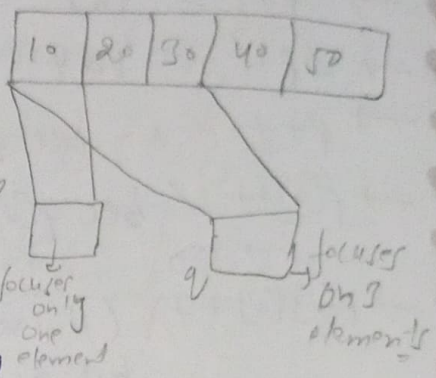
int (*q)[3]; // here (q) is a pointer, pointing to an array of 3 elements.

this will holds the address of 3 elements starting address

```
p = a;
q = a;
pf("%ld %ld \n", sizeof(*p), sizeof(*q));
pf("%ld %ld \n", sizeof(p), sizeof(q));
pf("%ld %u \n", p, *p);
pf("%u %u %u", q, *q, **q);
}
```

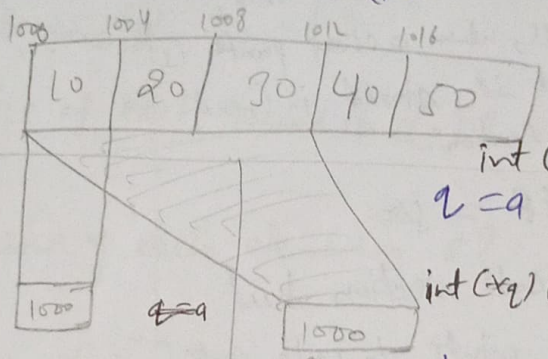
(4) (12) (8) (8) (1000) (10) (1000) (1000) (10)



→ q = 3 elements base add. (1000)

→ *q = In that array 1st ele base address (1000)

→ **q = 10



int *p;
p = a;

but this 1000 acts as one element address

Here, p = 1000
*p = 10
p+1 = 1004

int (*q)[3];
q = a

int (*q)[3];

this 1000 acts as 3 element starting address
q = 1000 (3 ele addr)
*q = 1000 (1st address of 3 elements)
q+1 = 1012 (bec 'q' is focusing on 12 bytes)
*q+1 = 1004 ('*q' is one ele add) bytes
**q+1 = 11

```
for(i=0; i<3; i++)
  pf("%d", (*q)[i]);
pf("\n");
```

o/p → 10 20 30

a warning occures bec q holds values & q can hold 3 values.

Session - 69

eg:-
```
void main()
{
  int a[5] = {10,20,30,40,50};
  int (*p)[3];
```

8 - 4 - 2.

```
p=(int (*)[])a;   Typecasted
Pf("%d %d %d\n",(*p)[0],(*p)[1],(*p)[2]);
}
```

Q1:- 10 20 30

→ (*p)[i] = *(*(p+0)+1) = 20

→ p+1 = 1012; gives the next array base address.

Now,

```
Pf("%d %d %d\n", p[0][0], p[0][1], p[0][2]);
```

(*p can be represents as p[0]).

⇒ when the address of a pointer which holds the address of another values it has to be caught by a double pointer.

⇒ In a 2-d array, the elements are 1-D array.

⇒ when a 2-d array is passed, it has to be caught with a special element that catches the passed array

⇒ we catch the 2-d array using pointer to array variable.

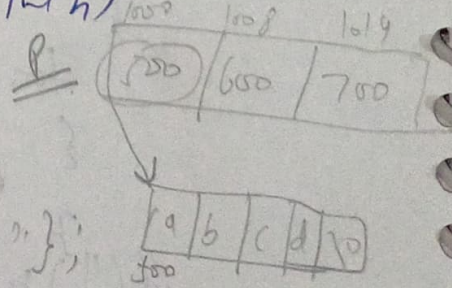
```
void print(int, int (*p)[]);
void main()
{
  int b[2][3]={{10,20,30},{40,50,60}};
  Print(2,3,b)
}
void print(int r, int c, int (*p)[c]).
{
  int i,j;
  for(i=0;i<r;i++)
  {
    for(j=0;j<c;j++)
    { Pf("%d", p[i][j]);
    }
  Pf("\n");
}
```

How to catch a 2-d array when passed?
→ Pointer to array

How to catch array of pointers when passed?
→ Using double pointer.

eg:-
```
void print(char (*)[], int);
void print1(char **, int);
void main()
{
  int i;
  char s[][10] = {"ABCD", "EFGH", "IJKL"};
  char *p[] = {"abcd", "efgh", "ijkl"};
  print(s,3);   ← 3 1-d arrays
  print(p,3);   ← 3 pointers.
}
void print(char **p, int n)
{
  int i;
  for(i=0; i<n; i++)
    pf("%s\n", p[i]);
}
```

eg:-
```
void main()
{
  char s[][10] = {"abcd", "efgh", "ijkl"};
  print(s,3);
}
void print(char (*p)[10], int n)
{
```



eg:-
```
void main()
{
  char *p[] = {"abcd", "efgh", "ijkl"};
  print(p,3);
}
```

```
Void print(char *xq, int n)
{
}
```

Session - 7 9-9-20

Command line arguments (Load time input)

→ There are two types of i/p, compile time i/p and run time i/p.

→ The program with compile time i/p executes faster than the program run time i/p. It does not wait for the user to give input.

→ Compile time i/p ⇒ Not dependent, No blocking function → favourable to OS

→ Runtime i/p ⇒ favourable to user (waits until the user gives i/p)

→ Load time → time is b/w compile time & run time

Load time → Time when the file loads into the RAM.

In C language, the i/ps can also be given during the load time.

Command line Arguments in the mechanism where we can supply the i/p to our program at load time.

→ Load time i/p :- favorable for both OS & User

⇒ Command is an exe. file which is present in bin directory

eg:- mkdir dir_name ↵ ← this takes i/p during load time

If mkdir takes i/p at run time

→ mkdir ↵

→ dir_name ↵

Note:- Almost all the commands take i/p at load time.

with respect to def: main() is user defined.

3 types of declaration of main() function.

1) void main()
{
}

2) void main(int argc, char **argv)
or char *argv[]
{
}

3) void main(int argc, char **argv, char **env)
or char *argv[], char *env[]
{
}

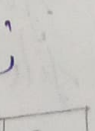
There are 2 cases to catch with double pointer
(1) passing address of pointer
(2) array of pointer base address.

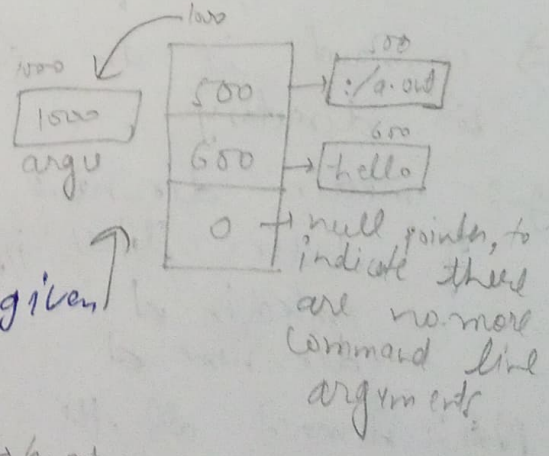
argc ⇒ argc holds no. of arguments we are passing through command line including a.out

argv ⇒ argv holds base address of array of pointers where each pointer pointing to one command line argument and atlast null pointer will be there.

eg:- void main(int argc, char **argv)
{
pf("%d\n", argc);
}

o/p:- ./a.out hello

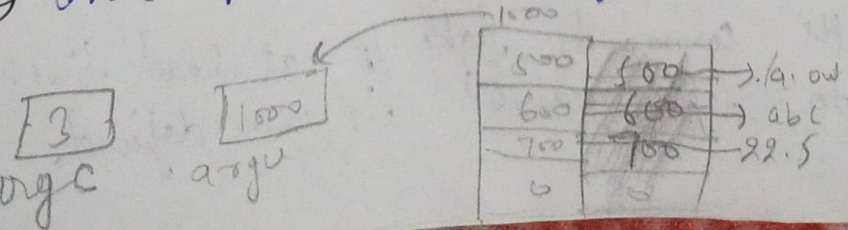




Q.

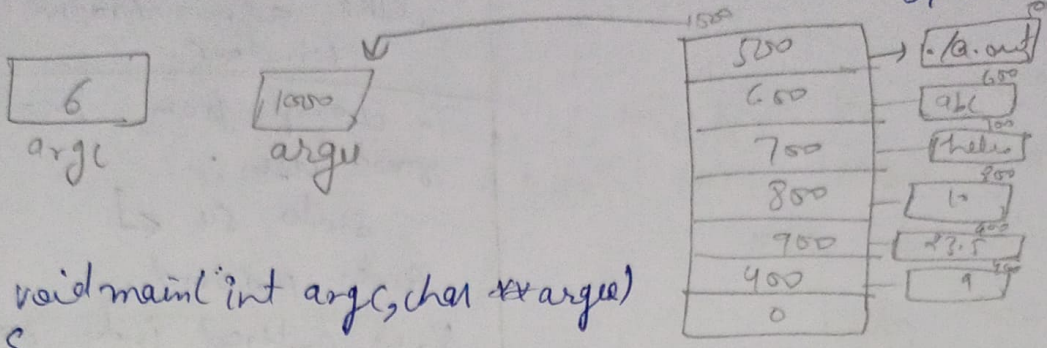
→ when ./a.out hello is given

→ when ./a.out abc 22.5 is given



This command prompt/os/shell is incharge of these operations, (arranging this and calling main())

e.g ./a.out abc hello 10 23.5 a (space is provided b/w all arguments)



eg:-
```
void main(int argc, char *argv)
{
  int i;
  Pf("%d\n", argc);
  for(i=0; i<argc; i++)
    Pf("%s", argv[i]);
}
```
→ neglect ./a.out so [1] not [0]

denotes the base address of the string entered

Note: Whatever is passed through command line is treated as string

→ Command line arguments are present above the main fun stack frame.

→ argv can be increment (argv++)

Session 7 — 10-09-20

Write a program for finding the length of the given string. Note:- Take input at load time.

```
#include<stdio.h>
#include<string.h>
void main(int argc, char *argv[])
{
  if(argc!=2)
  {Pf("usage:-./out string\n");
   return;
  }
```

If the exe file is copied into usr/bin directory then it becomes a command. Then no need of ./a.out

```
printf("The length of %s=%d\n",
        argv[i], strlen(argv[i]));
}
```

The program is only for one string apart from ./a.out

eg → For input like

→ ./a.out abcd

→ The length of abcd is 4

but when → ./a.out ↵

→ usage: ./a.out string

and when

→ ./a.out abc def ↵

→ usage: ./a.out string

→

To copy it into the bin directory, the user has to be root user like administrative in windows.

To change from local to root user :-

sudo su ↵

enter password

Now entered into root mood

eg. cp my_strlen /usr/bin/

To exit from root mode

exit ↵

The exe file is made as root

In linux '.' means pwd present working directory

'..' means parent directory

To show hidden files press → ls -a

Session - 12

11-09-20

atoi(); (ascii to integer) Predefined function

→ #include<stdlib.h> has to be given

→ atoi() Converts ascii to integer

```
int atoi(const char *nptr); // declaration
#include<stdio.h>
#include<stdlib.h> // stdlib
  Void main()
  {
   char s[] = "abcd";
    int i;
    i = atoi(s);
```

printf("s = %s i = %d \n", s, i); // s = abcd i = abcd

//	# (using atoi())
s = "1234@36";	i = 1234;
s = "-12:3";	i = -12
s = "abcd12";	i = -123
s = "-123 ab";	i = 0

	1	2	3	4	
	1	2	3	4	\0
ASCII →	49	50	51	52	

→ strings has characters stored and they are in their ascii format

User defined atoi() my_atoi();

```
int my_atoi(char *p);
void main()
{
 char s[] = "1234"
 int i;
 i = my_atoi(s);
 printf("s = %s  i = %d \n", s, i);
}
int my_atoi(char *p)
{ int i, num=0;
 if (p[0] == '-' || p[0] == '+')
  i = 1;
 else
  i = 0;
 for( ; p[i] ; i++)
 {
  if(p[i] >= '0' && p[i] <= '9')
   num = num * p[i] - 48;
  else
   break;
 }
 if (p[0] == '-')
```

Main logic

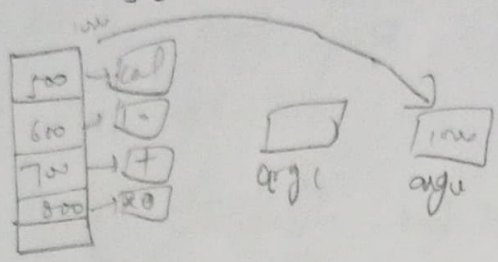
return num;

}

basic calculator program using load line i/f

eg: ./cal 10 + 20 ← 2 operands and 1 operator are taken at load time

res : 30



In the command prompt, some symbols have special memory

To neglect the special meaning '\' has to be used

eg: 10 * 20, will not work but

10 * 20; does the work of multiplication

eg: ls *.c ↵

→ displays all the files with '.c' in present working directory

```
#include<stdio.h>
#include<stdlib.h>
void main(int argc, char *argv[])
{
 if(argv!=4)
 {
  printf("Usage: ./cal num1 op num2 \n");
  printf("ex. ./cal 10 + 20 \n");
  return;
 }
 n1=atoi(argv[1]);
 n2=atoi(argv[3]);
 switch(argv[2][0])
  {
```

```
case '+' : n3 = n1 + n2;
           printf("n3 = %d\n", n3);
           break;
case '-' : n3 = n1 - n2;
           printf("n3 = %d\n", n3);
           break;
/* ... so on */
default: printf("Unknown operator\n");
}
```

or ./cal_10_-_20 → /* used bec. '*' has special work in cmd ; to remove speciality '/' used

space has to be given b/w the nos of operator else it will not give the designed value.

Session 73 — 12-09-20

Variable argument list function

⇒ Fixed argument list function
eg:- strlen(); } fixed the no. of args.
strcpy(); } given are fixed

⇒ Variable argument list function
eg:- printf("Hello\n"); 1 argu
printf("i=%d j=%d\n", i, j); → 3 arguments
scanf("%d %d", &i, &j, &k); → 4 arguments

// printf() ⇒ man page → $ man 3 printf ↵

printf(const char *format, ...); // min 1 arg is req.
sprintf(char *str, const char *format, ...); // min 2 arg req.

Note:- printf() returns integer, and it says no. of printable character.

eg: printf("hello\n"+1);

o/p:- ello

bec. this 'i' will point to h e l l o

eg:- int i;
{ const char s[] = "hello\n";
for(i=0; printf(s+i); i++);

o/p:- hello
ello
llo
lo
o

↳ the condition here is '\0' of the string

s+0 → hello\n
s+1 → hello\n
s+2 → hello\n

eg: printf("Hello %s\n", "abcd");

o/p:- Hello abcd

eg: printf("Hello", "Hai"); — A warning is thrown

o/p → Hello

Note:- printf() only worry about the end arguments, only when it finds any format specifier in the 1st array.

⇒ In a fun. call ',' - comma acts as a seperator

eg:- char s[] = "hello";
printf(s+1);

printf("hello\n"+1)

both are same

= In printf(), a string base address should be given

Note:- scanf() will return no. of sucessfull scanned data.

eg:-

```
int i,j,r;
printf("Enter \n");
r= scanf("%d \n",&i);
printf(" 1) r= %d\n",r);
printf("Enter i&j \n");
r= scanf("%d %d\n",&i,&j);
printf(" 2) r= %d \n",r);
```

o/p:- Enter i
5
r=1

o/p:- Enter i & j
5 10
r= 2

scanf() :- sprintf() :- sprintf(); does reverse of atoi() and atof()
↳ converts any type of data into ascii (string)

eg:

```
void main()
{
int i=1234;
float f=23.4;
char s[20];
sprintf(s,"i= %d f= %f \n",i,f);
printf("s= %s\n",s);
}
```

this is converted into string & stored in s

o/p
s= 1234 f= 23.400000

if 's' is a pointer, a valid address should be given to 's'.

Session - 74 16-9-20

Variable argument list function (Accessing number arguments)

Note:- Actual Arguments may/may not be contigeous, but the formal arguments are contigeous in a function stack frame.

Design a sum fun in such a way that how many integers arguments if we pass it will add them and send the result back.

while designing the variable arguments list fun., the function should have a minimum of one fixed aogument and at last 3 dots have to be provided. Method-I 1st method

[The fixed arg. may/may not indicates how many variables arguments are passed]

```
#include <stdio.h>
#include <stdarg.h>
int sum(int, ...);
void main()
{
  int i=10, j=20, k=30, l;
  l = sum(2, i, j); //30
  printf("l = %d\n", l);
  l = sum(3, i, j, k); //60
  printf("l = %d\n", l);
}
int sum(int n, ...)
{
  va_list v; //step 1
  int s=0, sum=0;
  va_start(v, n);
  for(i=0; i<n; i++)
  {
    s = va_arg(v, int);
    sum = sum + s;
  }
  return sum;
}
```

indicates no. of variable arguments present

indicates no. of variable arguments present

Accessing number arguments

this is implementation dependent

Last fixed argument name

this 'v' will point to the 1 no. of arg.

No need to increment 'v', it gets incremented automatically.

va-start (v, n);
↳ It is the declaration of the pointer 'v'.

Method-2

'Zero' has to be passed at last

```
#include <stdio.h>
#include <stdarg.h>
int sum(char *, ...);
void main()
{
  int i=10, j=20, k=30, l;
  l=("abcd", i, j, 0); //30
  printf("l=%d\n", l);
  l=("hai", i, j, k, 0); //60
  printf("l=%d\n", l);
}
int sum(char *n, ...)
{
  va_list v;
  int s=0, sum=0;
  va_start(v, n);
  for(;;)
  {
    s=va_arg(v, int);
    if(s==0)
      break;
    sum=sum+s;
  }
  return sum;
}
```

Session - 75 17-9-20

Dynamic Memory Allocation (DMA)

→ malloc() , calloc() , realloc() , free()

malloc ()

```
Void * malloc ( size_t size);
```

⇒ malloc() and calloc() is useful for allocating the dynamic memory.

⇒ realloc() is useful for reallocating the memory

⇒ free() is useful for unreserving the memory

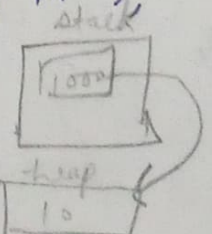
```
Void  * calloc (size_t nmemb, size_t size);
Void * realloc (Void *ptr, size_t, size);
Void free (Void *ptr);
```

malloc(), calloc(), realloc() → returns void pointer

Example:

```
#include <stdio.h>
#include <stdlib.h>
Void main()
int *p; // wild pointer
p=malloc(sizeof(int));
if (p==0)
{
printf("malloc unable to allocate memory \n");
return;
}
printf(" Enter the data \n");
scanf("%d", p);
printf("*p = %d \n", *p);
```

← This p is present in main() stack frame, but it is pointing to heap section.



In dynamic, this is the only way to access it. (i.e) it can be accessed indirectly. But when a static memory is allocated, it could be accessed by both direct & indirect method.

Session = 16 — 18-9-20

⇒ malloc() and calloc() will return the starting address of that dynamically allocated memory.

Write a program to allocate dynamic memory for 5 integers

```
#include<stdio.h>
#include<stdlib.h>
Void main()
{
static int *p;
int i;
p=malloc(sizeof(int)*5);
printf("Enter the ele--\n");
for(i=0;i<5;i++)
scanf("%d",p+i);//&p[i]
for(i=0;i<5;i++)
printf("%d ",*(p+i));//p[i]
printf("\n");
}
```

8 byte of static memory

20 byte of dynamic memory is allocated

memory is allocated for 5

memory

To access dynamic memory atleast one static pointer

To access dynamic memory, we static memory

Write a program to allocate dynamic memory for 1 string.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
{
char *p;
```

```
int l;
p= malloc(size(char)*10);
printf("Enter the string\n");
scanf("%s",p);
printf("p=%s\n",p);
l= strlen(p);
p= realloc(p,l+1);
printf("p=%s\n",p);
}
```

← 10 byte of dynamic memory is allocated

this +1 is given for the '\0'

← dynamic memory is reallocated in accordence with the length of the string.

O/P :-

Enter the string
hello
p= hello ← has 10 bytes of memory
p= hello ← has 10 bytes of memory.

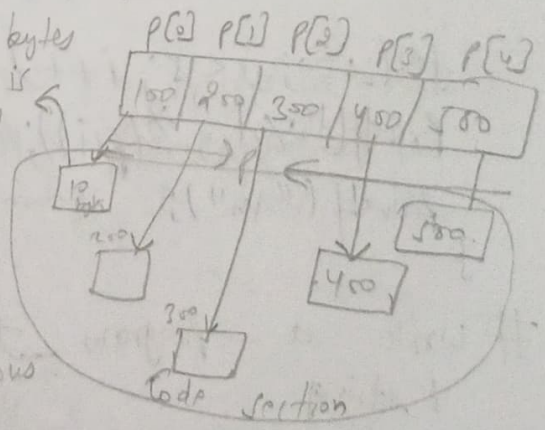
Allocate dynamic memory for 5 strings. Scan it and print() it.

An array of pointers are needed.

→ stack pointer

```
#include <stdio.h>
#include <stdlib.h>
Void main()
{
char *p[5]; ← p is an array of 5 pointers
int i;
for(i=0; i<5; i++)
 p[i] = malloc(10);
printf("Enter the strings\n");
for(i=0; i<5; i++)
 scanf("%s",p[i]);
for(i=0; i<5; i++)
 printf("%s",p[i]);
}
```

this 10 bytes of memory is contiguous



This are not contiguous

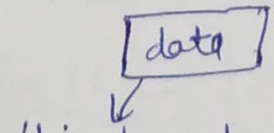
Allocate dynamic memory for n strings

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
  char **p;
  int i, n;
  printf("Enter n\n");
  scanf("%d",&n);
  p=malloc(sizeof(char *)*n);
  for(i=0; i<n; i++)
    p[i] = malloc(10);
  printf("Enter the strings-\n");
   for(i=0; i<n; i++)
   scanf("%s", p[i]);
   for(i=0; i<n; i++)
   printf("%s\n", p[i]);
}
```

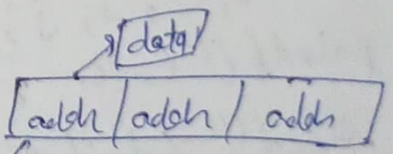
size of the char is given as an i/p.

Session - 77

19-09-20

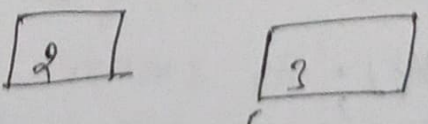


this has to caught with a single pointer



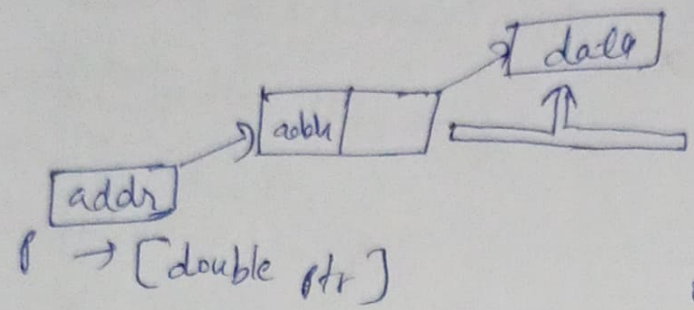
this has to be caught with double pointer.

Allocate dynamic memory for 2d array, where r and c need to take from user.



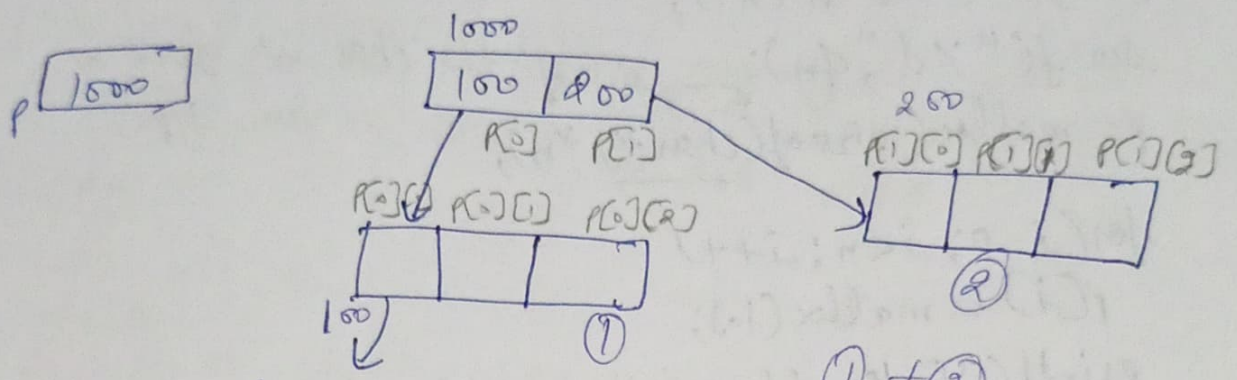
→ 2 1-D array & in each 1-D array, there are three (3) element

we are declaring a double pointer.



this data is stored in code section, which is read only data.

eg:



① + ② are not contigeous

to access this data we have to use p[0][0].

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
  int **p;
  int r, c, i, j;
  printf("Enter r & c\n");
  scanf("%d %d", &r, &c);
  p = malloc(sizeof(int *)*r);
```

↳ allocated memory for integer Pointer that's why int *

```
for(i=0; i<c; i++)
p[i] = malloc(sizeof(int)*c);
printf("Enter the data..\n");
```

```
for(i=0; i<r ; i++)
for( j=0 ; j<c; j++)
scanf("%d", &p[i][j]);
Pf("  ---  ---  ---\n");
for( i=0; i<r ; i++)
  {
    for( j=0; j<c; j++)
    Printf("%d ", p[i][j]);
    Print("\n");
  }
}
```

Difference b/w malloc() and calloc()

→ malloc() and calloc() are useful for allocating dynamic memory.

→ Both returns the starting address of the dynamically allocated memory.

malloc() → memory allocation

calloc() → Contiguous memory allocation.

→ Both returns null on failure

→ malloc() — 1 arg. arg1

→ calloc — 2 arg. arg2

[calloc(no. of members, each memory size)]

→ malloc() is faster than calloc(), but its not safer

calloc() after allocating memory, takes an extra step of initializing the allocated memory by zero

→ malloc() returns ptr. to uninitialized storage

The allocated region is initialized to zero in calloc() due to this, calloc() will not point to junk data.

→ malloc()

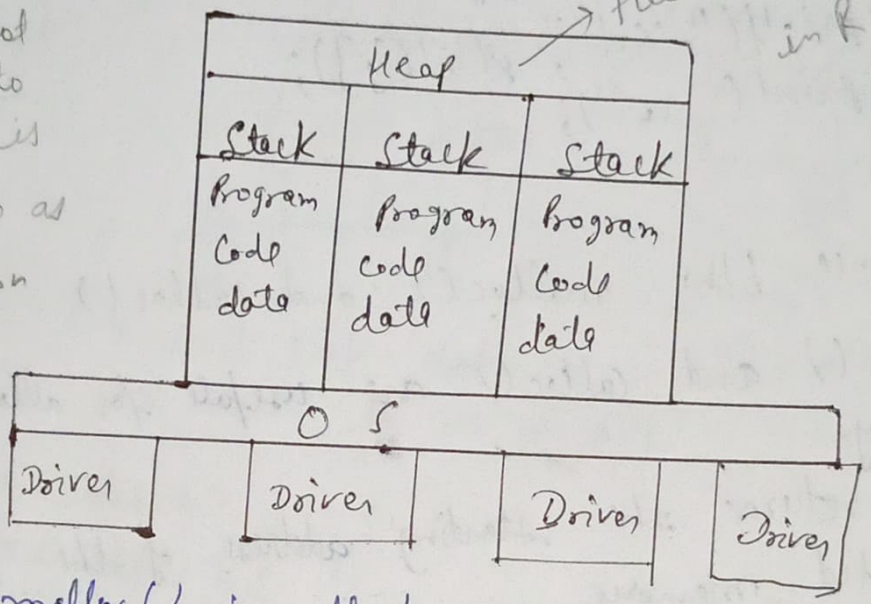
```
void *malloc(size_t n);   eg→ malloc(10*sizeof(int));
```

calloc()

```
void *calloc(size_t n, size_t size);
```

Memory allocation layout

In, RAM, the memory not allocated to anything is referred to as heap section



when malloc() is called, malloc() will communicate with the OS, and OS will reserve memory from heap section.

Once the memory is allocated from the heap section, it is called as extended data section

Life of dynamically allocated memory.

```
#include <stdio.h>
int i;
static int j;
main()
{
    int K; // Auto
    static int l; // local static
}
```

Life of a dynamically allocated variable is when it is freed, or when the program is terminated

Life → creation/destruction
Scope → visibility

⇒ life of DMA starts when memory is allocated when we allocate memory using malloc() or calloc().

Life ends when memory is ~~fixed~~ (free() or when this program terminate freed

consider.

```
p = malloc(10);
```

p can be reallocated.

```
p = realloc(p, 100);
```

//initially p is allocated with 10 bytes but now p is allocated with 100 bytes.

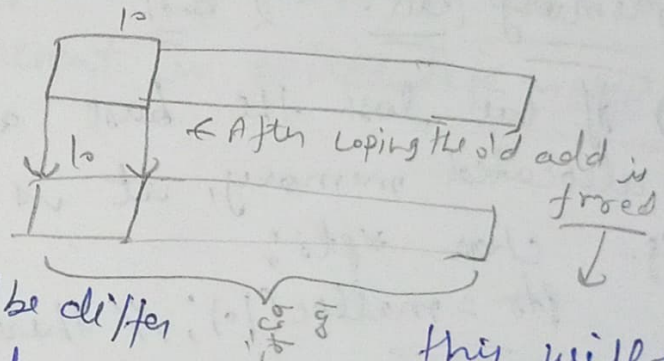
→ Here the memory is increased

This will allocate contigeous 100 bytes.

```
p = malloc(10);
```

=

```
p = realloc(p, 100);
```



both p's value (addr) will be differ [where memory is increased, p may / may not have the same value]

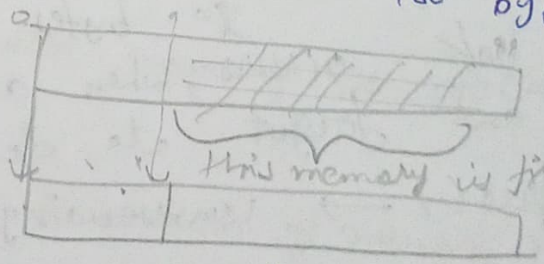
this will happen when there is no contigeous space for 100 bytes.

Now consider,

```
p = malloc(100);
```

=

```
p = realloc(p, 10);
```



In this case, always the value (add) of p is same.

× memory is decreased

When realloc() fails, already reserved dynamic memory remain untouched

eg:- p = malloc(10);

p = realloc(p, (oooooooooo)); // realloc fails

here

In this case catch with another ptr.

Session - 78 21-9-20

→ free() → memory leak → Dangling pointer

Upon failure realloc() will return zero.

```
p = malloc(10);
q = realloc(p,20);
```

⇒ ↳ realloc will search for contigious 20 bytes of memory, if 20 bytes of memory is available contigiously with p as starting address, q will also have the same value of p.

⇒ If realloc() does not find 20 contigious bytes of memory at p, it will search for a place where 20 byter of memory is available. Then the data in 'p' will be copied to the newly found 20 bytes of memory.

Memory leak:- 2 kinds

① → If we lose the base address of dynamically allocated memory, it is known as memory leaks.

eg- char *ptr;
ptr = malloc(10); // there 10 bytes are memory leaks as its address is lost.
ptr = malloc(20); // ptr is overwritten with address of 20 bytes.

② → Memory leak occurs, when an allocated memory in heap and forgot to delete it.

free() → unreserving the memory.

Dangling pointer:-

```
Void main()
{
  char *p;
  p = malloc(p);
```

```
printf("1) %p\n", p);
free(p); // p is freed here
printf("2) %p\n", p); // through p is freed, p still holds
                      the same address, this is due
                      to dangling ptr
```

⇒ After freeing the memory, if the pointer still points to the same memory, the pointer used is called the dangling pointer.

⇒ Another def:- If a pointer points to a variable whose life is ended it is called a dangling pointer.

#mtrace()

To check memory leak, a function called mtrace() is used.

→ mtrace() will hook or monitor the program to check the amount of memory allocated and leaked. This function is present in #include<mcheck.h>

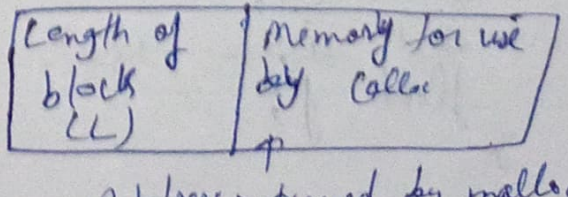
eg: To avoid memory leak

```
char *p;
p = malloc(10);
if("%p\n", p);
free(p);
p = 0; // making p a null pointer
```

→ When malloc() is called memory is allocated in the form of blocks.

#How the free() knows how much of memory needs to be freed?

eg:-
```
p = malloc(10);
free(p);
```



Address returned by malloc()/calloc()

When malloc() allocates the memory, it allocates an extra block to hold the length of the block in integer format, which gives the size of the block. This integer is located at the beginning of the block. The address actually returned to the caller points to the location just past this length value [The size of the block depends on implementation]

Session -19

22-09-20

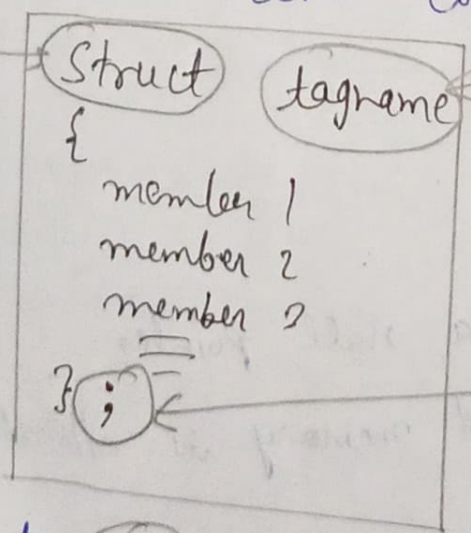
⇒ User defined datatype:

Structure:- Structure is a collection of different types of data which are in contigious memory location

How to declare a structure?

→ Use the keyword 'struct'

Note:- Keywords are the words which are already known to the compiler.



userdefined datatype name

Indicates the end of the structure.

eg:- struct one ← user defined datatype name
{
char ch;
int i;
float f;
};