advanced question   Bit operations

1). int a=5 ,b=3
print (a&b); = 1

	8	4	2	1
5 =	0	1	0	1
3	0	0	1	1
	0	0	0	1

2. x=6
x | 3
= 7

6)	0	1	1	0
3)	0	0	1	1
	0	1	1	1

3). x=4
x ^ x = 0

	8	4	2	1
4)	0	1	0	0
	0	1	0	0
	0	0	0	0

xor = ^
1 0 - 1   1 1 - 0
0 1 - 1   0 0 - 0

4) int x=1;
x = x<<2;
Print ("%d",x);
=4

x = 1

8	4	2	1
0	0	0	1
0	1	0	0

4

5) int x=8
x = x>>1;
Print ("%d", x);
= 4

	8	4	2	1
	1	0	0	0
1>	0	1	0	0

4

6) ~(5) equals to

(6)

1 0 1 0

(5) C 4 2 1
0 1 0 1

(-5) 1 0 1 0
msb (1)
0 1 0
(6) 0 1 1 0

step 1:

+5 = 0101

step 2:

~(0101) = 1010

we got

1010 → It is negative in 2's complement

8	4	2	1
0	1	0	1
			1
0	1	1	0

→ 6

1
1
10

---

7) check if the 3rd bit in x=10 is set.

```
int x = 10
x = x ^ (1<<3) //toggle 3 bit.
printf("%d\n", x);  op: 2
```

8	4	2	1
1	0	1	0

pro

```
x = 10;  // 0000 1010
int bit_p = 3;
if (x & (1<< bit_p)) {
    pf("it is set");
else
    pf("not set");
```

10 & (1<<3)
(10) 0000 1010
(1<<3)

Bit masking operation

It is basically like putting a mask over certain bits and hiding the other un-useful bits.

It is used to set, clear, or toggle (or) isolate.

example — creating a Bit mask

int x = 10; // 0000 1010
int mask = (1<<3); // 0000 1000 — index

index	value	x (10)	mask (1<<3)
3	8	1	1
2	4	0	0
1	2	1	0
0	1	0	0

x & mask = 0000 1010 & 0000 1000 = 0000 1000 →
Non-zero — Bit is set.

& → to check a bit
| → to set a bit
^ → to toggle a bit
& ~ → to clear a bit.

// check Bit 2
if (x & (1<<2))

0000 0100 (1<<2)

x = 10
10 → 0000 1010 — &
1<<2 → 0000 0100
0000

not a bit

Clear Bit

x & 2 = ~(1<<1)

this clears Bit 1 (makes it 0) even if it was 1 before

Homework

(1<<4) = 0000 0010

~(1<<1) = 1111 1101

x = ~~1111 1101~~ 0000 1010

1<<4

4th place one.

16 8 4 2 1

0001 0000 → 1<<4

0000 1010 → 10

0001 1010 → 26

→ x1 = (1<<

(1<<1)

0000 0010

= 1111 1101

10 = 0000 1010

x & = ~(1<<1)

1111 1101 &

0000 1010

0000 1000

= 8

8) Swap two numbers

a = a^b

b = b^a

a = a^b

a = 5 b = 10

8 4 2 1

a = 0101

b = 1010

a = a^b = 1111

b = b^a = 1111 1010 → 0101

a = a^b = 1111 0101 → 1010

(9) what does x & ~(1 << n) do?

x = 5 n = 2.

x 5 & ~(1 << 2).

~1

(1<<2) = 0000 0100

~(1<<2) = 1111 1011

5 = 0000 0101

0000 0001

(10) How to set nth bit of a number?

x | (1 << 5)

1 << 5 ->

0010 0000

x = 7

x = 7 : 0000 0111

0010 0000

0 0 1 0 0 1 1 1 = 39

Logical operations

1) int a=0, b=5;
Printf ("%d", a && b);

= 0

Because other than zero, all positive numbers are 1 only in && (op)

2) int a=1, b=0;
Printf ("%d", a||b);

= 1

Because of || (or) operator

3) !0 equal to.

= 1

Because this tells ! not operator.
So !0 = = 1

4) int x=0
Rif ("%d", x || ++x);

= 1

Because ++x (Prein)

5) && and ||

&& = compare each bit of two values

|| = compares the further (non-zero-true)

---

6) 0 && 1 || 1

0 && 1 = 0
0 || 1 = 1

= 1

---

7) evaluate !(a && b) = = ! a || ! b

a=1 b=2.

!(a && b) = ! (1)
= 0.

0 || 0

0 = 0

(8) Precedence && (or) || — && is high
|| is low

(9)
```
int a=1; b=1;
if (a-- && ++b)
    printf("%d", b);
```

a-- = 1
++b = 2.

so=2

(10) Can? Short circuit evaluation prevent a segfault in C? Yes

what is short-circuit?

&& Short circuits if the left side is false (0)

|| Short-circuits if the left side is true (non-zero)

The right side is not even evaluated if the left already determines the outcome.

ex:

```
char *ptr = NULL;
```

And

```
if (ptr != NULL && ptr[0] == 'A') {
}
```

This is safe because:

if ptr == null, the first condition ptr != NULL is false.

a) So && the second part ptr[0] == 'A' is never run

-> No crash, NO ga segmentation fault.

1's and 2's complement

(1) 1's complement of 0101 → 0101, is 1010

2) 2's complement of 0101

(or) 2's complement do 1's first and add (1)

```
1010
   1
1011
```

(3) find -5 in binary 2's complement

=1011

```
      8 4 2 1
5 ->  0 1 0 1
1's = 1 0 1 0
2's = 1 0 1 0
        1
      1 0 1 1
```

(4) is 2x+1 equal to -x?

x=3

```
-x = 1101
     0010
     0011
     1100
     1101
```

So they are equal

```
2x+1
= 23+1
= 0011
  1100
  1101
```

(5) Convert -10 to 2's comp in 8 bit

```
      8 4 2 1
10 -> 0 0 0 0 1 0 1 0
1's -> 1 1 1 1 0 1 0 1
2's -> 1 1 1 1 0 1 1 0
         1
       1 1 1 1 0 1 1 0
```

So = 11110110

In C++

```
signed char x=-10;
for(int i=7;i>=0;i--)
    Pb('1', (x>>i)&1);
}
```

6) Diff Btw signed and unsigned

signed → can represent both + and - values
uses MSB (0 - Positive, 1 = negative)

unsigned → can represent only positive values
All bits contribute to magnitude.

7) overflow detection in 2's

If 2 numbers of same sign are added and the result is opposite sign ~~then it is~~

+ (+) + = should (+) if result is (-) then it is overflow

- (+) - = should - if res is (+) then it is overflow

---

8) How is Subtraction Done using 2's complement?

Trick [a - b = a + (2's complement of b)]

ex: a=7, b=3

7 = 0111
3 = 0011

2's comp of B
= 0011
1' = 1100
2' = 1100 + 1
= 1101

a + (2's comp) = 0111 → a.
1101 ←
10100

Ignore → 0100 = 4

7 - 3 = 4 4 == 4

9) result of n(-1)

6/3

(111 1) |

$\Rightarrow$

not(1111) = 0000

-1) 0001

1's = 1110

2's = 1110

1

1111

10) 1's complement of -8 in 8 bit system?

-8 $\Rightarrow$ 1111 0111

-8

8 4 2 1

0000 1000

1's $\Rightarrow$ 1111 0111

x = 6

Array

1) int arr[] = {1, 2, 3};
printf("%d", arr[1]);

= 2

2) Accessing out of Bound : arr[10] = 5;
It has undefined Behaviour.

3] arr and &arr

arr => Array name (decays to pointer) — int* — Address of arr[0]

&arr — Address of the entire array — int (*)[5] — Address of whole array arr

arr = 0x1000 arr+1 = 0x1004
&arr = 0x1000 &arr+1 = 0x1014 (sizeof + 4 bytes).

arr:
1) decay to int*
2) points to arr[0]
3) You can access like arr[0], *(arr+1)

&arr:
→ is of type int(*)[5] → pointer to whole array
*) you need to differentiate
*(&arr) == arr
(*(&arr))[0] == arr[0]

4) Size of array using sizeof operator?

int arr[] = {10, 20, 30, 40, 56};

int size = sizeof(arr) / sizeof(arr[0]);

5×4

20/4 = 5.

5) int a[5] = {0};

what is a[4]:

Ans. int a[5] = {0}; so a[4] = 0.

---

6). How to Pass an array to a function?

```
void PrintArray(int *arr, int size){
    for(int i=0; i<size; i++){
        Print("%d", arr[i]);
```

call like

```
in arr[] = {1,2,3}
PrintArr(arr,3);
```

7) can you modify an array in a function.

Yes

```
void update (int arr[], int size){
    arr[0] = 99; // It will modify in OG.
}
```

8) what does *(arr+i). mean?

arr[i] == *(arr+i).

It uses pointer arithmetic

*) arr + i moves the pointer i elements ahead.

*) * deferens it

i.e).

9) int a[2][2] = {{1,2}, {3,4}};

Printf("%d", a[1][0]);

Sol)

a = [[1,2], [3,4]] ⇒ 3

Array of Pointer

10) int *arr[3];

int a=5, b=10, c=15;

int *arr[3] = {&a, &b, &c}

You can assign each pointer to a different int variable or arrays

Pointer to array

int (*ptr)[3];

*) points to the entire array block

*) used when passing full array to function.

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

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

Linked list

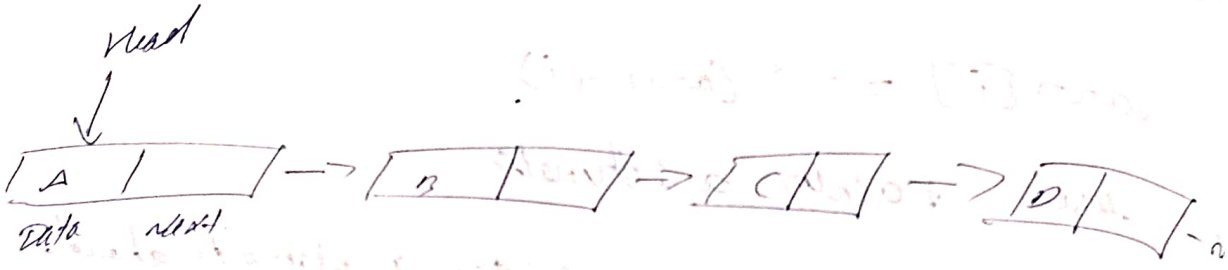
41 What is Linked list

A linked list is a dynamic data structure where

1) Each node holds some Data and a Pointer to the node

2). Nodes are dynamically allocated, so memory can be shrunk at runtime

3) The list ends when a node's next Pointer is null



Program

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

// Define the node structure using typedef

typedef struct Node {
    int data;
    struct Node *next;
} Node;

int main() {

    // Allocate 3 nodes in the heap
    Node *head = NULL;
    Node *second = NULL;
    Node *third = NULL;
```

```
node * temp = head;
printf ("Linked list values");
while (temp != NULL) {
    printf("%d", temp -> data);
    temp = temp -> next
}
free (head);
free (second);
free (third);
}
```

42 How To Traverse a linked list ?

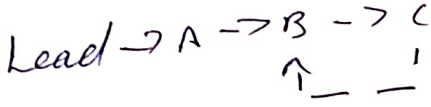
Traversal means visiting each node of the linked list from start to end. usually to

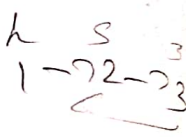
1) Print Data
2) Search for a value
3) count elements
4) (or) Perform other operat.

13) How to detect a loop in a linked list ?

* A loop in a linked list means :
  - Some node's next pointer links back to previous node
  - It creates an infinite cycle. Your program could get stuck
  - Loop is called detected.

ex:






```
lead -> data = 1;
Lead -> next = second
second -> data = 2
second -> next = Third
Third -> data = 3
third -> next = second; // loop
```

```
int detectLoop (Node * Lead) {
    Node * slow = Lead;
    Node * fast = Lead;
```

```
while (slow && fast && fast -> next)
    slow = slow -> next;
    fast = fast -> next -> next;
    if (slow == fast) {
        return 1;
    }
```

Linked list fully

struct fuction

```
typedef struct node {
    int data;
    struct node * next;
} node_s
```

global variable

```
node_s
* head *temp;
```

```
int main()
    init (100);
```

```
int init ( int data).
{
head = (node_s*)malloc (sizeof (node_s));
if (head == NULL)
 {
Print ("Head not created");
 return -1;
 }.
 head -> data = data;
 head -> next = NULL;
```

```
int insert_front (int data)
{
node_s * Newnode = (node_s*) malloc (sizeof (node_s));
if (Newnode == NULL)
  {
  printf ("New node not created");
  return -2;
  }
newnode -> data = data;
newnode -> next = head;
head = newnode;
```

insert_front (20) ->

insert_back() ; => node_s *n, *p;

```
n = (node_s*) malloc (sizeof (node_s));
if (n == NULL)
{
  Printf("\n node not created");
  return -3;
}
n -> data = data
n -> next = NULL;
for (p=head; p->next != NULL; p=p->next)
{
}
}
p->next=n;
return 0;
```

int delete_front() =>

```
temp = (node_s*) malloc (sizeof (node_s));
temp = head;
head = head -> next;
free (temp);
}
```

---

deleteback() -> node_s *n.

```
temp = (node_s*) malloc (sizeof (node_s));
for (temp=head; temp->next != NULL; temp=temp->next)
{
  n = temp;
}
free (temp);
n -> next=0;
```

Heap and memory errors

malloc vs calloc

	malloc(size)	calloc(num, size)
Parameter	→	
Init value	→ Doesn't initialize memory	initialize all bytes to zero
Use case	→ Use when init not required.	When you want zero-initialized memory.

e.g.!

malloc: Allocates a single block of memory of a specified size in bytes.

Return value: Return a void* pointer to the beginning of the allocated memory block. This pointer can be cast to any data type. Return Null if memory allocation fails.

calloc:

Allocates memory for an array of a specified number of elements, each of a specified size and initializes all bytes to zero.

Return value: Return a void* pointer to the beginning of the allocated memory block. This pointer can be cast to any data type. : Returns Null if memory allocation fails.

2) Segmentation fault:?

A Segmentation fault occurs when you try to access memory:

*) that your Program Doesn't own.

* that is invalid

*) (or) that's not allowed for that operation.

ex:

```
int *p= NULL;
*p = 10;
```

---

3) Accessing Freed memory

When you free() memory and still try to use it then it is undefined behaviour.

ex:

```
int *p = malloc (4);
free (p);
*p=10; // undefined behaviour
```

---

4) Double Free error

calling free() twice on the same memory location corrupts the program internal memory management data structures. This is undefined behaviour

ex:

```
int *p = malloc (10);
free(p);
free(p); // double free detected
```

5) memory leak example:

when you declared malloc and forgot to free() the memory it leads to memory leak.

```
void leak() {
    int *p = malloc(100);
}
```

we created the memory but we never freed it

---

6) Dangling pointer

A dangling pointer that points to a memory location that has been deallocated or freed. This situation typically arises in the following

simple → A dangling pointer points to memory that is already freed (or) out of scope.

ex 1

```
int *p;
{
    int x = 10
    p = &x;
}
x is out of scope here, p is dangling here.
```

ex 2.

```
int *p = malloc(4);
free(p);
// Now if you want to access the p
it is undefined behaviour.
```

wild pointer

A wild pointer, also sometimes referred to as an uninitialized pointer, is a pointer that has been declared but not initialized to a specific, valid memory address.

```
int *ptr;
// Accessing *ptr here would lead to undefined behavior.
```

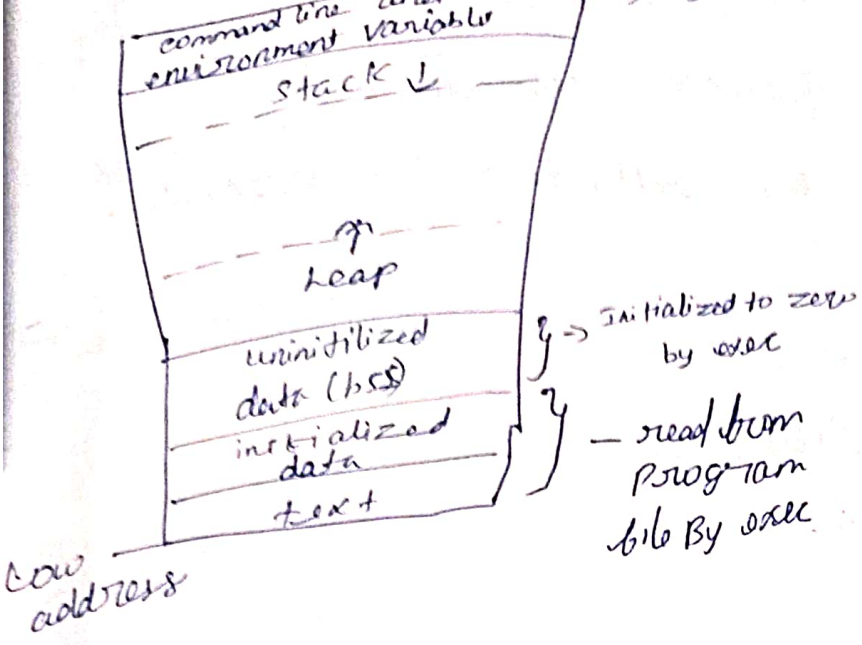
---

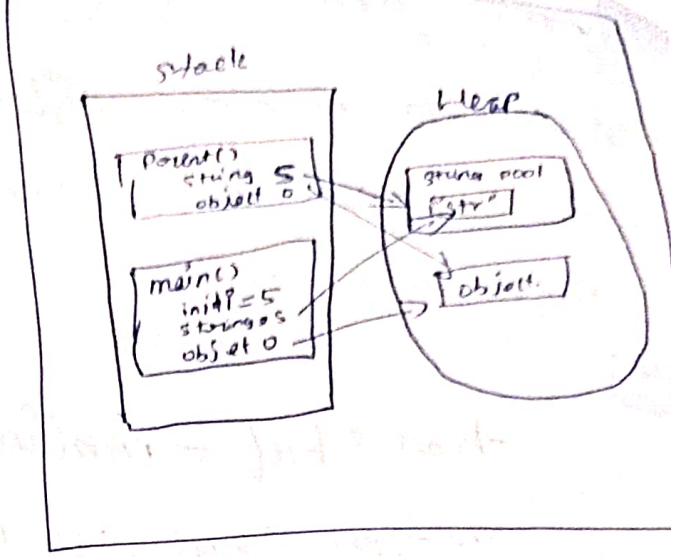
7) Heap vs Stack

feature	Stack	Heap
Allocated By	compiler	Programmer (malloc, calloc)
life time	Auto destroyed after function	You control (until free)
speed	faster (LIFO structure)	slower (manual mgmt)
Size	Smaller	Larger.
example:	int x = 5	int *p = malloc (sizeof (int));

---

memory	static memory	dynamic memory
Speed	faster	slower



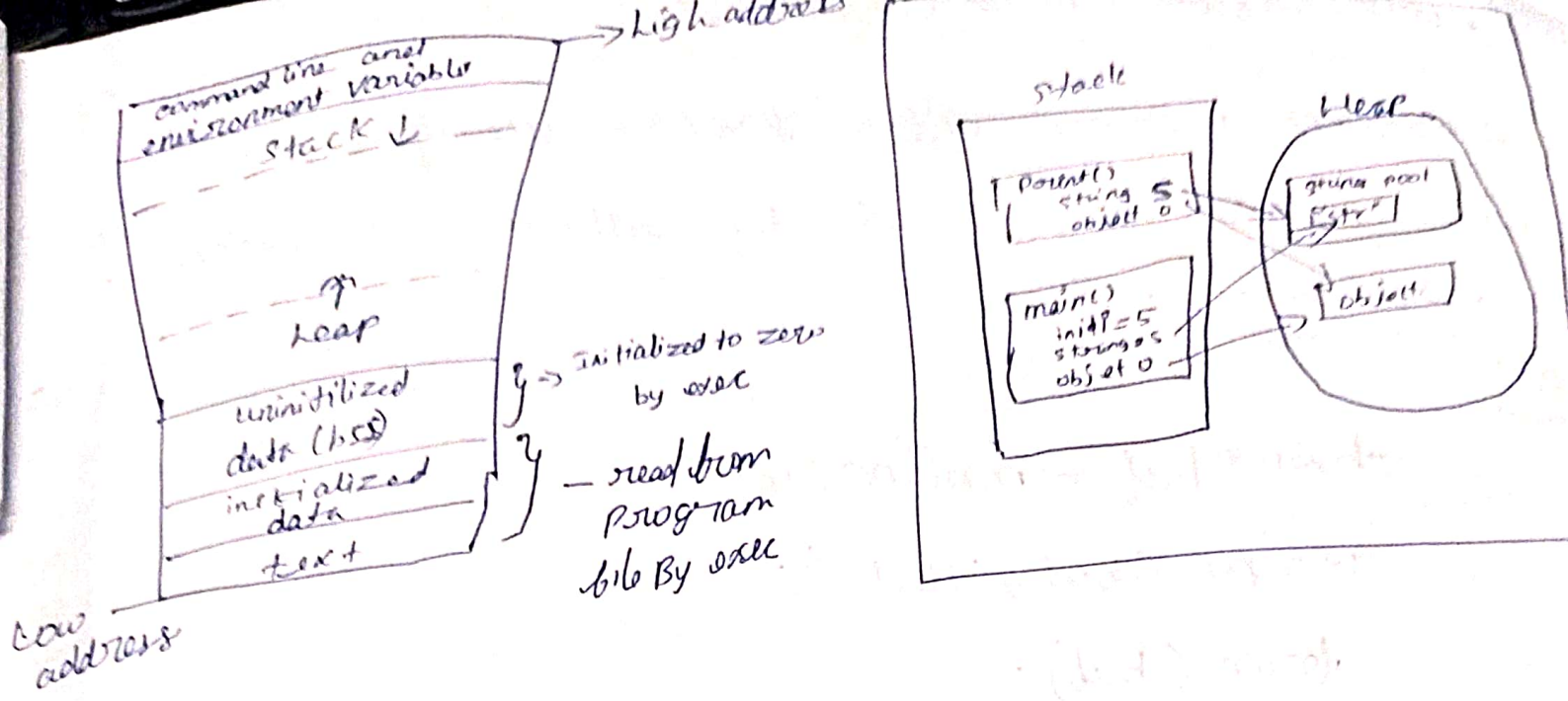


---

3) Segfault on Null pointer Dereference?

if you are Dereference Null on the pointer it will crash.

```
int *p = NULL;
Printf("%d", *p);  // crash.
Printf("%d", p);   // 0 - Print
Printf("%p", &p);  address print.
```



8) Segfault on Null pointer Dereference?

If you are Dereference Null on the pointer it will crash.

```
int *p = NULL;
Printf("%d", *p);   // crash.
Printf("%d", p);    // 0 - Print
Printf("%p", &p);   address print.
```

9). Use-After-Free Vulnerability

Accessing memory after free() = use-after-free
This can be exploited by attackers to execute arbitary code.

Ex:

```
char *buf = malloc(10);
strcpy(buf, "Hello");
free(buf);
printf("%s", buf);
```

~~It is pointing~~
Every time we run this it give random values

---

10) Tools to Detect memory errors

Tool	What it Detects
valgrind	leaks, invalid frees, use-after-free, etc
ASAN	(Address sanitizer) fast, runtime mem checker.
GDB	Debugging seg faults
Static Analyzer	clear static Analyzer, Cppcheck

---

cd' with valgrind in comp[/Downloads/PA-6-tmp1/week1]
Test Pro.

valgrind ./my-program

Function Pointers and Callbacks

1) Declare pointer to function returning int?

Ex] A function pointer stores the address of a function, just like an int pointer stores address of an int.

Syntax:

```
int (*fptr)(int, int);
```

fptr ↓ Variable NAME

| return_type (*ptr)(parameter-types)

ex:

```
#include<stdio.h>
int add(int a, int b){
    return a+b;
}
int main(){
    int (*fptr)(int, int);   // Declaration
                             // assignment
    fptr = add;
    int result = fptr(10, 20); // function call using Pointer
    Printf("Result %d", result);
    return 0;
}
```

2] How to pass function pointer to Another function:

In this function pointer to another function we will declare another function name into the required function.

ex:

```
compute(10, 20, add); // Passing function as argument.
```

Example Program

```
#include <stdio.h>
int add (int a, int b) {
    return a + b;
}
void compute (int x, int y, int (*add)(int, int)) {
    Printf ("Result = %d \n", add(x,y));
}
int main() {
    compute(10,20, add);
    return 0;
}
```

Another example

```
#include <stdio.h>
int add (int a, int b) {
    return a+b;
}
int sub (int a, int b) {
    return a-b;
}
int compute (int a, int b, int (*add)(int,int), int (*sub)(int,i
    int c = add (a,b);
    int d = sub (a,b);
    Printf ("addition is %d \n", c);
    Printf (" Sub is %d \n", d);
}
int main() {
    int a, b;
    Printf ("enter numbers");
    Scanf ("%d %d", &a, &b);
    compute (a, b, add, sub);
    return
```

3) Callback function example in C ?

Callbacks are just function pointers passed to another function. often for custom behavior (event, ISR, HAL).

ex:

```
#include <stdio.h>
void on_button_Press () {
    Printf (" Button was Pressed ");
}
void Simulate_button (void (* callback)()) {
 Printf(" Simulating Button Press ");
 call back();
}
int main() {
    simulate_button ( on_button_Press);
    return 0;
}
```

4) Normal vs callback function

Feature	Normal function	Callback function
Defined & called	Directly in main / logic	Passed to another function.
Fixed behaviour	Yes	By at runtime.
Use case	Any logic	event handling, API hooks, HAL

5) Array of function pointers

```
#include <stdio.h>
int add (int a, int b){
    return a+b;
}
int sub (int a, int b){
    return a-b;
}
int mul (int a, int b){
    return a*b;}
int main(){
    int (*op[3])(int,int)={add, sub, mul};
    int choice=0;
    int x=5, y=3;
    printf("%d", op[choice](x,y));
    return 0;
}
```

---

6) Function pointer to call

```
#include <stdio.h>
int add(int a, int b){
    return a+b;
}
int main(){
    int (*ptr)(int,int)=add;
    printf("Sum %d\n", ptr(4,6));
    return 0;
}
```

7) Can function pointers point to static function?

Yes, static functions can be pointed to by functions pointers within the same file

```
#include <stdio.h>
static void greet() {
    printf("Hello from static");
}
int main() {
    void (*ptr)() = greet;
    ptr();
    return 0;
}
```

---

8) What is typedef for function pointer?

using typedef make function pointer usage clean & readable

```
void (*handler)(int);
```

with typedef:

```
typedef void (*callback_t)(int);
void example(callback_t cb) {
    cb(100);
}
```

let us see full Program example for it

Prog

```
#include <stdio.h>
typedef int (*math_op_t) (int, int);
int add (int a, int b)
    {
    return a+b;
    }
int main() {
    math_op_t f = add;
    Printf("Sum = %d", f(7,3));
    return 0;
}
```

---

9) How is function Address stored in memory?

*) Function pointers store code Segment address (unlike variables, which are in stack / heap).

*) You can't modify function contents - only call via pointer

Code

```
#include <stdio.h>
Void hello() {
    Print("Hi");
}
int main() {
    Printf("Function address", hello); // address
    Void (*Ptr)() = hello;
    Ptr(); // call the function
    return 0;
}
```

(c) void (*fPtr)(); what is this?

This means:

- fPtr is a pointer to a function.
- The function takes no parameter
- The function returns void.

```
#include <stdio.h>
void show(){
    Printf("This is me");
}
int main(){
    void (*bPtr)(); //declare.
    bPtr = show;
    fPtr();
    return 0;
}
```

All in one Program

`int (*f)(int)` → f is pointer to function taking int and returning int

`typedef int (*fp)(int)` fp is a type alias for function pointer

`fPtr = add;` assign function to pointer

`bptr(10);` call function via pointer

`func(fPtr)` Pass function pointer as argument.

```
#include <stdio.h>
int add (int a, int b)
    { return a+b; }
int sub ( int a , int b)
    { return a-b; }
typedef int (*math_ptr) (int , int);

void call_math (math_ptr op , int x , int y){
    printf ("Result ", op(x,y));
}
void hello() {
    printf(" Hello there ");
}
int main() {
    math_ptr ptr = add;          // direct call
    call_math (ptr , 10 , 20);

    math_ptr ops[] = {add, sub};
    printf ("ops 1 = %d ", ops[1](50,30));      } Array of pointers

    void (*say_hi)() = hello;
    say_hi();
    return 0;
}
```

Queue

enqueue adds at the rear
dequeue removes from the front

1) FIFO

The Queue is working under Queue Principle

2) Array-based Queue implementation

```
#include <stdio.h>
#define MAX_size 100
typedef struct Queue {
    int queue [MAX_size];
    int front;
    int rear;
} Queue;
```

// Initialize queue is empty.

```
void initializeQueue (Queue *q) {
    q -> front = -1;
    q -> rear = -1;
}
```

// Check if the queue is empty

```
int isempty (Queue *q) {
    return (q -> front == -1);
}
```

// check if the queue is full

```
int isfull (Queue *q) {
    return (q -> rear == MAX size - 1);
}
```

// Insert element at rear

```
void enqueue (Queue *q, int data) {
    if (isfull (q)) {
        Printf ("Queue is full\n");
    return ;
    }
    if (is_empty (q)) {
        q -> front = 0;
    q-> rear ++.
    q -> queue [q -> rear] = data;
    Printf ("enqued :d", data);
}
```

// remove element from the front

```
int dequeue (Queue *q) {
    if (is empty(q)) {
        Printf ("queue is empty");
        return -1;
} int data = q -> queue [q-front];
int data = q -> front == q -> rear) {
    if (q -> front == q -> rear) {
        q -> front = q-> rear = -1;
} else {
    q -> front ++;
}
Printf ("Dequed :d", data);
return data;
}
```

```
void display (Queue *q) {
if(isEmpty (q)) {
    printf ("Queue is empty");
    return;
}
Printf ("Queue");
for (int i=q->front; i<=q->rear; i++) {
    printf ("%d", q->queue[i]);
}
Printf ("\n");
}
int main () {
    Queue q;
    initialize Queue (&q);
    enqueue (&q, 10);
    enqueue (&q, 20);
    display (&q);
    dequeue (&q);
    display (&q);
    return 0;
}
```

3.) Queue overflow

Queue overflow occurs when you try to insert an element into the queue but there is no more space left

4) enqueue and dequeue

5) Circular queue logic

In normal queue, once rear reaches the end, you can't reuse space.

In circular queue, the last position wraps around to the start forming a circular buffer.

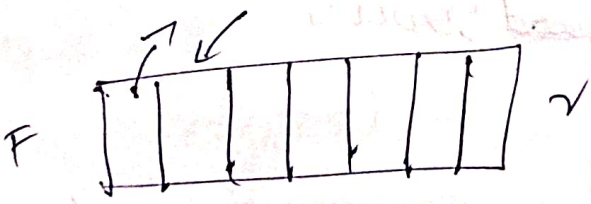
```
rear = (rear + 1) % MAX_size;
```

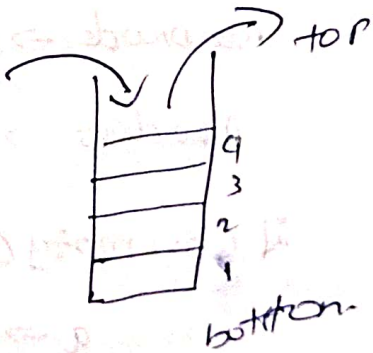
6) Detect full in circular queue:

```
int isfull (queue *q) {
return ((q -> rear + 1) % MAX_size) == q -> front.
```

7) Queue vs stack

Feature	Queue	Stack.
order	FIFO	LIFO
insertion	At - rear.	At top
Deletion	from front	from top.
use cass	Task scheduling, buffers	Recursion, undo/redo





Q.) Queue using linked list

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node {
    int data;
    struct node *next;
} node;
typedef struct {
    Node *front, *rear;
} Queue;

Void Initialize (Queue *q) {
    q->front = q->rear = NULL;
}

int Is empty (Queue *q) {
    return q->front == NULL;
}

Void enqueue (Queue *q, int data) {
    Node *newNode = (Node*) malloc (sizeof (Node));
    Newnode -> data = data;
    New done -> next = ~~head~~ NULL;
    if (Is empty (q)) {
        q -> front = q -> rear = newnode;
    } else {
        q -> rear -> next = newnode;
        q -> rear = newnode;
    }
}
```

```
int dequeue (Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty");
        return -1;
    }
    int data = q->front->data;
    Node *temp = q->front;
    q->front = q->front->next;
    free(temp);
    if (q->front == NULL)
        q->rear = NULL;
    return data;
}

void display (Queue *q) {
    Node *temp = q->front;
    while (temp) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL");
}

int main() {
    Queue q;
    initialize(&q);
    enqueue(&q, 10);
    enqueue(&q, 20);
    display(&q);
    printf("Deque id", dequeue(&q));
    display(&q);
```

Aug - 4

1) Stack [LIFO]

4 functions

1) Push
2) Pop
3) Top
4) Size

end = (end + 1) % SIZE

---

2) Implement stack using array

```
#include <stdio.h>
#define MAX 5

int stack [MAX];
int top = -1;
```

Push

```
void push (int value) {
    if (top == MAX - 1)
        printf ("Stack Overflow");
else {
    top ++;
    stack [top] = value;
    printf ("Pushed %d \n", value);
}
}

void pop() {
    if (top == -1) {
        printf ("Stack underflow");
    }
```

```
        printf("Popped %d\n", stack[top]);
        top--;
    }
}

void display() {
    if (top == -1) {
        printf("Stack empty");
    }
    else {
        printf("Stack");
        for (int i = 0; i <= top; i++) {
            printf(" %d", stack[i]);
        }
        printf("\n");
    }
}

int main() {
    push(10);
    push(20);
    push(30);
    display();
    pop();
    display();
    return 0;
}
```

Ans: 3) what is stack overflow?

1) Overflow happens when you try to push more of
4. than the stack can hold.

---

4) Recursive functions and stack

A recursive function is a function that calls itself
2) = solve smaller versions of the same problem.

```
void countdown (int n) {
    if (n == 0) {
        printf ("lift.");
        return 0;
    }
    printf ("%d \n", n);
    countdown (n - 1);
}
```

OP

if n=3

3
2
1
lift

How stack works with Recursion

every time you call a function, it creates a stack frame
a small memory box on the call stack.

This frame stores:

*) Function arguments
*) local variables
*) Return address

when a recursive functions call itself, new stack frames
are pushed. each time, and when reaches the set point

5) Push and pop operation

*) push means add to top

*) pop means remove from top.

6) stack in expression evaluation

stacks are used to :

*) evaluate postfix expressions like 23+ [means 2+3]

*) convert infix to postfix like (a+b)

You push numbers and pop them when you get an operator

7) call stack frame in recursion

every function call:

*) gets a stack frame, a memory box that stores variables and return address

*) once done, it's popped off.

9) check stack underflow

underflow happens when you try to pop from an empty stack.

if (top == -1).

10) complexity of push / pop :?

Time compl

push O(1)

pop O(1)

8) Stack using linked list

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

struct Node {
        int data;
        struct Node* next;
};

struct Node* top = NULL;

void Push (in val) {
    struct Node* Newnode = malloc (sizeof (struct Node
    newnode -> data = val;
    newnode -> next = top;
    top = Newnode;
    Printf (" Pushed %d in ", val);
}

void pop()
    if (top == NULL) {
        Printf ("Stack underflow");
        return;
    }
    Printf ("Poped %d in", top -> data);
    struct node * temp = top
    top = top -> next;
```

```
        free(temp);
    }
void display() {
        struct node* ptr = top;
        Printf("Stack");
    while (ptr != NULL){
            Printf("%d", ptr -> data);
        ptr = ptr -> next;
    }
    Printf("\n");
}
int main() {
        Push(10);
        Push(20);
        Push(30);
        display();
        Pop();
        display();
    return 0;
}
```