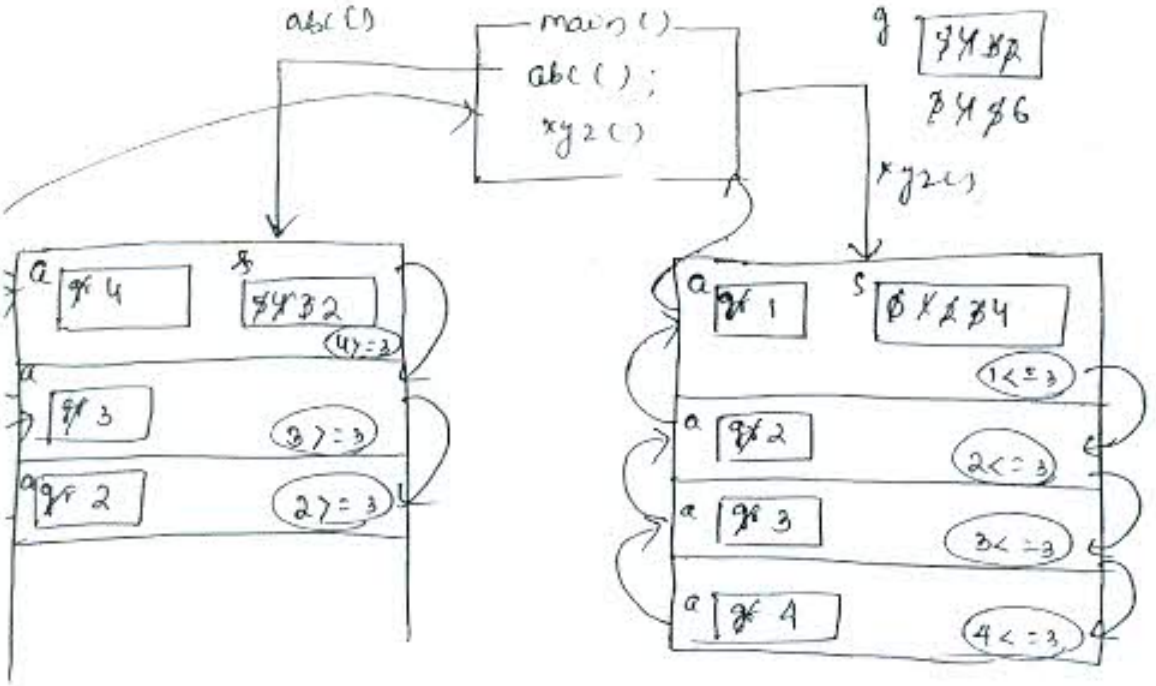
29/4/12

(Ex3)

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

4 4 4
3 3 2
2 2 2
3 2 2
4 2 2

1 1 3
2 2 4
3 3 5
4 4 6
4 4 6
3 4 6



- In C-Programming Language, Recursions are classified into 2 types :-
  1. internal recursive functions,
  2. external recursive functions.

- Whenever a function is calling itself, then it is called internal recursive function,
- One recursive function is calling another recursive function, then it's called external recursive process.

```
extern int g; // declaration
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); // declaration
    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);
}
int g = 3;
```

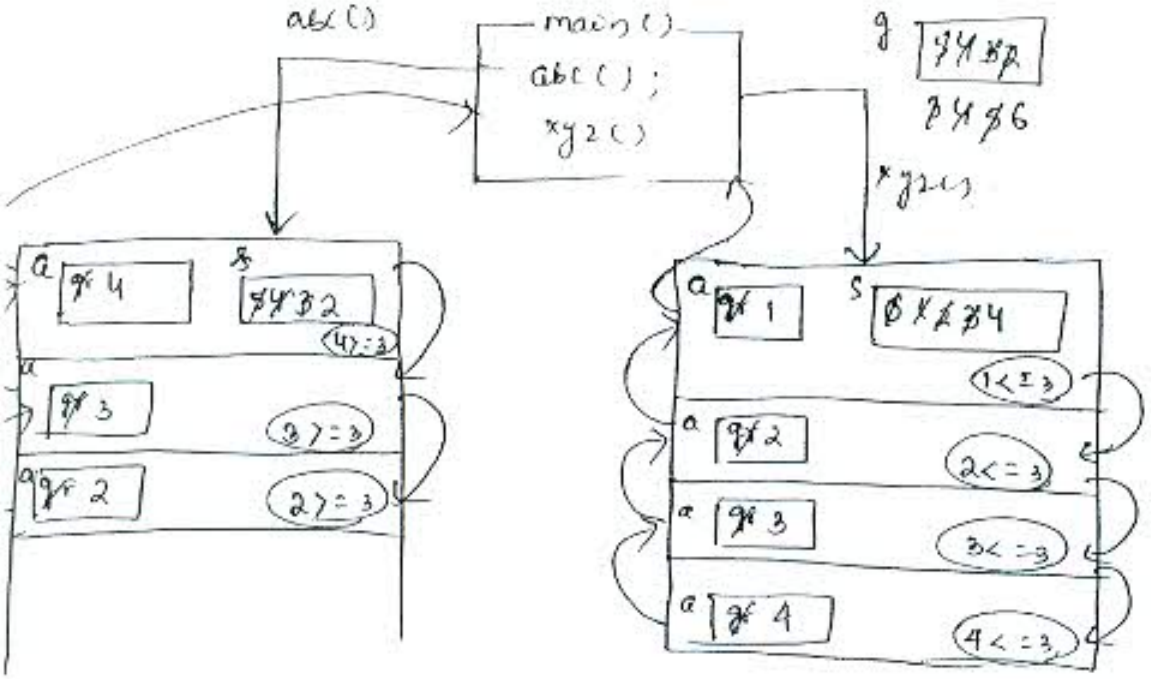
29/4/12

(ex3)

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

4 4 4
3 3 2
2 2 2
3 2 2
4 2 2

1 1 3
2 2 4
3 3 5
4 4 6
4 4 6
3 4 6



- In C-programming Language, Recursions are classified into 2 types :-
  1. internal recursive functions,
  2. external recursive functions.
- Whenever a function is calling itself, then it is called internal recursive function,
- One recursive function is calling another recursive function, then it's called external recursive process.

```
extern int g; // declaration
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); // declaration
    xyz();
}
```

```
void xyz()
{
    int a;
    static int s = 1;
    a = s+1;
    --g;
    Printf("\n %d %d %d", a, s, g);
    if(a<=2)
    {
        abc();
        xyz();
    }
    printf("\n %d %d %d", a, s, g);
}
int g = 3;
```

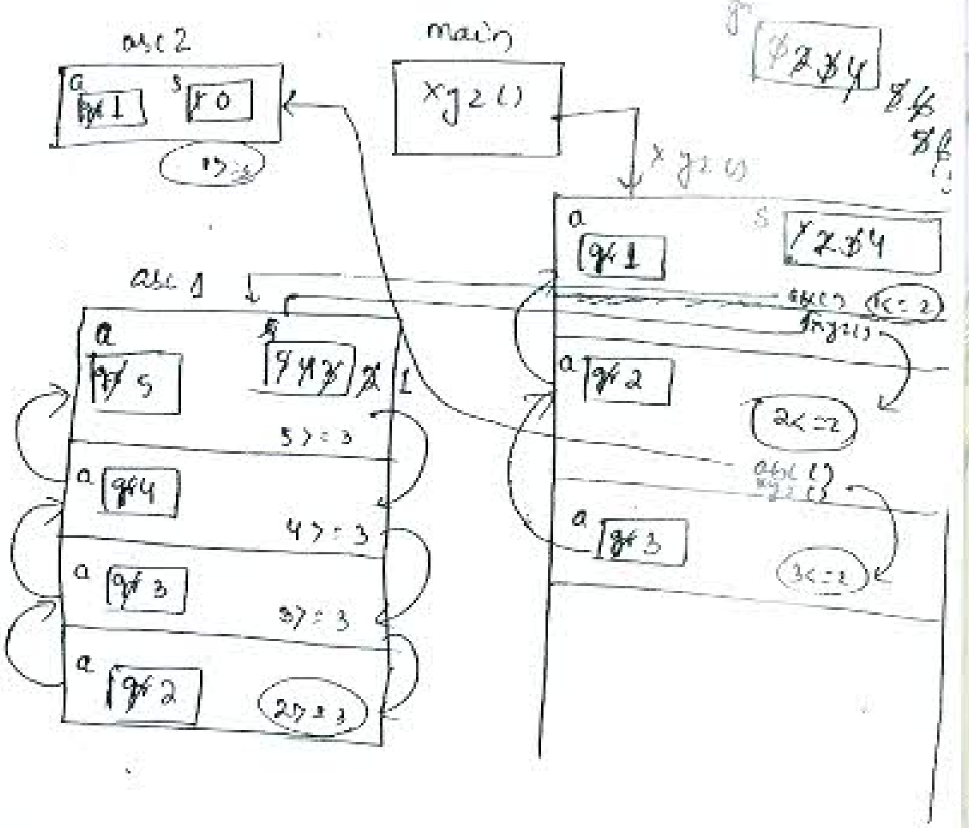
o/p:-

1 2 2 → xyz1
5 4 3 → abc1
4 3 4 → abc2
3 2 5 → abc3
2 1 6 → abc4
2 1 6 → abc4
3 1 6 → abc3
4 1 6 → abc2
5 1 6 → abc1

2 3 5 → xyz2

1 0 6 → 2nd abc
1 0 6 → 2nd abc

3 4 5 → xyz3
3 4 5 → xyz3
2 4 6 → xyz2
1 4 6 → xyz1



- main() function can be called by itself if it is required, but we will get stack overflow, if we are constructing auto variable.

(ex) void main()

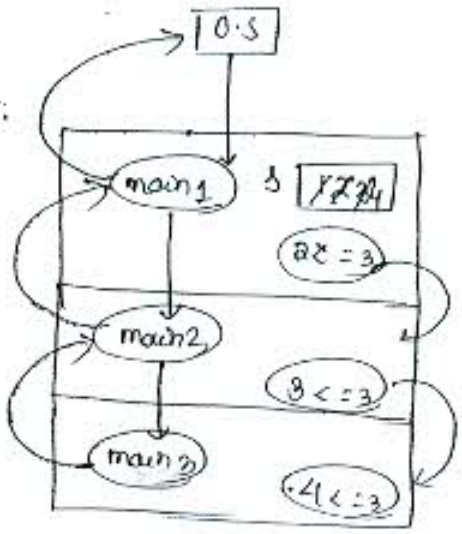
```
{
   int a=2;
   ++a;
   printf("%d",a);
   if(a<=3)
       main()
   printf("%d",a);
}
```

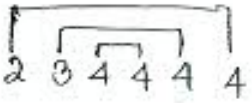
o/p → 3 3 3 3 3 ----- stack overflow

- According to the storage classes of 'C', by default any type of variable storage class specifier is <u>auto</u>, so whenever we are making the recursion in main() function it'll be created again & again, so condition is always true, so we'll get infinite loop.

5) void main()
{
    static int s = 1;
    ++s;
    printf("%d", s);
    if(s<=3)
        main();
    printf("%d", s);
}

op:
2 3 4 4 4 4

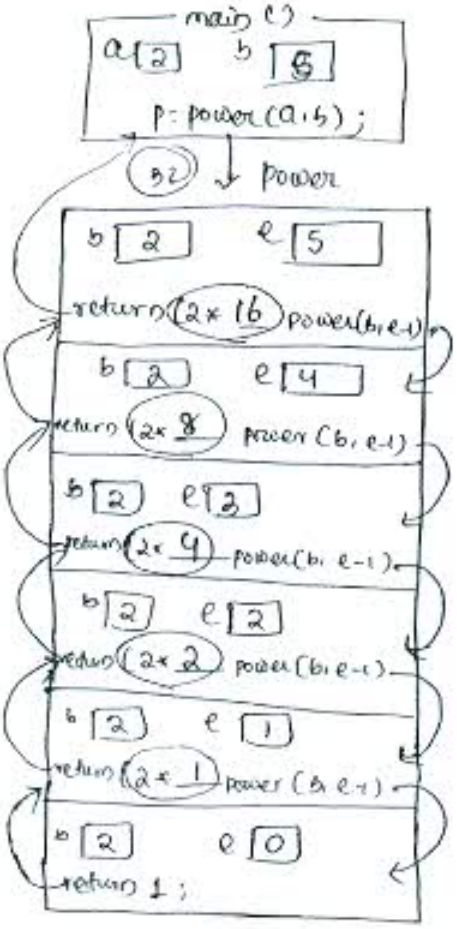




6)
Enter value of a : 2
Enter value of b : 3
2^3 value is : 8

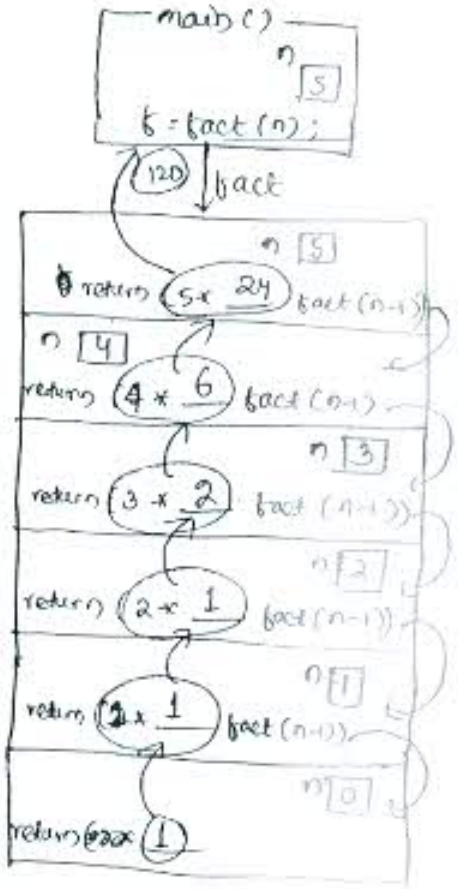
$\begin{cases} 2^{-3} = 0 \\ 2^{-2} = 0 \end{cases}$

```
int power(int b, int e)
{
    if (e<0)
        return 0;
    else if (e>0)
        return (b * power(b, e-1));
    else
        return 1;
}
void main()
{
    int a, b, p;
    clrscr();
    printf("Enter value of a : ");
    scanf("Enter to %d", &a);
    printf("\n Enter value of b: ");
    scanf("%d", &b);
    p = power(a, b);
    printf("\n %d ^ %d value is : %d", a, b, P);
    getch()
}
```



Qr-8

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



Whenever a function is not returning any values, then specify the return type as void.

'void' means nothing, i.e. no return value.

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

o/p: Hello abc

- If the function return type is void, then it is possible to place return statement, from 'void' function. When we are placing the empty return statement, then it passes the control back to the calling location.

i.e.

Ex:i)
```
void abc()
{
  printf("Hello abc");
  return;
}
void main()
{
  abc();
}
```

o/p: Hello abc

From 'void' function, we can specify the return statement with value also.

From 'void' function, when we are returning the value, then compiler gives a warning message, i.e. "void function may not return any value".

Ex:ii)
```
void abc()
{
  int a=10;
  printf("Welcome abc : %d",a);
  return a;
}
void main()
{
  abc();
}
```

o/p: Welcome abc : 10

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

ex:3
```
void xyz()
{
    int a=5;
    printf("Hello xyz : %d", ++a);
    return a;
}
void main()
{
    int x;
    x = xyz(); // error
    Printf("In x value = %d", x);
}
```

o/p
```
Error : not an allowed type
```

30/10/12

Dangling pointer :-

which pointer variable, pointing to a inactive or dead location, it is called dangling pointer.

function pointer :-

which pointer variable holds the address of a function, is called as function pointer.

advantage :-

- function pointers are faster than normal functions.
- by using function pointers, we can pass a function as an argument to another function.

syntax :-

→ Datatype (* ptr) () ;

eg :- int (* ptr) () ; non parrary

→ Datatype (* ptr) (Datatype)

int (*ptr)(int) params.

- When function doesnot take any parameters, then go for non-parameterized function pointer.
when the function requires the parameters, then go for parameterized function pointer.

Return by value, Return by Address :-

whenever a function returning value type data, then it's called return by value, i.e function returning value type.

ex:- printf(); scanf(), pow(), sqrt(), fact().

whenever a function returns address type data, then it is called return by address, i.e. function returning pointer type.

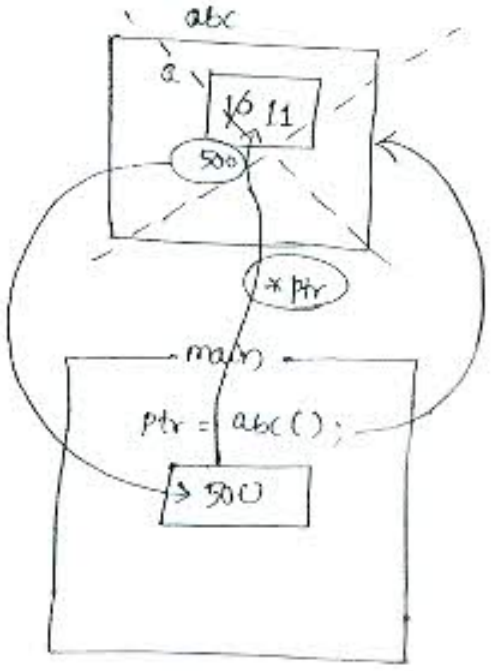
ex:- fopen(), strcpy(), strrev(), malloc();

- The basic advantage of return ~~type~~ by address is one function related local data can be accessed or modified from out side of the function.

An implementation, whenever a function is returning an integer value, then specify the return type as an int ; i.e. function returning value called return by value.

- whenever a function returning an integer variable address, then specify the return type as an int *, i.e. function returning pointer called return by address.

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

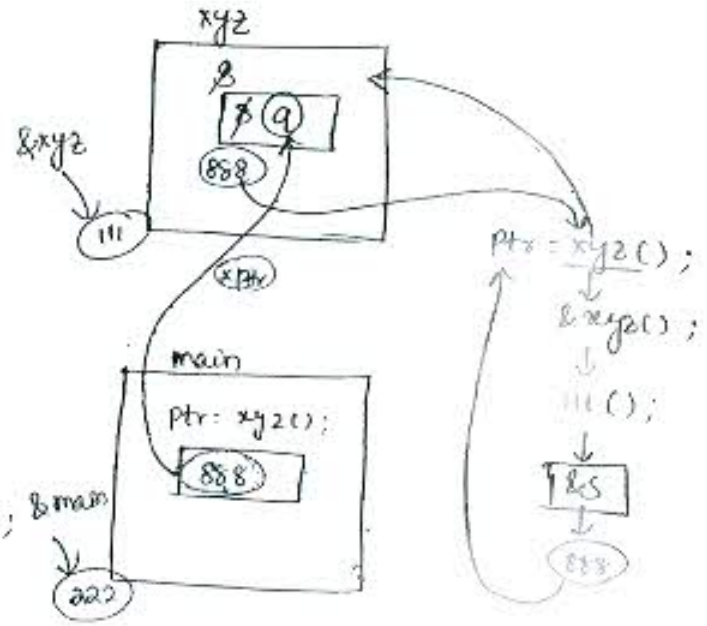


(o/p) a value = 11 (illegal)

- In the above program, ptr is called dangling pointer, because according to storage class of 'x', by default any variable storage class specifier is auto, that's why when we are accessing the data by using pointer, which is already destroyed.

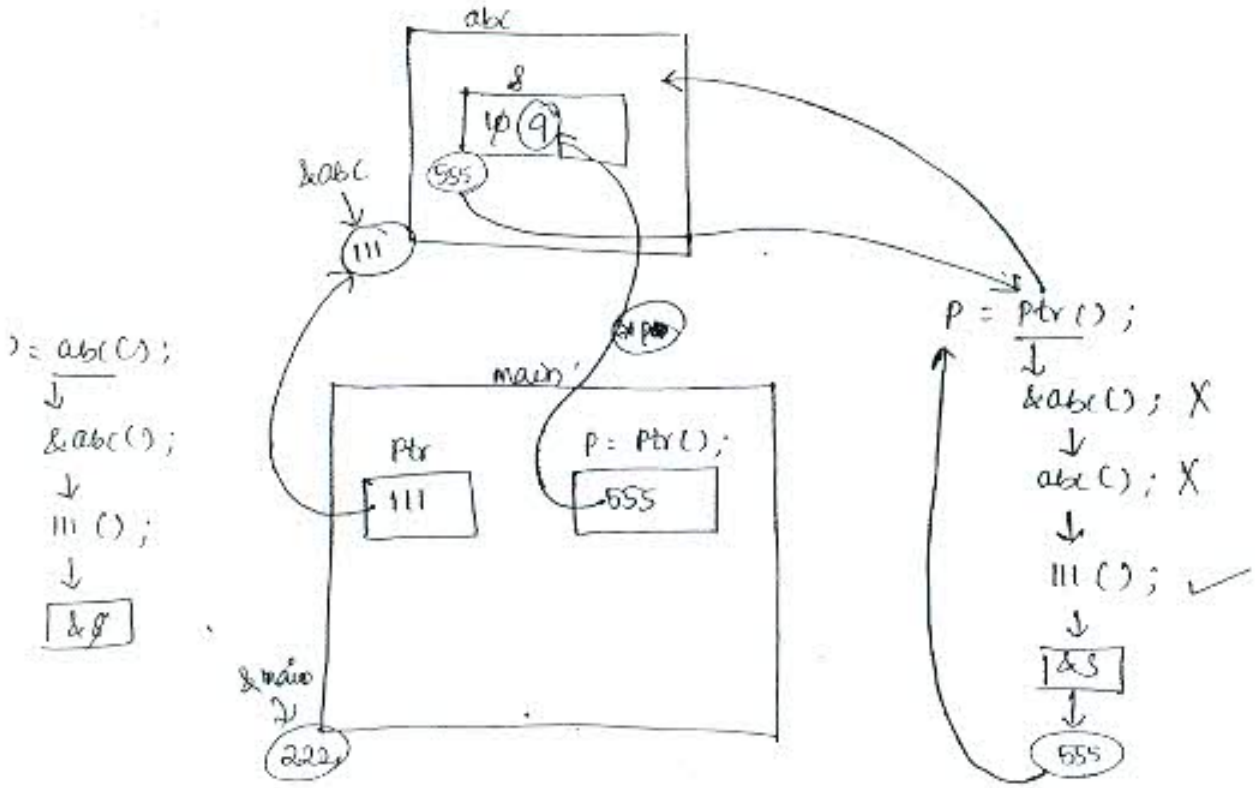
- The solution of the dangling pointer is, instead of creating auto variable, recommended to go for static, because lifetime of the static variable is entire the program.

(ex2)
```
int * xyz()
{
    static int s=8;
    ++s;
    return &s;
}
void main()
{
    int *ptr;
    ptr = xyz();
    printf("static data : %d", *ptr);
}
```



(o/p) static data : 9

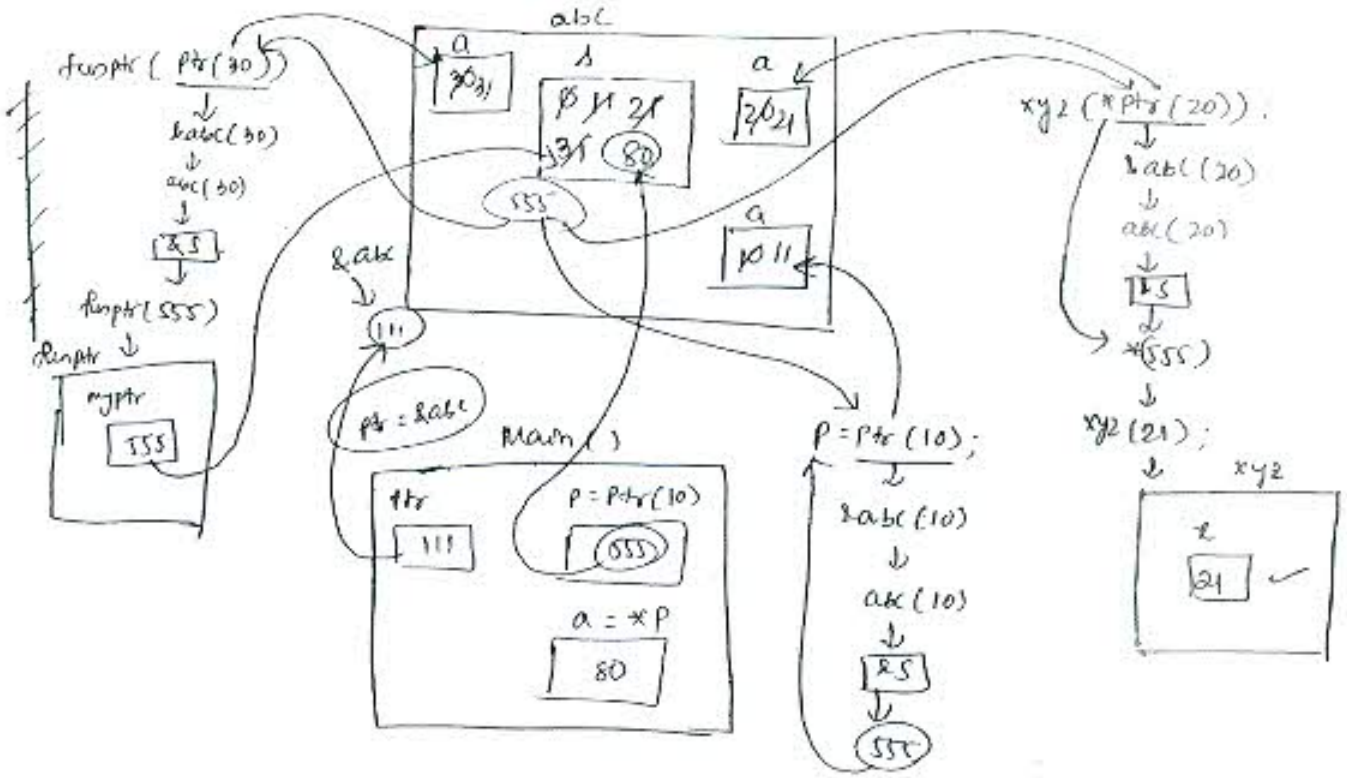
(ex3)
```
int * abc()
{
    static int s=10;
    --s;
    return &s;
}
void main()
{
    int *(ptr)();   // function pointer
    int *p;         // pointer to integer
    [illegible];
    ptr = &abc;  // collecting address of abc (&abc) into function pointer
    p = ptr;
    printf("\n static data : %d", *p);
}
/* without function pointer
 void main()
{
    int *p;  // pointer to integer
    p = abc();
    printf("\n static data : %d", *p);
}
```




```
int *abc(int a)
{
   static int s;
   s = s+a;
   return &s;
}
void xyz(int x)
{
   printf("\n data in xyz : %d", x);
}
void fun ptr( int *myptr)
{
   *myptr = 80;
}
void main()
{
   int *(*ptr)(int) ; // function pointer
   int *p ;    // pointer to integer
   int a ;
   ptr = &abc ; // collecting address of abc into fun. pointer
   p = ptr(10); // collecting &s into pointer to integer
   xyz(*ptr(20)) ; // passing value of 's' to xyz
   fun ptr(ptr(30)); //
   a = *p;
   printf("\n data in main : %d", a);
}
```

o/p

data in xyz : 21
data in main : 80



(i) void sum (int x, int y)

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

(op) Sum = 30

(ii)

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

(op) Sum = 30

warning: function should return a value

- when the function is returning the value, then specifying the return statement is optional. In the above case (ii), compiler will give only a warning message, i.e function should return a value.

(iii)

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

(op) Sum = 30

(not) → when the function is returning the value then collecting the value is also optional. In this case, compiler doesn't give any warning or error message.

(14) int sum (int x, int y)

```
{
    printf ("Sum = %d", x+y);
    return x+y;
}
void main ()
{
    int s;
    s = sum (10, 20);
    printf ("\n Sum = %d", s);
}
```

(o/p) Sum = 30
Sum = 20

o/p

Note:- When we are returning expression format data, then always recommended to place the return statement within the parenthesis only.

- when expression format data is not written within the parenthesis, then evaluation will take place in stack, which increases the burden on compiler, because evaluation takes place in stack.

(15) enter amount : 12759.57
TWELVE THOUSAND SEVEN HUNDRED FIFTY NINE RUPEES FIFTY SEVEN PAISE

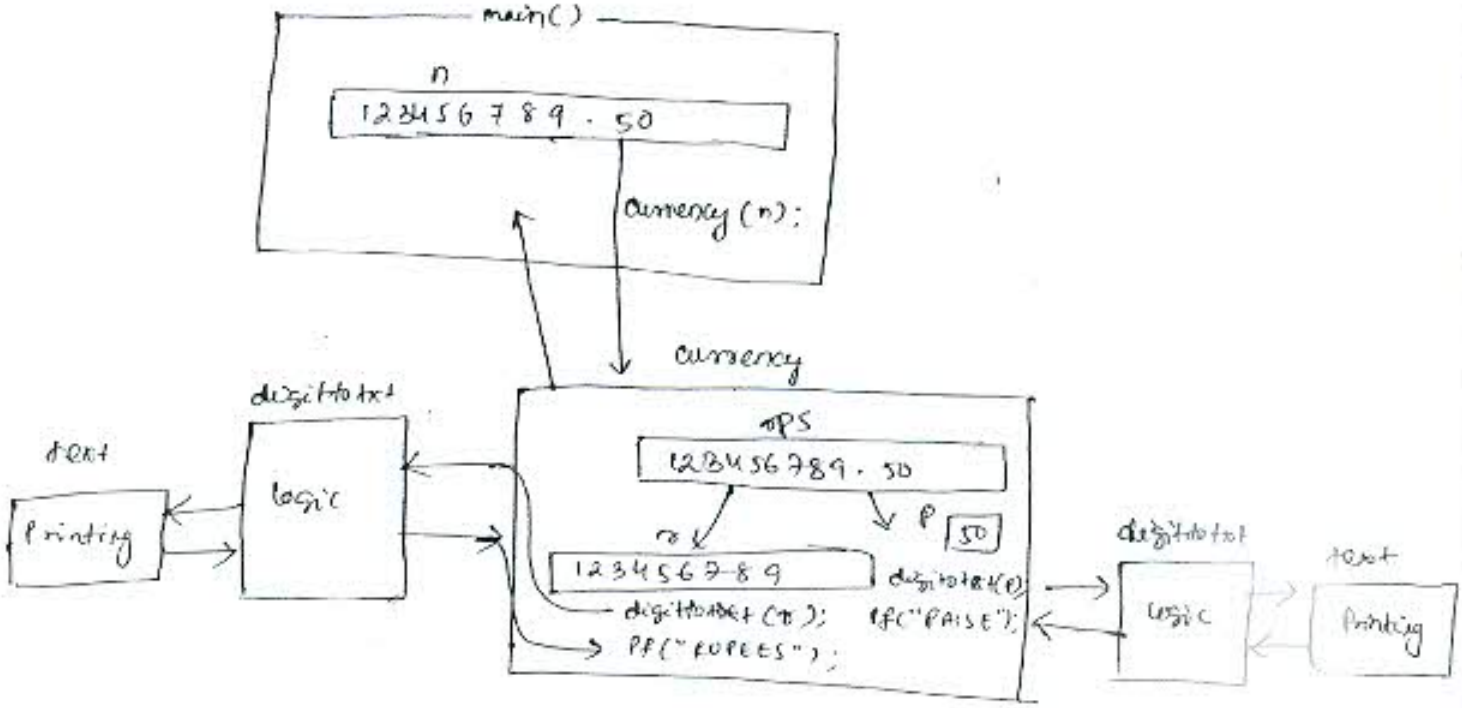
Ans:-

```
void text (long int no)
{
  switch (no)
  {
    case 1: printf ("ONE");
            break;
    case 2: printf ("TWO");
            break;
    case 3: printf ("THREE");
            break;
    ....
    ....
    case 19: printf ("NINETEEN");
             break;
    case 20: printf ("TWENTY");
             break;
    case 30: printf ("THIRTY");
             break;
```

```
        case 40 : printf (" FORTY ");
                  break;
        . . . .
        . . . .
        case 90 : printf (" NINTY ");
                  break ;
        case 100 : printf (" HUNDRED ");
                  break;
        case 1000 : printf (" THOUSAND " );
                  break;
        case 100000 : printf (" LACS ");
                  break ;
        case 10000000 : printf (" CRORES " );
    } // 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 (Lakhs);
        n = n % 100000;
    }
    if (n >= 1000)
    {
        t = n/1000;
        if (t <= 20)
            text (t);
        else
        {
            text (t/10 * 10);
            text (t % 10);
        }
        text (Thousand);
        n = n % 1000;
    }
    if (n >= 100)
    {
        text(n / 100);
        text (Hundred);
        n = n % 100;
    }
    if (n <= 20)
        text(n);
    else
    {
        text (n/10 x 10);
        text (n % 10);
    }
}
void currency (long double rps)
{
    long int r, p;
    r = (long int) rps;
    digit_to_text (r);
    printf (" RUPEES ");
    P = (rps - r) * 100;
    digit_to_text (P);
    printf ("PAISE");
}
```

```
void main()
{
    long double n;
    clrscr();
    printf("Enter amount:");
    scanf("%Lf", &n);
    currency(n);
    getch();
}
```



Declaration

1. `void abc(void);` or `void abc();`
2. `void abc(int);`
3. `int sum(void);`
4. `int sum(int, int);`
5. `float sum(int, float);`
6. `void swap(int, int);`
7. `void swap(int *, int *);`
8. `int power(int, int);`
9. `int *abc(void);`
10. `int *abc(int, int);`
11. `int **xyz(int *, int *);`

Calling

1. `abc();`
2. `abc(i1);`
3. `i1 = sum();`
4. `i3 = sum(i1, i2);`
5. `f2 = sum(i1, f1);`
6. `swap(i1, i2);`
7. `swap(&i1, &i2);`
8. `i1 = power(i2, i3);`
9. `ip1 = abc();`
10. `ip1 = abc(i1, i2);`
11. `ipp = xyz(&i1, &i2);`
    `ipp = xyz(ip1, ip2);`

(12) int abc (int **, int **); (12) c1 = abc (&ip1, &ip2);

PREPROCESSOR :-

- The basic diff. b/w internal static declaration & external static declaration is :-

The scope of the internal static declaration is restricted within the body, the scope of the external static declaration is restricted within the file.

case1

1.c

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

main.c

```
void main()
{
  ++g; ✓
}
```

case2

1.c (file scope extern variable)

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

main.c

```
void main()
{
  ++g; error
}
```

- In C- Prog. language, static functions are not-available.
- static functions are OOPL concept, which is required to use, when we are accessing static members of the class.
- In C'-prog language, static functions are not possible, but we can apply static storage class specifier for the function.
- When we are applying static storage class specifier for the function, then scope will be restricted within the file.

(ex) auto void abc()

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

o/p :- error : storage class 'auto' is not allowed here.

- By default, any function scope is extern, i.e. from any where in the project we can call the function.
- By using auto storage class specifier we can't restrict the scope of the function.

```
extern void xy2()
{
  printf("welcome xy2");
}
void main()
{
  xy2();
}
```

↳ by default, any function scope is 'extern'.

o/p :- welcome xy2

---

case 1

1.c

```
void abc()

void xy2()
```

main.c

```
void main()
{
    abc();  ✓
    xy2();  ✓
}
```

---

case 2

1.c

```
static void abc()
{
}
void xy2()
{
    abc();  ✓
}
```

main.c

```
void main()
{
    abc();  error X
    xy2();  ✓
}
```

— o —

PREPROCESSOR :-

- It's an automated program, which will be executed automatically before passing the source program to compiler.

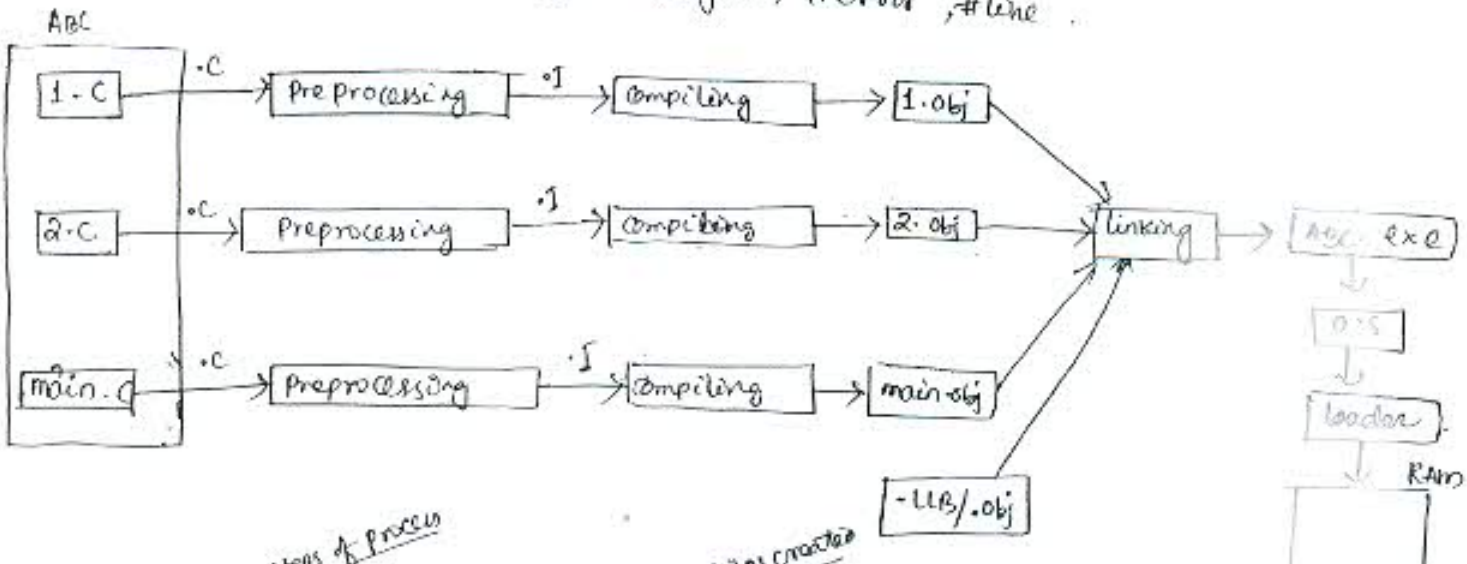
- Preprocessing is under the control of preprocessor directives.
- All preprocessor directives are starting with pound (#) sign symbol and should not be ended with semicolon (;).
- When we are working with preprocessor directives, it can be placed anywhere within the application, but generally recommended to place at the top of the program.
- In C-Programming language, 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



steps of process
① editing
② compiling } TC-IDE
③ linking
④ loading → O.S

files created
① .C
② .bak
③ .obj
④ .exe } automatically
⑤ .I → to create, or have to explicitly create

- When we are working with any C-application to create or execute the program, we require to perform 4 steps, i.e:-

① editing ② compiling ③ Linking ④ Loading

① - Constructing the source program and saving with '.c' extension, is called editing.
- When the editing process is completed, we'll get .c file.
- Editing can be performed by using any type of text editors like notepad, wordpad, or any other TC-IDE (Integrated Development Environment).

② - Compiling is a process of converting source code into object format data.
- When the compilation process is completed, then we'll get .obj file.
- The .obj file contains compiled code, that is understandable to System only.

③ - Linking is a process of compiling all obj files of current project including standard .LIB files to create an executable format data, i.e, .exe.
- When the linking process is completed automatically, we get .exe file.
- .exe file contains executable format data i.e. native code of operating system.
- For any C-application, we'll get five types of files. i.e.

① .c ② .bak ③ .obj ④ .exe ⑤ .I

- .c, .bak, & .I file contain user readable format data, .obj and .exe contains system readable format only.
- Among those all files, .I file needs to be created explicitly by using preprocessor command.

① Macro Substitution Directives (#define) :-

- By using #define, where an identifier is occurred, it can be substituted with replacement text.
- Generally, the replacement text can be created by using single or multiple tokens.
- When we are working with '#define', atleast single space should be required b/w #define, an identifier & identifier, Replacement text.

- Syntax: `#define identifier replacement text`

- By using '#define' just data will be substituted at the time of preprocessing.

Procedure to design C-program on DOS :-

- For editing the program, we require to use edit command.
  - syntax :- edit filename.c
  - ex :- E:\C9Am> edit p1.c
- code in p1.c

```
#define A 10
void main()
{
    int x;
    x = A;
    printf("%d %d", A, x);
}
```

// save p1.c (file → save)

// exit p1.c (file → exit)

- For preprocessing the program, we require to use 'cpp' command
- 'cpp' is an external command, which is available in "C:\TC\BIN"
  - syntax :- cpp filename.c
  - ex :- E:\C9Am> cpp p1.c
- code in p1.i :

```
p1.c 1 :
p1.c 2 : void main
p1.c 3 : {
p1.c 4 : int x;
p1.c 5 : x=10;
p1.c 6 : printf("%d %d", 10, x);
p1.c 7 : }
p1.c 8 :
```

- When the preprocessing is completed, where an identifier 'A' is occurred, it is replaced with '10'.
- After completion of preprocessing, no any preprocessor related directives will be present in source, because compiler can't understand any preprocessors.
- To compile & link the program, we require to use 'TCC' command
- 'TCC' is an external command which is available in "C:\TC\BIN"
  - syntax :- TCC filename.c
  - ex :- E:\C9Am> TCC p1.c

- when compiling & linking are completed, automatically we will get .obj & .exe file.

- To execute the program, we require to use program name or program name.exe

Ex:- E:\C9am> P1.exe

o/p : 10 10

(or) E:\C9am> P1

o/p : 10 10

Ex1) #define size 100

```
void main()
{
    int x;
    x = ++size; // x = ++100;
    printf("x = %d", x);
}
```

o/p error : L value required.

- By using #define, we can create symbolic constant values, i.e. can't be changed at the time of execution

Symbolic constants are decreasing the burden on compiler

- In the above program, size is an identifier, which is replaced with 100 at the time of preprocessing, so it is not possible to change the value.

Ex2)

```
#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

When we wrote :-

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

- In implementation, when we are composing the replacement text with multiple tokens, then always recommended to place within the parantheses only.

i.e (Ex3)

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

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

(Ex4)

```
# define pf printf
# define sc scanf
# define start main()

void start
{
    int a, b;
    pf("Enter two values :");
    sc("%d %d", &a, &b);
    pf("Sum = %d", a+b);
}
```

```
void main()
{
    int a, b;
    printf("Enter two values :");
    scanf("%d %d", &a, &b);
    printf("Sum = %d", a+b);
}
```

o/p:-

```
Enter two values : 10 20
Sum = 30
```

MACRO :-

- Simplified function is called 'macro'.
- Whenever a function body contains one or two statements, then it is called simplified function.
- Whenever the simplified function is occurred, always recommendent to go for 'MACROS'.

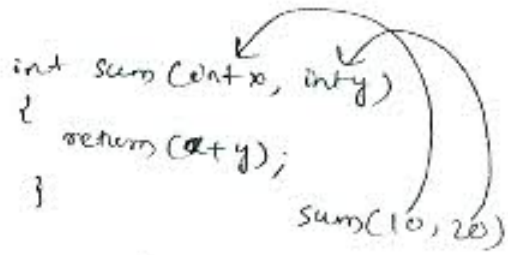
Advantages

- Macros are faster than normal functions.
- Macros don't occupy any physical memory.
- When we are working with macros, substitution will take place in place of bouncing process.

Drawbacks

- In macros, no any compilation errors will occur (syntax problem will not be considered)
- No any type checking process will be occurred in macros (Parameter types don't consider)
- From macros, we can't return the values.
- Control flow statements are not allowed in macro.

(x4) /* int sum (int x, int y)
{
return (x+y);
}
*/
#define sum(a,b) a+b
void main()
{
int s;
s = sum(10,20); // s = 10+20;
printf("sum = %d", s);
}



① Function memory → 2B/4B
② Actual Argument → 4B
③ Parameter → 4B
④ return statement → 2B
12B + Boundary marking

(o/p) sum = 30

note: s = sum(10+8+20); // s = 10+20=30;

- In the above program, at the time of preprocessing, 'sum' macro will be executed and simply it'll substitute the data in the form of a+b, i.e. 10+20.

(x5) /* int max (int x, int y)
{
if(x>y)
return x;
else
return y;
}
*/
#define max(a,b) a>b ? a : b
void main()
{
int m;
m = max(10,20);
printf("max value = %d", m);
}
(o/p) max value = 20

(ex6) #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
= 11

y to solve this problem :-

```
# 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)

(ex) #define CUBE(a) (a)*(a)*(a)

```
void main()
{
    int x, y;
    x = CUBE(2);
    y = CUBE(2+3);
    printf("x = %d  y = %d", x, y);
}
```

x = CUBE (2)
= (a) * (a) * (a)
= (2) * (2) * (2)
= (8)

y = CUBE (2+3)
⇒ y = (a) * (a) * (a)
⇒ y = (2+3) * (2+3) * (2+3)
= 5 * 5 * 5
= (125)

(o/p) x = 8 y = 125

(ex) #define SQR(a) (a) * (a)
#define CUBE(a) SQR(a) * (a)

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

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

(o/p) i = 125

(2) File Inclusion (#include) :-

- By using this preprocessor, we can include a file into another file
- By using this preprocessor, we can include any type of files like .c, .cpp, .xc, and .h, but generally we are including '.h' files.
- Header file is a source file, which contains forward declaration of predefined functions, global variables, constant values, predefined datatypes, predefined structures, predefined macros and inline functions.
- Header file doesnot contain any implementation part of predefined functions. It contains prototype only.
- A C-program is collection of functions.
- '.c' file is a combination of predefined and userdefined functions

- Normally source code contains implementation part of userdefined function, calling statement of predefined function.
- Coding part should should be required for predefined & userdefined functions also
- '.obj' file contains implementation part of userdefined function & .Lib file contains implementation part of predefined functions, and this file will be loaded into the application at the time of linking.
- As per the function concept, whenever we are calling a function before defination, it should require declaration, so all predefined functions are called before defination only, so we required to go for forward declaration.
- By using '.h' files we can provide forward declaration of predefined functions.
- We can't provide the forward declaration of predefined functions explicitly, because when we are providing forward declaration for predefined function, then it becomes user-defined, so mandatory to provide implementation part also, but it is not possible.

Syntax) `#include <filename.h>`

or

`#include "filename.h"`

#include <filename.h> :-

- By using this syntax, when we are including the header file, then it'll be loaded from default location; i.e. "C:\TC\INCLUDE".

Generally, by using this syntax, we are including predefined headerfiles.
When we need to load the userdefined headerfile by using this syntax, then userdefined headerfile should be placed in predefined location, i.e. "C:\TC\INCLUDE".

#include "filename.h" :-

- By using this syntax, when we are including the header file, those'll be loaded from current project directory, i.e. "...\projectname\include".
- Generally by using this syntax, we are including userdefined headerfiles.
- When we are including the predefined headerfile by using this syntax, first it searches in current project location, if it's not available then loads from default location.

```
#include <stdio.h>      #include <dir.h>      #include <alloc.h>
#include <conio.h>      #include <malloc.h>   #include <mem.h>
#include <stdlib.h> → standard conversion
#include <math.h> → mathmatics
#include <string.h> → String related
#include <limits.h>
#include <signal.h>
#include <dos.h>
#include <process.h>
```

③ conditional compilation preprocessor :-

- By using this preprocessor, depending on the condition, block needs to be pass for compilation or not, which will be decided at the time of preprocessing.
- In conditional compilation preprocessor, if condition is true, then block will be passed for compilation; if condition is false, then correspondent block will be remove from source.
- The basic advantage of this preprocessor is reducing the .exe file size.

(ex1)
```
void main()
{
    printf("welcome");
    printf("A");
    #if 5<2
        printf("C");
        printf("B");
    #endif
    printf("NIT");
}
```

↦ Preprocessed file (.I)
```
1. void main
2. {
3.     printf("welcome");
4.     printf("A");
5.
6.
7.
8.
9.  printf("NIT");
10. }
11.
```

(op):- welcome A NIT

- In the above program at the time of preprocessing condition became false, so corresponding block will be removed from source.

(ex2)
```
void main()
{
    printf("A");
    # if 1!=2<5
        printf("B");
        printf("C");
    # else
        printf("X");
        Printf("Y");
    #endif
    printf("welcome");
}
```

(op) AXY welcome

ex3)
```
void main()
{
    printf("NIT");
    #if 2<5 && 2>5
        printf("A");
        printf("B");
    #elif 1!=2>5
        printf("X");
        printf("Y");
    #else
        printf("C");
        printf("D");
    #endif
    printf("Hello");
}
```

op) NITXY Hello

{ #ifdef → #if defined.
  #ifndef → #if not defined
  ⇓

#ifdef and #ifndef are called macro testing condetional compilation preprocessor.

By testing this preprocessors, we can avoid multiple substitutions of header file code.

ex4)
```
void main()
{
    printf("welcome");
    #ifdef Test
        printf("Hello");
        printf("NIT");
    #endif
    printf("AB");
}
```

- In this program, 'Test' macro is not defined, that's why correspondent block will not be passed for compilation process.

op) welcome AB

ex5)
```
#define NIT
void main()
{
    printf("Hello");
    #ifdef NIT
        printf("NIT");
        printf("C");
    #endif
    printf("welcome");
}
```

op) Hello NITC welcome

ex6)
```
#define Test
void main()
{
    printf("A");
    #ifndef Test
        printf("B");
        printf("C");
    #endif
    printf("welcome");
}
```

op) A welcome

- In the program(ex6), Test is called null macro, because it doesnot contains replacement text.
- whenever a macro doesn't has replacement text, then it's called null macro

#undef :-

- By using this preprocessor, we can close the scope of existing macro.
- Generally, by using this preprocessor, we are redefining existing macro.

(ex:) #define A 10
```
void main()
{
    printf("%d", A); // 10
    #undef A
#define A 20
    printf("%d",A); //20
    #undef A
#define A 30
    printf("%d", A); //30
    #undef A
printf("%d", A); // error.
}
```
(note) → After closing the scope of an identifier, we can't use that identifier, until we are redefining it.

(o/p) ~~102030~~ | Error: undefined symbol 'A'

#error :-

By using this preprocessor, we can create userdefined error messages.

(ex6)
```
#define NIT
void main()
{
    #ifndef NIT
        #error please define NIT
    #endif

    #ifdef NIT
        printf("NIT");
        printf("welcome");
    #endif
}
```
- In the above program, if 'NIT' macro is not defined, then it gives an error at the time of preprocessing.

#line

↳ By using this preprocessor, we can create user defined line sequences in .I file.

eg)
```
void main()
{
    printf ("welcome");
    printf (" A");
    #if 5 < 2
        printf (" C");
        printf (" B");
    #endif
  #line 5
    printf ("NIT");
}
```

```
1. void main() #line 1
2. {
3.     printf ("welcome");
4.     printf(" A");
5.
6.
7.
8.
5.  #line 5
6.  printf(" NIT");
7. }
```

o/p welcome A NIT

- In the above program, at the time of preprocessing line sequences are reset to '5' in .I file.

#pragma

↳ This is compiler dependent headerfile, i.e some of the compilers will not recognise this preprocessor.

A preprocessor directive that is not specified by the ISO standard.

pragmas after control actions of the compiler & 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 & exit functions should not receive or return any values.

#pragma warn used to supress (ignore) specific warning msgs from compiler.

```
#pragma warn -rvl
        return value warnings
#pragma warn -par
        parameter not used warnings.
#pragma warn -rch
        unreachable code warnings.
```

- In the program (ex6), Test is called null macro, because it doesn't contain replacement text.
- whenever a macro doesn't has replacement text, then it's called null macro

#undef :-

- By using this preprocessor, we can close the scope of existing macro.
- Generally, by using this preprocessor, we are redefining existing macro.

(ex7)
```
#define A 10
void main()
{
    printf("%d", A); // 10
    #undef A
#define A 20
    printf("%d",A); //20
    #undef A
#define A 30
    printf("%d", A); //30
    #undef A
printf("%d", A); // error
}
```

(note) → After closing the scope of an identifier, we can't use that identifier, until we are redefining it.

(o/p) Error : undefined symbol 'A'

#error :-

By using this preprocessor, we can create userdefined error messages.

(ex8)
```
#define NIT
void main()
{
    #ifndef NIT
        #error please define NIT
    #endif

    #ifdef NIT
        printf("NIT");
        printf("welcome");
    #endif
}
```

- In the above program, if 'NIT' macro is not defined, then it gives an error at the time of preprocessing

#line

↳ By using this preprocessor, we can create user defined line sequences in .I file.

eg)
```
void main()
{
    printf ("welcome");
    printf (" A");
    #if 5 < 2
        printf ("C");
        printf (" B");
    #endif
#line 5
    printf ("NIT");
}
```

```
1. void main(); #line 1
2. {
3.     printf ("welcome");
4.     printf(" A");
5.
6.
7.
8.
5.  #line 5
6.  printf(" NIT");
7.  }
```

welcome A NIT

- In the above program, at the time of preprocessing line sequences are reset to '5' in .I file.

#pragma

↳ This is compiler dependent headerfile, i.e some of the compilers will not recognise this preprocessor.

A preprocessor directive that is not specified by the ISO standard.

pragmas after control actions of the compiler & 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 & exit functions should not receive or return any values.

#pragma warn used to supress (ignore) specific warning msgs from compiler.

#pragma warn -rvl
    return value warnings

#pragma warn -par
    parameter not used warnings.

#pragma warn -rch
    unreachable code warnings.

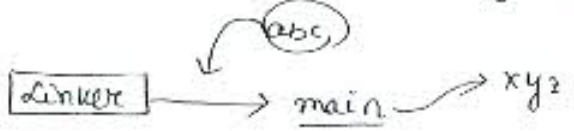
5/11/12

ex)
```
void abc (void) ; // declaration
void xyz (void) ; // declaration
#pragma startup abc
#pragma exit xyz
void abc ()
{
  printf ("\n from abc()");
}
void xyz ()
{
  printf ("\n from xyz()");
}
void main()
{
  printf ("\n from main()");
}
```

o/p

```
from abc()
from main()
from xyz()
```

- In the above program, before execution of main() function, program will start from abc(). After execution of main() function, program will be terminated with xyz().
- Without startup & exit options, automatically main() function will be executed.



- In implementation, when we are having more than one startup & exit function then according to the priority, functions are executed.
- In # pragma startup option, which function contains highest priority, that one will execute first, and which function contains least priority will execute at last.
- In implementation, when equal priority is occurred in startup option then which function is specified at last that one will execute first.
- In # pragma exit option, which function is having highest priority, that one will execute at last, function having least priority will execute first.
- In # pragma exit option, if equal priority is occurred, then which is specified at first will execute at first.

ex2)
```
void start1()
{
   printf("\n from start1 ");
}
void start2()
{
   printf("\n from start2 ");
}
void end1()
{
   printf("\n from exit1");
}
void end2()
{
  printf("\n from exit2");
}
#pragma startup start2 1
#pragma startup start1 2
#pragma exit end2 1
#pragma exit end1 2
void main()
{
   printf("\n from main()");
}
```

o/p
```
from start 2
from start1
from main()
from exit 1
from exit 2
```

ex3)
```
# pragma warn -rvl
#pragma warn -par
#pragma warn -rch
int abc(int a)
{
   printf("\n Hello abc");
}
void main()
{
   printf("\n Hello main()");
   abc(10);
   return ; // back to OS
   getch();
}
```

o/p
```
Hello main()
Hell abc
```

ARRAY

- An array is a derived datatype in C, which is constructed from fundamental datatype of C-language.
- An array is a collection of similar type of data element in a single variable.
- In implementation, when we require n no. of variables of same datatypes, then go for an array.
- When we are working with arrays, compile time memory management happens, that's why always memory will be constructed in continuous manner.
- When we are working with arrays, randomly we can access the data, because memory will be there in continuous manner.
- When working with array, all elements share same name, with unique identification value called index.
- When working with array, we require to use array subscript operator, i.e. [].
- Array subscript operator will require one argument of type unsigned integer constant, which value is always greater than zero (0).
- In arrays, always array index starts from '0' and ends with size-1

Syntax: `Datatype arr_name [size];`

Properties of 1D array :-

① `int arr[5];`

size → 5;

sizeof(arr) → 10 Bytes (5 × 2B = 10B)

② `int arr[5];`

size → 5

sizeof(arr) → 10 Bytes

5 int variables

	index / Pos^n of element
arr[0] →	0
arr[1] →	2
arr[2] →	3
arr[3] →	4
arr[4] →	5

③ `int arr[]; // error: size is unknown`

- In declaration of array, mandatory to mention size of the array or else it gives an error, i.e. "unknown size" or "size is unknown".

④ int arr[0] ; // error

⑤ int arr[-5] ; // error

- In declaration of the array, size must be unsigned integer type, which value is always greater than '0'.

⑥ int arr[5] = {10, 20, 30, 40, 50} ;

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

⑦ int arr[5] = {10, 20, 30}; valid ✓

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

- In initialization of the array, if specific no. of elements are not initialized, then rest of all values will be initialized with 0's.

⑧ int arr[3] = {10, 20, 30, 40, 50} ; // error X

In array initialization process, we can't initialize more than size of array elements. In this case, compiler will give an error, i.e. "too many initializations".

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

size = 5 ;
size of (arr) → 10 B

- In initialization of the array, mentioning the size is optional. In this case, how many elements are initialized, that many elements are created.

⑩ int arr[2] ;

arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
arr[3] = 40;

In Java / C++, → error

valid ✓ in C'

- In C' & C++, there is no any upper boundary checking error process occurs, that's why, when we are crossing the upper limit, depending on the O.S. security level, anything can happen.

⑪ int arr[40000] ; // error
4 40000 * 2B = 80000 Bytes

C' program is created in Data Segment &
Data Segment = 64 KB = 65,536 Bytes

- In DOS based compiler, at compile time, maximum of 64 KB of data, i.e. 65536 Bytes only can be created, but in the above program / syntax, we require 80000 Bytes.

(12) long int arr[20000]; // error
↳ 20000 × 4B = 80000 Bytes

(13) char arr[40000]; valid ✓
↳ 40000 * 1B = 40000 Bytes < 65536 Bytes

(14) float arr[13.8]; // error

(15) float arr[14];
size → 14;
sizeof(arr) → 14×4 = 56 Bytes

(16) int size = 10;
int arr[size]; // error

(17) const int size = 10;
int arr[size]; // error

- On declaration of the array, size must be unsigned constant only, i.e variables or constant variables are not allowed.

(18) #define size 10
int arr[size]; // valid } at compilation
int arr[10]; ✓

- On declaration of array, size can be symbolic constant value, because at the time of preprocessing, identifier will be replaced with constant value.

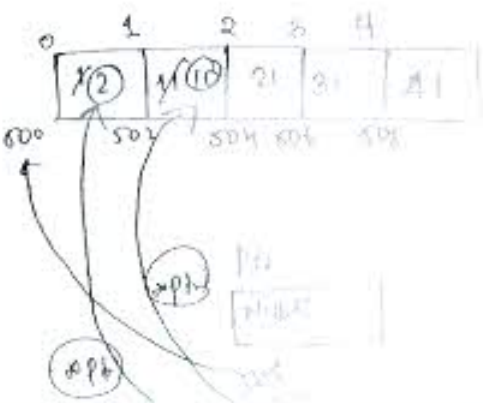
(19) int arr[2+3]; valid → int arr[5];

(20) int arr[2*5]; error → int arr[0];

(21) int arr[5/2]; valid → int arr[1];

(ex.)
```
#include<stdio.h>
#include<conio.h>
int main()
{
    int arr[5] = {1, 11, 21, 31, 41};
    int near * ptr = (int near*) NULL;
    ptr = &arr[0];
    ++ptr;
    --*ptr;
    --ptr;
    printf("%d %d", arr[0], arr[1]);
    getch();
    return 0;
}
```

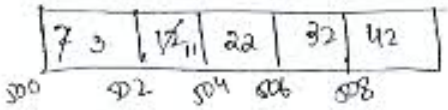
o/p: 2 10



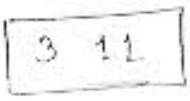
x3) 6|11|12

```
#include <stdio.h>
#include <conio.h>
int main()
{
    int arr[5] = { 2, 12, 22, 32, 42 };
    int *ptr = (int *) NULL;
    ptr = &arr[0];
    --ptr;
    ++*ptr;
    ++ptr;
    --*ptr;
    printf("%d %d", arr[0], arr[1]);
    getch();
    return 0;
}
```

(RDE) When we are working with unary operators, if equal priority is occurred, then we require to evaluate towards from right to left



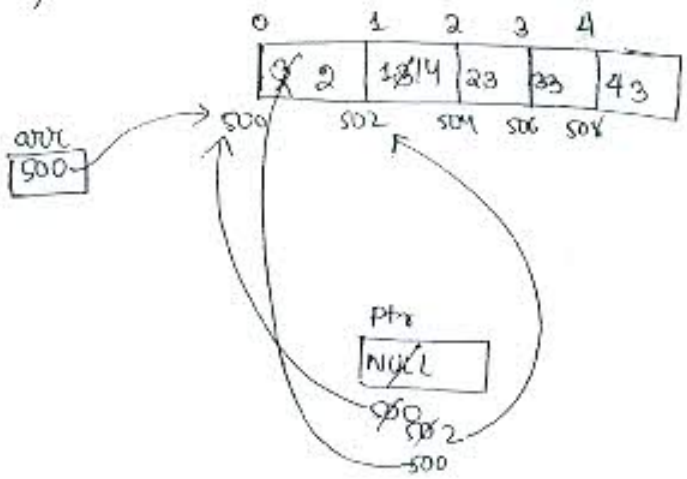
(*/)



(Ex 3)

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

O/P → 8 14

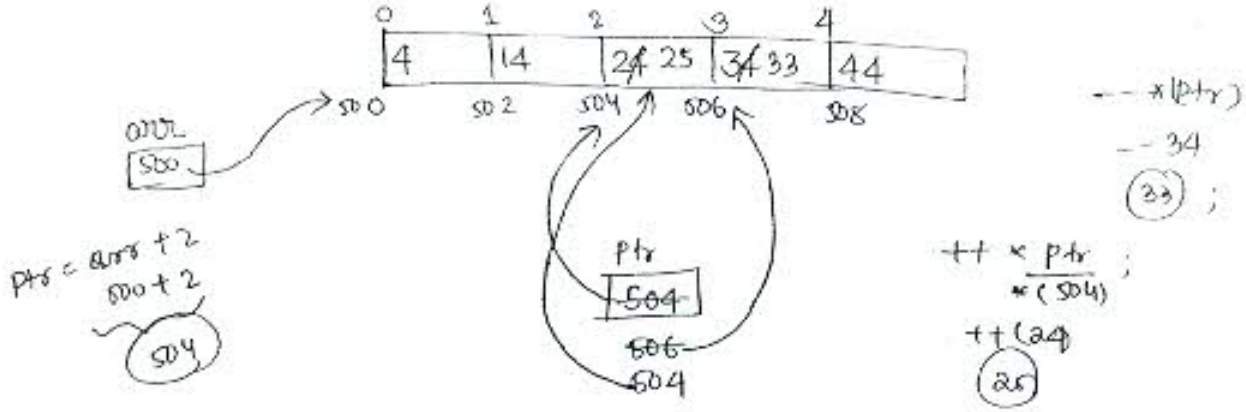


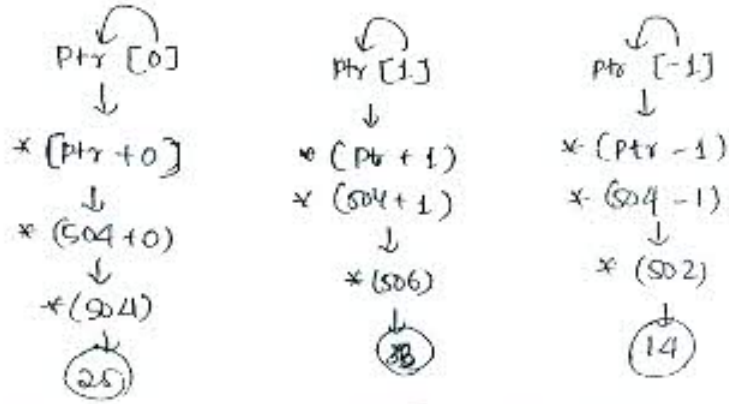
- An array is an implicit pointer variable, which always holds base address of an array.
- 'arr' name always provides base address of an array, i.e. "&arr[0]".
- 'arr+1' will provide next address of an array, i.e. &arr[1].

(Ex 4)

```
#include <stdio.h>
#include <conio.h>
int main()
{
    int arr[] = { 4, 14, 24, 34, 44 };
    int *ptr = arr + 2; // &arr[2];
    ++ptr;
    --*ptr;
    --ptr;
    ++*ptr;
    printf("%d %d %d", ptr[0], ptr[1], ptr[-1]);
    getch();
    return 0;
}
```

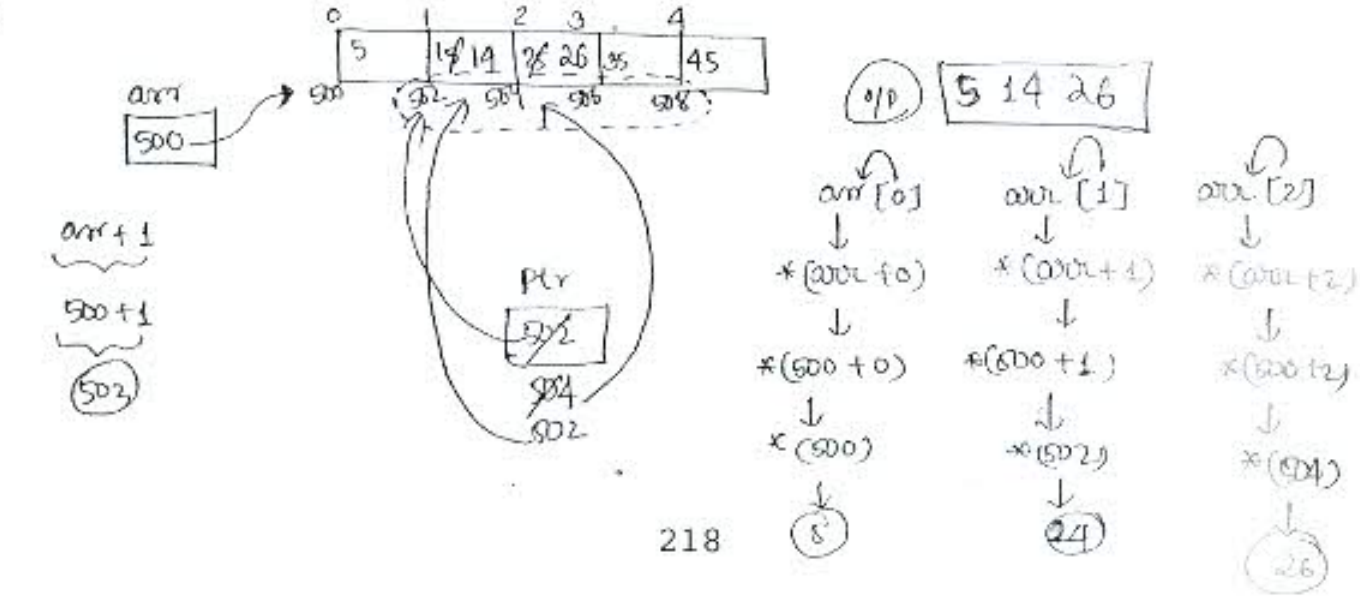
O/P → 25 23 14



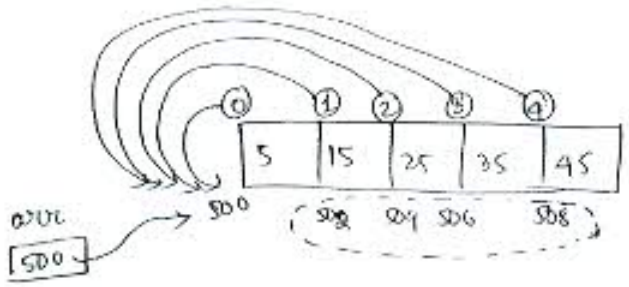


- When we are working with pointer, or pointer variable, when we are applying the subscript operator, then index value will be mapped with current pointer data and it applies indirection operator.

```
#include <stdio.h>
#include <conio.h>
int main()
{
    int arr[] = {5, 15, 25, 35, 45};
    int *ptr = arr + 1;
    ++ptr;
    ++*ptr;
    --ptr;
    --*ptr;

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



Array Architecture



arr [3]
↓
* (arr + 3)
↓
* (500 + 3)
↓
* (506)
↓
(35)

Why the index is started with '0' ?

if index = 1
arr [1]
↓
* (arr + 1)
↓
* (500 + 1)
↓
* (502)
↓
(15) → 2nd element

we can't get (5)
↓
1st element

if index = 0
arr [0]
↓
* (arr + 0)
↓
* (500 + 0)
↓
* (500)
↓
(5) 1st element

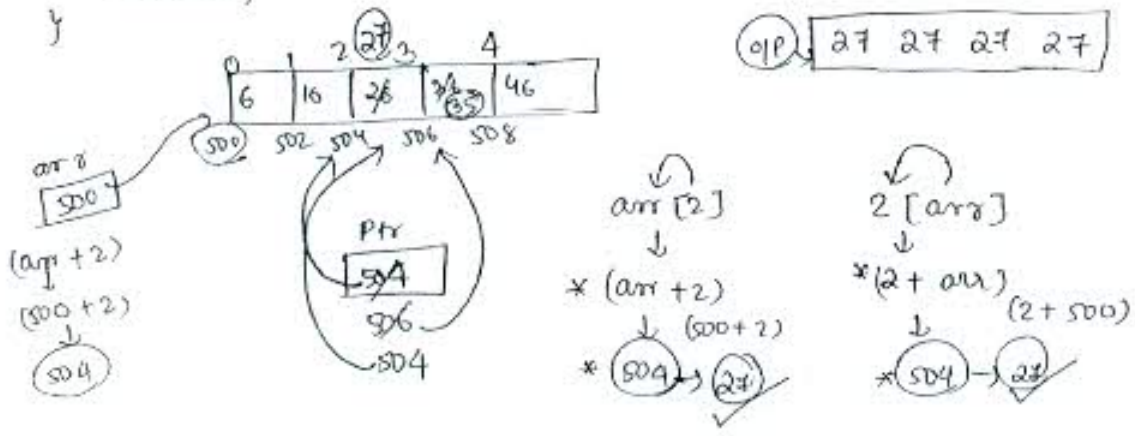
- According to the architecture of Array, when we are dealing with array, it doesn't maintain all element information except base address of an array, because an array is an implicit pointer variable, which holds base address.
- In arrays, to manage remaining element information, base address is mapped with index value
- On array, when we are applying the subscript operator, index value will be mapped with base address of an array and it applies indirection operator on newly calculated address.
- Always array index should start from 0' only, because if we are starting the index with 1', then as per the architecture, it'll access the 2nd element in place of first.

i.e.
arr [1]
↓
* (arr + 1)
↓
500 + 1
↓
* (502)
↓
2nd ✗

arr [0]
↓
* (arr + 0)
↓
(500 + 0)
↓
* (500)
↓
1st ✓

206) #
#

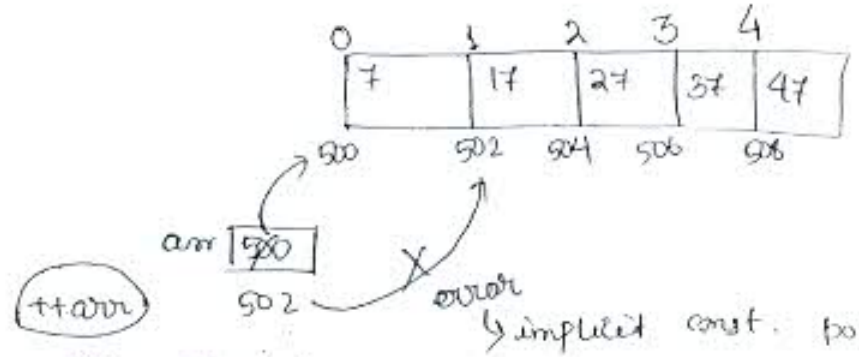
```
int main()
{
    int arr[] = { 6, 16, 26, 36, 46 };
    int *ptr = arr + 2;
    ++ptr;
    --*ptr;
    --ptr;
    ++*ptr;
    printf("%d %d %d %d", arr[2], *(arr+2), 2[arr], *(2+arr));
    getch();
    return 0;
}
```



207) #
#

```
int main()
{
    int arr[] = { 7, 17, 27, 37, 47 };
    ++arr; // arr = arr+1;
    ++*arr;
    --arr; // arr = arr-1;
    --*arr;
    printf("%d %d", arr[0], arr[1]);
    getch();
    return 0;
}
```

o/p → error

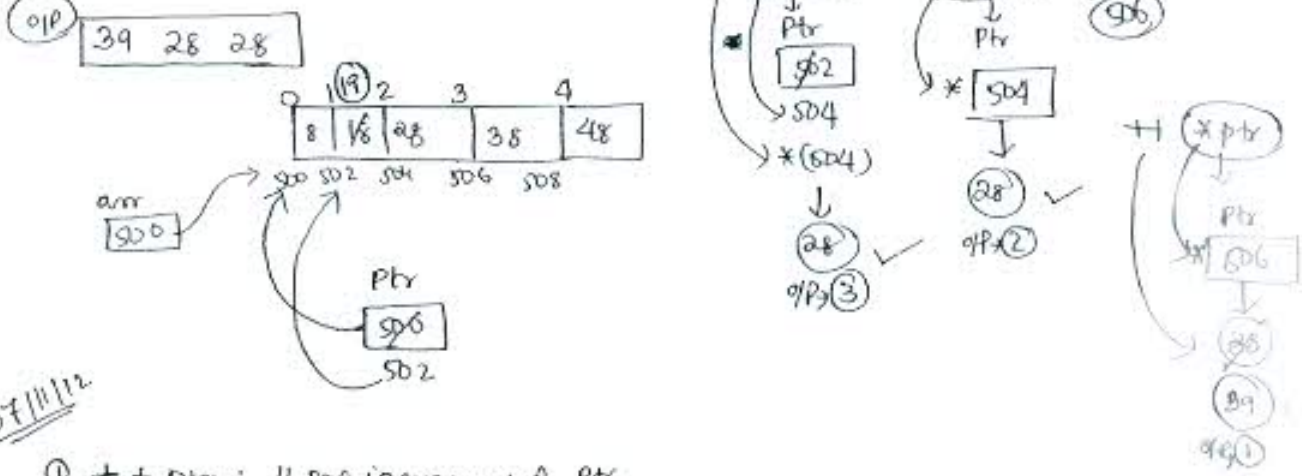


arr = arr + 1
= (500 + 1)
= (502) X

- An array is an implicit constant pointer variable, so it's not possible to change the base address of an array.
- On 'arr', we can't apply increment, decrement operations, because it is of constant type.

(ex8)
```
#include<stdio.h>
#include<conio.h>
int main()
{
    int arr[] = {8, 18, 28, 38, 48};
    int *ptr = arr;
    ++ptr;
    ++*ptr;
    printf("%d %d %d", ++*ptr, *ptr++, *++ptr);
    getch();
    return 0;
}
```

L ← R



07/11/12

① ++ptr ; // pre increment of ptr
② ptr++ ; // post increment of ptr.
③ --ptr ; // pre decrement of ptr
④ ptr--; // post dec. of ptr

① ++ *ptr ; // pre increment of obj
② *ptr++ ; // post " "
③ --*ptr ; // pre decrement of obj
④ *ptr-- ; // post " "

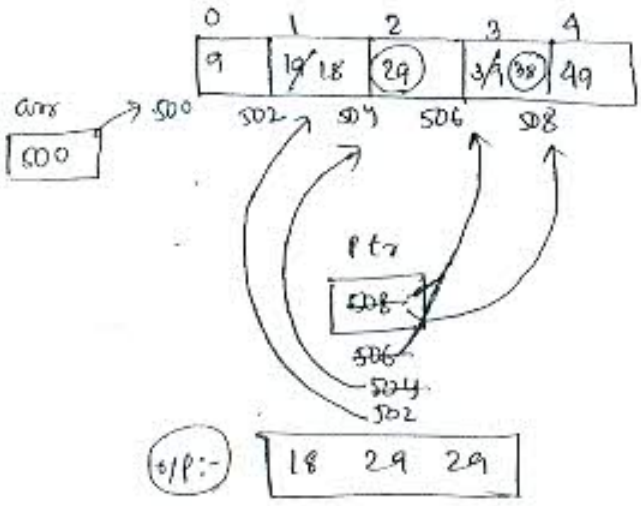
1. *++ptr ; // Pre inc. of ptr & accessing data.

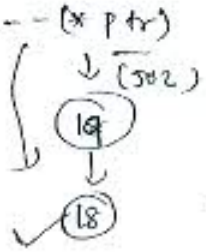
2. *ptr ++ ; // accessing data & post inc. of ptr

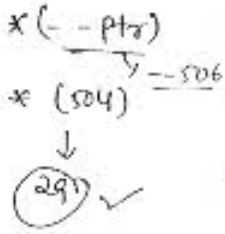
3. *--ptr ; // Pre dec. of ptr & accessing data.

4. *ptr--; // accessing the data & post decrement of ptr

ex9)
```
#include<stdio.h>
#include<conio.h>
int main()
{
    int arr[] = { 9, 19, 29, 39, 49 };
    int *ptr = arr + 4 ;
    --ptr ;
    --*ptr ;                                   L ← R
    printf("%d %d %d", --*ptr, *ptr--, *--ptr);
    getch();
    return(0);
}
```



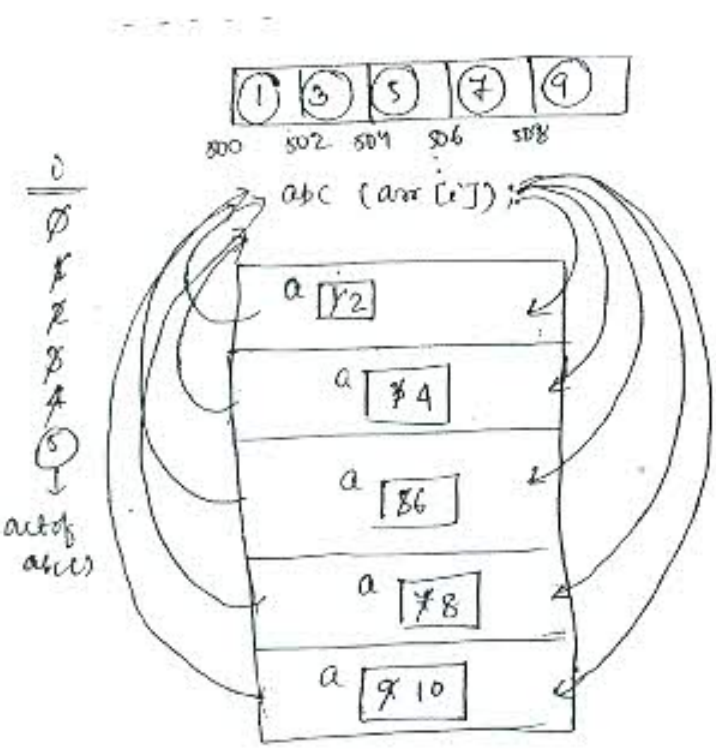




Array & function

Ex1)

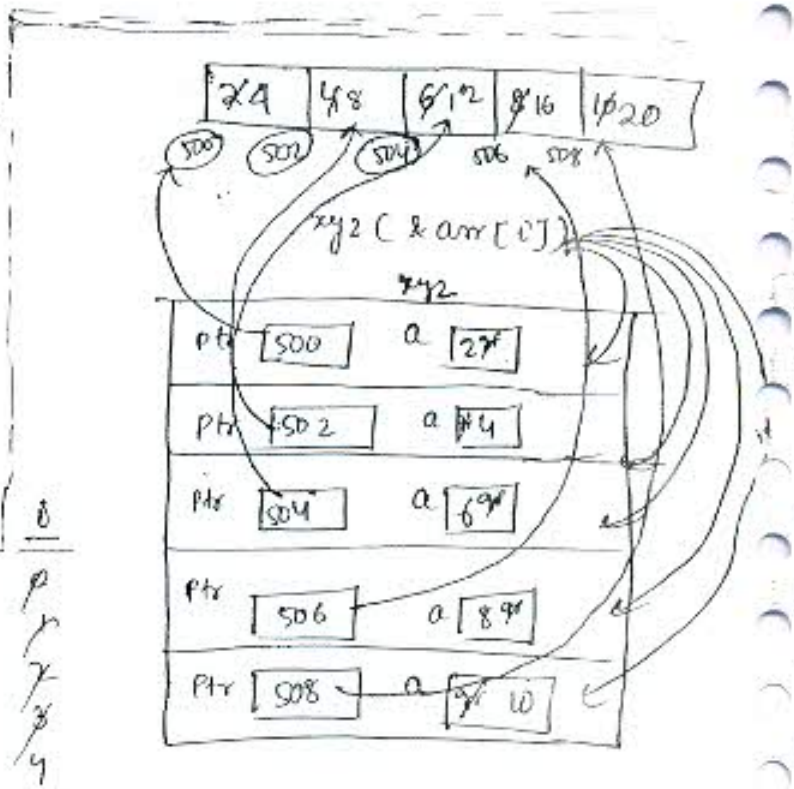
```
#include<stdio.h>
#include<conio.h>
void abc( int a )
{
   ++a;
   printf("  %d", a);
}
void main()
{
   int arr[5] = { 1, 3, 5, 7, 9 };
   int i;
   printf(" Data in abc : ");
   for(i=0; i< 5 ; i++)
         abc(arr[i]);
   printf(" In Data in main :");
   for( i=0 ; i<5 ; i++)
         Printf(" %d", arr[i] );
   getch();
   return 0;
}
```



- In the previous program, array elements are passing by using call by value mechanism, that's why, no any modification of abc() function will Barry back to main().

- In implementation, when we are expecting the modifications back to calling location, like go for "call by Address".

ex:
```
#include<stdio.h>
#include<conio.h>
void xyz(int *ptr)
{
    int a;
    a = *ptr;
   *ptr = a*2;
    printf("%d ", *ptr);
}
int main()
{
    int arr[5] = {2,4,6,8,10};
    int i;
    printf("\n Data in xyz(): ");
    for(i=0; i<5; i++)
        xyz(&arr[i]);
    printf("\n Data in main(): ");
    for(i=0; i<5; i++)
        printf("%d ", arr[i]);
    getch();
    return 0;
}
```



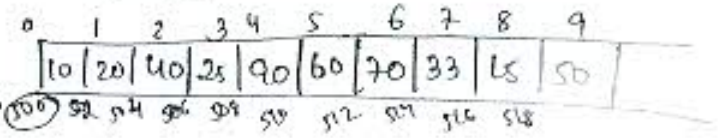
O/P:
```
Data in xyz(): 4 8 12 16 20
Data in main(): 4 8 12 16 20
```

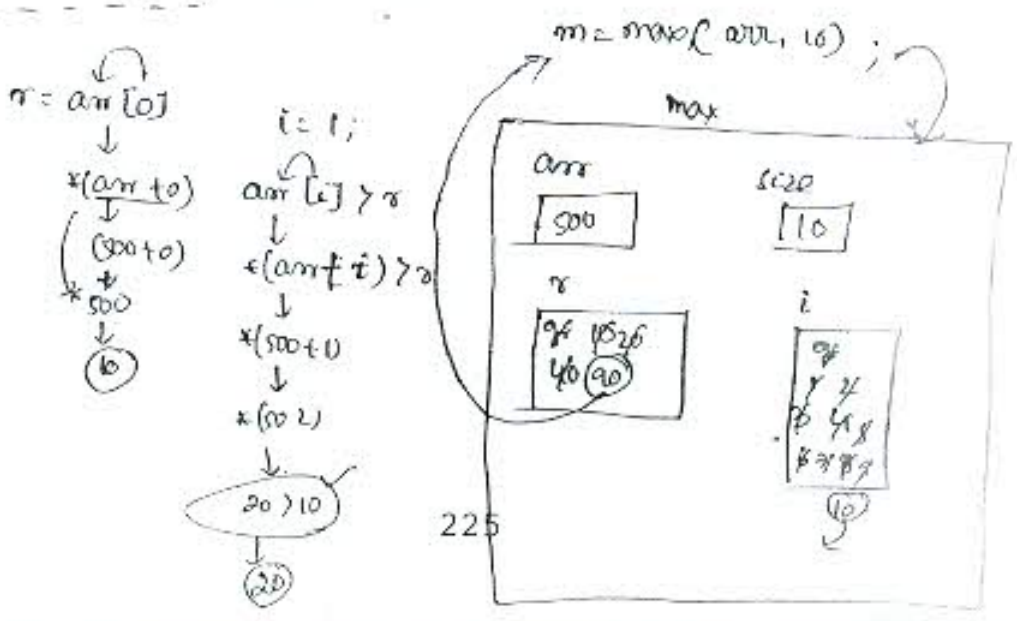
- In above program, array elements are passed by using call by Address mechanism, that's why all modifications xyz() function will carry back to main().

(Ex3)

```
Enter 10 values : 10 20 40 25 90 60 70 33 15 50
Max value of list : 90
```

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





- In any programming languages, it is not possible to pass entire array as an argument to the function, because it creates memory wastage.
- In C-programming language, it is not possible to pass entire array, but it's posible to access entire array from outside of the function with the help of pointers.
- In implementation, when we need to access entire array from outside of the function, then we requires to pass base address along with size of the array
- By using base address of the array with the help of indexes, we can access complete array elements by using size of the array.
- In parameter locations, it's not possible to create array elements.
- In parameter location, if 'int arr[]' syntax is available, then it creates a pointer variable and syntax is indicating that one dimensional array base address is holding.

Ex:

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

< min value program using array & function >

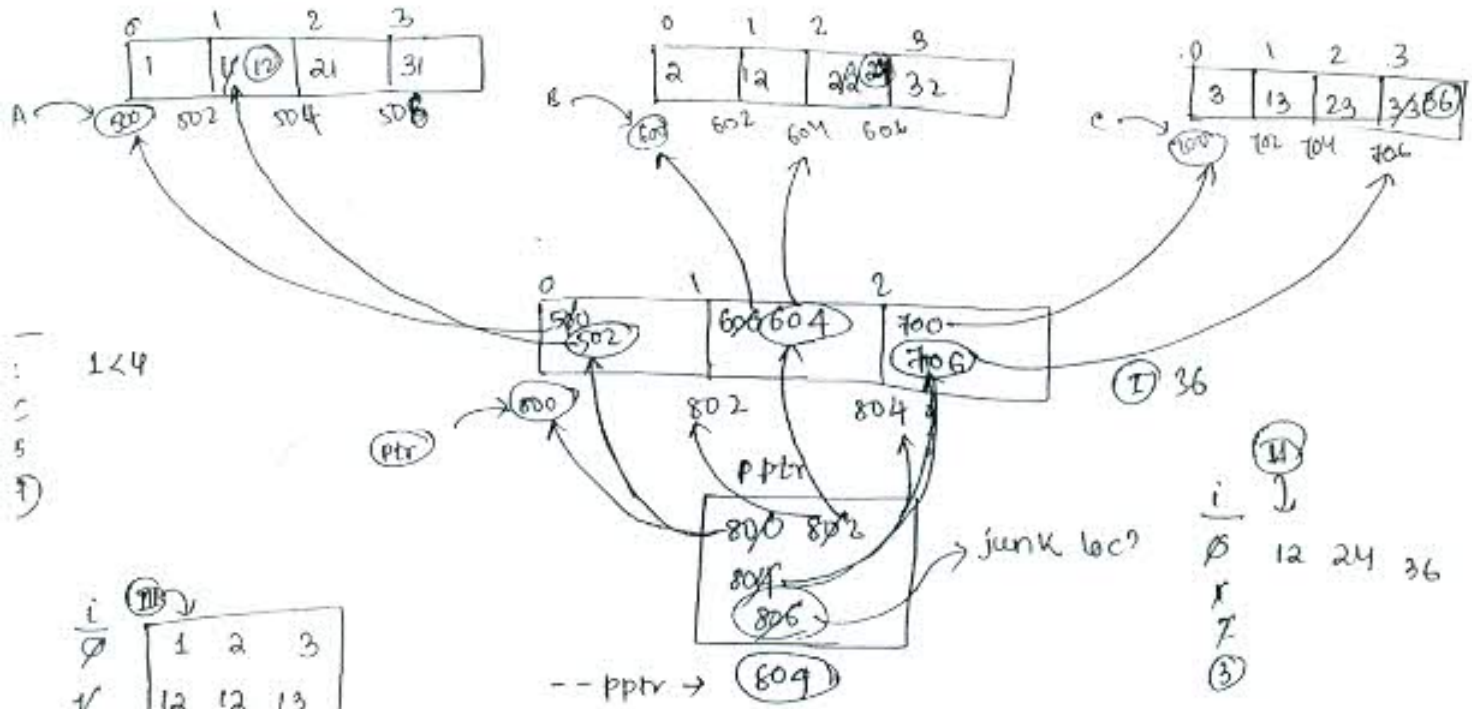
(o/p)

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

Min value of list : 10

(ex5)

```
#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;
    int i;
    clrscr();
    ptr[0] = A; // ptr[0] = &A[0]
    ptr[1] = B; // ptr[1] = &B[0]
    ptr[2] = C; // ptr[2] = &C[0]
    pptr = ptr; // pptr = &ptr[0]
    for(i=1; i<4; i++)
    {
        *pptr += i; // *pptr = *pptr + i;
        **pptr += i; // **pptr = **pptr + i;
        ++pptr;
    }
    --pptr;
    printf("\n %d \n", **pptr);
    for(i=0; i<3; 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;
}
```



1<4

i
0
1
2
3

0	1	2	3
1	12	12	13
2	21	24	23
3	31	32	36

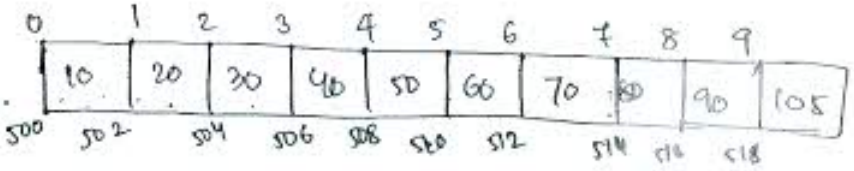
96			
12	24	36	
1	2	3	
12	12	13	
21	24	23	
31	32	36	

Arr

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

```
#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", &a[i]);
```

```
for (i=0 ; i< size ; i++)
        sum += arr[i] ;  // sum = sum + arr[i] ;
avg = (float) sum / size ;
printf("\n Sum of list : %d", sum);
printf("\n Avg of list : %.2f", avg);
getch();
return 0;
}
```



i
0 1 2 3 4 5 6 7 8 9 10

size
10

sum
0
10
30
60
100 150 210 280 360 450 550

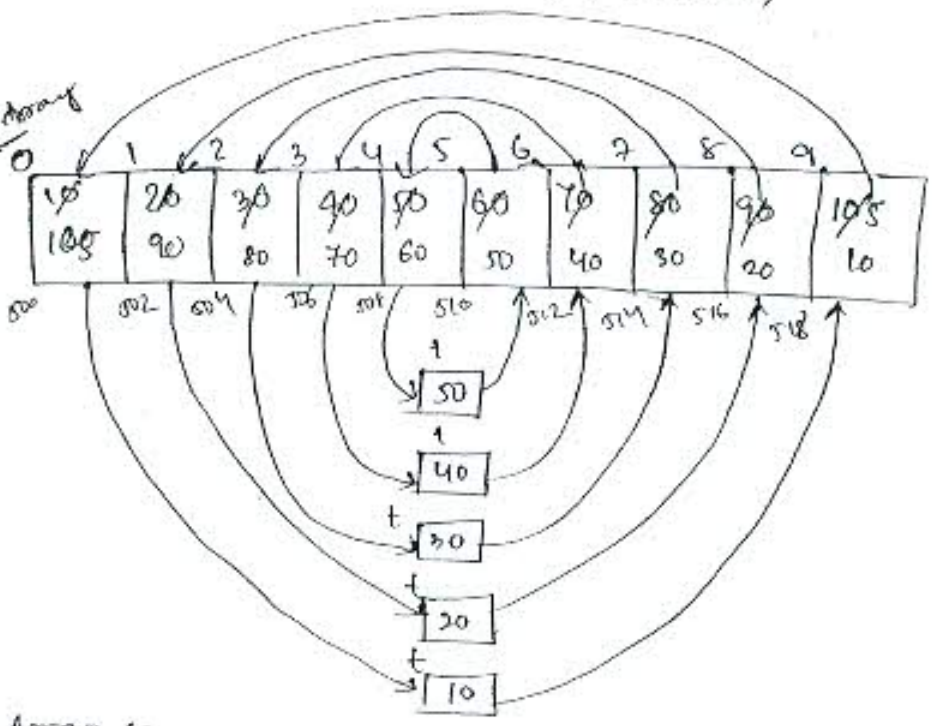
ext

```
Enter 10 values : 10 20 30 40 50 60 70 80 90 105
Reverse Arr Data : 105 90 80 70 60 50 40 30 20 10
```

```
#include <stdio.h>
#include <conio.h>
#define size 10
int main()
{
    int arr[size];
    int i, t;
    clrscr();
    printf("\n 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 : ") ;
    for ( i=0 ; i<size ; i++ )
        printf ("%d", arr[i]);
    getch();
    return 0;
}
```

```
/* reversing array without using third variable */
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;
}
```

Logic for reversing array

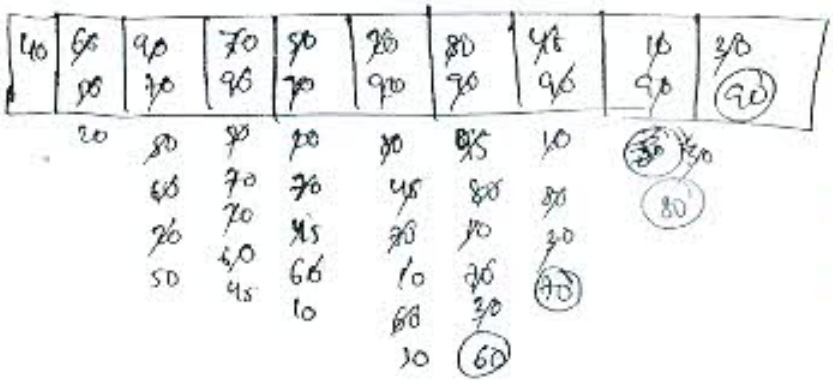


→ Sorting of Array ←

Enter 10 values : 40 60 90 70 90 20 80 45 10 30
Sorted list : 10 20 30 40 45 50 60 70 80 90

```
/* Bubble Sort */
for (i=0; i<size; i++)
{
    for (j=0; j<size-i-1; j++)
    {
        if (arr[j] > arr[j+1])
        {
            t = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = t;
        }
    }
}
```

// logic to fix the last element i.e max^m element.



/* bubble sort without using third variable */

```
⇒ for ( i = 0 ; i<size ; i++ )
  {
     for ( j=0 ; j<size-i-1 ; j++ )
     {
        if (arr[j] > arr[j+1])
        {
            arr[j] = arr[j] + arr[j+1] ; // a = a+b;
            arr[j+1] = arr[j] - arr[j+1] ; // b = a-b;
            arr[j] = arr[j] - arr[j+1]; // a = a-b;
        }
     }
  }
```

9/11/12

Ex9

Enter 10 values : 50 30 70 10 60 80 20 40 90 35

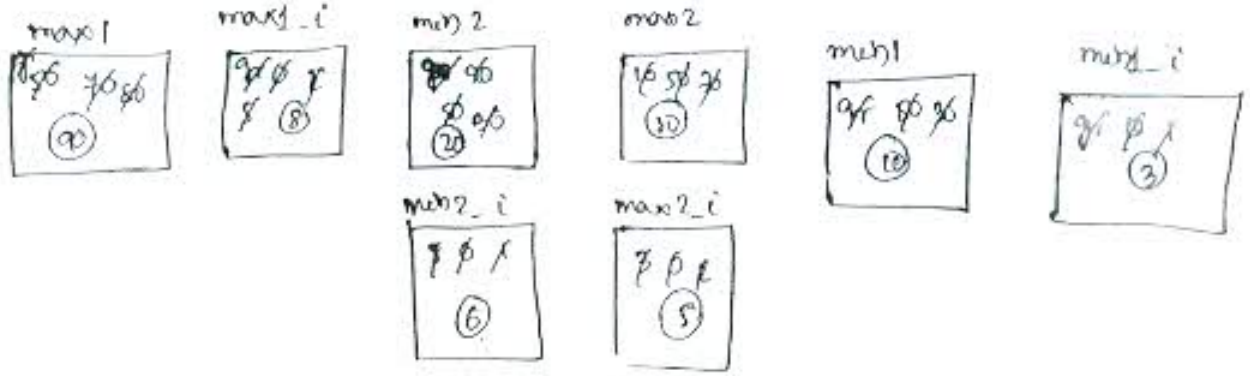
Max1 = 90 Index = 8
Min 1 = 10 Index = 3
Max 2 = 80 Index = 5
Min 2 = 20 Index = 6

→ without sorting the array

```
#include <stdio.h>
#include <conio.h>
#define size 10
int main()
{
    int arr[size]
    int max1, min1, max2, min2 ;
    int max1_i, min1_i , 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)
    {
        min1 = arr[i];
        min1_i = i;
    }
}
max2 = min1;
max2_i = min1_i;
min2 = max1;
min2_i = max1_i;
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;
    }
}
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;
}
```

0	1	2	3	4	5	6	7	8	9
50	30	70	10	60	80	20	40	90	35



2D ARRAY :-

- In 2D-array, elements are arranged in rows, columns format.
- when we are working with 2D array, we require to refer two subscript operators, which indicates row size, column size.
- The main memory of 2D array is rows, elements are available in columns.
- When we are working with 2D array, arr_name gives main memory location, i.e. 1st row base address, arr+1 gives next main memory of 2D-array, i.e. 2nd row base address.
- when we are working with 2D-array, for accessing the elements, we require to use two subscript operators,
- On 2D array, when we are refering one-subscript operator, then it gives row address.

Syntax:

```
DataType arr_name [RSIZE][CSIZE] ;
```

Properties of 2D Array :-

① int arr[3][4] ;

size → 3×4

sizeof(arr) → 24B (3×4×2B)

② int arr[3][2] ;

size → 3*2

sizeof(arr) → 12B ((3×2)×2B = 12B)

3 rows

each row → 2 int variables

6 int variable

arr[0][0] → 1st element

arr[0][1] → 2nd "

arr[1][0] → 3rd "

arr[1][1] → 4th "

arr[2][0] → 5th "

arr[2][1] → 6th "

④ int arr[][] ; Error

⑤ int arr[3][] ; Error

⑥ int arr[][3] ; Error

↳ In declaration of 2D array, it's mandatory to mention row & column size or else it gives an error.

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

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

arr[1][0] → 40
arr[1][1] → 50
arr[1][2] → 60

- By using the above initialization process, we required to initialize the elements in sequence only, i.e. selected no-of elements can't be initialized.
- When we need to initialize the elements randomly, then go for row-of-row initialization process, i.e. Place multiple single dimensional array in a single unit.

⑦
```
int arr[4][3] = {
                  {10, 20};
                  {30},
                  {40, 50, 60};
                  {0}
                };
```

- In initialization of the 2D-array if specific no of elements are not initialized, then rest of all elements will be initialized with '0'.

⑧
```
int arr[][] = {
                {10},
                {20, 30},
                {40}
              };
```
error

⑨
```
int arr[3][] = {
                 {10, 20, 30},
                 {40, 50},
                 {60}
               };
```
Error

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

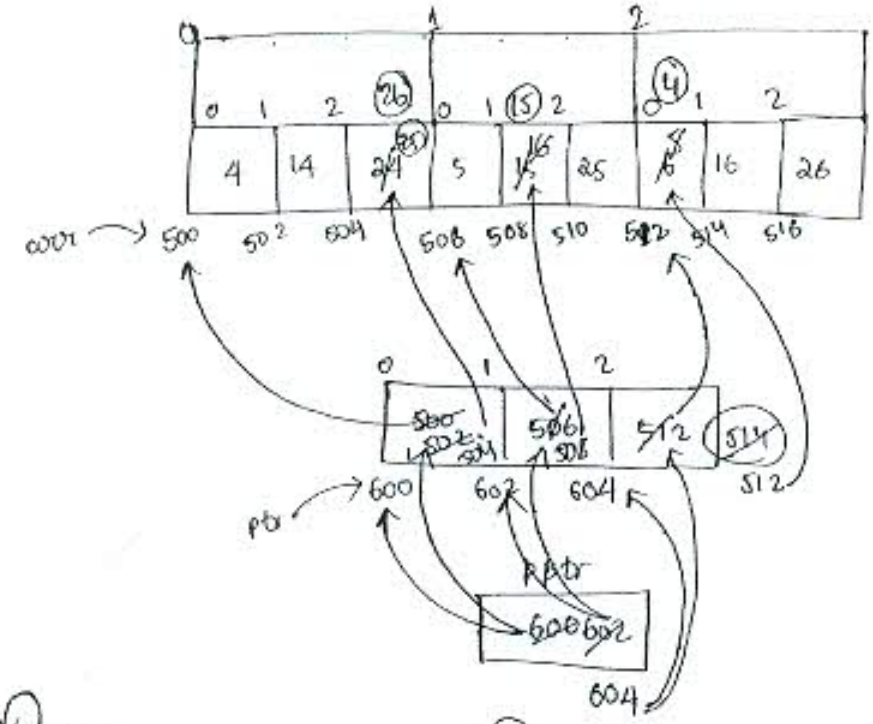
- In initialization of the 2D array, specifying the row size is optional, but column size is mandatory. But in declaration, row & column sizes, both are mandatory.

(Q.1)

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

int main()
{
    int arr[3][3] = {
                        {4, 14, 24},
                        {5, 15, 25},
                        {6, 16, 26}
                    };
    int *ptr[3]; // Array Pointer
    int **pptr; // Pointer to Pointer
    ptr[0] = arr; // &arr[0][0];
    ptr[1] = arr+1; // &arr[1][0];
    ptr[2] = arr+2; // &arr[2][0];
    pptr = ptr; // &ptr[0];

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

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

26	15	4
26	15	4
26	15	4



arr[0][2]

↓

*(arr+0)[2]

↓ main memory

((arr+0)+2)

↓

(500+0)

↓ row address

*(500)

↓ column address

(500+2)

↓

*(504)

↓

26

arr[1][1]

↓

*(arr+1)[1]

↓ main memory

((arr+1)+1)

↓

(500+1)

↓ row address

*(506)

↓ column address

(506+1)

↓

*(508)

↓

15

arr[2][0]

↓

*(arr+2)[0]

↓ main memory

((arr+2)+0)

↓

(500+2)

↓ row address

*(512)

↓ column add

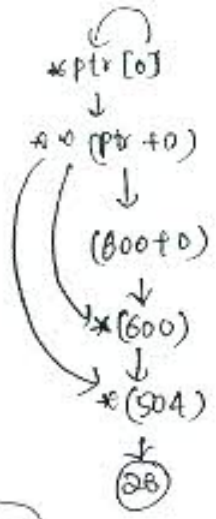
*(512+0)

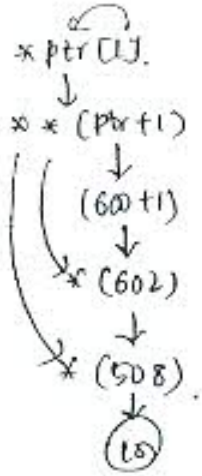
↓

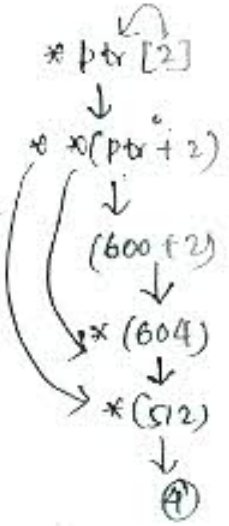
*(512)

↓

4







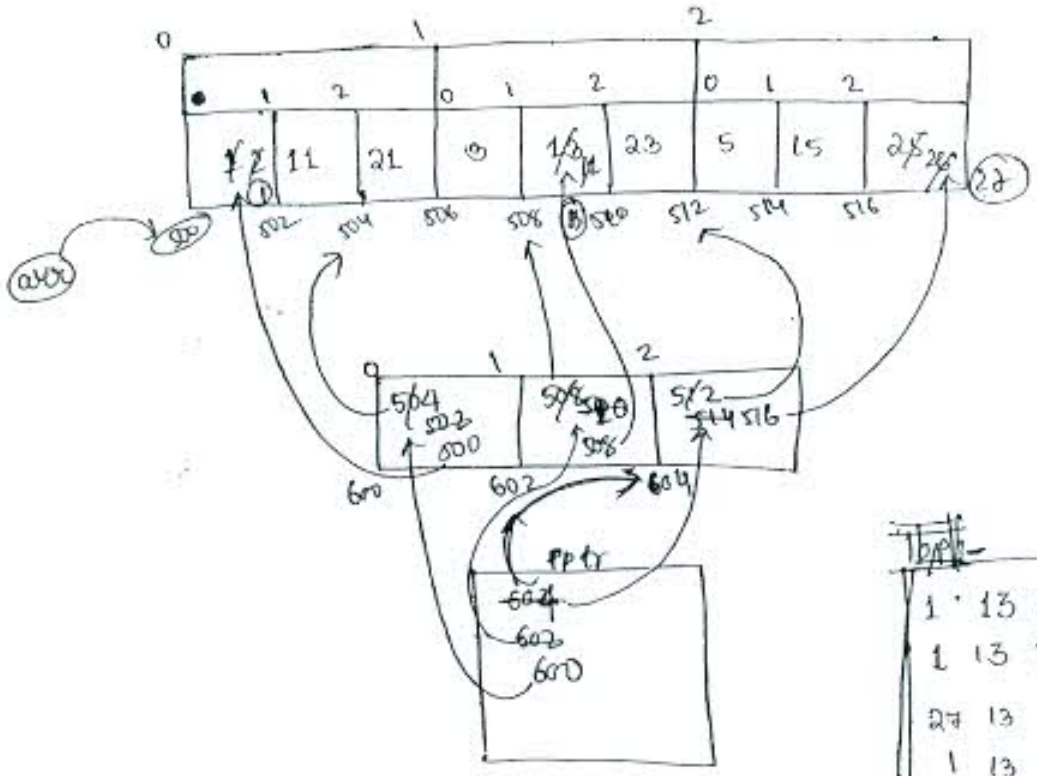
Notes:

- Collection of similar type of pointers in a single variable is called Array pointer
- whenever we require n' no of pointers of same datatype, then go for array pointer.
- Array pointer name always gives base address of array pointer.

10/11/17

Ex1)

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int arr[][3] = {
                    {1, 11, 21},
                    {3, 13, 23},
                    {5, 15, 25}
                   };
    int *ptr[3];
    int **pptr;
    clrscr();
    ptr[0] = arr[0]+2; //&arr[0][2]
    ptr[1] = arr[1]+1; //arr[0][1]
    ptr[2] = arr[2]+0; //arr[2][0]
    pptr = ptr+2; //&ptr[2]
    ++*pptr; ++ptr[2];
    ++*ptr[2];
    ++**pptr;

    --pptr;
    ++*pptr;
    --ptr[1];
    --**pptr;
    ++*ptr[1];
    --pptr;
    --*pptr;
    --ptr[0];
    ++*ptr[0];
    --**pptr;
    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;
}
```



1	13	27
1	13	27
27	13	1
1	13	27

arr[0][0]
↓
*(arr+0)[0]
↓
((arr+0)+0)
↓
*(500+0)
↓
*(500)
↓
1

arr[1][1]
↓
*(arr+1)[1]
↓
((arr+1)+1)
↓
(500+1)
↓
*(506)
↓
(506+1)
↓
*(508)
↓
13

arr[2][2]
↓
*(arr+2)[2]
↓
((arr+2)+2)
↓
(500+2)
↓
*(512)
↓
(512+2)
↓
*(516)
↓
20

*ptr[0]
↓
**(ptr+0)
↓
(600+0)
↓
*(600)
↓
*(500)
↓
1

*ptr[1]
↓
**(ptr+1)
↓
(600+1)
↓
*(602)
↓
*(508)
↓
13

*ptr[2]
↓
**(ptr+2)
↓
600+2
↓
*(604)
↓
*(516)
↓
20

**pptr
↓
*(604)
↓
*(516)
↓
20

**pptr++
↓
*pptr
↓
*(602)
↓
*(508)
↓
13

pptr: 604

L ← R

(*pptr)++;
↓
*pptr
↓
*(600)
↓
*(500)
↓
1

pptr: 602

**pptr
↓
*(600)
↓
*(500)
↓
1

L ← R

(*pptr)--
↓
*(602)
↓
*(508)
↓
13

pptr: 600

**pptr--
↓
*(604)
↓
*(516)
↓
20

pptr: 602

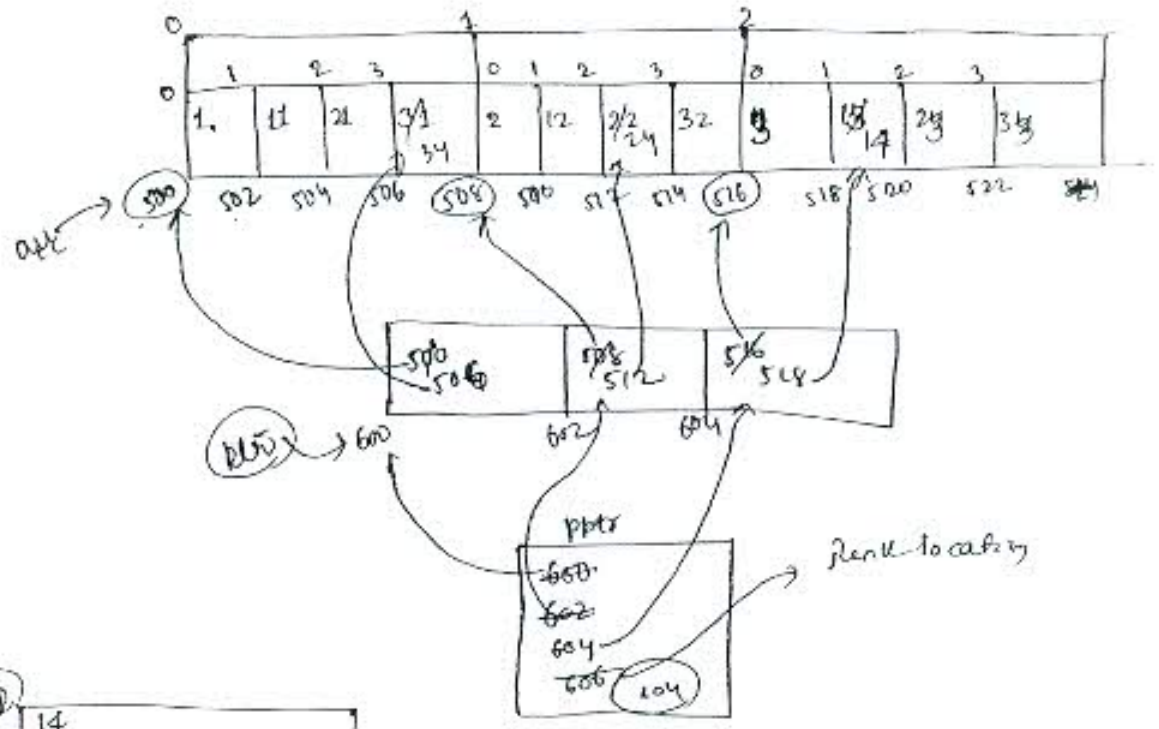
ex2

```
#include <conio.h>
#include <conio.h>
int main()
{
    int arr[][4] = {
```

- In 2D-array, when we are applying one subscript operator ([]), then it gives row address
- Row address + numeric value will provide column address.
- Arr name gives main memory, arr+1 address gives next main memory of the array i.e row address.

ex3

```
#
#
int main()
{
    int arr[][4] = {
                    {1, 11, 21, 31},
                    {2, 12, 22, 32},
                    {3, 13, 23, 33}
                   };
    int *ptr[3];
    int **pptr;
    int i, r, c;
    clrscr();
    ptr[0] = &arr[0][0]; // arr[0]; // arr;
    ptr[1] = &arr[1][0]; // arr[1]; // arr+1;
    ptr[2] = &arr[2][0]; // arr[2]; // arr+2;
    pptr = &ptr[0]; // ptr
    for(i=3; i>=1; i--)
    {
        *pptr += i; // *pptr = *ppt + i;
        **pptr += i; // **pptr = **ppt + i;
        ++pptr;
    }
    --pptr;
    printf("\n%d\t", **pptr);
    for(i=0; i<3; i++)
        printf("%d", **(ptr+i));
    // Printf("%d", *ptr[i]);
    for(r=0; r<3; i++)
    {
        printf("\n");
        for(c=0; i<4; c++)
            Printf("%3d", arr[r][c]);
        // Printf("%3d", *(*(arr+0)+c));
```



2D Array & function =>

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

Output:
Enter 3 * 4 values:
1 2 3 4
5 6 7 8
9 10 11 12
Sum of list: 78

4/11/21

3D Arrays :-

- In 3D array, elements are arranged in blocks, rows, & columns format.
- When we are working with 3D array, we required to mention 3 subscript operators, which indicates blocksize, rowsize and column-size.
- The main memory of 3D array is blocks, sub main memory is rows, elements are available in columns.
- On 3D array, when we are referring one subscript operator, then it gives block main, two subscript operators give row name & three subscript operator will provide an element.
- On 3D array, arr name gives baseaddress of the array, i.e. main memory, (arr+1) gives next main memory of 3D array.

- Syntax:- Datatype arr name [Bsize][Rsize][Csize];

properties of 3D array :-

① int arr[2][3][4];

size → 2*3*4
sizeof(arr) → 48B (2*3*4 = 24*2 = 48B)

② int arr[3][2][3];

size → 3*2*3
sizeof(arr) → 36B

3 blocks
each block 2 rows
each row 3 columns
is int variable.

③ int arr[2][3][2];

12 integer operator

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

arr[1][0][1] → 8 | arr[2][1][0] → 9 | arr[2][1][1] → 10 | arr[1][2][0] → 11 | arr[1][2][1] → 12

④ int arr[2][2][2] = {10, 20, 30, 40, 50, 60, 70, 80};

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

- By using the above initialization process we require to initialize all elements in sequence only.

⑤
```
int arr[3][3][4] = {
                    {
                      {10,20},
                      {30},
                      {40,50,60}
                    },
                    {
                      {7,8};
                      {9},
                    }
                    {
                      {70,80}
                    }
                   };
```

- In initialization of 3D arrays if specific no. of elements are not initialized, then rest of all values will be initialize with 0's.

⑥ int arr[][][]; error

⑦ int arr[][][5]; error

⑧ int arr[][2][5]; error.

- In declaration of 3D array, it is mandatory to specify block size, row size, and column size, or else it gives an error

⑨
```
int arr[][][] = {
                  {
                    {10,20},
                    {30},
                  },
                  {
                    {40,50,60},
                    {70}
                  }
```

(10) int arr[][][3] = {
{
{10},
{20,30},
},
{
{40,50,60},
{70}
}
}; // valid.

(11) int arr[0][2][3] = {
{
{10, 20, 30},
{40},
},
{
{50},
{60}
}
}; // valid.

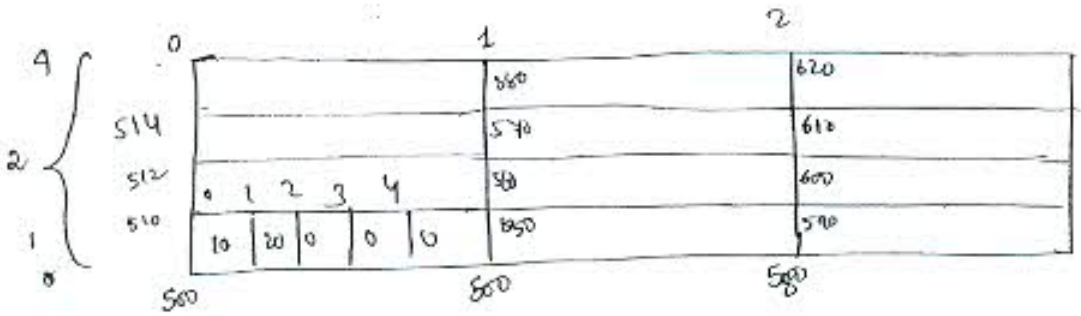
- On initialization of 3D array, specifying the block size is optional; but row & column sizes are mandatory.

4D Array :-

- In 4D-array, elements are arranged in sets, blocks, rows & columns format.
- When we are working with 4D array, we require to specify 4 subscript operators, which indicates set size, block size, row size, & column size.
- The main memory of 4D array is sets, sub memory is blocks, next to main memory is rows and elements is columns.
- On 4D array, when we are referring one subscript operator, then it gives set name, two subscript operators will give block name, three subscript operators will give row name, and 4 subscript operators gives element.
- On 4D array, arr name provides main memory address, i.e. first set base address, (arr+1) provides next set base address.
- (Syntax) Datatype arr[SetSize][BSize][RSize][CSize];
- On declaration of 4D array, it's mandatory to specify, set size, block size, row size & column sizes, whereas in initialization specifying the first size is optional.

Memory Mgmt in Multidimensional Array or 3D Array :-

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



① arr → 500 ① arr+1 → 540 ② arr+2 → 580

② arr[0] → 510 arr[1] = 540 arr[2] → 580

③ arr[0]+1 → 512 arr[1]+1 → 550 arr[2]+2 → 600

④ arr[0][0] → 500 arr[1][0] → 540 arr[2][0] → 580

⑤ arr[0][0]+1 → 502 arr[0][1]+1 → 552 arr[2][2]+2 → 604

arr[0][0][0] = 10

arr[0][0][0]+1 = 11

&arr[0][0][0]+1 = 502

*(&arr[0][0][0]+1) = 20

ex's) Enter mats/ outputs : 2

Enter no of rows :

#

#

```
int main()
{
    int m1[10][10];
    int m2[10][10];
    int m3[10][10];
    int r, c, r1, c1, r2, c2, r3, c3, t;
    clrscr();
    printf("\n Enter mats details :");
    printf("\n Enter no of rows :");
    scanf("%d", &r1);
    printf("\n Enter no of columns :");
    scanf("%d", &c1);
```

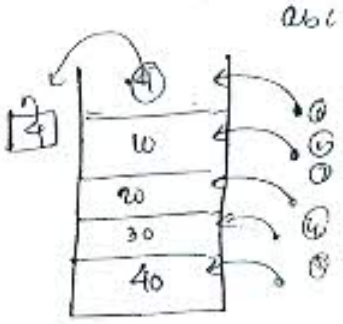
```
printf("\n Enter MAT2 details :");
printf("\n Enter no of rows :");
scanf("%d", &r2);
printf("\n Enter no of columns ");
scanf("%d", &c2);
printf("\n
if(c1 != r2)
{ printf("\n mat mul not possible ...");
getch();
return 0;
}
printf("\n Enter mat1 %d * %d elements ", r1, c1);
for(r=0; r<r1; r++)
for(c=0; c<c1; c++)
scanf("%d", &m1[r][c]);
printf("\n Enter mat2 %d * %d elements ", r2, c2);
for(i=0; i<r2; r++)
for(c=0; c<c2; c++)
scanf("%d", &m2[r][c]);
r3=r1;
c3=c2;
printf("\n the result ");
for(i=0; i<r3; r++)
{ 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]);
}
printf("\n");
}
getch();
return 0;
}
```

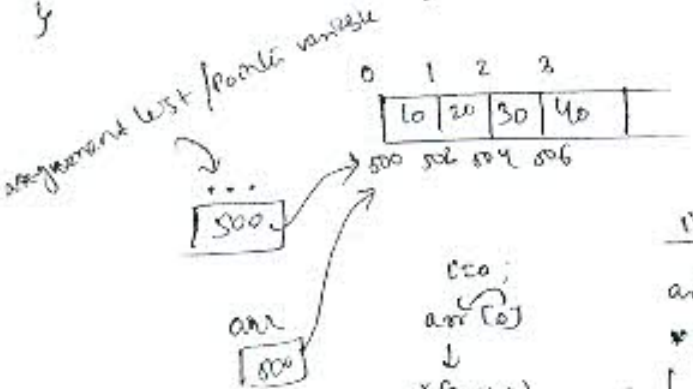
- C' prog. language doesn't support function overloading concept.
- Function overloading is a OOPL feature, which comes with the help of polymorphism.
- C' prog. language doesn't support function overloading, but similar kind of behaviour is possible by using argument list concept.
- printf(), scanf() related functions accept any no of arguments, because those are implemented by using argument list concept.

```
#include<stdio.h>
#include<conio.h>
void abc(int n,...)
{
    int i; int *arr=...;
    printf("in data is abc:");
    for(i=0;i<n;i++)
        printf("%d",arr[i]);
}
int main()
{
    abc(0);
    abc(1,10);
    abc(2,10,20);
    abc(3,10,20,30);
    abc(4,10,20,30,40);
}
```

1 ← R

abc(4, 10, 20, 30, 40);





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

i=1
arr[1]
↓
*(arr+1)
↓
(500+1)
↓
*(502)
↓
20

i=2
arr[2]
↓
*(arr+2)
↓
(500+2)
↓
*(504)
↓
30

i=3
arr[3]
↓
*(arr+3)
↓
(500+3)
↓
*(506)
↓
40

```
data in abc:
data in abc: 10
data in abc: 10 20
data in abc: 10 20 30
data in abc: 10 20 30 40
```

2/11/12

Read Only Variables :-

- Constant variables (Const) are called read only variable.
- Readonly is a property of a variable, which doesn't allow to modify the value of a variable by using variable name.
- By using 'const' modifier, we can apply readonly property to a variable.

ex)
```
#include<stdio.h>
int main()
{
    int a=10;
    ++a;
    printf("a=%d", a);
    return 0;
}
```
a=11

ex)
```
#include<stdio.h>
int main()
{
    const int a=10;
    ++a; //error
    Printf("a=%d",a); //e
    return 0;
}
```
(o/p) error: can't modify constant object

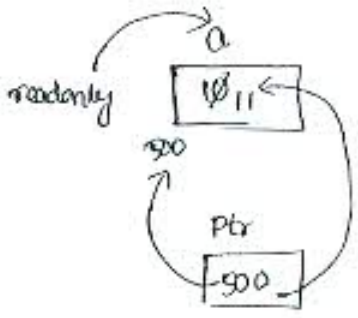
⇒

① const int a=3; 'a' is a variable of type const int

② int const a=3; 'a' is a variable of type int const.

If the variable is 'const int' or 'int const', both contain read only property, so it's not possible to change the value.

ex)
```
#include<stdio.h>
int main()
{
    int const a=10;
    int *ptr;
    ptr=&a;
    //++a; Error
    ++*ptr;
    printf("a=%d", a);
    return 0;
}
```
o/p a=11



readonly
++a; error

++*ptr;
int *ptr;
Normal type

↳ `int * ptr;` 'ptr' is called pointer to integer (int).

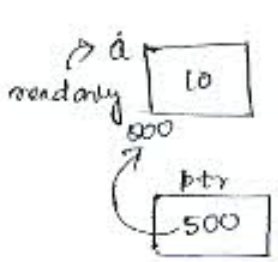
- By using pointer to int variable, it is possible to change normal or readonly variable data.
- In implementation, when we are working which read only variables, then always its recommendent to go for "pointer to const int" or "pointer to int const"

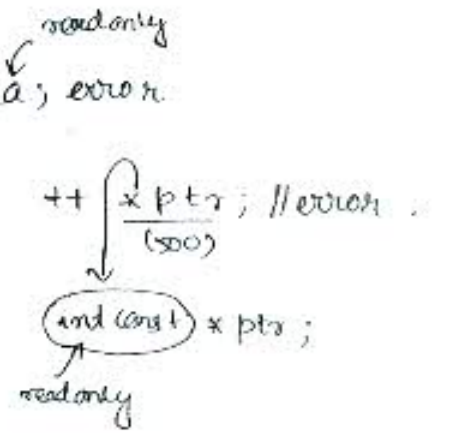
↳ `const int *ptr;` pointer to const integer
`int const *ptr;` pointer to integer const

- When we are working with 'pointer to const int' or 'pointer to int const', then normal variable data and readonly variable data also not possible to change.

(ex3) #

```
int main()
{
    const int a = 10;
    int const *ptr;
    ptr = &a;
    ++a; error
    ++*ptr; Error
    printf("a = %d", a);
    return 0;
}
```





(ex4) #

```
int main()
{
    int a = 10;
    const int *ptr;
    ptr = &a;
    ++a; // yes
    ++*ptr; //error
    printf("a = %d", a);
    return 0;
}
```

(o/p) Error

(ex5) #

```
int main()
{
const int arr[2] = {10, 20};
```

(Q5) #
```
int main()
{
    const int arr[2] = {10, 20};
    int *ptr;
    ptr = &arr[0];
    ++ptr;
    ++*ptr;
    --ptr;
    --*ptr;
    printf("%d %d", arr[0], arr[1]);
    return 0;
}
```
(op) 9 21

(Q6) #
```
int main()
{
    int const arr[2] = {5, 10};
    const int *ptr;
    ptr = &arr[0];
    ++ptr ; // yes
    ++*ptr ; // error
    --ptr ; // yes
    --*ptr ; // error
    printf("%d %d", arr[0], arr[1]);
}
```
(op) error

- when we are working with 'pointer to const int' or 'pointer to int const', then value modification only restricted, but pointer can be changed.
- whenever we require to restrict the modification of the pointer, then go for 'const pointer'.

i.e. `int * const ptr ;` 'ptr' is called const pointer to integer.

when we are working with 'const pointer', it should be initialized because address can't be reassigned after constructing pointer.

Ex7 #

```
int main()
{
    const int arr[2] = {1, 2};
    int * const ptr = &arr[1];
    --ptr; // error
    ++*ptr; // yes
    ++ptr; // error
    --*ptr; // yes
    printf("%d %d", arr[0], arr[1]);
}
```

o/p:- error

- When we are working with const pointer, then pointer modification is restricted, but value modification is allowed.
- Whenever we required to restrict the modification of pointer & value, then go for "const pointer to const int" or "const pointer to int const".

↳ `const int * const ptr;` 'ptr' is called const pointer to const integer

↳ `int const * const ptr;` 'ptr' is called const pointer to int const.

When we are working with const pointer to int const or const pointer to const integer, then value and address both are restricted from modification.

Ex8 #

```
int main()
{
    int const 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]);
}
```