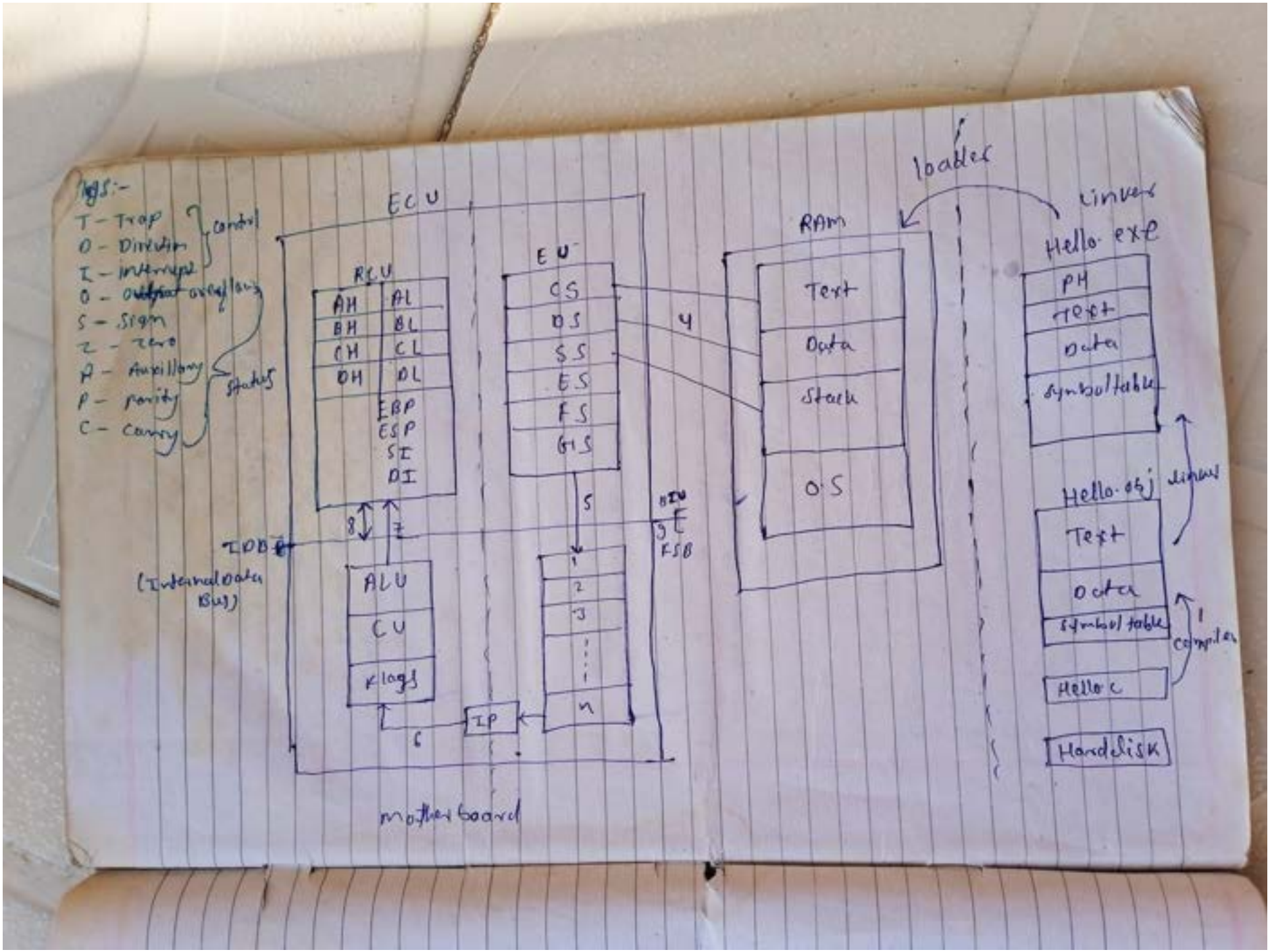
ECU - Execution control unit
RCU - Register control unit
EU - Execution unit
EBP - Base pointer
ESP - stack pointer
SI - Source index
DI - Destination index
CS - Code Segment
DS - Data Segment
SS - Stack Segment
ES - External Segment

~~EAX EAX~~ H - High L - low
EAX - Accumulator Register
EBX - Base register
ECX - Count register
EDX - Data register
ALU - Arithmatical Logic unit
CU - Control unit
IP - Instruction pointer
FSB - front Side Bus
BIU - Bus interface unit
IOB - internal data Bus

1) editer
2) preprocessor .i intermediate file
3) Compiler .obj object file
4) Assembler .Asm assembly file / .s
5) linker .exe
6) loader load on RAM


Flags:-
T - Trap
O - Direction
I - Interrupt
S - Sign
Z - zero
A - Auxillary
P - parity
C - Carry
control
status
ECU
RCU
AH
AL
BH
BL
CH
CL
DH
DL
EBP
ESP
SI
DI
EU
CS
DS
SS
ES
FS
GS
ALU
CU
Flags
IP
IDB
(Internal Data Bus)
EDB
FSB
motherboard
loader
RAM
Text
Data
Stack
O.S
Hello.exe
PH
Text
Data
Symbol table
Hello.obj
Text
Data
Symbol table
Hello.c
HardDisk
linker
compiler

Computer Architecture

tool chain:- it is a set of programming tools which are used in chain to convert our program into executable file.

editor:- editor is a program which is used to write something on it and which is used to edit that written things eg: wordpad, notepad

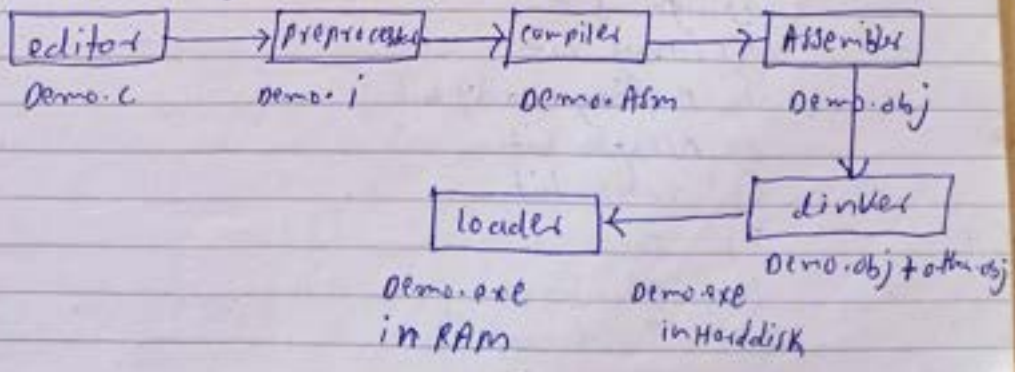
Compiler:- compiler is a program which is used to convert high level programing language (C) in to machine dependent programming language.
eg: GNU c compiler, Turbo C, Borland, visual studio

Assembler:- it is a program which is used to convert machine dependent language into machine understandable language (binary) eg: Asm, MASM, TASM

Linker: it is a program which is used to link an object file with other object file

loader:- it is a part of operating system which is used to load executable file from harddisk into RAM

Diagram of x86 tool chain



* Types of CPU registers -

1) general purpose registers -
   i) EAX (Accumulator register)
   ii) EBX (Base register)
   iii) ECX (Count register)
   iv) EDX (Data register)

2) Index register
   1) ESI (Source index)
   2) EDI (Destination index)

3) Pointer register
   i) ESP (Stack pointer)
   ii) EBP (Base pointer)
   iii) IP (Instruction pointer)

4) Segment register
   i) CS (Code Segment
   ii) DS (Data Segment)
   iii) SS (Stack Segment)
   iv) ES (Extra Segment)
   v) FS, GS (No full form)

5) flag register:-

1) Parity bit
2) Sign bit
3) Carry bit
4) Auxillary carry bit
5) overflow bit
6) zero bit
7) Trap
8) Direction
9) Interrupt

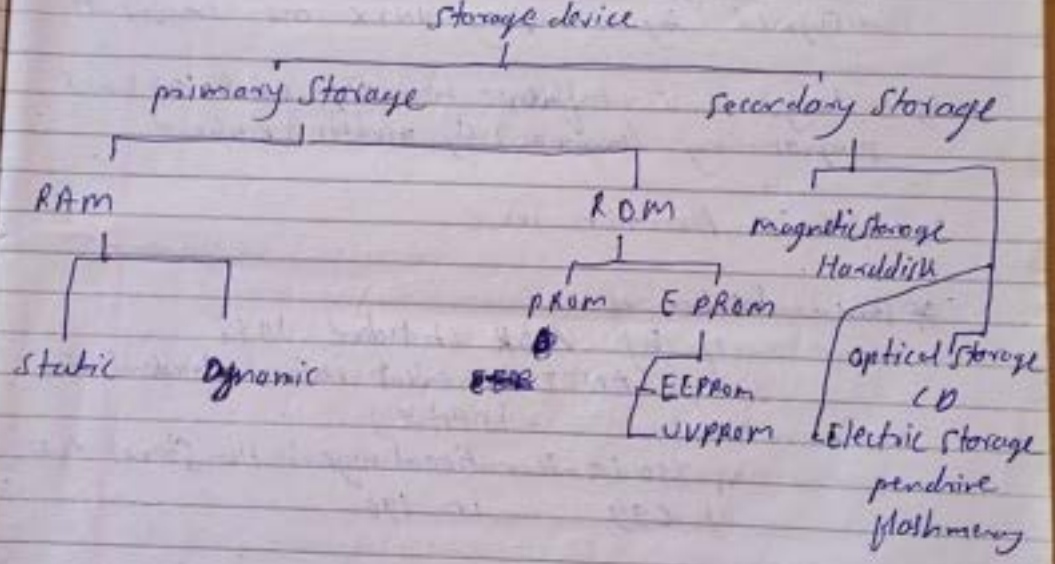
RAM

Stack

* Components of Computer

1) microprocessor :- it is used to take decisions and action
2) multi co-processor :- used to perform Arithmetic computation
3) Storage device :- used to store the data
4) input devices :- used to accept the input
   eg: keyboard, mouse
5) output devices :- used to display the output
   printer, monitor

* Storage devices in computer :-



* programming language –
It is a language which is used to communicate with machine (computer)

Type of programing language

- procedural → C
- object oriented → C++
- virtual machine → java

C programing language:

1) it is developed and deligned by Dennis Ritchie Brian Kernighan in AT&T Bell it is deligned by developing UNIX O.S in 1969-1973

2) C language is influence from BCPL (Basic combined programming language) by martin Richard.

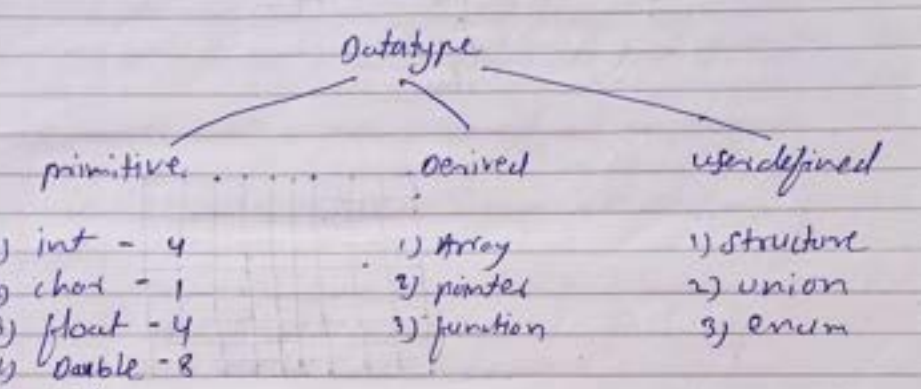
3) unix is rewritten in C.

* Standardization of C
  1) By K&R standard 1978
  2) ANSI (American National standardization institute)
  3) ISO (International organization standardization)
  4) C99 in 1991

* Characteristics of C language
  1) high level programming language
  2) procedural programming language
  3) general purpose —11—
  4) block structured —11—
  5) Case sensitive
  6) compiled
  7) platform dependent

8) free-flow programing language
9) it uses top down approach
10) it is standardize language

* Datatype in "C"



* Datatype qualifiers

it is used to provide new quality for our Datatype

1) const (non modifiable / value)
2) volatile (Compiler optimization is removed)

* Datatype modifiers:-

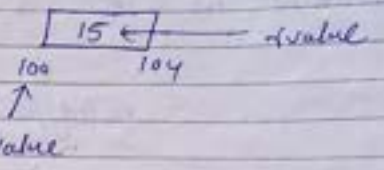
it is used to modify the range & size of datatype

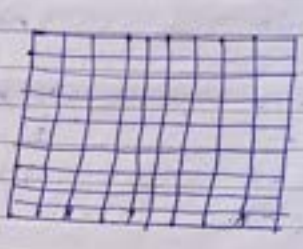
1) Signed
2) unSigned
3) short
4) long

* lvalue & Rvalue of variable

1) lvalue :- it is the location of that variable
2) rvalue :- it is the object at that location

```
int i = 15
```





* Declaration of variable:

it is a location at which the size of variable is decided by the compiler but memory is not allocated yet

* Defination of variable

it is a location at which actual memory gets allocated.

18 7 15

* Storage classes :-

1) This concept is language independent concept means it is not specific to any programing language

2) The concept of storage class can be used in C & C++ also without any change in the concept

3) The concept of storage class is applicable for the variables which are used in our program or the functions which are defined in a program

4) There are 4 mostly storage classes in C & C++ are

i) auto
ii) extern
iii) static
iv) register

5) if we want to understand the concept of storage class then we have to consider 5 different characteristics such as

* 1) i) memory region in which our variable is allocated
   ii) there are multiple memory option such as
      i) stack section
      ii) data section (BSS or non BSS)
      iii) static segment
      iv) CPU registers

According to the type of storage class memory can be allocated in any of the above memory locations

2) Default value :-

it indicates by default initialization is performed by the memory manager if that variable not initialized

Default value of variable can be
i) it may contains some garbage value
ii) it may contains 0 or 0.0 or '\0'

3) Scope:-
Scope of a variable is depend on the region in which our variable is accessible
Scope of a variable can be, block scope
2) file scope 3) program scope.

4) lifetime:-
lifetime of variable is the gap in between memory allocation and deallocation of that variable

lifetime of a variable can be, 1) local lifetime
2) Global lifetime

5.) linkage:-
linkage of variable indicates the region in which our variable can be linked

There are 3 types of linkage such as
1) no linkage
2) internal linkage
3) external linkage

1) auto storage class:-

i) memory :- Stack
ii) default value - garbage
iii) Scope - Block scope or function scope
iv) lifetime - local lifetime
v) linkage - no linkage

a) Any variable which is declared or defined inside any block or function then the storage class of that variable is considered as auto.
b) for local variables auto is the default storage class.
c) as it is default storage class use of auto keyword is optional.
d) function argument, local variable as a default auto storage class
e) to avoid the drawback of garbage values auto variable should be initialized explicitly

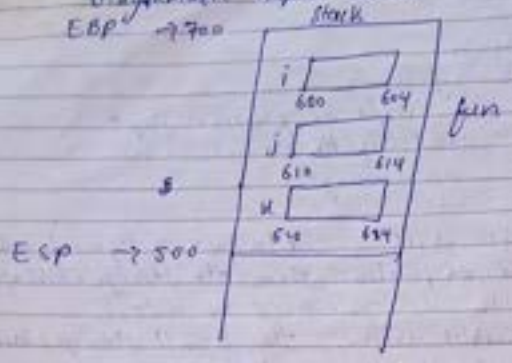
eg:
```
void fun c int i )
{
    int j = 11;
    int k;
}
```

in above function i, j, k considered as auto variables.
This variable also can be defined as
```
auto int j = 11;
auto int k;
```

in above syntax auto keyword is not use because it is optional.

memory for this auto variable is allocated inside the stack frame of a function in which it is defined.

Diagramatic representation:



in above diagram stack frame of function fun() is displayed which contains the memory for all local variables.

According to the standard processor architecture our stack grows downward.

The starting address of stack frame is stored in EBP and end address is stored in ESP

2) register Storage class:-
   i) memory :- cpu registers
   ii) default value = garbage
   iii) Scope :- block scope or function scope
   iv) lifetime - local lifetime
   v) linkage - no linkage

a) The concept of the register storage class is almost same auto storage class.

b) the only difference betn auto and register storage class is the memory allocation region.

e) if any of the variable is frequently required in an program then that variable should be declared as register variable

d) when we try to access any variable the value of that variable gets copied from RAM to CPU register.

e) if the memory for variable is already allocated inside the register then time which is required to copy a value from RAM to CPU register gets minimized.

f) Register Storage class is considered as a request because memory gets allocated inside CPU registers.

g) if there is no CPU register available then the request for register Storage class gets denied and then that variable gets default Storage class as auto.

h) whether the CPU register gets allocated or not is decided at runtime.

i) there is no such direct technique by using we can predict the register Storage class is applied or not.

j) if we create the variable as a register variable then we can not fetch the address of that variable

eg: void fun()
{
register int i=0;
for(i=0; i<1000; i++)
{
printf("%d", i);
}
}

in above example i variable which is loop counter is considered as a register variable.

As this loop counter required 1000 times in our program due to which instead of accessing from ram we can store it into cpu register.

eg: register int i = 11; // allowed
register int arr[10]; // not allowed
register float f = 3.14f; // not allowed
register double d = 3.14; // not allowed
register char ch = 'a'; // allowed

According to above syntax we can conclude that register storage class can be applied only for "integral" datatypes.

means it is only applicable for char and int but it is not applicable for float and double.

we can not create derived datatype as array which contains register variables.

eg: register int i = 11; //
int main()
{
// code
}

the above syntax is not a compiletime error but it is a logical problem.

if we declare the variable as global variable then memory for that variable gets allocated throughout the program and it is illogical to acquire

the CPU register throughout the program.

* extern storage class :-
  i) memory - data
  ii) default value - 0 or \0 or 0.0
  iii) scope - throughout program scope
  iv) lifetime - global (throughout program)
  v) linkage - external linkage

If we declare or define any variable as a global variable then default storage class gets allocated for that variable is extern.

The concept extern storage class is generally used if our project contains multiple program files.

eg: // first.c

```
int i=11;    // variable defination
void fun()   // function defination
{
    int x;
}
```

// second.c

```
int main()
{
    extern int i; // variable declaration
    printf("%d", i); //
    extern void fun(); // function declaration
    fun(); // call
}
```

in above project there are 2 .c files as first.c and second.c.

if we want to write the project in multiple files then main function should be in one file only and in other files we can write anything.

to compile this multifile program we have to use command as

gcc first.c second.c ↵

if we want access the contents from some other files then we have to declare that variable or function as a extern in our file

if it is not declared by using extern keyword then compiler generates error

by using the extern keyword our compiler search externally for that variable.

as like above application we can write multiple files in an project & we can access the contents one file into other file by using extern storage class.

in this program variable a is local variable to the function fun due to which we can not access that variable by using extern keyword.

4) Static Storage class:-

i) memory - data section of static segment
ii) default value - 0, 0.0 & '\0'
iii) scope - file scope
iv) lifetime - global lifetime (throughout program)
v) linkage - internal linkage.

C Startup routine

4/7/15

1) ~~memory is lost~~ In case of auto storage class memory is allocated for the variable inside the stack frame of that function.

2) when the function terminates memory gets automatically free due to which we can not preserve values of that variables.

3) if we want preserve that values then it should be allocated outside the stack frame.

4) for this purpose static storage class is used which is used to preserve the values of local variables across the function calls

5)
```
void fun ()
{
    int i=10;
    i++;
    if (i<15)
    {
        fun();
    }
}
int main ()
{
    fun ();
    return 0;
}
```

6) in above program inside fun function there is one local variable named as i which is initialize to 10.

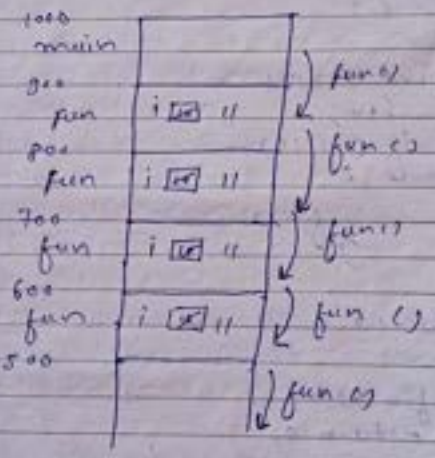
7) this function gets called recursively from that function again and again

8) for every function call separate stack frame gets created which contains value of i 10.

9 due to which the condition which is written inside that function will not get executed successfully because memory for local variable i gets allocated for every function call.

(a) due to this drawback our program abnormally terminates when memory for stack gets finished it generates an runtime exception as stack overflow

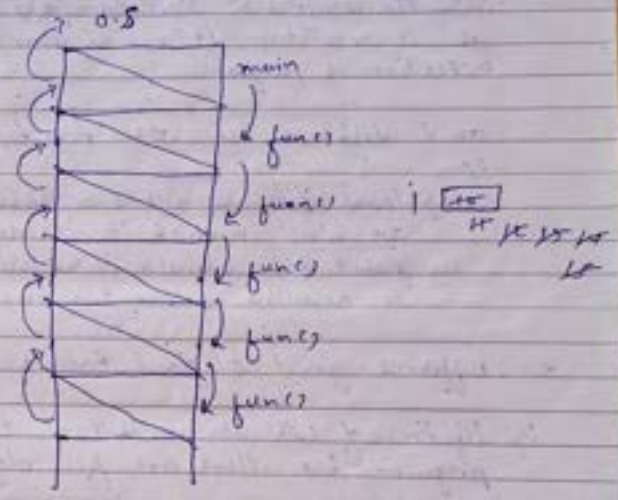
(1) we can avoid this problem by declaring that i variable as static.



(2) to avoid this problem our modified code should be

```
void fun()
{
    static int i = 10;
    i++;
```

```
if (i<5)
{
    func();
}
```



* There are two types of Static variables as
  1) local Static
  2) global Static

eg:
```
Static int global=10;
int x=10;

void fun()
{
    Static int local=10;
}
int main()
{
    fun();
    return 0;
}
```

in above program local is considered static local variable
global is considered as static global variable

the similarity betn these two types of variable is it maintains its values through out the execution of program.

and diffence betn these two types of variable is

1) local static variable is accessible only inside the block in which it is return.
2) global static variable is a variable which is accessible inside file in which it is written

* Differnce global & global static variable

1) lifetime of both the variables is throughout the program but there are few diffrences such as

a) global variable can be accessible outside the file by using extern keyword.
b) global static variable can not be accessible outside the file because it had an internal linkage.

** Symbol table:-

1) this table maintains metadates about the datas of our program.

2) this variable table maintain all the identifiers which are used in our program this table gets created by the compiler as well as operating system.

a) all the information which are required for identifiers is maintain in symbol table

Name of identifier	Address (virtual)	value	from	To	Maintenance

a) this table contains the entries list of name of the identifier used in the program.

b) the address which is allocated by the compiler which is virtual address

c) the value of the variable if it is initialized

d) the starting address of variable is accessible

e) end address till which we can access our variable

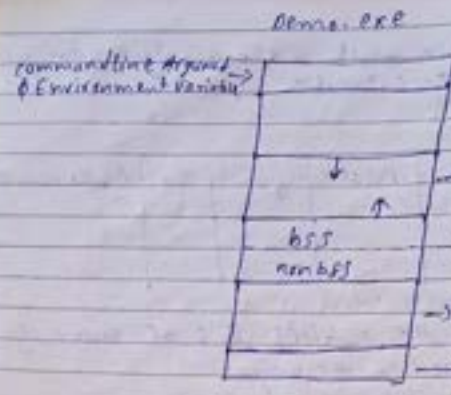
f) If variable has a reference variable which is used in C++ then it's name stored in another name column.

* layout of program

Below diagram explains the layout of our executable file.

1) The topmost part of our layout contains the memory for environment variables & commandline argument

2) Environment Variable are such variable whose values are by running the program.



3) after the commandline arguments memory memory for stack gets allocated then executable file gets loaded into memory.

4) Size of stack gets fixed by the compiler.

5) below the memory of stack there is a potential gap which is used at a runtime by the stack or by data section.

6) if our stack use all the memory of potential gap then there is stack overflow error.

7) and if our data section or heap uses all the memory of potential gap then it is "there is no enough memory error".

8) Data Section is divided into 2 parts bss and non bss which is used to store values of global variable and the memory for dynamic memory allocation technique.

environment variable - path, command line argument, if else,

g) kpt folder contains all the instruction in binary format which are required at runtime.

25/7/18

* Arrays in C or C++

* Array - Array is a linear data structure which store homogenous elements
Array is considered as an derived datatype

* datatype :- Datatype indicates the memory allocation which are used store some values and the operations which are applied on that values.

* Derived datatype :- Derived datatype is such datatype which is derived from primitive datatype or user-defined datatype

eg. - Array, pointer, function

* Datastructures - Datastructure is a way of storing and representing the data in some appropriate manner.
There are 2 types of datastructures
a) linear Data structure
b) Nonlinear Data structure.

3) in case of linear data structure data accessing is performed in linear fashion where as in nonlinear data linear structured data accessing is performed in nonlinear fashion

eg. linear - Array, Stack, linkedList, Queue
Nonlinear - Tree, hashtable

- if we want to store multiple elements of same datatype then we have to create array because array stores homogeneous elements.

- Homogeneous means every element is of same datatype

eg: if i want to store marks of 5 students in mathematics then there are 2 programatical approaches of

approach 1: we can create 5 diffent integer variable such as

```
int marks1;
int marks2;
int marks3;
int marks4;
int marks5;
```

approach 2: instead of creating 5 diffent variables we can create array of 5 integer as

```
int marks[5];
```

- Both the approaches are working programatically but in first approach as a programmer we have to remember five different name which is difficult.

- To avoid this problem we can create an array in which we have to remember only one name.

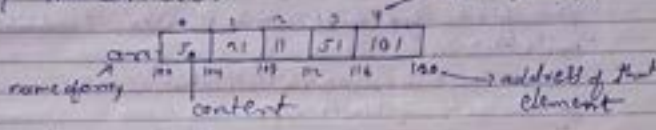
- In above 2 approaches if we create N different variables then there are N diffrent symbol table entries

But if we create array of n elements then there is only one Symbol table entry

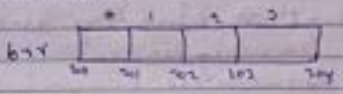
- there are 2 types of array of
  1) Single dimensional array
  2) multi dimensional array

* diagramatic representation of array

1) int arr[5];



2) char brr[4];



3) if we want to access the element of array then we have to use 2 things of
   1) Name of the array
   2) index of that element

4) Accordiry to above diagram if we want to access element 51 then Syntax should be

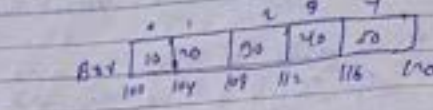
```
arr[3];
```

Similarly if i want to initialize character 'A' in 2nd index of array brr then Syntax should be

```
brr[2] = 'A';
```

5) there are multiple ways in which we can initialize the content of array

a) int Arr[5] = { 10, 20, 30, 40, 50 };



b) int Arr[5];
Arr[0]=10;
Arr[1]=20;
Arr[2]=30;
Arr[3]=40;
Arr[4]=50;

The first way initialization is called as member initialization list
The 2nd way of initialization is called as member by member initialization technique

6) if we know all the elements of array then we can use member initialization list but if we want to initialize only some of the elements then we have to use member by member initialization technique.

7) eg: int Arr[5];
Arr[1] = 20;
Arr[3] = 40;

in above syntax we want to initialize only 2 members due to which member by member initialization technique is used.

8) if array is uninitialized then the contents in that array depend on storage class of that array

eg: int arr[4]; // extern storage class
int main ()
{

```
    int Brr[4];   // auto storage class
    return 0;
}
```

9) As both the arrays are uninitialized contents of array arr is zero and content Brr[] is garbage like auto storage class

10) if first element of array is initialize by using member initialization list then all the remaining elements of that array gets its default value of 0, '\0', or 0.0f

```
int arr[5] = {5};
```

This above concept is not applicable if we are using member by member initialization technique

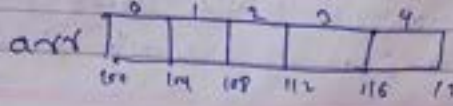
11) in case of member initialization technique we can not initialize random element of

```
int arr[5] = {1, 4, 3, 7}; //error
```

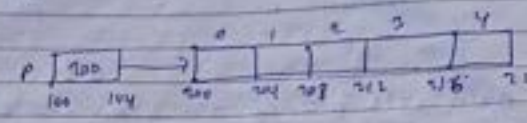
12) There are 2 ways in which we can allocate the memory for array

i) Static memory allocation.
ii) dynamic memory allocation

Syn `int arr[5];` // Static allocation



// dynamic memory allocation
int *p = (int *) malloc (5 * sizeof (int));



13) In above call memory for array is allocated on heap and arr pointer p holds the Base address of that allocated memory.

14) incase of dynamic memory allocation all the values are by default set to zero.

15) incase of Static memory allocation Size of array should be compile time integral constant otherwise compiler generates an error.

eg: int arr[5]; // allowed
a) int no = 5;
int arr[no]; // not allowed
it is not allowed because no is variable.

3) const int no = 5;
int arr[no]; // error
it is allowed on some of the platforms and some of the compilers

4) #define MAX 5
int arr[MAX]; // allowed
it is allowed because after preprocessing arr Syntax should be
int arr[5];

5) int arr[]; // not allowed
6) int arr[3+2-1]; // allowed

while specifying the size of array we can use any expression that will get evaluated to compiletime integral constant.

16) Specifying the size of array is optional if we are using member initialization list

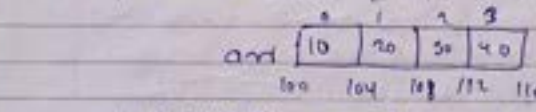
```
int arr[] = { 1,2,3,4,5 }; //allowed
```

in this syntax compiler calculates no of elements which are initialize and depend on which it decides the size of array.

17) Name of array is considered as Base address of that array

As Base address is the only entity which is available to the compiler to access every element of array due to which we can not modify base address of array.

```
int arr[4] = { 10,20,30,40 };
```




```
printf("%d", arr);  // 100
arr++;  // error
```

18) after understanding concept of array we have to read the Syntax of array properly as.

```
int arr[5];
```

arr is one dimensional array which contains 5 elements each element is of type integer.

int brr[3][4];

brr is a dimensional array
which contain 3 one dimensional arrays
each one dimensional array contains 4 elements
each element is of type integer.

int crr[3][2][4];

crr is 3 dimensional array
which contain 3 two dimensional array
each 2 dimensional array contains 2 one dimensional array
each one dimensional array contains 4 elements and each element is of type integer.

"pointer in C & C++"

1) pointer is a language independent concept means it exactly same in C & C++.

2) pointer is considered as a system programming primitives in C & C++.

3) pointer is a derived datatype which can be derived from any primitive datatype, any user defined datatype, any derived datatype.

4) pointer:- pointer is a variable which stores address & the capability of that variable is to fetch the content whose address is stored in that variable.

eg: int no = 11;

* operators used in pointers:-

'&' :- & is a unary operator which is called as address of operator which is used to retrive address of any type of variable.

'&' operator is used to retrive the virtual address of the variable.

'*' : it is unary operator which is called as content of operator. this operator is used to fetch the contents whose address is stored in pointed variable.

Some important point about pointer

1) As pointer is a variable which store address and address is always integer due to which sizeof any pointer is always 4 byte
2) means size of pointer is 4 byte irrespective of type of pointer
3) pointer is also considered as a normal variable just the extra capability is added to our pointer with is used to fetch the contents where address is stored in pointer

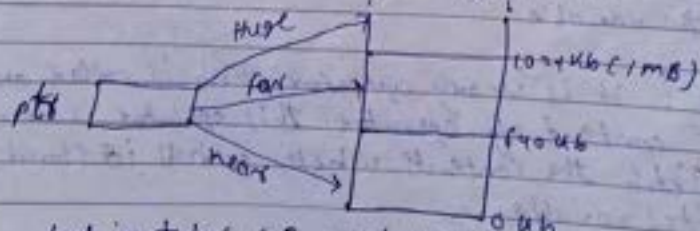
int 4

* Types of pointer:-

1) In real mode operating system there are 3 types of pointers as
   1) near pointer
   2) far pointer
   3) huge pointer

2) Real mode o.s are such operating system which can access only one 1mb of memory from RAM
3) In Real mode o.s only single process run at a time

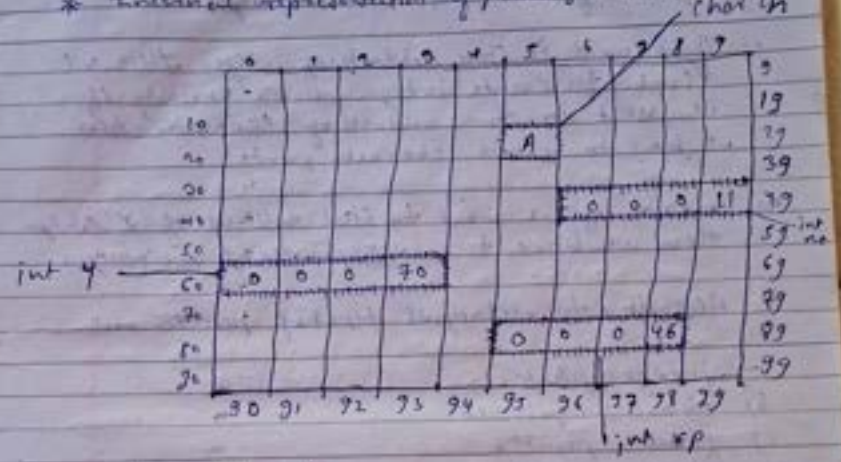
0 - 640kb - near pointer
640kb - 1024 kb - far pointer
1024kb - above - huge pointer



but in todays o.s which works in protected mode there is no such type of pointer that o.s uses the concept of flat addressing scheme where there is no partition of memory

due to which today's compiler and o.s there is no need concept type of pointer.

* Internal representation of pointer in RAM




```
char ch = 'A';
int y = 70;
int no = 11;
int *p = &no;
```

above diagram is 100 byte portion of RAM every small block indicate 1 byte memory when we declare any of the variable in our program memory gets allocated in this manner

According to this diagram our pointer gets acquired 4 byte memory like a normal integer variable

This pointer stores only address but by looking at the contents of the pointer we can not predict it is pointer or normal variable

Some important point about pointer

1) As pointer is a variable which store address and address is always integer due to which sizeof any pointer is always 4 byte
2) means size of pointer is 4 byte irrespective of type of pointer
3) pointer is also considered as a normal variable just the extra capability is added to our pointer with is used to fetch the contents where address is stored in pointer

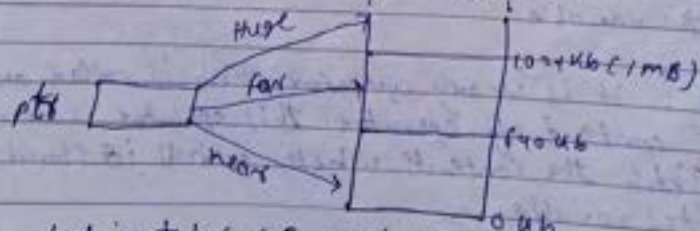
int 4

* Types of pointer:-

1) In real mode operating system there are 3 types of pointers as
   1) near pointer
   2) far pointer
   3) huge pointer

2) Real mode o.s are such operating system which can access only one 1mb of memory from RAM
3) In Real mode o.s only single process run at a time

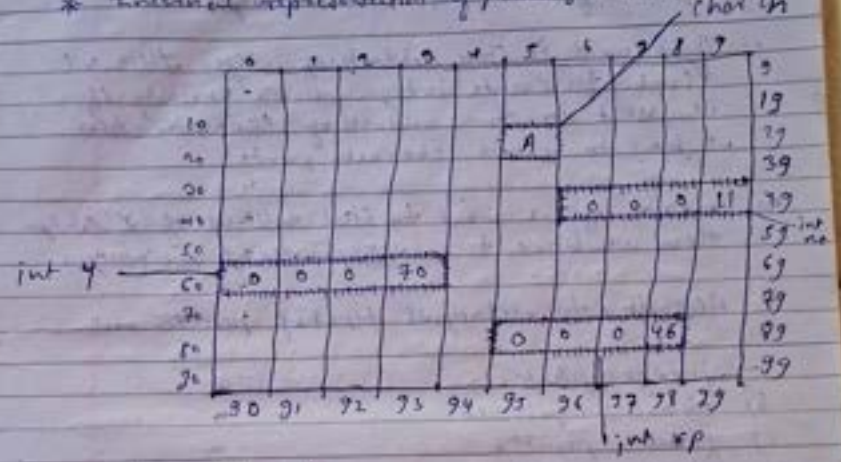
0 - 640kb - near pointer
640kb - 1024 kb - far pointer
1024kb - above - huge pointer



but in todays o.s which works in protected mode there is no such type of pointer that o.s uses the concept of flat addressing scheme where there is no partition of memory

due to which today's compiler and o.s there is no need concept type of pointer.

* Internal representation of pointer in RAM




```
char ch = 'A';
int y = 70;
int no = 11;
int *p = &no;
```

above diagram is 100 byte portion of RAM every small block indicate 1 byte memory when we declare any of the variable in our program memory gets allocated in this manner

According to this diagram our pointer gets acquired 4 byte memory like a normal integer variable

This pointer stores only address but by looking at the contents of the pointer we can not predict it is pointer or normal variable

Types of pointer depend on datatype

1) if we want to store address of integer then we have to create integer pointer. Similarly if we want to store address of character then we have to create character pointer

2) in general if we want to store address of 'x' datatype then we have to create pointer of 'x' datatype

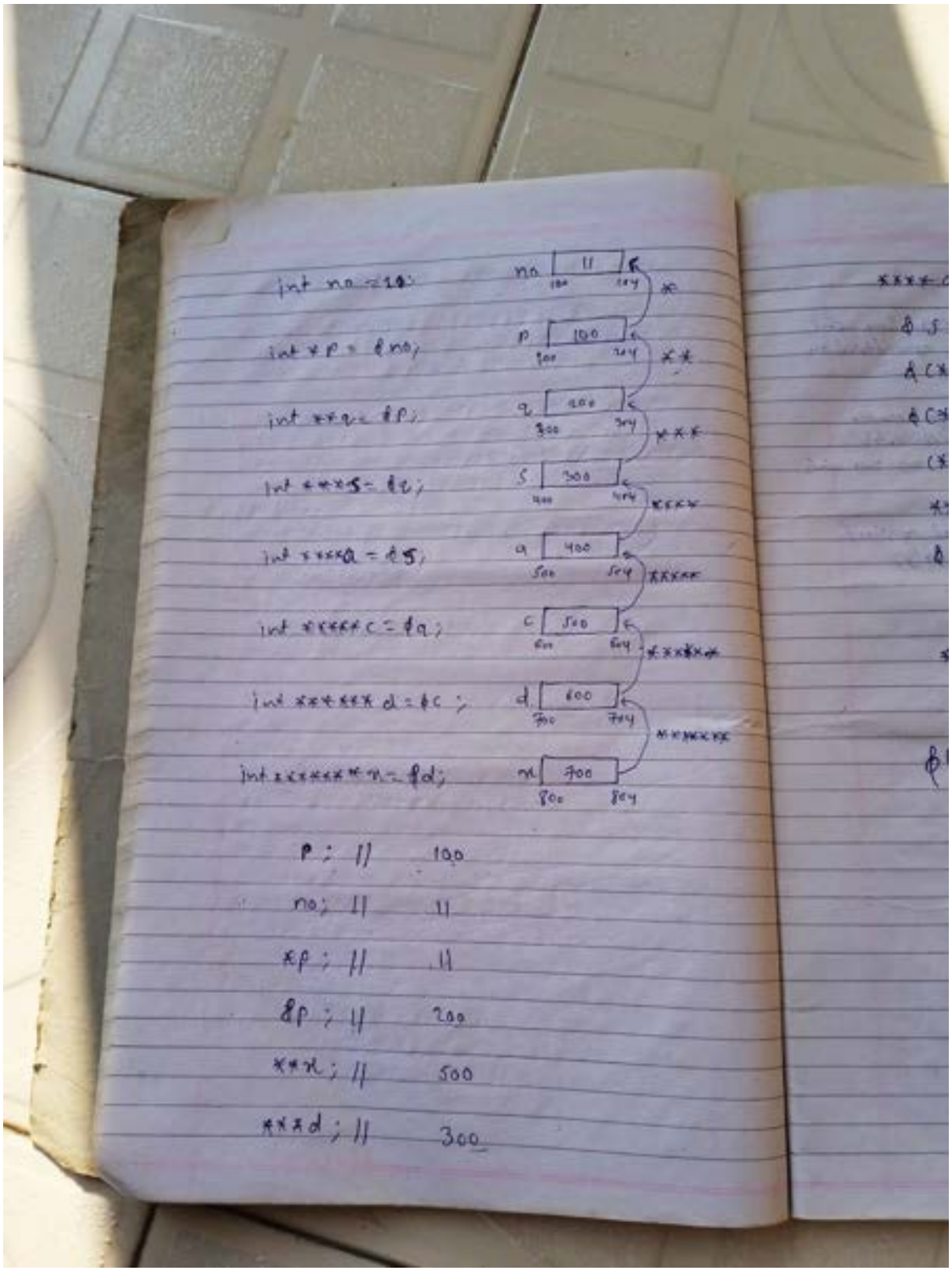
3) According to datatypes types of pointer are

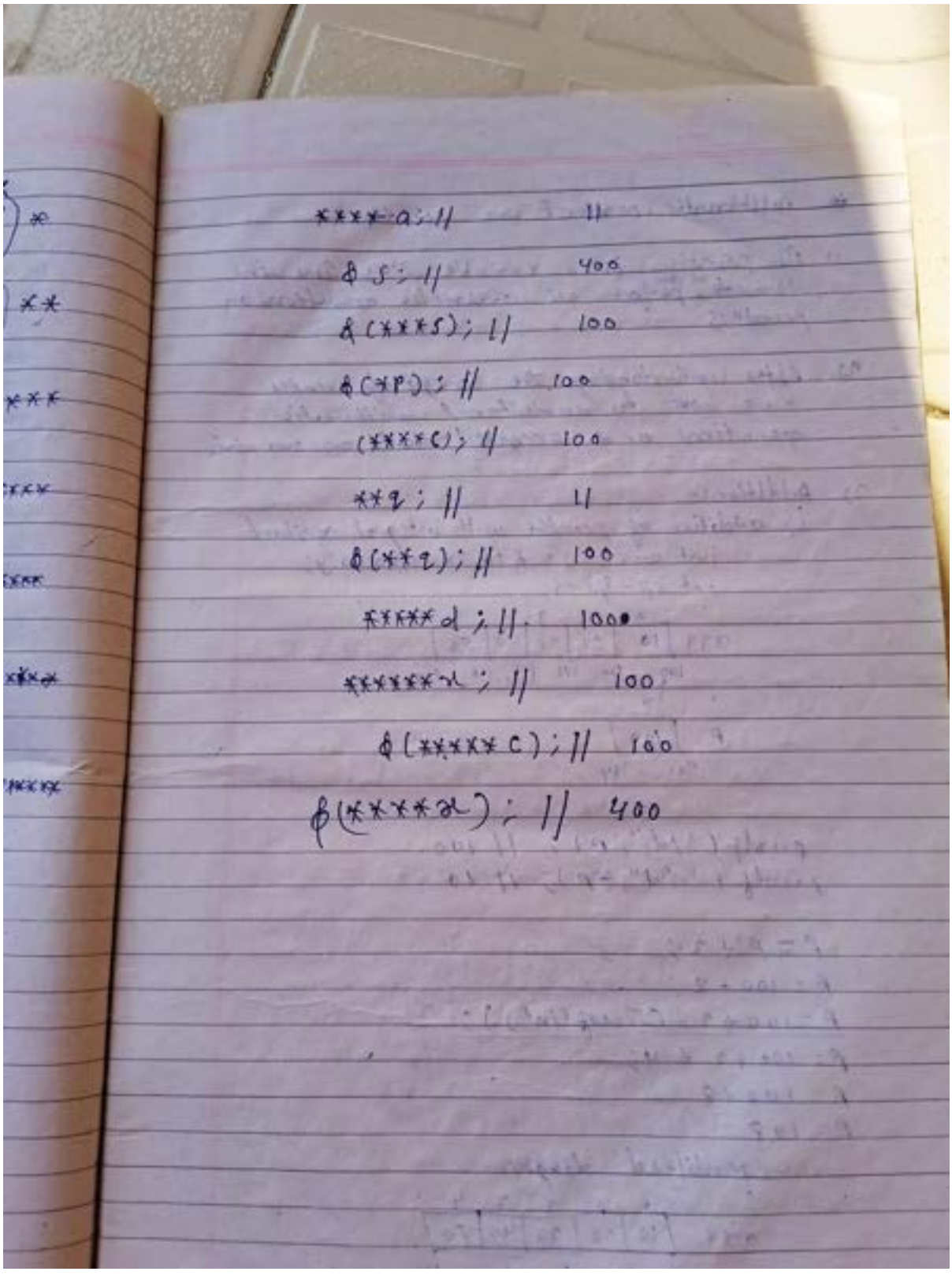
a) integer pointer
b) character pointer
c) float pointer
d) double pointer
e) structure pointer
f) union pointer

4) all this type of pointer require 4 bytes of memory irrespective of type of pointer.

5) if we store address of pointer then it is considered as pointer to pointer.

6) To understand this concept Considered below example as.


int no = 11;
int *p = &no;
int **q = &p;
int ***s = &q;
int ****a = &s;
int *****c = &a;
int ******d = &c;
int *******m = &d;
no 11 100 104 *
p 100 200 204 **
q 200 300 304 ***
s 300 400 404 ****
a 400 500 504 *****
c 500 600 604 ******
d 600 700 704 *******
m 700 800 804
p ; // 100
no ; // 11
*p ; // 11
&p ; // 200
**m ; // 500
***d ; // 300



* Arithmetic operations on pointers:-

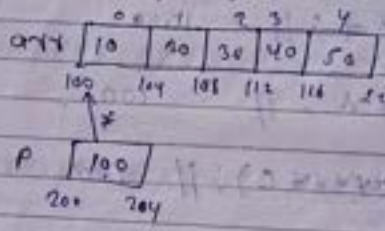
1) As pointer is a variable like some other variable we can perform all arithmetic operations on pointers

2) After understanding the concept of pointers we have to understand arithmetic operations and excepted result on our data

a) Addition:-

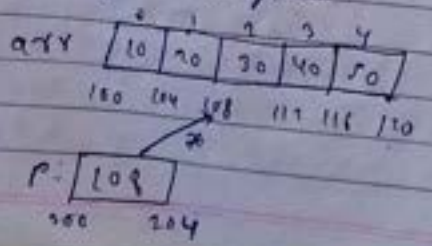
i) addition of pointer with integral constant

```
int arr[5] = {10, 20, 30, 40, 50};
int *p = &arr;
```




```
printf("%d", p); // 100
printf("%d", *p); // 10

p = p + 2;
p = 100 + 2;
p = 100 + 2 * (sizeof(int));
p = 100 + 2 * 4;
p = 100 + 8
p = 108
```

modified diagram



printf("%d", p); // 108
printf("%u", xp); // 120

the above syntax use the concept of addition of pointer with integral constant

when we add any integral constant with pointer then we have to multiply that constant with sizeof pointer type

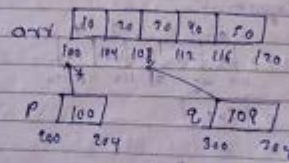
In our example pointer type is integer and pointed datatype which is array is also integer

ii) Addition of two pointers:-

Addition of two pointers is not allowed it generate compile time error because after addition it may generate some invalid address.

eg: int arr[5] = {10, 20, 30, 40, 50};
int *p = &(arr[0]);

int *q = &(arr[2]);



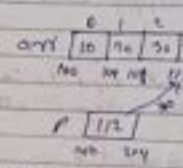
p+q; // not allowed
compilation error.

2) Subtraction:-

i) Subtracting integral constant from pointer

eg: int arr[5] = {10, 20, 30, 40, 50};

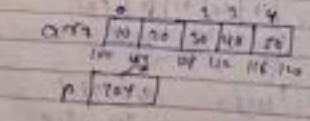
int *p = &(arr[3]);



printf("%d", p); // 112
printf("%d", *p); // 40

p = 112 - 2;
p = 112 - 2 * sizeof(int);
p = 112 - 2 * 4;
p = 112 - 8;
p = 104

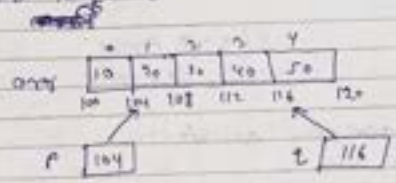
modified diagram:-



printf("%d", p); // 104
printf("%d", *p); // 20

ii) Subtract one pointer from other pointer

eg: int arr[5] = {10,20,30,40,50};
int *p = &(arr[1]);
int *q = &(arr[4]);



q - p;
= 116 - 104 = (116 - 104)/(sizeof(int));
= 116 - 104 / 4
= 12 / 4
= 3

when we subtract two pointers then we have to divide it by sizeof pointer type.

if we want to perform subtraction of pointer then
1) both pointer should be same point
2) both pointer must point to same memory which is related in nature

c) multiplication :-
multiplication of two pointer and multiplication of integral constant not allowed

p * q ; // Compilation error
p * 2 ; // Compilation error

d) division:

division of 2 pointers and dividing pointer with integral Constant is not allowed

```
p/q; // error
p/2; // error
```

e) increment operator:-

we can apply increment operator on ++ on pointer

f) decrement operator:-

we can also apply decrement operator on -- on pointer

++ & -- operators internally Method operator they use addition and subtraction technique indirectly

```
int arr[5] = {10, 20, 30, 40, 50};
int *p = &(arr[1]);
int *q = arr[4];
```

p++ ; //

p = p+1;
p = 104+1;
p = 104 + 1 * sizeof(int);
p = 104 + 1*4
p = 104 + 4
p = 108;

q--; //

q = q-1;
q = 116 - 1;
q = 116 - 1 * sizeof(int);
q = 116 - 1*4;
q = 116 - 4;
q = 112;

1/8/15

Void pointer - it is considered as generic pointer because that pointer stores address of any datatype

1) means void pointer can store address of integer, address of float, address of double, address of char

2) if pointed datatype is not known then we can use the concept of generic pointer i.e (void *).

eg: int no = 10;
void *p;
p = &no;
printf("%d", (int*)*p);

4) according to above syntax at the time dereferencing the content we have to use specific typecasting

5) we can assign any type of pointer to void pointer directly.

eg: int no = 10;
int *iptr = &no;

char ch = 'A';
char *cptr = &ch;

void *p;

p = iptr; // allowed
p = cptr; // allowed

6) we can not apply any pointer arithmatic operations on void pointer

7) eg: void *p, *q;

p+3; //error
q-p; // error

* Memory allocation :-

There are 2 ways in which we can allocate memory
1) Static memory allocation
2) dynamic memory allocation

Static memory allocation is also called compiles time memory allocation or early memory allocation

dynamic memory allocation is also called as runtime memory allocation or late memory allocation

In both the typed of memory allocation memory is allocated in main memory i-e RAM

There are some difference in static memory allocation and dynamic memory allocation such as

Static	dynamic
1) memory is allocated at runtime how much memory should be allocated is decided at compile time	memory is allocated at runtime but how much memory should be allocated is decided at runtime
2) memory allocated inside stack or data	memory is allocated inside heap
3) the default value of allocated memory is dependon storage class	default value is always zero
4) there can be memory shortage or memory wastage	There can not be memory shortage or memory wastage
5) if there is no enough memory then our program is unable to execute	if there is no enough memory then our program may abort at runtime
6) there is no need to use any special function for memory allocation or deallocation	we have to use some special function such as malloc(), calloc(), realloc() new() free(), delete()
7) there is no need to free the memory	it is programmer responsibility to free the allocated memory

8) we can not increase or decrease the allocated memory	we can increase or decrease the allocated memory by using realloc()
9) once memory gets allocated statically it remains there in main memory till the lifetime of that storage class ends.	After the use of dynamic memory gets completed we can free that memory.
10) it requires less time to access the data	it requires more time to access the data.

* memory allocation and deallocation function

1) malloc() :- memory allocation

a) this is a Basic function which is use to allocate memory dynamically.

this function accept one parameter which is no of bytes that we want to allocate and it returns the byte address of allocated memory.

prototype

```
void *malloc (size_t no);
```

size_t is a typedef for unsigned int

as

```
typedef unsigned int size_t;
```

as a malloc is a library function which internally calls brk() - system call and that brk() system call internally calls growing algorithm.

if our malloc() function fails to allocate memory it returns NULL otherwise it returns a valid base address of allocated memory.

malloc → brk() → growregion

growregion - algorithm
brk() - system call

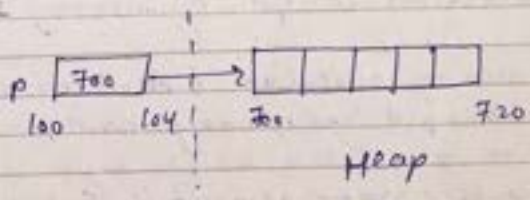
syntax

eg. if we want to allocate memory for five integers array then syntax should be

```
int *p = NULL;
p = (int*) malloc (5 * sizeof(int));
```

p → base address of allocated memory
(int*) → typecasting
malloc → function name
5 → no of elements
sizeof(int) → size of each element

diagram:-



in above diagram memory for pointer p is allocated in either stack or data section depend on it storage class and the actual memory to stored the data is allocated inside Heap

the below syntax indicate the static memory allocation of same memory.

```
int arr[5];
```

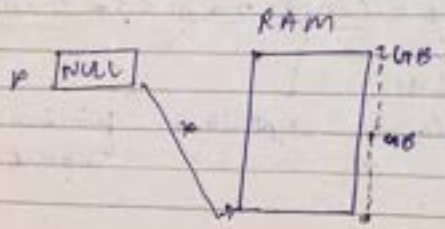
If the above dynamic memory allocation fail it returns NULL otherwise it returns valid byte address from Heap

* Null pointer:-

1) Null pointer is such pointer which stores zeroth address of our memory

2) generally at this address there is source code of operating system.

3) if pointer is uninitialized then it may generate runtime accident to avoid that we have to initialize with it NULL



* Internal concept of memory management

1) memory allocation & deallocation is performed and managed by memory manager of O.S

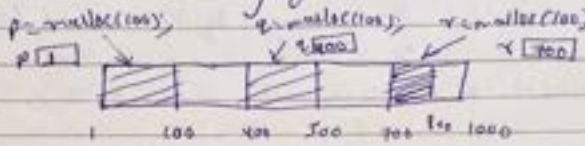
2) memory management is built a unit of operating system which internally maintains record of allocated memory and unallocated memory.

3) memory manager maintains 2 data structure
   1) used list of memory
   2) free list of memory.

- used list of memory contains all the information about currently allocated memory whole of the free list of memory contains the information about such a memory which is unallocated

4) Both two DataStructure contains 2 elements
   1) Base address
   2) no of bytes

which are currently free or in used



ULM

Base address	no of bytes
1	100
400	100
700	100

FLM

Base address	size
100	300
500	200
800	200

5) when we call malloc(), calloc(), realloc() function entry gets removed from free list of memory and it's get added in used list of memory

6) when we call free() function entry gets removed from ULM and it gets added in free list of memory

** Calloc():- Calculated allocations

1) The internal mechanism of malloc() & calloc() is exactly same

2) generally calloc() function is used to allocate memory for array

3) This function is called as calculated alloc because it performs internal calculation to decide the no of bytes should be allocated

4) void * calloc (size_t no, size_t size);

5) first parameter of calloc indicate no of elements and 2nd parameter indicates size of each element

6) if we want to allocate memory for array of float which contains 10 elements then syntax should be

```
float *p = NULL;
p = (float *) calloc (10, sizeof (float));
```

7) we can use the same logic by using malloc function as

```
p = (float *) malloc (10 * sizeof (float));
```

8) in old operating systems the memory which is allocated by malloc contain garbage and memory which is allocated by calloc contain zero in it

2) But in todays operating System there is no such difference.

**3) realloc() :- Reallocation :-

1) This function is generally used to increase the size of already allocated memory or decrease its size

void * realloc (void *p, size_t size);

2) first parameter of realloc is base address of already allocated memory
2nd parameter is the new size that we want to allocate that size can be greater than the previous or less than previous size.

eg: int *p = NULL;
p = (int *) malloc (5 * sizeof (int));
int *q = NULL;
q = (int *) realloc (p, 7 * sizeof (int));

3) in above syntax we are using q as a new pointer which is used to store address of expanded memory

4) at this place we can use p pointer also but if that expanded memory is not available contigiously then it returns NULL.

5) due to which we lost the base address of previously allocated memory. to avoid this problem we can take new pointer as q.

6) There are 2 scenarios in which we can use realloc()

i) To increase the existing memory
ii) To decrease the existing memory

7) if we want to increase already allocated memory then memory manager checks whether that much amount of memory is available immediately after that memory and if it is available then it returns the same base address

8) But if it is not available then it allocates the memory sequentially in another memory slot and returns the new base address and frees previously allocated memory implicitly.

9) But all the scenarios are not applicable if we want decrease the existing memory.

10) Some extra behaviours in realloc:-

if generally realloc() function is used to reallocate existing memory but there are 2 scenarios in which realloc() function works in different manner.

a) if first parameter of realloc is NULL then it behaves as a malloc.

eg: int *p = NULL;
p = (int*) realloc (NULL, 40);

in above syntax new 40 bytes memory get allocated and its base address gets copied inside p.

b) int *p=NULL;
p = (int*) malloc (40);

p = (int *) realloc (p, 0);

by using above realloc syntax the allocated memory by malloc gets free.

** free ():—

1) it is a good programing practice to free the dynamically allocated memory after its used gets completed.

2) if we hold that memory throughout the execution of program then it becomes memory wastage.

3) To free that physically allocated memory we have to use free () function.

void free (void *p);

4) return value of free is void but it accept one parameter which is the base address of existing memory which can be allocated by malloc, calloc, realloc.

5) when we call free () function memory manager of that operating system used list of memory table to ~~search~~ search it's base address

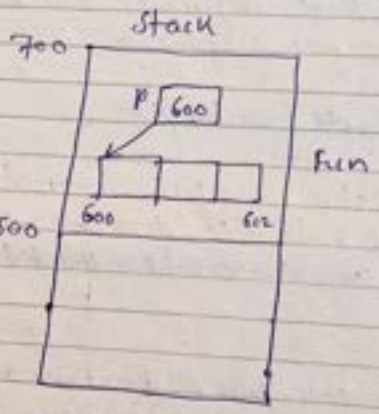
6) if that base address is found then that entry gets removed used list of memory table and it gets added in free list of memory table.

7) if we pass base address to free function that address should not be modified address because memory manager unable to search in used list of memory table.

* alloca() :- This function is same as malloc() but it is used to allocate a memory inside the stack frame of function inwhich that alloca() function called

in case of alloca there is no need to free the memory explicitly because that memory is freed implicitly when our function terminates.

```
void fun()
{
    int *p = NULL;

    p = (int *) alloca(3*sizeof(int));
```



* function pointer:-

1) function pointer is such a pointer which stores the address first instruction of our function

2) when we create a function pointer we can store address of any function whose prototype gets matched with prototype of that function pointer.

```
int add (int, int);
↓   ↓ ↓ ↓ ↓ ↓ ↓
int (*p) (int, int);
```

3) accordiry to above syntax p is a pointer which may store address of a such function which accepts two parameter first is integer and 2nd is integer and that function returns integer.

4) At this point only the pointer gets created but it contains nothing

5) As we want to store address of function then there is no need to use address of operator because name of the function is internally consider as its base address.

```
p = add;
p(10,20);
```

6) when we use this p pointed by passing 10 & 20 the function gets called implicitly whose address is stored in pointer p.

eg:
```
int add (int no1, int no2)  //1000
{
    printf("add");
}
int sub (int no1, int no2)  //2000
{
    printf("sub");
}
int mult (int no1, int no2) //3000
{
    printf("mult");
}
```

```
int main()
{
    int (*p)(int, int);
    p = add;
    p(10,20); // 1000   add
    p = sub;
    p(30,20) // 2000   sub
    p = mult;
    p(5,2) // 3000   mult

    return 0;
}
```

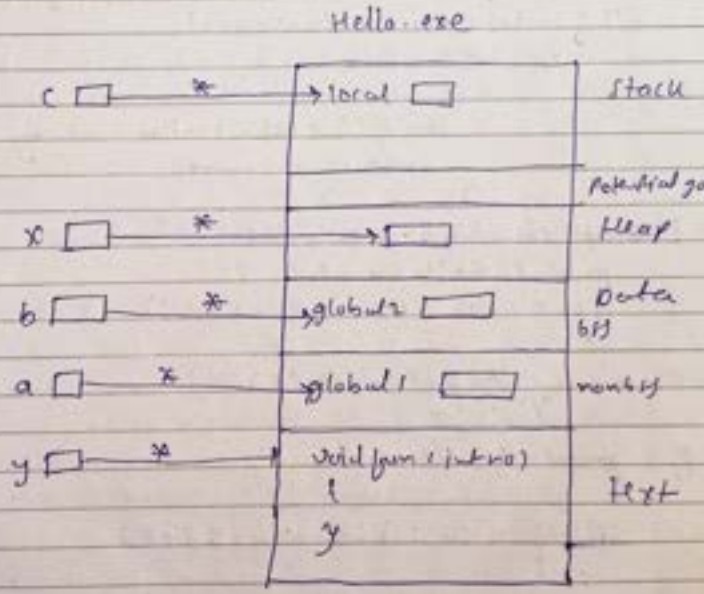
7) the concept of function pointer used in libraries such as dynamic link library where we have to receive address of function due to which acceptor should be function pointer.

8) like normal function ~~memory~~ memory for function pointer is always 4 bytes but it stores the address from text section of a process

* program which is used to demonstrate the pointers which may point to any section of a process(text, data, stack, heap).

```
int global1 = 10;
int global2;

void fun(int no)
{
    printf("fun");
}
```

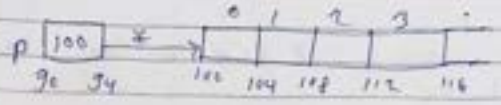
```
int main()
{
    int local = 11;
    int *a = &global1;
    int *b = &global2;
    int *c = &local;
    int *x = (int*)malloc(9);

    void (*y)(int ***);
    y = fun;

    return 0;
}
```



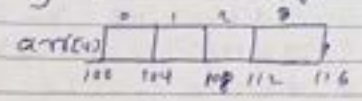
8/8/15 Dynamic memory allocation

* one dimensional array:-

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

int main()
{
    int element=0, i=0;
    int *p=NULL;

    printf("Enter no of elements");
    scanf("%d", &element);
// allocate memory
    p=(int*)malloc(element*sizeof(int));
// if memory is not allocated
    if(p==NULL)
    {
        printf("unable to allocate memory");
        return -1;
    }
// accept elements from user
    printf("Enter element");
    for(i=0; i<element; i++)
    {
        scanf("%d", p[i]);
    }
// print the element

    for(i=0; i<element; i++)
    {
        printf("%d", p[i]);
                // p[i]=*(p+i)
    }
// free allocated memory
    free(p);
}
```



Static memory allocation diagram:-



In above program we have to allocate memory dynamically for one dimensional array.

for that purpose we have to accept no of elements from the user & then allocate the memory by using malloc.

after use of that memory gets completed we have to free the allocated memory by using free function

* 2 Dimensional array:-

Consider the static memory allocation syntax of

arr[4][5]

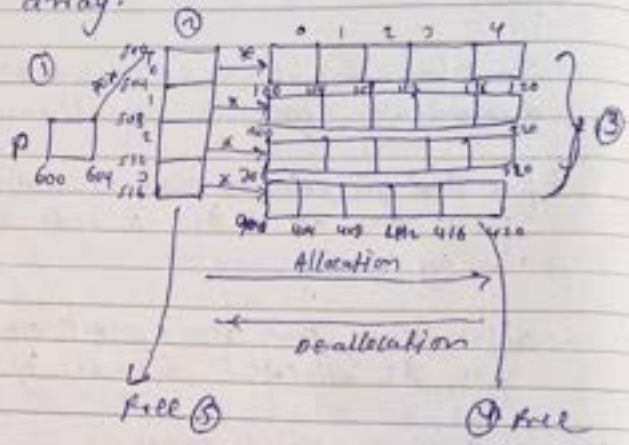
according to above syntax arr is 2 dimensional array, which contains 4 one dimensional array each one dimensional array contains 5 elements each element is of type integer.

arr[2][2]=10;

According to above concept we can conclude that

2 D array is collection of one dimensional array.

As we know Dynamic memory allocation for one dimensional we can use that same concept repeatedly to allocate memory for 2D array.



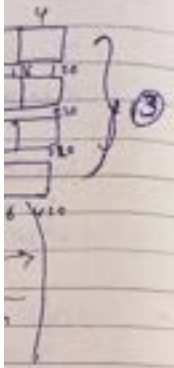
According to above diagram for memory allocation we have to follow 3 steps as

Step1: allocate memory for pointer p which is used to hold base address of array of pointers.

Step2:- allocate memory for array of integer pointers where each element of that array holds the base address of 1-D array.

Step3:- allocate actual memory to store our data elements.

to free that allocated memory which is allocated by using above 3 steps we have to follow reverse approach.



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

int main()
{
    int row=0, col=0, i=0, j=0;
Step1   int **p = NULL;

    printf("enter rows & cols");
    scanf("%d%d", &rows, &cols);

Step2:
    int p = (int**) malloc(sizeof(int*));

Step3:  for(i=0; i<row; i++)
        {
            p[i]=(int*) malloc(col * sizeof(int));
        }

    printf("enter the elements");
    for(i=0; i<row; i++)
    {
        for(j=0; j<col; j++)
        {
    scanf("%d", &p[i][j]);
        }
    }

printf("your entered elements are");
    for(i=0; i<row; i++)
    {
        for(j=0; j<col; j++)
        {
            printf("%d", p[i][j]);
        }
```

// below code is used to free deallocated memory
allocated memory

Step4: for(i=0; i<row; i++)
{
free p[i];
}

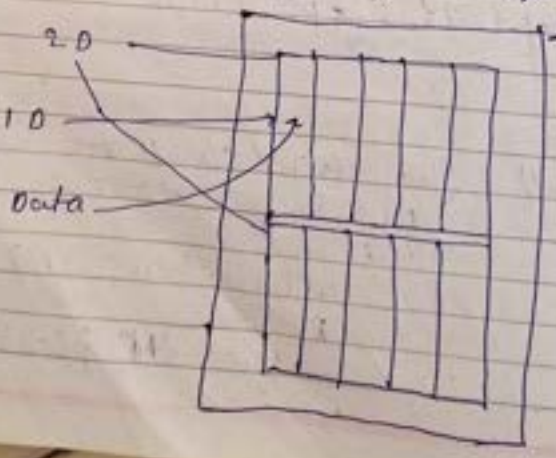
Step5:
free(p);

}

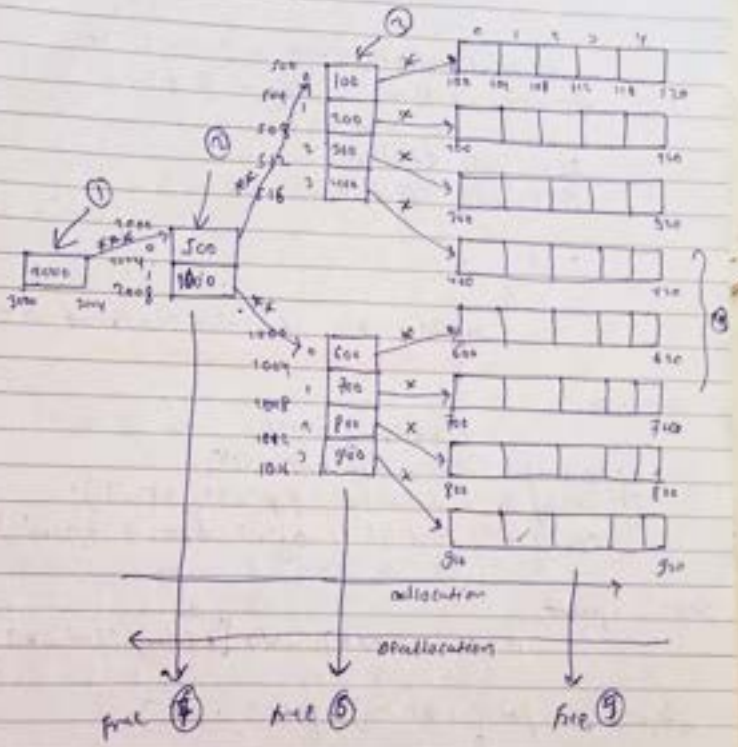
* 3-Dimentional array:-
Consider the below syntax of

int arr[2][4][5];

This array is considered of 3 dimensional array which contains 2 ~~two~~ dimensional array. Each 2 dimensional array containing 4 one dimensional array, each one dimensional array containing 5 elements each element is of type integer
below diagram is layout of above syntax of.



if we want to allocate that memory dynamically then our memory representation should be



```
int main()
{  int first=0, second=0, third=0;
step1:   int ***p = NULL;
   int arr[first][second][third]
```

memory allocation step - 1, 2, 3, 4
memory deallocation step - 5, 6, 7.

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

int main()
{
    int first=0, second = 0, third=0;
        i=0, j=0, k=0;
step1:
    int ***p = NULL;
  printf(" enter dimensions");
// scanf(":%d%d%d", &p[i][j][k]);
  scanf("%d%d%d", &first, &second, &third);
step2: point
    p = (int***) malloc(first*sizeof(int**));
step 3:  for(i=0; j<first; i++)
    {
      p[i] = (int**) malloc(second*sizeof(int*));
    }
step 4:  for(i=0; i<first; i++)
    {
      for(j=0; j<second; j++)
      {
        p[i][j] = (int*) malloc(third*sizeof(int));
      }
    }
```

```
printf("enter elements");
for(i=0; i<first; i++)
{
    for(j=0; j<second; j++)
    {
        for(k=0; k<third; k++)
        {
            scanf("%d", &p[i][j][k]);
        }
    }
}
//same logic used print the elements.
```

steps:

```
for(i=0; i<first; i++)
{
    for(j=0; j<second; j++)
    {
        free p[i][j];
    }
}
```

step 6:

```
for(i=0; i<first; i++)
{
    free p[i];
}
```

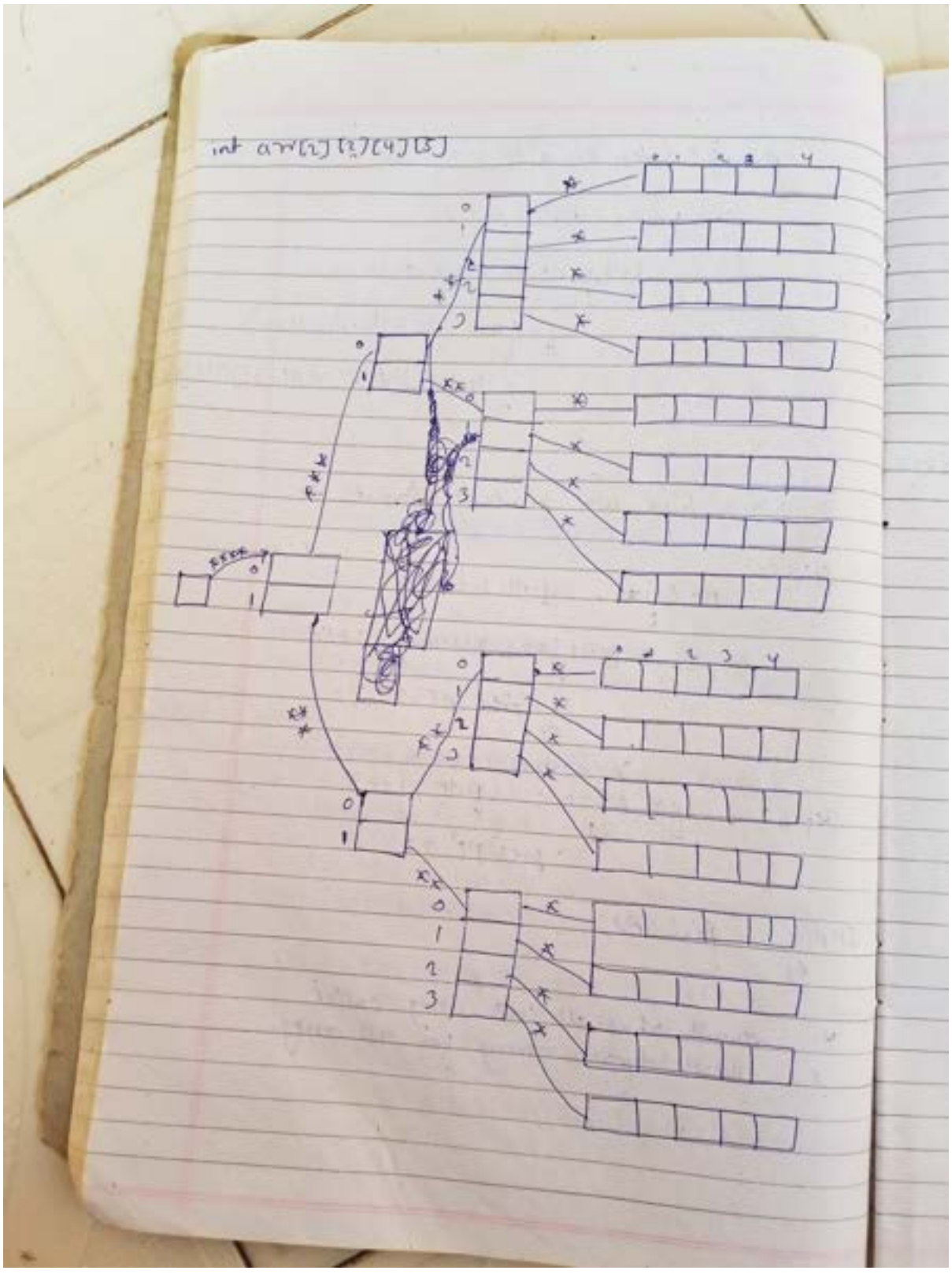
step 7 `free(p);`

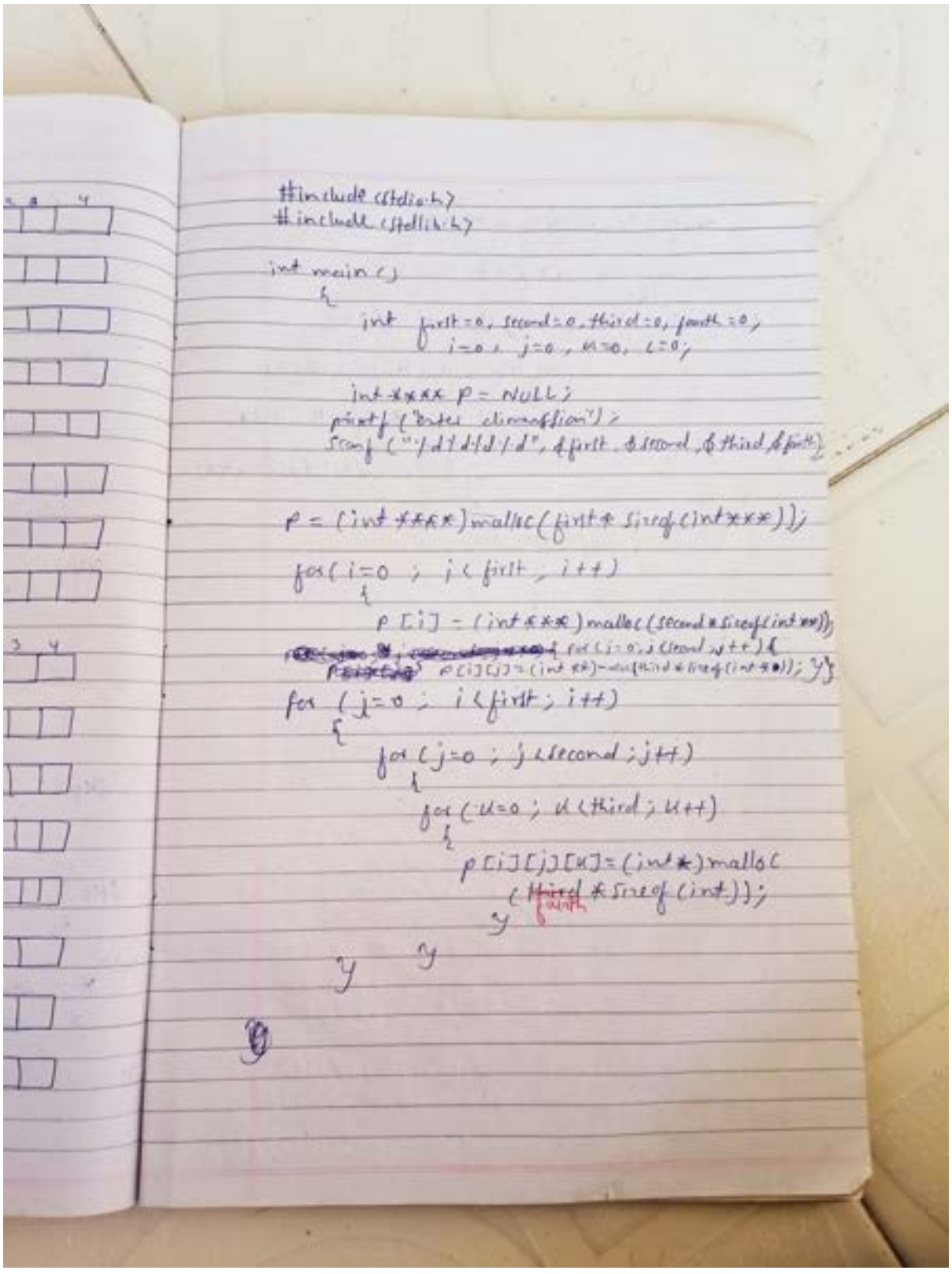
}

note:

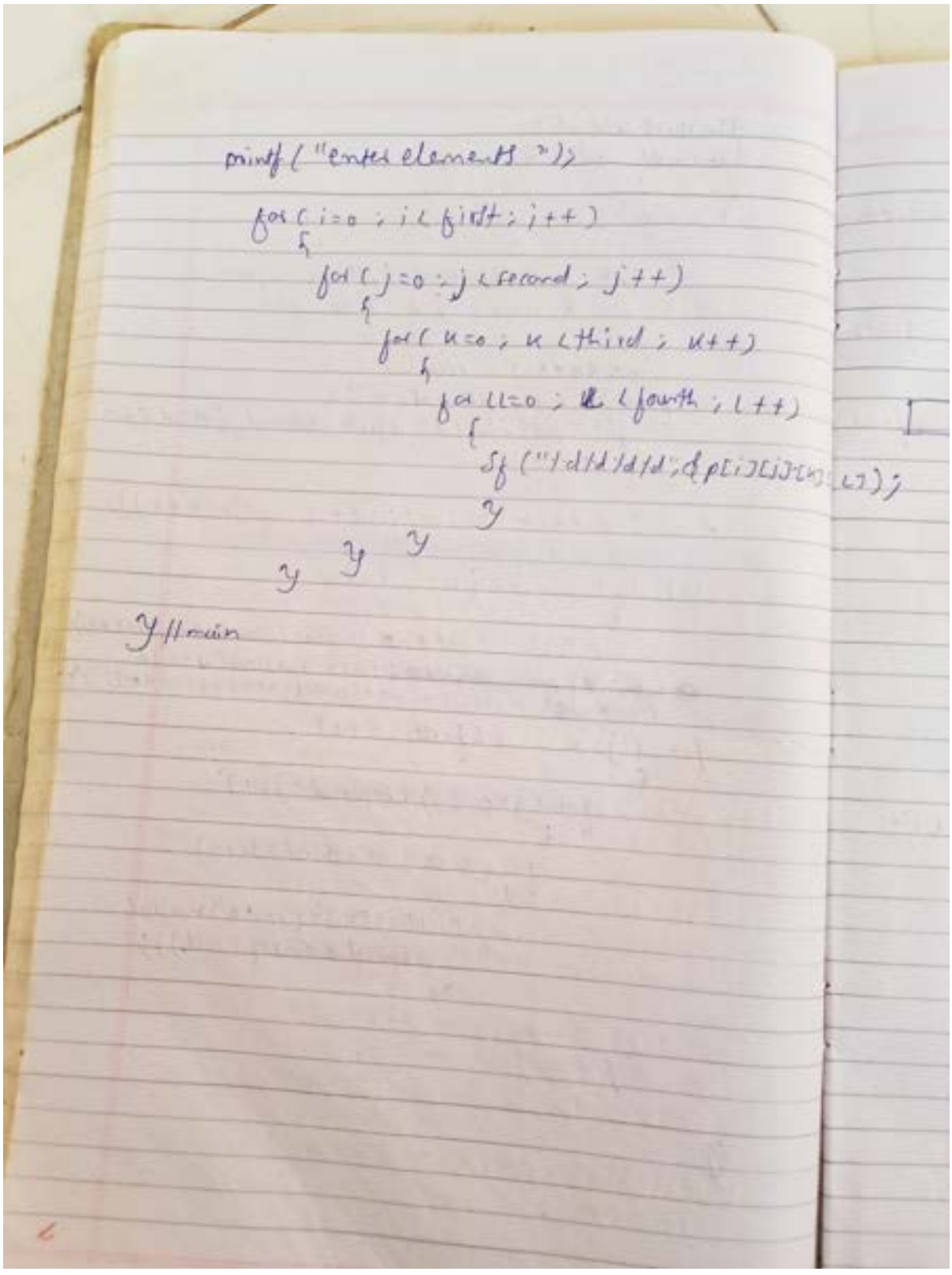
* rewrite above all code using calloc
* allocate dynamic memory for 4D array.

`rd * sizeof(int));`


int arr[2][4][5]

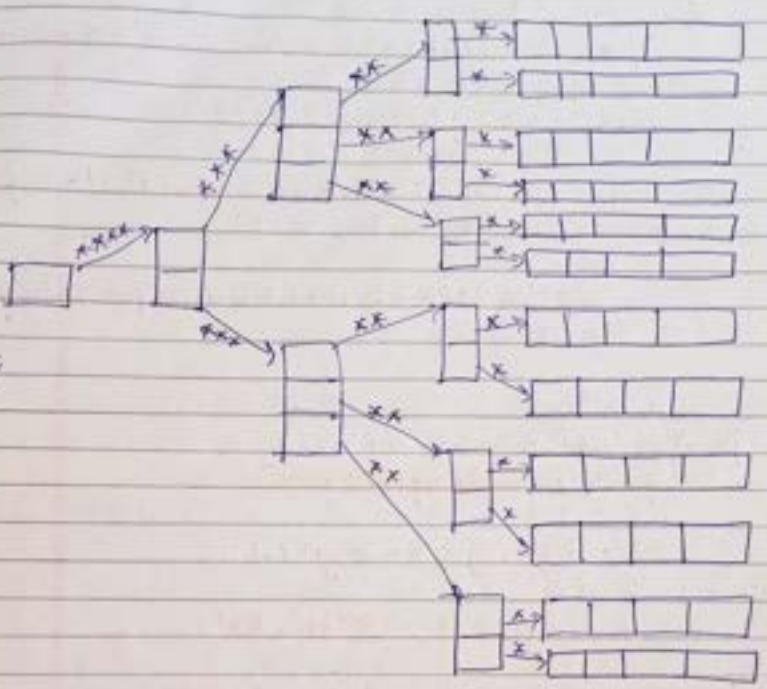


```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int first=0, second=0, third=0, fourth=0,
        i=0, j=0, k=0, l=0;

    int ****p = NULL;
    printf("enter dimmension");
    scanf("%d%d%d%d", &first, &second, &third, &fourth);

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

for(i=0 ; i<first ; i++)
{
    p[i] = (int ***)malloc(second * sizeof(int**));
    for(j=0; j<second; j++){
    p[i][j] = (int **)malloc(third * sizeof(int*));}}
for (j=0 ; i<first; i++)
{
    for(j=0 ; j<second; j++)
    {
        for(k=0 ; k<third; k++)
        {
            p[i][j][k] = (int*)malloc
                (fourth * sizeof(int));
        }
    }
}
}
```

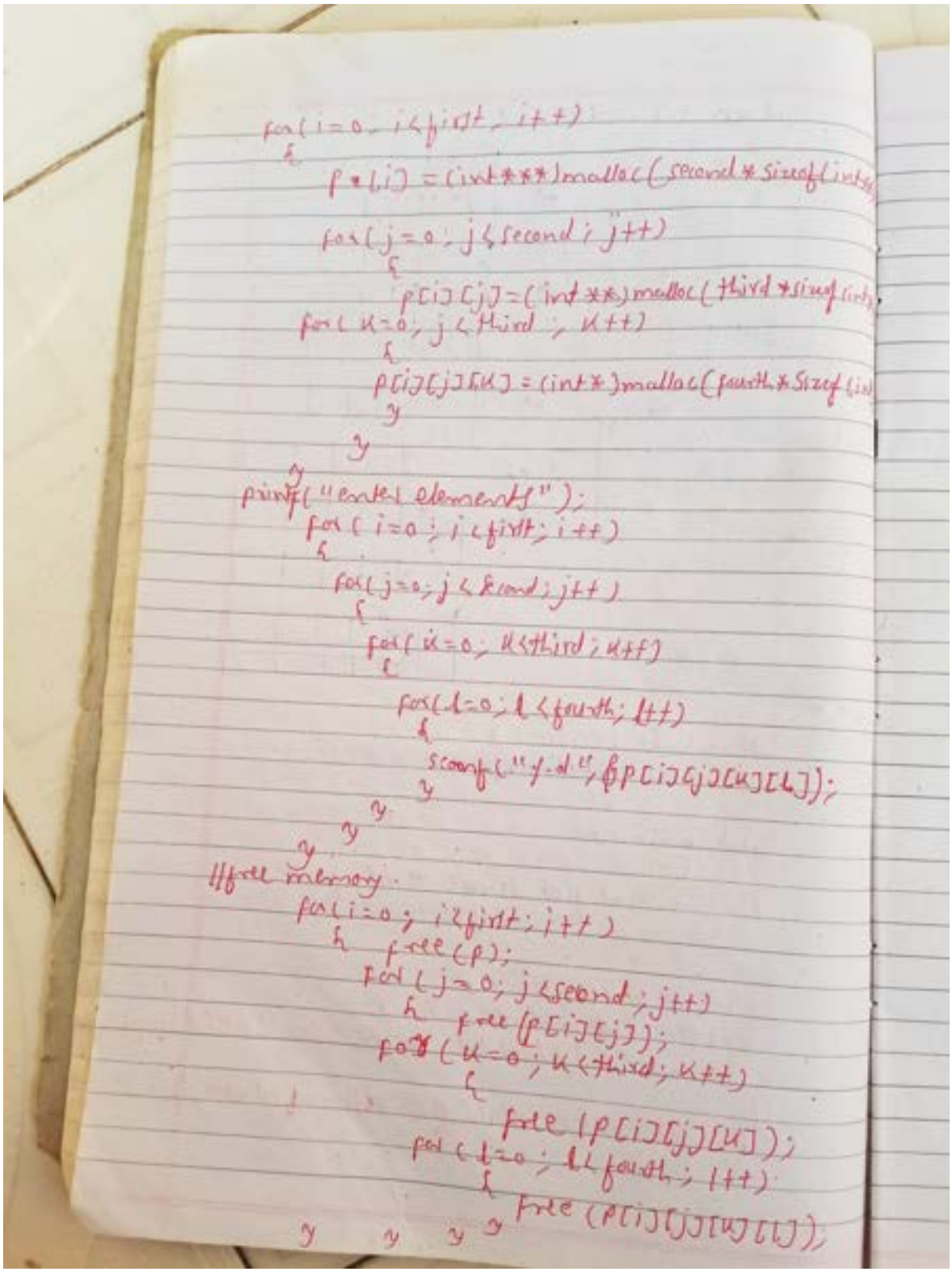


```
printf("enter elements");
for(i=0; i<first; i++)
{
    for(j=0; j<second; j++)
    {
        for(k=0; k<third; k++)
        {
            for(l=0; l<fourth; l++)
            {
                scanf("%d%d%d%d",&p[i][j][k][l]);
            }
        }
    }
}
}//main
```

int arr [2] [3] [2] [4]



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

int main()
{
    int first, second, third, four;
    int i, j, k, l;

    int ****p = NULL;
    printf("Enter dimension");
    scanf("%d %d %d %d", &first, &second, &third, &four);

    p = (int ****) malloc(first * sizeof(int ***));
```



```
for(i=0; i<first; i++)
{
    p[i] = (int***)malloc(second * sizeof(int**));
    for(j=0; j<second; j++)
    {
        p[i][j] = (int**)malloc(third * sizeof(int*));
        for(k=0; k<third; k++)
        {
            p[i][j][k] = (int*)malloc(fourth * sizeof(int));
        }
    }
}
printf("enter elements");
for(i=0; i<first; i++)
{
    for(j=0; j<second; j++)
    {
        for(k=0; k<third; k++)
        {
            for(l=0; l<fourth; l++)
            {
                scanf("%d", &p[i][j][k][l]);
            }
        }
    }
}
//free memory.
for(i=0; i<first; i++)
{
    free(p);
    for(j=0; j<second; j++)
    {
        free(p[i][j]);
        for(k=0; k<third; k++)
        {
            free(p[i][j][k]);
            for(l=0; l<fourth; l++)
            {
                free(p[i][j][k][l]);
            }
        }
    }
}
```

* Array and pointers *

1) One Dimensional array:-
Name of array is internally considered as Base address of that array.

when we access any element of array then that normal syntax gets converted into pointer syntax

Consider Below one dimentional array as
int arr[5] = { 50, 40, 30, 20, 10 };

when we create this kind of array Symbol table contains its name address and scope.

Consider the below syntax as
arr[3];
can be converted as.

```
*(arr+3);
*(100+3);
*(100+3*sizeof(int));
*(100+12);
*(112);
  20
```

above syntax indicates that when we access any element of the array it gets converted into the pointer syntax.

all the below syntax gives the same result as

A[B];	Arr[3];
*(A+B)	*(Arr+3);
*(B+A);	*(3+Arr);
B[A];	3[A];

} 20

* Array and pointers *

1) One Dimensional array:-
Name of array is internally considered as Base address of that array.

when we access any element of array then that normal syntax gets converted into pointer syntax

Consider Below one dimensional array as
int arr[5] = {50, 40, 30, 20, 10};

when we create this kind of array symbol table contains its name address and scope.

Consider the below syntax as
arr[3];
can be converted as.

```
*(arr+3);
*(100+3);
*(100+3*sizeof(int));
*(100+12);
*(112);
  20
```

above syntax indicates that when we access any element of the array it gets converted into the pointer syntax.

all the below syntax gives the same result as

A[B];	Arr[3];
*(A+B)	*(Arr+3);
*(B+A);	*(3+Arr);
B[A];	3[A];

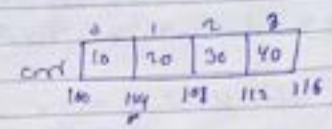
} 20

* Why array index starts from zero.

we can access every element of array by using its name and appropriate index.

Consider the below syntax of

int arr[4] = {10, 20, 30, 40};



If we start array index from one instead of zero then our syntax to access the first element should be arr[1];

This syntax gets converted into pointer format as

*(arr+1);
*(arr+1*4);
*(100+4)
*(104);

according to above syntax due to starting index one we can directly access the element which is at address 104.

The element at address 104 is not the first element of our first element stored it is 100. due to the starting index 1 we can not access the element at address 100

hence array index start from zero.

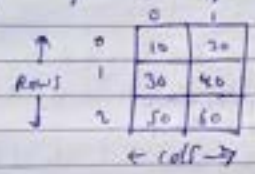
Two dimensional array

Consider the below syntax a dimensional array of

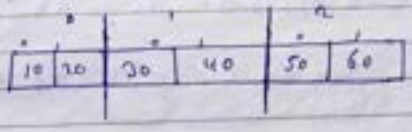
int arr[3][2];

for this array which is multidimensional array we can draw its diagram in two ways as.

1) diagramatic representation.



2) memory layout / memory representation:-



There are 2 ways in which the contents of array get stored in memory

1) Row major storage
2) column major storage

in case of row major storage contents of multi-dimensional array get stored in Row by Row format

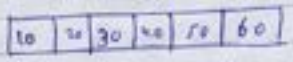
where as, in column major representation contents gets stored column by column format

The storage type technique is depend on the programming language used.

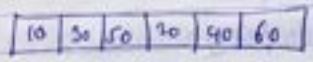
C, C++ uses row major storage where as Python uses column major storage (some of the versions)

Accessing mechanism of the contents array is irrespective of the storage mechanism.

Row major :-



column major :-



Similarly as one dimensional array we can also convert the normal syntax of accessing the elements into the pointer syntax.

Consider if i want to access the element of

arr[2][1];

the above normal syntax can be converted into pointer syntax as

Arr[i]:X// (Arr[2])[1] → X[1]; → *(X+1);

((A+X+2)+1)

In generic terms our 2D array syntax can be converted into pointers syntax as

X[A][B];

((X+A)+B)

((A+X)+B)

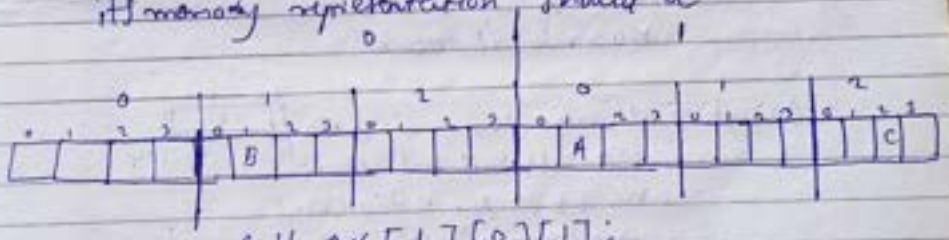
(B+((A+X)));

* 3 dimensional array:-

int arr [2][3][4]

Consider below 3 dimensional array of in diagramatic representation should be

its memory representation should be



A // Arr [1][0][1];
B // Arr [0][1][1];
C // Arr [1][2][2];

H.W

problem on pointer 1
problem on C
problem on array
problem on Control instructions
problem on storage

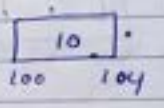
* Constant and pointers:-

Constant is considered as datatype qualifier in C & C++

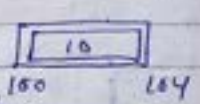
If const keyword is used while declaring the variable then we can not change the content of that variable.

Consider the below syntax as

```
int no = 10;
no = 11; // allowed
no++;    // allowed
no--;    // allowed
```




```
Const int no = 10;
no = 11; // not allowed
no++;    // not allowed
no--;    // not allowed
```



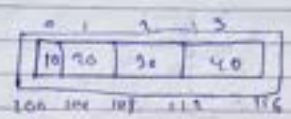
According to the concept of constant the variable which is initialized at the time of declaration can not be change throughout execution of program.

Like normal constant variable we can also create constant array.

Constant array should be by using initialization list otherwise we can not initialize it.

Eg: const int arr[4] = {10, 20, 30, 40};

arr is one dimensional array which contains 4 elements and each element is of type constant integer.

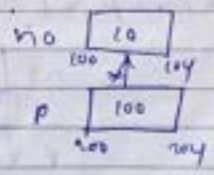


arr[1] ++; // not allowed
arr[2] = 15; // not allowed

As the concept of constant is applied on any datatype due to which it is also applicable for pointers.

There are some scenarios to understand the concept of constant with pointers.

Scenario 1:- int no = 10;
int *p = &no;



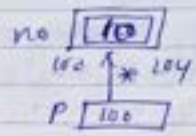
p is a pointer which holds address of integer

no = 11; // allowed
no ++; // allowed
p = &x; // allowed
p++; // allowed
(*p) = 11; // allowed

In this scenario pointed type is nonconstant and pointer type is nonconstant

Scenario 2:-

```
const int no = 10;
const int *p = &no;
```



p is pointer which holds address of integer constant

```
no = 11;   //not allowed
no++;      // not allowed
p = &x;    // allowed
p++;       // allowed
(*p) = 11; // not allowed
```

In this scenario pointed type is constant but pointer type is nonconstant

Scenario 3:-

```
int no = 10;
int * const p = &no;
```

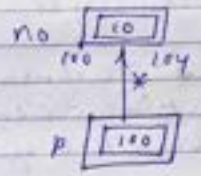
p is a constant pointer which holds address of integer.

```
no = 11; // allowed
no++; // allowed
p = &x; // not allowed
p++; // not allowed
(*p) = 11; // allowed
```

in this scenario pointed type is non constant but pointer type is constant

Scenario 4:-

```
const int no = 10;
const int * const p = &no;   } both same
int const * const p = &no;   }
```




```
no = 11;      // not allowed
no++;         // not allowed
p = &x;       // not allowed
p++;          // not allowed
(*p) = 11;    // not allowed
```

in this scenario pointed type is constant as well as pointer type is constant

* Complicated Statment reading in C

1) int no=10;

no is a variable of type int which is initialize to value 10.

2) int arr[4];

arr is a one dimensional array,
which contains 4 elements.
each element is of type integer.

3) struct demo obj[5];

obj is a one dimensional array,
which contains 5 elements
each element is of type struct demo.

4) int arr[3][5];

arr is a 2 dimensional array
which contain 3 one dimensional array,
each one dimensional array contains
5 elements
each element is of type integer.

5) Int arr[3][2][4];

arr is 3 dimensional array,
which contains 3 two dimesional array.
each 2 dimensional array contains 2 one
dimensional array
each one dimensional array containg 4 element
each element is type integer

6) int *p;

p is a pointer which can hold address of integer

7) char *p;

p is a pointer which can hold address of character

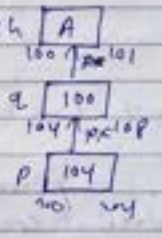
8) void *p;

p is a pointer which can hold address of any datatype

9) char **p;

p is a pointer which holds address of character pointer

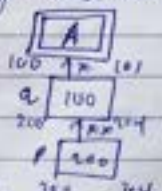
10) eg: char ch = 'A';
char *q = &ch;
char **p = &q;



11) const char **p;

p is a pointer which holds address of such a character pointer,
that pointer holds address of constant character

const char ch = 'A';
const char *q = &ch;
const char **p = &q



ch++; // X
ch = 'B'; // X
*q = 'B'; // X
(*q)++; // X

*p = &x; ✓
*p++; // ✓
**p = ch2; // X
(**p)++; // X

11) char * const *p;

p is a pointer which holds address of constant pointer and that pointer holds address of character.

```
char ch = 'A';
char *const q = &ch;
char * const *p = &q;
```

12) const char * const *p;

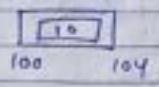
p is a pointer which holds address of constant pointer, and that pointer holds address of character constant

```
const char ch = 'A';
const char *const q = &ch;
const char *const *p = &q;
```

15/8/15

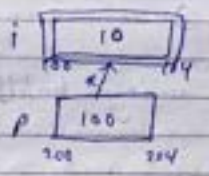
* const int i = 10;

i is a variable of type constant integer. which is currently initialize to 10.



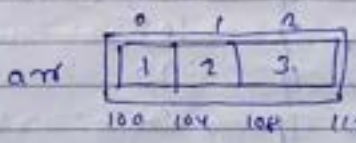
* Const int *p = &i;

p is a pointer which holds address of integer constant. and currently it holds address of variable i



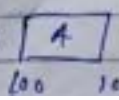
* Const int arr[3] = {1, 2, 3};

arr is one dimensional array. which contains 3 elements, each element is of type constant integer, and that array is initialize to values 1, 2, 3.



* char c = 'A';

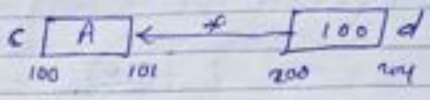
c is a variable of type character which is initialize to value A.



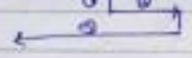
deep c secret, expert in c by peter vander linden

* char *d = &c;

d is a pointer which holds address of character and currently it holds address of variable c

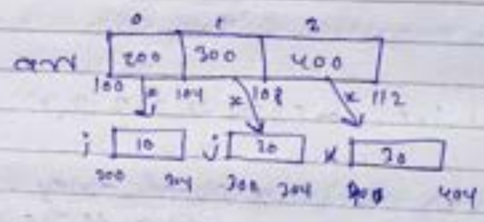


* int *(arr[3]) = {&i, &j, &k}; i = 10, j = 20, k = 30



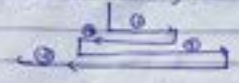
① arr is a one dimensional array, ② which holds contains 3 elements, ③ each element is type of int *

currently that array holds address of variable i, j, k.

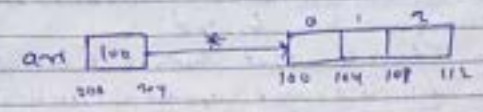


* arr; // 100
&arr; // 100
arr[0]; // 200
&(arr[2]); // 108
*(arr[1]); // 20
*(arr[3]); // 30

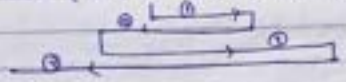
* int (*arr)[3];



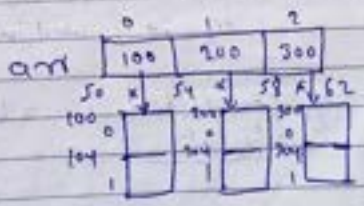
① arr is,
② pointer which holds address of
③ one dimensional array which contains 3 elements
④ each element is of type integer.



* int (*(arr[2])[3]);



① arr is one dimensional array which contain 3 elements
② each element is of type pointer
③ ~~which holds address of one~~ which holds address of one dimensional array which contain 2 elements
④ each element is of type integer.



* Rules of Statement reading by Greedy parsing algorithm

The rules which we are using to read the statement are internally used by compiler for tokenization activity.

Step
1) while reading start from identifier name
2) traverse the statement on right hand side till we get closing circular bracket
3) again traverse that statement on left hand side till we get opening circular bracket
4) repeat step2 & step3 till our statement ends.

* void fun(void);

fun is a function which accept nothing and return nothing.

* void fun();

fun is a function which nothing and which return nothing (if it is used in c)

fun is a function which accept any no of argument and it returns nothing
(if it is used in c++)

this type of convention is allowed in c++ only for function declaration.

* int fun(int, char);

parameters fun is a function which accept two parameters
first parameter is value of integer
2nd parameter is value of character
and that function return integers.

caller

```
int main()
{
    int i=11, no=0;
    char ch='A';
    no=fun(i, ch);
    // no=fun(11, 'A');
    return 0;
}
```

callee

```
int fun(int x, char c)
{
    ;
    return is;
}
```

According to above program main is considered as a caller function & fun is callee function

* function calling techniques

There are 3 types of function calling techniques available in C & C++ depend on the contents of parameter such as

1) call by value
2) call by address
3) call by reference

1) call by value : in this type function calling technique we have to pass the value to callee function

2) call by address : in this type of function calling technique we have to pass address of global or local variable, acceptor in callee function should be pointer because we are sending the address

3) call by reference (only in C++) :-
in this concept is similar as

call by address concept in this concept we have to use concept of reference in C++ which is consider as another name to the variable

* function returning mechanism:-

All four function calling mechanism there are 3 ways in which we can return something from the function

1) return of a value:-
in this returning mechanism we have to return some value which should be stored in some another variable of called function.

2) return of a address:
in this technique we can return address of any global or local variable in this case acceptor in called function should be pointer.

3) return of a reference only in C++):-
in this technique we can return name of the variable which is accepted in called function in some another variable of reference variable.

* int fun (int, char)

in this function we are using concept of call by value and return by value technique

* int * fun (int);
fun is a function which accept one parameter i.e value of integer and that function returns address of a integer.

caller

```
int main()
{
  int *p = NULL;
  int y = 111;
  p = fun(y);
  pf("%d", *p);
  return 0;
}
```

callee

```
int no=1;
int *fun(int x)
{
  return &no;
}
```

above code use) concept of call by value and return by address technique

in above program no variable is taken as global variable because we want to return address of that variable.

if we take that variable as a local variable while returning that variable scope of that variable gets terminated.

* int fun(int *, char*);
fun is a function which accepts two parameters first is address of integer, and is address of character and that function returns integer

caller

```
int main()
{
  int no = 11, x = 0;
  char ch = 'A';
  x = fun(&no, &ch);
  return 0;
}
```

callee

```
int fun(int *p, char *q)
{
  :
}
```

in above program we are using call by address technique and return by value technique

* int fun(int a[]);

fun is a function which accept one parameter i.e base address of array of address ~~of array~~ of integer.

int fun (int *a);

this syntax is also same as above syntax

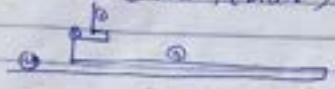
```
int main ()
  {
     int arr[2] = {10,20};
      fun (arr);
       return 0;
  }
```

* int **fun ( );

fun is a function which accept nothing and it return address of integer pointer

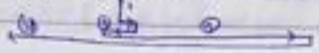
eg: int main()
```
{ int no=11;
  int *p = &no;
  int **fun ( )
    {
      return &p;
    }
  return 0;
}
```

* int (*p)(int, char);



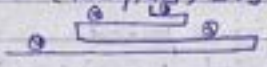
(1) p is
(2) pointer which holds address of
(3) such a function which accept two parameters first is value of integer, 2nd is value of character
(4) and that function returns integer.

* int (*p) (int **);



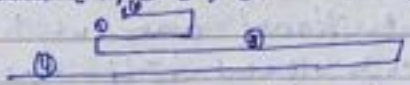
(1) p is
(2) pointer which holds address of
(3) such a function which accept one parameter i.e which accept address of integer pointer
(4) and that function returns integer.

* int (*daytab) [13]; // int *a[13];



① daytab is
② a pointer which holds address of
③ one dimensional array which contains 13 elements
④ and each element is of type integer.

* void (*p[3]) (int, int);

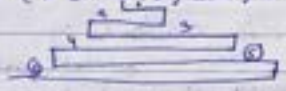


(1) P is a one dimensional array which contain 3 elements and each element is of type function pointer which holds address of such function each function accept two parameters and function return nothing.

* above program is used to create array of function pointer

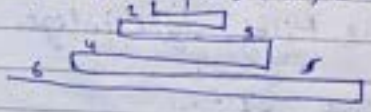
```
void fun (int x, int y);
void func (int x, int y);
void sunc (int x, int y);
void (*F[3])(int, int) = {fun, gun, sun};
F[1](10,20); // call gun

for (int i=0; i<=2; i++)
  {
    F[i](10,20); // call fun, gun, sun
  }
```

* char (*(*x())[4])();



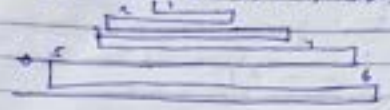
(1) x is a function which accept nothing
(2) which returns pointer to
(3) one dimensional array which contains 4 elements
(4) each element is of type pointer
(5) which holds address of such function which accept nothing
(6) and returns characters.

* char (*(*x[3])())[5]



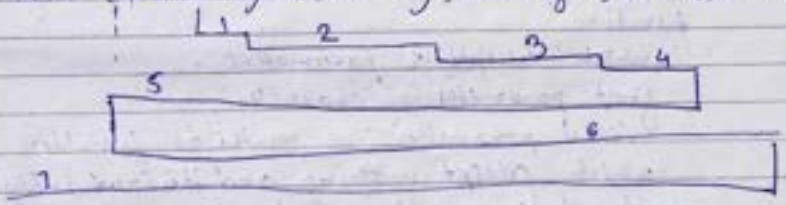
(1) x is one dimensional array which contain 3 elements (2) each element is of type pointer which holds address
(3) such a function which accept nothing
4) and which returns pointer to
5) one dimensional array which contain 5 elements (6) and each element is of type characters.

* int *(*(* arr[5])()) (int);



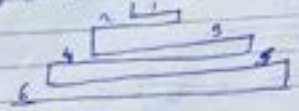
1) arr is one dimensional array which contains 5 elements
2) and each element is of type pointer
3) which holds address of such a function which accept nothing
4) that function returns pointer to
5) such function which accept one parameter
6) i.e int
7) that function returns address of integer

* void (*bsd_signal(int sig, void(*fun)(int)))(int);



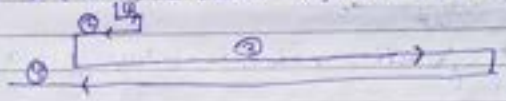
(1) bsd_signal is a function which accept 2 parameter
(2) and
(3) first parameter is integer,
(4) and 2nd parameter is pointer to function which accept one parameter as a integer and it returns nothing
(5) and bsd_signal function returns pointer to
(6) such a function which accept one parameter i.e integer
(7) and that function returns nothing

* float (*(*b[])[])();



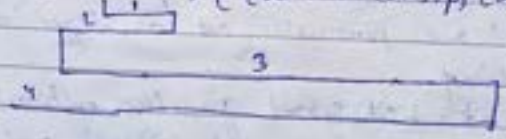
b is a function which accept nothing which returns pointer to an array which contains 5 elements and each element is of type pointer which holds address of such fun which accept nothing and return float

* void *(*c)(char, int(*)());



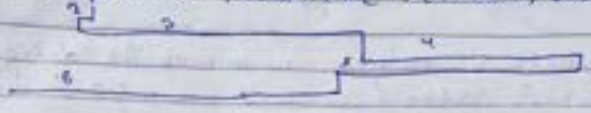
c is a pointer which holds address of such function
which accepts 2 parameters
first parameter is character
second parameter is a pointer to function which accept nothing and it returns integer and whole function returns address of any datatype.

* void *(*a[5])(char * const p, char * const);



a is one dimensional array which contains 5 elements each element is of type pointer, which holds address of such a function, which accept two parameters first parameter is constant pointer which holds address of character, 2nd parameter is also constant pointer which holds address of character and whole fun returns address of any datatype

* void *(*p)(int, char **(*)(char *, char **));



p is a pointer
which holds address of such a function
which accepts two parameters
first parameter is integer
2nd ~~point~~ parameter is pointer to such a function
which internally accepts two parameters
where first parameter is address of character,
and parameter is address of character pointer of
that function returns address of character pointer
and whole function return address of any datatype

16/8/15

* user defined datatypes in C

1) user defined datatype is one of the datatype in C whose containing contents are depend on user of the programming language ie programmer.

2) depend on the programmatic requirement we have to design our user defined datatype.

3) There are 3 user defined datatypes in C such as
   i) structure
   ii) union
   iii) enumeration

4) Allocated contents of structure, union & enumeration are directly depend on the programmer.

5) Before using any of the datatype first we have to think about its content and then we have to declare that appropriate datatype.

6) if we create single variable of any datatype then it is manageable but if we create multiple such variable then it is not manageable to avoid this problem we can use derived datatype i.e array.

7) The major restriction on array is the homogenous elements of that array.

8) means in array we can store elements of same datatype.

9) if we want to store related element together under one name then we have to use any of the user defined datatype.

eg: if we then want to store marks of 10 student then we create array of integer.

if we want to store division of 10 student then we have to create array of character.

if we want to store birth year of student then we have to create array of integers.

all above practical approaches are correct but we can not map this 3 different data structure together.

means we can retrive division of student, marks of that student, Birth year of student simultaniously but if we want to perform that activity then we have to create structure which contains name of the student, marks of student, it's division etc.

The above technical example indicates need of user defined datatypes.

Some logical concept of user defined datatype is used in C++ which is named as class.

* important points about structure and union

structure - structure is user defined datatype which contains all related things under single name

structure can contains homogenous or heterogenous elements. means structure can contains elements of different datatypes also.

1) Structure can contain any of the primitive datatype.

eg: struct demo
{
int i;
char c;
float f;
double d;
};

in above syntax struct is keyword, demo is name of struct structure. i, c, f, d are members of struct structure and ; after last closing curly bracket indicate end of structure declaration.

2) structure can contain any derived datatype in it such as array, pointer etc.

eg: struct demo
{
int arr[5];
char *p;
};

3) structure can contain any user defined datatype as its member means it contain structure & union inside it

struct outer
{
int i;
struct inner
{
int j;
};
};

in above example inner named structure is considered as an inner structure.

Before using any of the structure first we have to declare that structure

eg: struct demo
{
int i;
float f;
};

when we declare the structure memory is not allocated for that structure members

at the time of structure declaration new entry gets inserted inside symbol table by the compiler.

that symbol table entry contains name of the structure and no of bytes which are required for that structure

y) * Actual memory for the structure when we get allocated when create object or variable of that structure

struct demo obj1, obj2;

obj1, obj2 are objects of struct demo.

5) * we can create object of variable of a structure at the time of declaration itself.

eg: struct demo
{ int i;
float f;
} obj;

6) we can not initialize any member of structure at the time of declaration because at that point memory is not allocated.

eg: struct demo
{
int i = 10; // error
};

7) if we try to initialize that member it generates compiletime error.

8) we can initialize members of structure after creating variable of that structure.

9) we can initialize the member by using member by member technique both of

eg: struct demo
{
int i;
int j;
};
struct demo obj;
obj.i = 10;
obj.j = 20;

10) we can also initialize content of structure by using initialization list.
eg: struct demo
{
int i;
char ch;
};
struct demo obj = {10, 'a'};
struct demo obj = { .ch = 'a' }; // compiler dependent

11) we can use initialization list at the time of declaration also

```
struct demo
{
    int i;
    char ch;
} obj = {10, 'a'};
```

if we use initialization list for initialization purpose we have to initialize all the members in sequence in which they are declared.

12) if we want to initialize specific elements in initialization list the some compilers allow this syntax of.

eg:
```
struct demo
{
    int i;
    char ch;
};
struct demo obj = {.ch = 'a'};
```

13) Every name of member of a structre should be different to avoid ambiguity while accessing the member.

if names are same then it generates compile time error

eg:
```
struct demo
{
    int i;   } compile time error
    char i;  }
};
```

14) in same program we can use same identifier name as well as local or global variable as well as structure member.

eg:

```
int i = 11;
struct demo
{
    int i = 10;
} obj = {10};
```

above code is allow according to every compiler because the variable name of i inside the structure is not directly accessible by it's name but it is accessible by its structure's variable name.

15) like normal statement reading, we can also read structure declaration as.

eg:

```
struct demo
{
    int i;
    char ch;
};
struct demo obj;
```

obj is a variable of type demo, which is of user defined datatype i.e structure, which containing two elements of first is integer and 2nd is character.

16) As array is derived datatype we can create array of user defined datatype i.e structure or union.

eg: struct Demo
{
int i;
int j;
};
struct Demo arr[3];

arr is one dimensional array which contains 3 elements, each element is of type Demo, where Demo is a structure which internally contains two elements where first element is integer and 2nd is also integer.

17) we can initialize above array in multiple way such as

i) struct Demo arr[3] = {{10,20},{30,30},{30,40}};
ii) arr[0] = {10,20};
iii) arr[1].i = 11;
iv) arr[1].j = 21;

18) if members of structure are not initialized then it contains it's default values according to the storage class of structure variable not structure declaration.

eg: void fun()
{
struct Demo
{
int i;
};
struct Demo obj;
printf("%d", obj.i); //garbage
}

```
Struct Demo
{
  int i;
}Obj;
Struct demo

int main()
{
  pf("%d", obj.i ); //o
  return 0;
}
```

19) if we have Structure variable to access its members then we have to use . dot (direct accessing operator)

```
eg: Struct Demo
    {
      int i,j;
    };

    Struct Demo obj;
    obj.i = 10;
    obj.j = 20;
```

20) if we have Structure pointer which holds address of Structure object then to access its member we have to use → operator (indirect access operator)

eg. Struct Demo
{
int i;
int j;
};

Struct Demo obj = {10, 20};

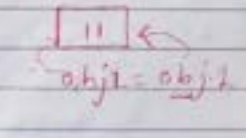
Struct Demo *ptr;
ptr = &obj;

printf("%d %d", ptr->i, ptr->j);

21) we can use assignment operator to assign contents of one structure variable into other structure variable.

But this copy of one structure variable to other structure variable is shallow copy.

Eg: Struct Demo
{
int i;
};
Struct Demo obj1, obj2;
obj1.i = 11;
obj2 = obj1;



printf("%d", obj2.i); // 11

22) we can not use any comparision operator, any relational operator or any equality operator to compare two variables of same structure.
means if we use "==" operator to compare two variables it generates compile time error.

eg: struct Demo
{
int i,j;
};
struct Demo obj1={10,20};
struct Demo obj2={10,20};

if (obj1 == obj2) // compilation Error
{
pf("equal");
}
else
{
pf("not equal");
}

23) as equality operator is not applicable for structure variable then we can apply equality operator on every member of that structure and then we can compare that structure successfully.

eg: if ((obj1.i == obj2.i) && (obj1.j = obj2.j))
{
pf("equal");
}
else
{
pf("not equal");
}

24) allocate memory for structure of variable in two ways
1) static allocation
2) dynamic allocation

eg. Struct Demo
{
int i, j;
};
Struct Demo obj1; // Static memory allocation

Struct Demo *p = (Struct Demo *) malloc
(sizeof(struct Demo));
// dynamic memory allocation

25) we can create structure inside structure in two ways

1) Struct Demo
{
int i;
};
struct outer
{
int i;
int j;
struct Demo var;
} obj;

2) Struct outer
{
int i, j;
Struct Demo
{
int i;
} var;
} obj;

in both the above scenarios we are using concept of nested structure.
But in first scenario our inner structure i.e Demo is declared independently outside the outer structure due to which we can also create

object of that demo structure independently

But in 2nd scenario demo structure is declared inside the outer structure due to which we can not create object of demo structure independently.

26) If we are using the concept of nested structure in which the inner structure is declared inside outer structure then it is necessary to create variable of inner structure immediately otherwise memory for inner structure is not allocated

```
eg: struct outer
    { int i,j;
      struct inner
      {
        int x,y;
      }y;
    }obj;
```

in this scenario sizeof obj is 8 bytes only because variable of inner structure is not created inside the outer structure.

27) we can use same identified name in case of nested structure

```
eg: struct outer
    { int i,j;
      struct inner
      { int i;
      }var;
    }obj;
```

in this eg. member of outer structure i can be accessed obj.i where as member of inner structure i can be accessed as obj.var.i.

1) we can not create variable of same structure inside that structure.

eg struct demo
{
int i;
struct demo obj; // error
};

above syntax generated compiletime error because when compiler tried to create variable obj inside structure demo it generates error

error is generated because at that point compiler is unable to find out sizeof(structure demo because it not yet completely declared

2) we can not create variable of same structure inside structure but we can create pointer of same structure inside that structure because compiler easily predict size of pointer as it is always 4 bytes

eg: struct demo
{
int i,j;
struct demo *p;
};
// allowed

this type of structure is called as self-referential structure.

18 8 15

* unnamed structure:-

1) unnamed structure is a structure which is considered as normal structure but without its name if our program contains unnamed structure then we have to create variable of that structure at the time declaration itself.

2) eg: struct
{
int i,j;
}obj;

3) in this example obj is considered as an unnamed structure.

4) After this structure declaration we can not create any variable that structure because structure identifier is missing

5) incase of unnamed structure it's symbol table entry is not created by the compiler

* Similarly as unnamed structure we can create unnamed union also.

Union is considered as an user defined datatype which contains any primitive datatype, any user defined datatype, any derived datatype.

Syntactical way of writing union & structure both are same but internal memory allocation of structure & union is different.

eg: union demo
{
int i;
char ch;
}obj;

18 8 15

in this union there are 2 members of integer and character according to its size integer requires largest memory according to its size due to which only 4 bytes get allocated for our variable obj.

5) when we create variable of union memory for only for largest datatype get allocated due to which we can only initialize one element of it at a time

6) if we initialize multiple elements then finally union contents the value which is initialized at the end

eg:1) eg: obj.i = 97;
obj.ch = 'b';

according to above syntax character 'b' is the last value in our union due to which the previous value i.e 97 gets override with the ASCII value of 'b'.

7) if we set only single value in union and if we try to retrive by using some diffent member name of union then it gives some value which is initialize previously

eg: obj.i = 97;

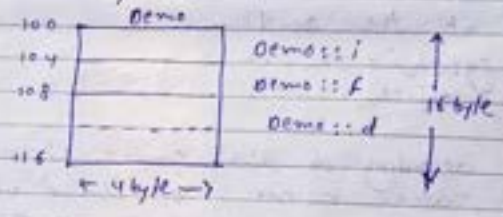
printf("%c", obj.ch); // a

above printf generate outputs as a because 97 is the ASCII value of character a.

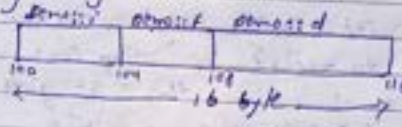
After designing any structure we can draw its diagram in two ways as
i) Diagramatic representation
ii) memory layout

eg struct demo
{
int i;
float f;
double d;
} obj; // 16byte

i) Diagramatic representation



ii) memory layout



* memory allocation strategy for structure or union

1) memory allocation strategy is purely depend on type of compiler and memory management strategies used by operating system.

2) generally on 32 bit compiler and 32 bit O.S every members memory of structure and union gets allocated if word size format.

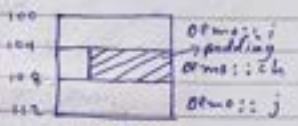
3) WORD is concept which used in Assembly lang. which indicates 4 byte size.

4) means every element of structure gets it's memory in terms of an word size.

5) due to this concept accessing mechanism of Structure or Union becomes faster.

* Consider the below structure we explain the concept of padding

```
Struct Demo
{
    int i;
    char ch;
    int j;
};
```



11 12 byte

padding :- padding is the memory allocation which are allocated due to memory management strategies but that memory locations are not accessible to the programmer.

Generally if our structure contains characters as a data member then the concept of padding gets introduced.

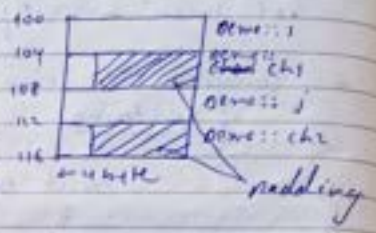
eg: Struct Demo
{
    int i;
    char ch1, ch2;
    int j;
};

In this structure 2nd member is character due to which now 4 byte memory get allocated allocated that character for that 4 byte if one byte is useful for that character

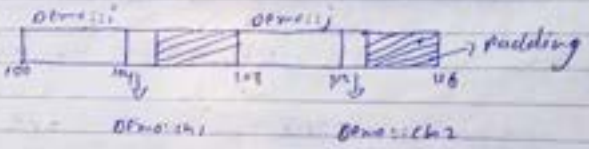
after that char there is another character named as ch2 & this new char require one byte memory we have 3 byte of memory free due to which from that three byte 1 byte get allocated for that char & remaining 2 byte padding.

eg: struct Demo

{
int i;
char ch1;
int j;
char ch2;
};



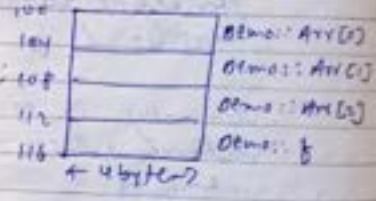
1116 byte



* Structure Contain array:-

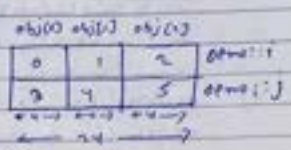
Struct Demo
{
int Arr[3];
float f;
};

1116 byte



1) if Structure Contain array as it has its data member consider array member of array is separate data member.

eg: Struct Demo
{
int i;
int j;
} obj[3];

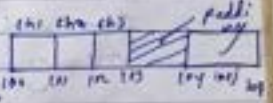


sizeof(obj); // 24 byte.
sizeof(obj[0]); // 8 byte
sizeof(obj.i); // 12 byte
sizeof(obj[0].i) // 4 byte

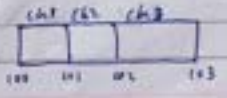
eg. Struct Demo
{
int a[2][2];
int j;
} obj; // 20 byte

1) In this structure 2 dimensional array it use the concept row medium representation we have to apply that representation technique while drawing the layout of that structure.

eg: 1 Struct Demo
{
char ch1, ch2, ch3; int j;
}; // 8 byte



eg. 2 Struct Demo
{ char ch1, ch2, ch3;
}; // 3 byte.



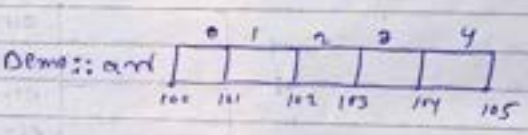
Size of structure may vary if we change the compiler or the OS with architecture by standard memory management strategy

3) if structure contain all characters of its datamember then the rule of WORD size is not applicable for that structure.

4) according to this concept above structure requires 3 bytes of memory without any padding in eg:2.

eg: Struct Demo
{
char arr[5];
} // 5 byte

5) To this a structure also above rule is applicable because it contains array of characters.



6) According to above all the examples which are concern with size of structure there is a concept of padding.

7) As every structure contains the padding means our every structure variable waste our memory

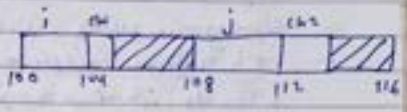
8) Due to this concept of padding it may generate some problems on embeded platform where memory is less.

9) To avoid this drawback of padding we have to use custom packing request technique.

eg ~~#pr~~

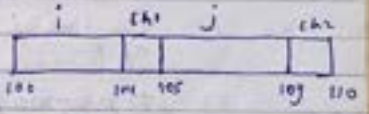
10 To apply this concept of packing we have to use preprocessor directive named as #pragma pack

eg: struct demo
{
int i;
char ch1;
int j;
char ch2;
};



11 This structure required 16 bytes memory from that 16 bytes 6 bytes memory gets wasted for padding means actual 10 bytes memory is useful for storing a data to avoid this wastage we can rewrite our structure as

#pragma pack(1)
struct demo
{
int i;
char ch1;
int j;
char ch2;
};



12 According to this diagram & concept size of our structure is 10 byte

13 when we write #pragma pack ; in our structure request of 1 byte packing is applicable to every structure which is declared after that prototype.

14 if we want to stop packing request we have to write syntax at the end of declaration of -

#pragma pack()

15 #pragma pack concept should be use before structure declaration otherwise there is no use of that concept

16 if we use #pragma pack at the time of creating a variable of that structure then its concept is not applicable because how much memory shall be allocated for that structure is already decided at the time declaration itself.

eg: Struct demo
{
int i;
char ch;
char ch2;
int j;
};
#pragma pack(1) ← not applicable
Struct demo obj;
sizeof(obj); // 12 byte

17 we can provide any parameter to #pragma pack of 1 or 2 or 4 or 8 or 16.

18 if we provide some other values then compiler ignore that parameter and use it's default packing concept of 4 bytes.

3) The concept of packing is also applicable for union also.

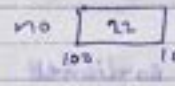
Binary Numbering System and its Concept

1) Binary numbering system is such a numbering system which is used internally to store data in memory.

2) To understand the concept of binary numbering systems we have to consider its Base and range Base of binary numbering system is 2 and range is 0 to 1.

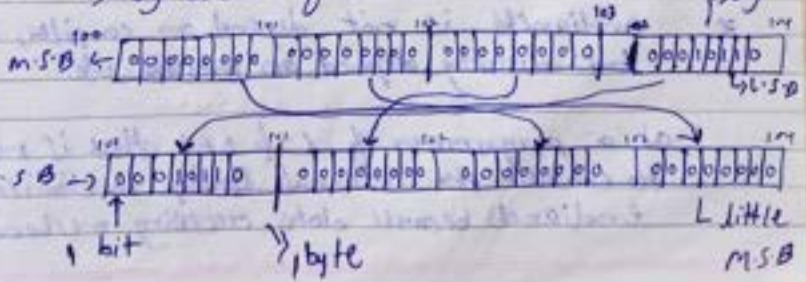
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^9	2^{10}
256	128	64	32	16	8	4	2	1	512	1024
				1	0	1	1	0		

Consider integer variable no which declared as int no = 22;



To store that no in memory it require 4 bytes according to storage mechanism of multibyte datatype there are 2 ways in which we can store our data in memory of

1) little endian format
2) big endian format



* L.S.B - least significant Bit

it is a bit having least magnitude.

* least significant Byte:-
if Byte contains least significant Bit then it is called as least significant Byte

* M.S.B - most significant Bit

* it is a bit certain highest magnitude

* most significant Byte:-
if byte contains most significant Bit then it is called as most significant Byte

* little endianess:-
it is an endianess in which L.S.B is stored at lower addr. and MSB is stored at higher addresss.

eg. intel processor family, some of the process and family uses little endianess.

* Big endianess:-
it is an endianess in which m.s.B is stored at lower addresss and L.S.B is stored at higher addresss.

eg: arm processor
and some of the processor and family uses Big endianess.

* endianess is not depend on compiler, O.S but directly depend on processor use

a programmer of C++ there is no need to consider internal storage mechanism of endianess because data accessing mechanism

of any of the endianess is exactly same.

if we interact with Assembly language then Endianess important means endianess is just way of storing the data internally.

* Bitfield in Structure or Union

1) According to previous topic of #pragma pack we can save the memory which gets wasted due to the padding.

2) Consider the below layout of a structure which is used to store date, month & year.

```
Struct calender
{
    int date;
    int month;
    int year;
};
```

3) As this structure certain 3 integer variable it requires 12 bytes of memory i.e 96 bits.

4) As this structure is used to store date the range of every data element should be fixed.

5) our date variable can store values of 1 to 31
month variable stores values of 1 to 12
year variable stores values of 2000 to 2040
as a range is fixed instead of taking 4 byte i.e 32 bit for every variable we can only take required no of bits.

7) to store date are 6 bit enough
to store month 5 bit are required we can
store year value 13 bit enough

8) To apply this concept our modified syntax should be

eg. #pragma pack(1)

```
Struct calender
{
    int date:6;
    int month:5;
    int year:13;
};
```

when we calculate size of structure we get 3 byte
means we can save 9 byte.

* Some important points about bitfield.

1) we can not set -ve or zero value of a bitfield

2) we can not set bitfield to float or double

3) we can not set bitfield to an array.

4) we can not set bitfield value which is greater
than sizeof that datatype

```
Struct demo
{
    int i:0;    -① // error
    int j:-3;   -① // error
    float f:6;  // error  -②
    double d:7; // error  -②
    int arr[7]:4; // error  -③
```

```
int x:35 ; // error   -②
};
```

all the above syntax generates error

* if structure contains bitfield then we can not apply & (address of operator) on the members of that structure.

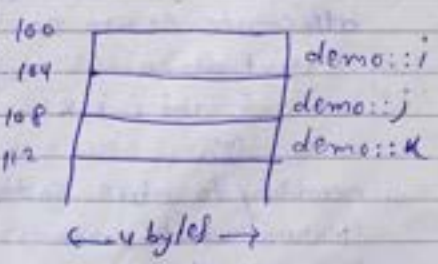
```
eg: struct demo
    {
      int i:7;
    };
    struct demo obj;
    printf("enter value");
    scanf("%d", &obj.i );// error

int *p = &obj.i; // error
int *p = &obj; // allowed
```

22/8/25

All the above points are applicable as it is for union also means we can use only concept such as #pragma, bitfield for union also only the major difference between union & structure is according to memory allocation
- when we create structure which contains some data elements & when we try to fetch address of that data it shows in sequential manner

eg: struct demo

```
{
  int i;
  int j;
  int k;
};
struct demo obj;
```

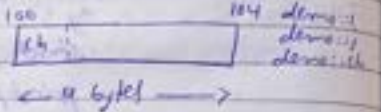


Consider base address of that object is 100 when output of following printf() statements

```
printf("%d", &obj.i); //100
printf("%d", &obj.j); //104
printf("%d", &obj.ch); //108
```

if we consider union and its data members then address of every element is same because there is no separate memory allocation for every element

eg: Union demo
{
int i;
int j;
char ch;
};



Union demo obj;
consider base address of obj is 100 Then output of below printf() is

```
printf("%d", &obj.i); //100
printf("%d", &obj.j); //100
printf("%d", &obj.ch); //100
```

* Usedefined Datatypes:-
Enumeration:- 1) Similar as Structure & union, enum is also consider as usedefined datatype 2) Enum is also called as an enumeration constant because the value get stored into enum is always constant 3) As like structure & union we can also create enum as usedefined datatype

eg: struct demo
{ int i,j,k;
};

enum hello
{ i,j,k;
};

4) according to above syntax demo is considered as a structure which contains 3 integer coordinates as i,j,k;
5) Similarly hello is name of enum where i,j,k are considered as enumeration constants or integral constant

6) if structure contains multiple elements then separate memory is allocated for every elements of structure but in case of enum there is no separate memory allocation is required 7) Generally memory for enum is allocated inside text section of process 8) According to above we can not apply size of operator on variable of enum.
eg enum demo {a,b,c};

9) In this enum every variable a,b,c are not initialized with any value due to which 1st variable of enum gets its default value start from zero & every other variable gets values incremented by 1.

a; //0 b; //1 c; //2

10) enum demo {a=5, b, c};
In this enum 1st variable is initialized with value 5 due to which concept of default initialization is not used

11) As 1st variable is initialized with value 5 due to which next variable gets its next value i.e 6 & so on

a; //5 b; //6 c; //7

12) enum demo {a=7, b, c, d, a};
above enum generates error because a variable declared twice in enum which is not allowed because it generates ambiguity means same named variable is not allowed in enum.

13) enum demo {a=11, b, c=11};
In this enum both variables a & c gets its value as 11 which is allocated means same value can be used in enum

14) enum demo {a=3+2, b, c};
In this case 1st variable is initialized with only expression which gets evaluated is 5 due to which remaining variable gets its value 6 & 7

a; //5 b; //6 c; //7

* To initialize enum variable we can use any expression that should evaluate to an integral constant.
enum demo {a=-6, b, c};
To enum variable we can store -ve value also

a; //-7 b; //-6 c; //-5

we can not fetch addel of any variable from text section

7) enum {a,b,c};
This enum is consider as unnamed enum and similar like structure & union

8) enum demo {a,b,c} obj;
In this enum obj is variable of type enum demo.
According to the above syntax our object variable can be initialize with a values if enum variable only otherwise it generate error

```
obj = 1; // error
obj = b; // allowed
```

9) enum demo {a,b,c};
```
a = 15;  // error
b++ ;  // error
```

Above two statments generate compiletime error because we cannot change value of enumeration variable

10) enum demo {a,b,c}
```
int *p = &a;  // error
```
As we can not fetch address of enum variable it generates compiletime error.

11) enum demo {a = 'a', b, c}
above syntax is allowed because 1st variable a gets initialized to ASCII value of a; i.e 97 which is integral constant
```
a; // 97
b; // 98
c; // 99
```

12) enum demo {a, b, c};
struct hello
{
int i;
enum demo obj;
};

Any structure / union contains object of enumeration constant. This member of structure should be initialized with value of enum only.

13) struct demo obj;
obj.j = 11;
obj.obj = 'a';
obj.obj = 15; // error

/* typedef at the last */

/* String in C:-

String is not considered as a direct datatype

1) In c++, string is considered as an existing hierarchy class which contains some predefined functions.

2) In java, string is considered as datatype. In C & c++ string is considered as character array which ends with '\0'.

3) '\0' is considered as a delimiter for the string

4) There are multiple ways in which we can create string in c & c++ as.

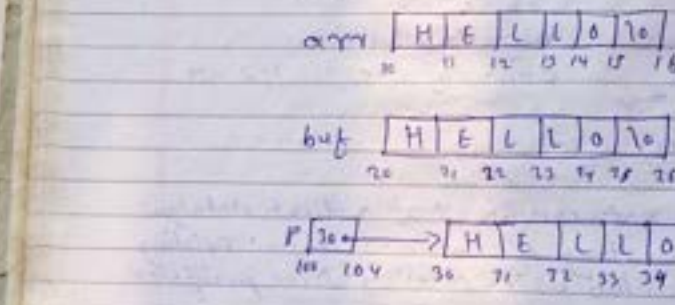
eg. 1) char arr[] = {'H', 'e', 'l', 'l', 'o', '\0'};
2) char buf[] = "Hello";
3) char *p = "Hello";

In this 3 ways we can create string in c or c++

5) if we use 1st way for creation of a string, we have to insert '\0' at the end explicitly. But this explicit insertion of '\0' is not need in case of syntax 1 & syntax 3 because it gets implicitly added by compiler.

6) If string is created by using member initialization list then '\0' is expected at the end of string.

7) But if string is enclosed is " " then there is no need to insert '\0' at the end.



- According to all the above diagrametic representation every string requires 6 bytes of memory '\0' is initiated at the end of every string.

* strlen:- it is a string library function which accept base address of string & it counts no of characters from that base address till we get first '\0'.

* sizeof:- it is a unary operator which gives the no of bytes which should be allocated for every datatype sizeof operates returns the no of bytes by taking help of symbol table, datatype table.

```
int i=10;
sizeof(i); //4 info fetched from symbol table
sizeof(int);//4 info fetched from datatype table
```

Consider the above syntax of string & predict the o/p

```
sizeof(arr); //6
strlen(arr); //5
sizeof(buf); //6
strlen(buf); //5
sizeof(p); //4  due to pointer
strlen(p); //5
sizeof(*p); //1
strlen(ptr);//3
```

As we have syntax we can create string from that 3 syntax. we can reinitialize the string by using 3rd syntax only as

```
p="world";
arr[]={'a','b','a'}; //error
arr="world"; //error
```

In above all syntax arr we can only change contents of string in 1st syntax as

```
arr[1]='x';
```

23/8/15

23 & 15

* 'C' Preprocessor

1) Preprocessor is a first component of ~~x86~~ tool chain which gets involved after editor in a tool chain.

2) preprocessor is a program which accepts in high level programming language such as C and C++ and convert that input into expanded source code.

3) instead of scanning whole program preprocessor scans only such lines which starts with # Symbol

4) According to this '#' Symbol gives direction to the preprocessor due to which '#' is called as preprocessor directive symbol

5) This preprocessor program performs multiple tasks such as :
   1) Tokenization and whitespace removal
   2) file inclusion
   3) Macro Expansion
   4) Conditional compilation

6) After performing all this task our expanded source code is still in high level programming language, then that source code is passed towards the compiler for remaining processing.

* 1) Tokenization and whitespace removal :-

a) Tokenization is a first process which is performed by preprocessor.

b) the process of tokenization our preprocessor accepts statement of expression from our source code

c) that statement gets converted into multiple words and that word is called of token. and above process is called of tokenization.

d) After performing the process of tokenization our token gets forwarded towards the compiler.

e) in this phase of preprocessor, preprocessor removes extra whitespaces which may contain normal space \n, \t in our program, that all things gets removed by the preprocessor.

f) if a programmer writes a comment in a program then all that comments gets removed by the preprocessor.

g) after passing through this phase of a preprocessor our program is considered as a normal file according to an operating system i.e "unformatted uniform stream of bytes".

Hello.c

```
#include<stdio.h>
int main()    // main function
{
    printf("Hello");
}
```

↓

Hello.i

#include<stdio.h> int main() { printf("Hello"); } ← tokenization

↓

add contents in .i file

file inclusion

2) file inclusion :-

1) This is a preliminary task of preprocessor which is used to include the contents of header file in our source file.

2) Header file is a such a file which is normal text file which contains
   i) prototypes of a function
   ii) declarations of user defined datatype of structure, enum, union, class
   iii) typedefs which are used in our program
   iv) some user defined MACROS (#define)
   v) header file inclusion for other header file.

3) when preprocessor gets #include keyword in our program then the preprocessor search that specified header file and copy the contents of that header file in our source file.

eg: Hello.c

```
#include (stdio.h)
int main ()
{
    pf ("Helloworld");
}
```

↓

Hello.i

```
int printf (const char * format, ...);
int scanf (const char * format, ...);
#define NULL 0
int main ()
{
    PF ("Hello word");
}
```

} contents of stdio.h

4) According to above code snippet after preprocessing the contents of stdio.h headerfile gets added in our source code.

5) There are two syntactical ways in which we can include user headerfile such as

1) #include <Demo.h>
2) #include "Hello.h"

6) if filename is ~~written~~ written in angular bracket then preprocessor search that file in compiler specified or system specified path.

7) if file name is written in double code then compiler search that file in current directory of a process

8) if that file is not found in a current directory then preprocessor search that file in compiler specified path.

* current directory :- current directory of process is a directory in which our parent process resides

* parent directory :- parent directory of a process is directory in which our file resides.

#3 MACRO Expansion:-

1) There are 2 types MACRO such as

1) predefined MACRO - provided by Compiler
2) user defined MACRO - provided by programmer

3) Every type of MACRO is divided into 2 types as.

i) object like MACRO:-
eg: #define MAX 10

ii) function like MACRO:-
eg #define MULT(X,Y) X*Y.

4) we have to use MACRO for 2 reasons such as
i) A big MACRO program become much more readable
ii) If we want to change value of any MACRO then there is no need to scan whole program but we just change value of one position it gets automatically replaced in whole program by preprocessor itself.

5) MACRO is divided in to two parts such as macro template & MACRO definition
eg: #define MAX 10
↓ MACRO template
→ MACRO definition

6) To perform this MACRO expansion table preprocessor creates one data structure named as MACRO expansion table which contains 2 columns in it.

7) first column contain MACRO template & 2nd column containg MACRO defination.

MACRO template	MACRO definition
MAX	10
NULL	0

8) while performing the task of MACRO expansion preprocessor replace every macro template of our program with its definition by taking help of macro expansion table.

eg: Hello.c

```
#define INTEREST 9
void calculate (int no)
{
    int finalAmount = 0;
    finalAmount = (no * INTEREST)/100;
}
```

↓

Hello.i

```
Void calculate ( int no)
{
    int finalAmount = 0;
    finalAmount = (no * 9) /100;
}
```

9) Below eg explains the concept of function like macro.

Hello.c

```
#define multi(x,y) x*y
int main()
{
    int no = 0;
    no = MULT(10,20);
}
```

↓

Hello.i

```
int main()
{
    int no=0
    no = 10*20;
}
```

eg: The is no such concept of scope of MACRO.

we can define same MACRO multiple time which is called as redefination of MACRO

eg:

```
void fun ()
    {
        #define MAX 10
        pf ("%d", MAX); 10
    }
gun ();
pf ("%d", MAX);   10
}
void gun ()
{
    #define MAX 20
    pf ("%d", MAX); 20
}
int main ()
{
    fun ();
    return 0;
}
```

10) whenever redfine the MACRO in a program that is no seperate entry gets added in MACRO expension table but it only replace old defined one with new defination.

11) when preprocessor gets #define keyword new entry gets added in MACRO expension table and when it gets #undef keyword that entry gets remove from MACRO expension table.

after getting #undef we can not use that MACRO because it gets removed from MACRO expansion table

```
void fun()
{
    #define MAX 10
    pf("%d", MAX); //10

    #undef MAX
    pf("%d", MAX); //error
}
```

we can also write MACRO which contains multiple lines init.

If we want to include next line in our MACRO then at end of previous line there should be \ at the end.

eg.

```
#define MAX 10\
printf("Hello");
```

in this example printf statement is also included in a MACRO.

* Some predefined MACROS in C and C++

i) Stringized MACRO:-

This MACRO is used to convert our parameter into string.

eg:

```
#define Demo(Str) #Str
int main
{
    pf( Demo(Helloworld));
}
```

↓

```
pf("Helloworld");
```

ii) Concatination MACRO
this MACRO is used to concat input arguments
eg: #define Demo (x,y) x##y

```
int var = 10;
pf ("%d", Demo(var));
↳pf ("%d", var);
```

iii) MACRO which give filename of a program

eg printf("%s", __FILE__);
// Demo.c (filename)

iv) MACRO which gives the line no

```
14   eg:
15        printf("%d", __LINE__ ); //15
16
```

↓ (line no)

v) MACRO which gives the System time

printf("%s", __TIME__);
// System time
(11:20 Am)

vi) MACRO which gives current DATE

printf("%s", __DATE__); // System date
23 Aug 2015

vii) MACRO which is used to check our compiler is Standard compiler or not

```
if(__STDC__ == 1)
    pf("Std compiler");
```

* miscellenious directive of preprocessor.

1) #pragma pack()
This MACRO is used to instruct compiler about memory allocation techniques.

2) #pragma warn -rvl
This MACRO is used to display the warning or hide the warning
rvl = return value required warning

3) #pragma startup fun
this pragma startup MACRO is used to call the function before calling main.
in this syntax fun function gets called before main

```
void fun()
{ // code
}
```

4) #pragma exit fun
This exit pragma directive is used to called the function after terminating main function

4 Conditional Compilation:

1) Concept of conditional is used to instruct the compiler that which part of code should be compiled and which part of code should not be compiled to used this concept there are multiple MACROS available such as

#if
#endif
#else
#elif

eg. int main()
{
int no=5;
#if (no==5)
{
pf("Hello");
}
#else
pf("world");
#endif

in the above program pf("world") statement is not the compile by compiler because condition is false.

eg. #define MAX 10

#ifdef MAX
pf("Hello");
#endif

~~#ifdef and #ifndef~~

* #ifdef and #ifndef are used to check whether the specified macro is defined or not in program.
of above syntax of MAX macro is define due to that the below printf statement is compiled successfully.

* operator in "C"

1) Expression :- Expression is a statement which gets internally evaluated by the processor

Expression contains more than one operator & more than one operands

2) operator :-

operator is considered as a symbol which decides the action taken on the identifiers

3) operands :-

operands are such entity on which we have to perform the operation.

*) Types of Expression :-

There are two types of expression according no of operators in it.

1) Simple expression :-

if Expression contains single operator in it then it is called as Simple Expression.

eg: no = 15;
operand ← → operator

2) Compound Expression

if expression contains more than one operator init then it is called as compound expression.

eg: Ans = a+b;

2/8/15

* precedence of an operator:-

if expression contains more than one operator in it then precedence of an operator decides the sequence in which subexpression gets evaluated

* Associativity of an operator:-

if expression contains multiple operators in it having same precedence then associativity of an operator decides in which sequence we have to evaluated our sub expression i.e L->R OR R->L

* Types of operators:-

operators are classified according to two types

1) Types of operators according to no of operands

according to the no of operands operators are divided into 3 types of.

1) unary operator - required single operand
2) binary operator - requires two operands
3) ternary operator - requires three operands

2) Types of operators it depend on its behaviour or operations

1) Arithmatic operators:-
eg: +, -, *, %, /.

2) relational operators:-
eg. <, <=, >, >=,

3) equality operators
eg. ==, !=

4) logical operators:-
&&, ||, !

5) bitwise operators:-
&, |, ^, ~

6) misclleneous operator
*, →, sizeof, (), [], .

7) Shorthand assignment operators:-
+=, -=, *=, /=, %=, <<=, >>=

8) Shorthand operator
++, --

9) Shift operators:-
<<, >>

precedence & Associativity table

precedence	operators	type	Associativity
1	() [] -> . ++ --	miscellaneous	L -> R
2	+ - ! ~ * & sizeof	unary	R -> L
3	* / %	multiplicative	L -> R
4	+ -	Additive	L -> R
5	<< >>	shift operator	L -> R
6	< > <= >=	relational	L -> R
7	== !=	equality operators	L -> R
8	&	bitwise and	L -> R
9	^	bitwise EX-OR	L -> R
10	|	bitwise OR	L -> R
11	&&	logical and	L -> R
12	||	logical OR	L -> R
13	? :	Conditional	R -> L
14	=, +=, -=, *=, /=, %= <<=, >>=, &=, ^=	Assignment	R -> L
15	,	comma	L -> R

Associativity
→R
→L
→R
→R
→R
→R
→R
→R
→R
→R
R
R
→L

1) Miscellaneous operators :-

Miscellaneous operators are sub operators which are provided by language designer

a) ( ) - this operator generally used to increase precedence of operator

eg: Ans = (a+b) * d - c

in above expression according to the normal precedence rule multiplication operator evaluate first but due to the ( ) bracket first + operator evaluated first

if any expression contains any sub expression in a ( ) bracket then that sub expression gets evaluated first without considering the precedence rule.

operator
Expression - Ans = x + (y - z) ← Sub expression
operands

in above compound expression ; sub expression i.e (y-z) gets evaluated first before + operator.

b) [ ] :- Subscript operator -

this operator generally used in 'C' programming language if we want to declare an array or if we want to access any element of array.

eg. int arr[7];
printf ("%d", arr[2]);

c) '.' operator (direct access operator)
it is used to access the members of structure and union if we have variable of that structure

d) '->' operator (indirect access operator)
it is used to access members of structure and union if we have pointer which points to structure or union

```
eg: struct demo
    {
        int i, j;
    };
    struct demo obj;
    struct demo *p = &obj;
        obj.i = 11;
        p->i = 21;
```

i) we can use . & -> operator in different manner such as
```
obj.i = 21;
p->i = 21;
(&obj)->i = 11;
(*p).i = 21;
```

(e) ++ & -- (Shorthand increment or decrement op)

```
eg: int no = 10;
    no++;
    no = no+1

    no--;
    no = no-1;
```

increment and decrement operator is divided into 2 types as

4) postincrement or ~~postdecrement~~ postdecrement
5) preincrement or predecrement

this two types are applicable only if increment or decrement operator used in expression

if it used independently then both this types of pre & post behaves in a same manner

eg: the below eg indicates diffrence bet" preincrement and postincrement operator as

1) int no = 10;
   no++; // 11

2) int no = 10;
   ++no; // 11

3) int no = 10; ans = 0;
   ans = no++;
   printf("%d%d", ans, no); // 10, 11

4) int no = 10; ans = 0;
   ans = ++no;
   printf("%d%d", ans, no); // 11 11

according to above four example in first & 2nd example there is no diffrence preincrement and postincrement but if same concept is used in Expression then preincrement & postincrement are considered as a diffrent concept.

* all the concept are applicable for decrement operator also. *

2) Uniary operators

all this operators are uniary operator because this operator requires single operand.

a) +, -, :- this operators are used to assign sign to our variable

eg: int no = 10, ans;
ans = +no;
ans = -no;

b) ! ~

both operator called as not operators but ! is considered as logical not operator where as ~ is considered as bitwise not operator.

logical not operator not going to change any of the data whereas bitwise operator change state of our data

eg. if (!(no<10))
{
//code
}

int no = 10
ans = ~no;

as this is a bitwise operator first we have to consider bit representation value 10

0 0 0 0 1 0 1 0
~ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
1 1 1 1 0 1 0 1

(c) * & (dereference operator and address of operator)

& operator is called as address of operator which is used to return the virtual address of any identifier used in our program.

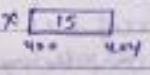
& operator gives the virtual address which is available in symbol table.

virtual address is such address which is provided by the compiler at the time of compilation but it is not an existing address.

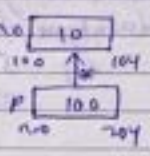
* i.e dereference operator which is used to return the contents whose address is stored in our variable.

Consider below syntax & symbol table to understand the concept of & and * operator

```
int x = 15;
```




```
int no = 10;
int *p = &no;
```



Symbol table:-

Name	address	value	size	from	to	another name
x	400	15	4	6	24	-
no	100	10	4	6	24	-
p	200	100	4	6	24	-

when we use name of identifier only then parser search it name in name column of symbol table and return the contents of value column

1) printf("%d", x); // 15

2) if we use & operator before identifier name then processor search that name in name column & returns the contents of address column

eg: printf("%d", &no); // 100

3) if we use * dereference operator then first we have to search the name in name column, fetch the value of correspondingly that name

- that value should be search in address column of symbol table and returns its correspondingly contents from value column

eg: printf("%d", *p); // 10

4) sizeof is unary operator which returns its result by considering the contents of symbol table or datatype table.

eg: sizeof(no); // 4 consider symbol table
sizeof(double); // 8 consider datatype table

```
int no = 10;
printf("%d", sizeof(++no)); // 4
printf("%d", no); // 10
```

the expression written inside sizeof operator is evaluated but its result not reflected in original operand

3) multiplicative operators :-

(a) * - this is binary operator which gives multiplication of two operands.

eg: int ans = 0, a = 5, b = 2;
ans = a * b; // 10

(b) / - division operator :-
ans = 10 / 3 // 3

(c) % - it gives the remainder which gets generated after division

eg: no = 10 % 3 ; // 1
no = -10 % 3 ; // -1
no = -10 % -3 ; // -1
no = 10 % -3 ; // 1
no = 10.0 % 3 ; // error

mod (%) operator is only applicable for integral constant it is not applicable for floating point variables

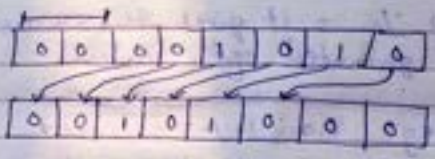
4) Additive operators :- (+, -)

no = 10 + 3 ; // 13
no = 10 - 3 ; // 7

30 | 12

5) Shift operator:- This type of operator are used to shift bits of our either in left direction or right direction.
generally this shift operator are used to manipulate our data & modify that data

i) << left shift operator:-
int no = 10
no = no << 2;



if we shift our no in left direction by n no of bits then n empty bits created on right side

always empty bits should be initialize with zero only.

we can directly calculate result of shift operator by using formula as x<<y

$$x << y = x * 2^y$$

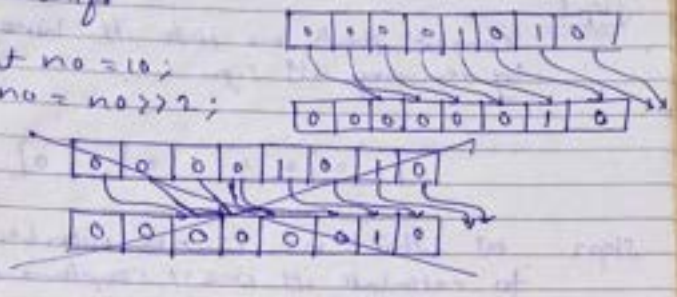
in our eg. x is 10 & y is 2 due to which
$10 << 2 = 10 * 2^2$
$= 10 * 4$
$= 40$

we can not apply any shift operator on floating point no. because floating point no are not stored in a sequential format

12

2) >> right shift

```
int no = 10;
no = no >>> 2;
```



if we shift our no to right side magnitude of our no gets decreased.

if we want to calculate result by using formula then consider

$$[x >>> 2 = x/2^y]$$

in our example x is 10 and y = 2 due to this

$10 >>> 2 = 10/2^2$

$= 10/4$

$= 2$

* negative no representation of RAM

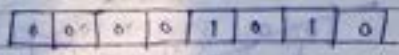
1) if we store the no then that no gets stored by directly converting into its binary format

2) if our data is -ve then processor performs some special task while storing that no into RAM

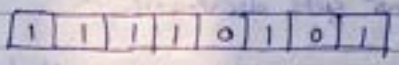
eg: Consider no as -10

```
int no = -10;
```

Step 1
convert that no into its binary representation by removing its sign.

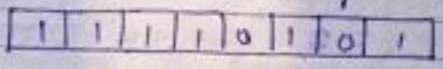


Step 2. as that no is +ve number first we have to calculate its one's complement by toggling every bit of a number.

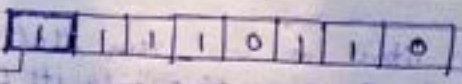


Step 3. apply the concept of 2's complement on above data.

incase of 2's complement we have to add one in our actual data.



+ 1



Signbit

the output which is generated by the above step gets store in memory.

after Step 3 the first bit of no. one which is our actual data but that bit is used the represent whether our no is +ve or -ve.

if first bit is one then our no is consider -ve otherwise consider +ve.

the below steps are performed when we read the no. whose first bit is one i.e sign bit is on.

Step1 Consider that no as it is which gets stored in a memory.

1	1	1	1	0	1	1	0

Step2 apply one's complement logic on above no.

0	0	0	0	1	0	0	1

Step3 apply concept of 2's complement of above no

0	0	0	0	1	0	0	1

+1

0	0	0	0	1	0	1	0

as this no becomes -10 when we try to print that no on screen it display -10 because 1'st bit of no is one when we retrive that no from RAM.

* diffrence betn signed & unsigned variables

signed & unsigned are considered as datatype modifier due to which range of our datatype gets modified.

Consider the example of character which requires 8 bit.

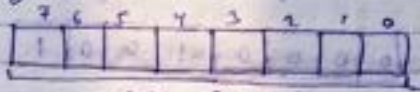
if our character is signed then we can use only 7bits from that 8 bits because one bit is reserved for signed representation

char datatype

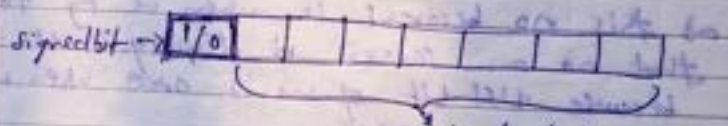
signed unsigned

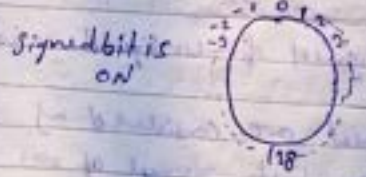
consider the example of character as.

1) unsigned character



2) Signed character





6, Relational :- this operator are binary operator which are to check relationship b/t two operator result of this two operator is either true or false

```
if (10>7)
  {
    printf("yes");
  }
else
  {
    printf("No");
  }
```

in above example relational operator is used in if condition and condition is true over control comes inside if block

7. Equality operator :-
this two has == & != are used to check whether contents of both the operands are equal or not.

Similar as the concept of relational operator it returns either true or false depend on equality.

8.9.10 bitwise (&, |, ^) And, OR, EX-OR :-
all this operator are use to manipulate bits of our data.

op1	op2	&	|	^
1	1	1	1	0
0	1	0	1	1
1	0	0	1	1
0	0	0	0	0

it is not truth table

int no1 = 15, no2 = 21, Ans = 0;

convert no1 into binary

2	15	
2	7	1
2	3	1
2	1	1
2	0	1

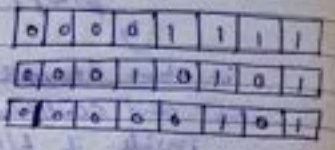
2	21	
2	10	1
2	5	0
2	2	1
2	1	0
2	0	1

1) Ans = no1 & no2

no1 = 0 0 0 0 1 1 1 1

& no2 = 0 0 0 1 0 1 0 1

Ans = 0 0 0 0 0 1 0 1



printf("%d", Ans); // 5

2) Ans = no1 | no2

no1 = 0 0 0 0 1 1 1 1

0 0 0 0 1 1 1 1
0 0 0 1 0 1 0 1
0 0 0 1 1 1 1 1

printf("%d", Ans); // 31

3) Ans = no1 ^ no2

0 0 0 0 1 1 1 1
^
0 0 0 1 0 1 0 1
0 0 0 1 1 1 0 0

printf("%d", Ans); // 26

11,12) logical operators

example to demonstrate concept of precedence & associativity

Ans = A+B*C && x || y-(z+7)%3

out1 = z+7 Ans = A+(B*C) && x || y - out1 %3

out2 = B*C Ans = A+out2 && x || y - (out1 %3)

out3 = out1%3 Ans = (A+out2) && x || y - out3

out4 = A+out2 Ans = out4 && x || (y - out3)

out5 = y-out3 Ans = (out4 && x) || out5

out6 = out4 && x Ans = out6 || out5

out7 = out6 || out7 Ans = out7

logical operator apply on the condition this operators not applicable on any of the data

there are 2 logical operator in c such as && || both this operator are binary operators due to which it requires two operands. both the operands can be an expression the result of our operator is depend on truth or falsity of operand1 & operand2

To apply the concept of logical operator we have to consider truth table of

op1	op2	&&	||
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

incase of logical operators preprocessor internally uses short circuit evaluation such as

i) if operation to be performed is logical && then

a) if the result first operand is true then we have to check result of 2nd operand

b) if result of first operand is false then there is no need to check the result of 2nd operand because overall result become false

ii) if operation to be performed is ||

a) if result of first operand is true then there is no need to check the result of 2nd operand because overall result is true

b) if result of first operand is false then we have to check result of 2nd operand.

eg: int no1=10, no2=7, no3=6;

(1) if ((no1==10)^T && (no2!=6)^T)
{ pf("yes");
}
// yes

(2) if ((no1==10)^T || (no2--)^{not evaluated})
{
pf("%d", no2); // 7
}

13 conditional operator :-

this is only one ternary operator in C means it requires 3 operands

this operator is termed as an conditional operator because everything which can be written by using if else construct can be written by using ternary operator.

Consider the below example of if else condition it corresponding conditional operator

```
if (no==5)                    (no==5)?(A=7):(A=6);
{
  A=7
}
else
{
  A=6
}
```

Similarly as above example we can also convert nested if else into an ternary operator such as.

```
if (no==7)                    (no==7)?((X=5)?(A=5):(A=6))
{                                     :((Y==7)?(B=7):B=6);
  if (X==5)
  {
    A=5;
  }
  else
  {
    A=6;
  }
}
else
{
  if (Y==7);
  {
    B=7;
  }
  else
  {
    B=6;
  }
```

(no==7)?((x==5)?(A=5):(A=6)):((x<7)?
(B=6):(B=7));

14) Assignment & Short hand operator

if we want to assign value to of one operand to another operator then assignment operator should be use

eg: int no1=10
no2 = no1
assignment

there are some shorthand assignment operator such as

C=C*D	A=A/B	A=A+B	A=A-B	A=A<<2
C*=D	A/=B	A+=B	A-=B	A<<=2

B = B!2
B!=2

15) ',' Comma operator:-

Comma is a such operator which is not always considered as a operator but some place it is considered as a separator.

in below example , is worked as a separator.
eg: int a[ ] = {10, 20, 30, 40, 50};

fun(10, 20)

void add(int a, int b);

The below example ',' is work as operator

int a = 10, 20, 30, 40; // 10

int a = 10

int b = (10, 20, 30, 40); // 40

int b = 40

22/6/22

Continue

Typedef:-

1) Typedef is not a storage class.

2) Typedef is concept which is used in C & C++ which is used to assign some diffrent name to an existing datatype

3) by using typedef we can not create new datatype

4) when we create typedef it entry gets added into datatype table which is used to manage all information concern with the datatype.

5) typedef int Integer;
According to above syntax Integer is considered as new name for datatype int

6) after this typedef statement can use Integer as a datatype
Integer no = 10;
above statement is considered as
int no = 10;

7) we can apply concept of typedef for any primitive datatype, any derived datatype, any user defined datatype.

eg: Structdemo
{
int i;
int j;
int k;
};

8) If we want to create variable of structure demo then we can create variable as
Struct demo obj;

9) According to this syntax we have to write Struct keyword & name of struct everytime to avoid this we can create its typedef as
typedef Struct demo newtype;

10) According to this syntax newtype is new name for existing datatype named as Struct demo after this syntax we can create variable of that structure as newtype obj;

11) we can create typedef for structure & union at the time of declaration itself
eg typedef Struct demo
{ int i;
int ji
} new_type;

12) newtype is not object for Struct demo but it is typedef for Struct demo As it is a typedef we can create variable of structure as new_type obj;

13) If we are creating typedef for structure & union at the time of declaration then we can create that structure as unnamed structure also
eg typedef Struct
{
int i, j;
} new_type;

14) According to above syntax new_type is typedef for that unnamed structure

15) Using typedef we can create multiple names for some datatype such as

```
typedef struct demo
{ int i,j;
}newtype1, newtype2;
```

In this syntax, newtype1 & newtype2 are considered as new type names for structure demo

a. We can compare the concept of #define with typedef

eg typedef int *iptr1;
#define iptr2 int*

iptr is considered as datatype like int* where as in case of #define int* is an replacement text for keyword iptr.

```
iptr1 p,q;
iptr2 x,y;
```

According to above syntax p & q are considered as 2 variables of type int*

where as x is a variable which is considered as int* & y variable is only considered as int

Accordingly to above diffrence, we can conclude that typedef is applicable for every variable that we are creating but #define is not applicable for all

* There are some major diffrences between #define & typedef just considered as keyword.

1) all typedef are handled by compiler where as #define handled by preprocessor

2) entry for typedef gets added in datatype table where as entry for #define macro gets added in macro expansion table

3) Replacement of macro is just blind replacement where as replacement of typedefs is logical replacement

```
typedef int (*FP) (int,int)
        FP (X,Y)
int add (int i, int j)
{
    return (i+j);
}
int sub (int i, int j)
{
    return (i-j);
}

X = add;
Y = sub;
X (10,20) ; // call to function add
Y (20,10); // call to function sub
```

In above syntax FP is not considered as a variable of fun^n pointer but it is considered as new datatype which is similar as function pointer which stores address of such a function which accepts 2 parameters both are integer & it return integer.

Module 1

Basic Description of kernel module

```
// Hello1.c
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
    printk(KERN_INFO "module is loaded");
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_INFO " module is removed");
}
```

1) this headerfile is required for the flags which are used in our module.
   init_module and cleanup_module() function prototype is written in that header file

2) this header file contains some predefined symbols of kernel which are used at runtime.

4) this function gets automatically invoke by insmod utility at the time of loading the kernel module after insmod command memory gets allocated inside kernel address space and the specific .ko file gets loaded into that memory.

after invoking this function name of our kernel module gets added in the list of modules which is maintained by kernel at location /proc/modules.

6) it is not possible to perform user interaction from the kernel module program but the status of kernel module can be maintained in kernel log file.
printk function is used to write the data in that particular file.

8) this file gets automatically created when boot os at location /var/log/syslog
printk() is variable no of argument function in which the first argument should be fix that is the priority of the msg and 2nd argument is the msg what that we want to write.

9) if we return 0 value from init function then it indicate the success. if there is any problem at the time of loading kernel module return value is -1.
this return value is not used for programmer but the corresponding error msg gets displayed in file kernel log file.

10) cleanup_module is a keyword which is used for such a function which gets implicitly called at a time of removing module from memory
when we rmmod command internally this function gets invoked.
generally this function is used to deallocate the resources for the kernel module.

Module 2

in above program init_module and cleanup_module is the name of function which gets accessed by insmod and rmmod command

we can also provide any user defined name for this function but we have to register that user defined name with module_init and module_exit macro to use that macro successfully in our program we have include <linux/init.h> header file

we can also used __init and __exit flag at a time of defining this function.

this flags are used to optimize the memory for our kernel module. due to this flag memory gets immediately deallocated when the execution of that function gets completed.

* module : inside kernel module we can create local variables which can be used in our kernel module.

datatype and name of variable is same normal c programming technique.

static int no __initdata = 21;

in this syntax due to the static we can not access variable the outside the file

int is the datatype of that variable
no is the name of that variable
__initdata is the macro which indicates that the memory for that variable should be allocated at the time of loading the kernel module

****int int

int arr [2] [3] [4]

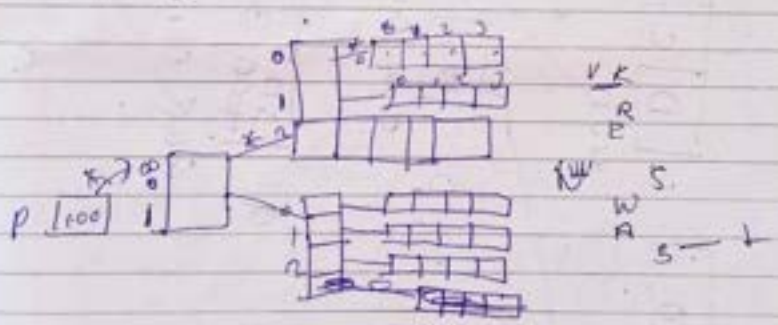
```
int ***p = (int ****) malloc (2 * sizeof (int));
for ( i=0; i<2; i++)
{
    int ***p = (int ***) malloc (3 * sizeof (int));
    for ( j=1; j<3; j++)
    {
        int *p = (int **) malloc (4 * sizeof (int));
    }
}
```

1770
598

80 → 150
70 → 9

DCMNT

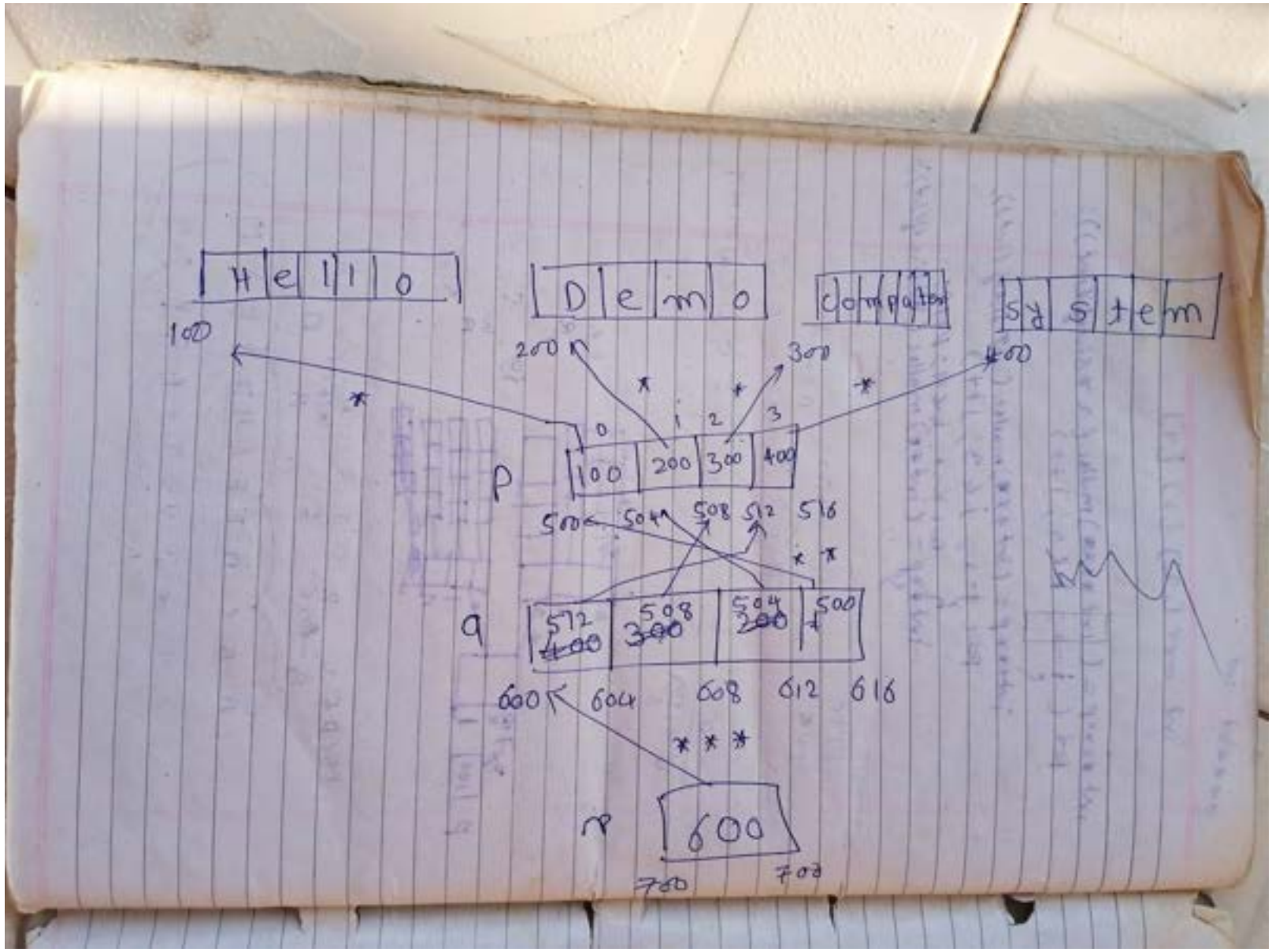
EOUCOMNT



ABCDE 1 2 3 4
B C E A D
last 2nd

A B C D E F G H I J K L M

Z Y X W V U T S R Q P O N


H e l l o
D e m o
c o m p u t e r
S y s t e m
100
200
300
400
0 1 2 3
P
100 200 300 400
500 504 508 512 516
q
572 508 504 500
600 604 608 612 616
p
600
700 708