18.01.25
20.01.25

# 100 Logical Reasoning Questions in C

1. Bit Operations :-

1. int a=5, b=3
   printf("%d", a^b);

Ans 1.

2. int x=6;
   int result = x | 3;
   printf("%d", result);

Ans: 7

3. int x=4;
   printf("%d", x^x);

16.05.25

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

Ans: 4

5. int x=8;
   x = x>>1;
   printf("%d", x);

Ans: 4

XOR Rule

| |
|---|
| 0 ^ 0 = 0 |
| 1 ^ 1 = 0 |
| 0 ^ 1 = 1 |
| 1 ^ 0 = 1 |

>> right shift
<< left shift

6. ~(5) equals to?

~ (NOT

| | | |
|---|---|---|
| 0 | → | 1 |
| 1 | → | 0 |

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

1 0 1 0

→ bit position counted from 0 (right most).

0 1 0
3 2 1 0

Ans: 3rd bit 1.

2 | 10 — 0
2 | 5 — 1
2 | 2 — 0
2 | 1

XOR

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

8. Swap Two Numbers using XOR.

```
int main()
{
int a=5, b=10;
a=a^b;
b=a^b;
a=a^b;
printf("%d %d", a, b);
}
```

O/P:-

a=10
b=5.

a=15

b= 1111 / 1010 → 0101 ; 1010 — 10 / 0101 — 5 / 1111

b=5

a= 1111 / 0101 → 1010

a=10

1 × 2^0 = 1
1 × 2^1 = 2
1 × 2^2 = 4
1 × 2^3 = 8
15

9. what does x & ~(1<<n)

x = 5
n = 2

⇒ 0101 & ~(1<<0010)

⇒ 0101 & ~(0100)

⇒ 0101 & 1011

= 1 //.

0100 — 4

1 × 2^0 = 1
1 × 2^1 = 2
0 × 2^2 = 0
1 × 2^3 = 8
11

0101
1011
0001

10. How to set the $n^{th}$ bit of a number?

To use OR operator

Logical operations :-

11). int a=0, b=5;
printf("%d", a && b); Ans: 0

0000
0101 (&&)
0000

12. int a=1, b=0;
printf("%d", a || b); Ans: 1.

0001
0000
0001

1 (logical NOT)
0 -> 1 -> True
1 -> 0 -> false

13). !0 equals to? Ans: True.

14. int x=0;
printf("%d", x || ++x); Ans: 1.

0001
0000
0001

15. Difference between '&' and '&&' in C?

0001
0000
0000

16. what is the output of '0 && 1 || 1'?

0000 || 0001

Ans: 1

17. !(a && b) == !a || !b.

int a = 5 b = 8

0101 - 5
1000 - 8

0101
1000
0000

1010
0111 (||) 1111
(1111)
↳ 1

Ans: True.
↳ equal.

18. What is precedance: && or ||?

19. int a = 1;
int b = 1;
if (a-- && ++b)
{ printf("%d", b);
}.

0010
0000
0000

Ans: b = 0.

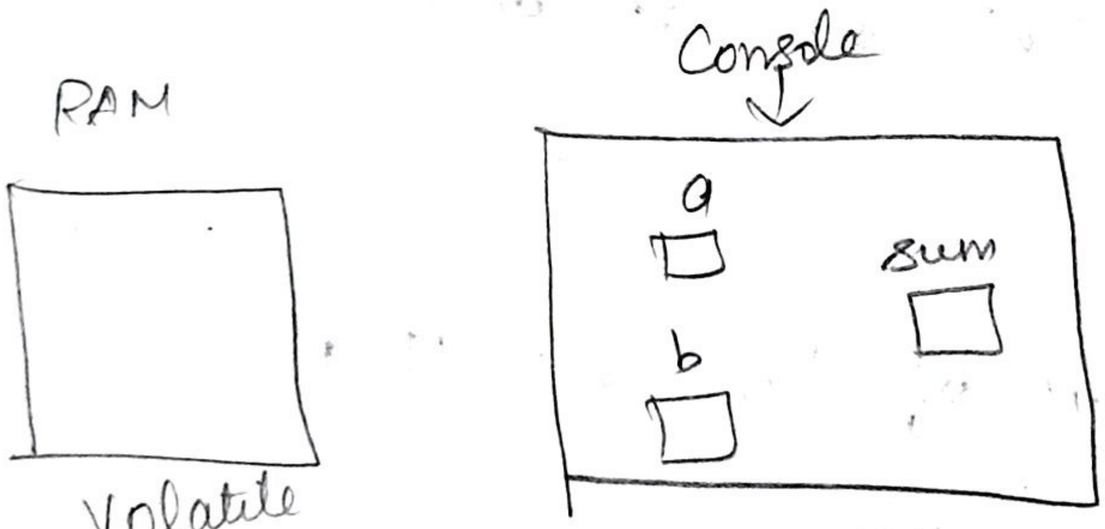
20. can short-circuit evaluation prevent seg fault?

1's and 2's complement.

21.07.25

---

File Handling :-




When the PC switch off / on the all files are gone.

It contains Three mode

r → read the file data
w → write a file
a → write

Difference between w & a w → it create a new file and write. a → it point the old file data to add something lines.

1S, 2S complement

1S Complement

0 → 1

1 → 0.

find the 1S complement of

$(11010100)_2$.

1S → $(00101011)_2$

2S complement.

the given binary to change the 1S complement then to add the 1.

find the 2S compliment of $(11000100)_2$.

1+1 in binary = 10.

Step 1 :-

1S → 00111011

Step 2 → Add 1.

$$\begin{array}{r} 00111011 \\ +\quad\quad 1 \\ \hline 00111100 \end{array}$$

Ans :- $(00111100)_2$.

21). What is the 1's complement of 0101 ?

Ans :- 1010.

22). what is the 2's complement of 0101 ?

Ans :-

1S → 1010

Add 1 → 1

1011

Ans :- 1011

23). Find -5 in binary using 2's complement :-

Ans: 00000101

1S → 11111010

2S → 11111010 + 1

1011

Ans :- 11111011.

24) Is ~x+1 equal to -x?

x = 0110

~x = 1111 1001

+ 1

1111 1010

= x = 0000 0110

$2_2$ = 1111 1001

+ 1 (+)

$2_3$ → 1111 1010

Condition is true.

25) Convert -10 to 2's complement in 8-bit.

x = 0000 1010 is → 1111 0101

2s → 1111 0110.

2 | 10 - 0
2 | 5 - 1
2 | 2 - 0
1

1

1111 0110

5

26). Difference between signed and unsigned representation?

27). overflow detection in 2's complement addition :-

When the sum is not representable # bits

to get overflow.

+ (positive no) + + (no) if result it - then it overflow

- + (-(no)) → the result return - then it overflow

28). How is subtraction done using 2's complement?

A = 6 B = 5

to change this part 1s, 2s. then mins

0110 → 0101 ⇒ 1010 → 1010 / 1 / 1011

↓ minuend ↓ Subtrahend.

0110 / 1011 / 0001

Ans:- 0001

29. what is the result of ~(-1)?

1 → 0001

1s → 1110

1

2s → 1111

~(1111) → 0001 / 1 / 0010

Ans:- 0000

30). 1's complement of -8 in 8bit system?

8 → 00001000

⇒ -8 → 11110111

2 | 8 - 0
2 | 4 - 0
2 | 2 - 0
1

~~Ans~~: 11111000. // Ans.

+8

-8 → 11111000

Ans :- 11110111 // Ans.

Array:

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

32. Accessing out of bounds: arr[10] = 5; this return the undefined behaviour.

33. Difference between arr and &arr?

| | type | point to |
|---|---|---|
| arr | int * | The first element (&arr[0]) |
| &arr | int (*)[5] | The entire array. |

34. Size of array using sizeof operator.

```
int num[] = {20, 30, 40, 50};
int len = sizeof(num) / sizeof(num[0]);
```

↓ gives the total size of array in bytes.

↓ gives the size of element

35. `int a[5] = {0};` a[0] = 0, the rest of a[1] to a[4] are automatically initialized 0.

What is a[4]?

The array is partically initialized, the remaing elements to are set to zero automatically.

36. How to pass array to function?

```
#include <stdio.h>
int fun_array(int arr[], int len)
{
    for(int i=0; i<len; i++)
    {
        printf("%d %d", i, arr[i]);
    }
}
int main()
{
    int arr[] = {20, 40, 70, 90};
    int len = sizeof(arr) / sizeof(arr[0]);
    printf("Array values are :");
    fun_array(arr, len);
    return 0;
}.
```

Example :-

```
int arr[] = {2, 4, 6, 8, 10};
int *q;
q = arr; // It means the q point the base address of array.
```

```
int b = 10;
int *p;
p = &b; ✓
p = b; x → the pointer does not store the integer value.
```

To print address to use the "-/-p".

printf("-/-p" arr+1);

("-/-p", &a+1); // Print the garbage value - 104 4 / 132

```
int a[5], i;
int *q = a;
for(i=0; i<5; i++)
  { scanf("%d", &a[i]);
  }
for(i=0; i<5; i++)
{
  printf("-/-d", a[i]); *(q+i), *(a+i)
                          i[a], i[q];
  printf("-/-p", *(a+i));
                  printf("  ", *(a+i));
```

| 104 | 108 | 112 | 116 | 120 | 124 | 128 | 132 | 136 | 140 | 144 | 148 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

104 + 12 / 116

1) 104

2) 104 + 12 * 4 // garbage

3) 104 + 3 * 4 = 104 + 12 = 116

4) 104.

6) 1

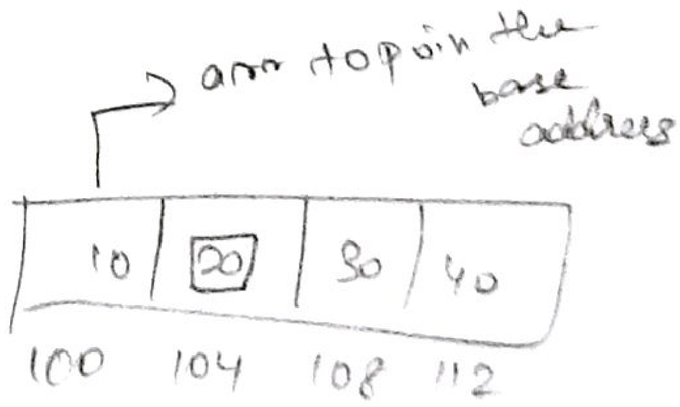
7)

8). 1 9. 2 10. 3

11.

38) What does *(arr+i) mean?

```
int main()
{ int i = 0
  int arr = {10, 20, 30, 40};
  printf("%d", *(arr+i));
}
```




100 + 1 * 4 = 100 + 4

= 104.

Ans = 20

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

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

Ans = 3.

(40). Array of pointers vs pointer to array?

Array of pointer :-

Syntax:

```
int main. datatype . *arrayname [size];
{
int  a[5] = {10, 20, 30, 40, 50};
int  *q[1];
for (int i=0 ; i<5 ; i++)
    {
        q[i] = &a[i];
    }
for (int i = 0 ; i<5 ; i++)
    {
printf("Array of elements :%d ", *q[i].
    }
}
```

Pointer to array:-

```
void  Poin_array (int (*P) [5])
  {
     for (int i=0 ;; i< 5; i++)
         {
           Printf ("%d", (*P) [i]);
  } }
int  main()
 {
   int arr[5] = {1, 2, 3, 5, 5};
     Poin_ array (& arr);
       return 0;
 }.
```

24.07.25
Section - 3.

8). How do you allocate a 2D array dynamically?

```
int **arr;
int row = 3, col=4;
arr = (int **) malloc (row * sizeof (int *));

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

arr[0][0] = 10;

for (int i=0; i<row; i++)
  {  free (arr[i]);
  }
free (arr);
```

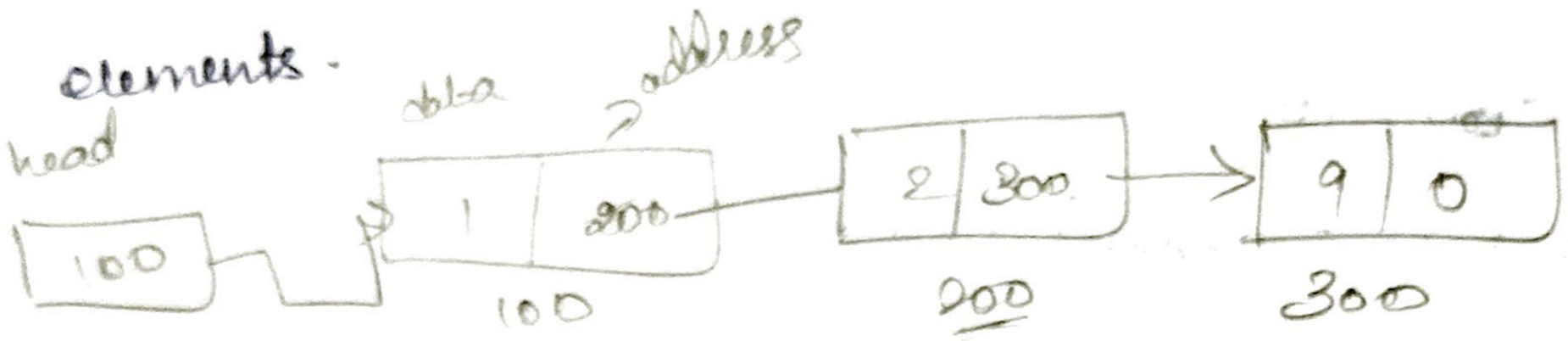
4 x 4 = 16

9. What happens if you free() memory twice?

calling free() twice on the same pointer is known as a double free, and it causes like a undefined behaviour.

41. What is Linked List :-

A Linked list is a dynamic data structure made of node. where each node contains data and pointer to the next node. It allows efficient insertion and deletion without shifting elements.




```
Struct node
{ int data;
  Struct node * next;
};
Struct node *head, *newnode, *temp;
int choice
head = 0; or NULL; → while(choice){
(Struct node *)
newnode = (node *) malloc (sizeof (struct node));
Printf ("Enter data");
scanf ("%d", &newnode->data);
newnode -> next = NULL;
if (head == 0)
{
temp = head = newnode;
}
else
{ .
```

newnode

temp

```
head -> next = newnode;
    temp -> next = newnode;
    temp = newnode;
}
printf("Do you want to continue (0,1) ?");
scanf("%d", &choice);
}
temp = head;
while (temp != 0)
{
    printf("%d", temp -> data);
    temp = temp -> next;
}
getch();
}
```

42). How to traverse a Linked list?

```
temp = head;
while (temp != 0)
{
    printf("%d", temp -> data);
    temp = temp -> next;
}
```

43). Detect loop in Linked list?

44. Insert node at end in singly list.


int Insertion at front ( )

{

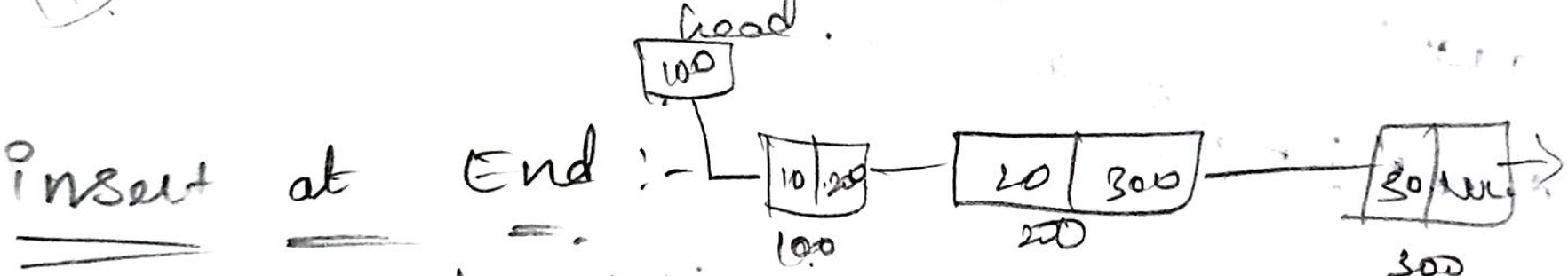
newnode = (struct node *) malloc (sizeof (struct node));

Printf ("Enter data you want to insert:");

Scanf ("%d", &newnode->data);

newnode -> next = head;

head = newnode;

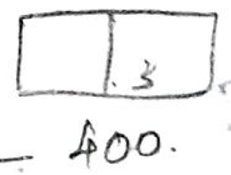



Insert at End :-

struct node *temp;

{

newnode = (struct node *) malloc (sizeof (struct node))




Printf ("Enter a data you want to insert at End");

&scanf ("%d", &newnode->data);

newnode->next = 0;

temp = head;

while (temp->next != 0)

{

temp = temp->next;

}

temp->next = newnode;

}.

Insertion at Mid :-

```
int pos, i = 1;
struct node *head, *newnode, *temp;
newnode = (struct node *) malloc (sizeof(struct node));
printf ("Enter the data & position");
scanf ("%d", &newnode->data);

If ( pos > count )
{
  Plint ("Invalid position");
}
else
{
    temp = head;
    while ( i < pos )
    { temp = temp -> next;
      i++;
    }
Plintf ("Enter data");
Scanf ("%d", &newnode->data);
newnode->next = temp -> next;
temp -> next = new node;
}
```

45. Delete node in singly linked list :-

Delete from Front.


```
Delete_From_t ( )
{
    struct node *temp;
    temp = head;
    head = head -> next;
    free (temp);
}
```

Delete_End ( )

{

struct node *temp, *prev_node;

temp = head;

while (temp->next != 0)

{

prev_node = temp;

temp = temp->next;

}

if (temp == head)

{

head = 0;

}

else

{ prevnode->next = 0;

} free(temp);

}.

Delete at mid :-

Delete_End ( )

{ struct node * temp, *nextnode;

int pos, i = 1;

temp = head;

printf ("Enter the position ");

scanf ("%d", &pos);

while (i < pos-1)

{ temp = temp->next;

i++;

}

nextnode = temp->next;

temp->next = newnode->next;

free (nextnode);

}.

46. Reverse a Singly Linked list.

```
#include<stdio.h>
#include<stdlib.h>
struct Node {
    int data;
    struct Node *next; };
struct Node *front=NULL;
struct Node *real=NULL;
void enqueue(char val){
struct Node * newnode=(struct node*) malloc(sizeof(struct
                                                Node));
newnode->data = val;
newnode->next = NULL;
if(real==NULL){
    front=real=newnode;
}else
{
    real->next=newnode;
    real=newnode;
}}
char dequeue(){
if(front==NULL){
    return '\0';
}
struct Node *temp=front;
char val = temp->data;
front=front->next;
if(front==NULL){
    real=NULL;
}
free(temp);
}
void revQueue(){
 struct Node * revfront=NULL;
 struct Node *revreal=NULL;
 while(front!=NULL)
 {
    char ch=dequeue();
struct Node *newnode=(struct Node*)
        malloc(sizeof(struct Node));
newnode->data=ch;
newnode->next=revfront;
revfront=newnode;
if(revreal==NULL)
  {
    revreal=newnode;
  }
  front=revfront;
  real=revreal;
}
void display(){
  struct Node *temp=front;
while(temp!=NULL){
  printf("%c",temp->data);
  temp=temp->next;
}
}int main()
{
  int size;
  printf("Enter the size of string");
  scanf("%d",&size);
  char ch;
  printf("Enter the string");
  for(int i=0;i<size;i++){
      scanf("%c",&ch);
      enqueue(ch);
  }
  reverse_queue();
  display();}.
```

47. Difference : singly vs doubly list.

| Feature | Singly linked list | Doubly linked list. |
|---|---|---|
| Node Structure | Each node contain data and a pointer to the next node | Each node contains data, a pointer to the node, pointer to the previous node. |
| Traversal | It can only be traversed in one direction : forward | It can be traversed in both directions backward, forward. |
| Memory usage | use less memory per node (one pointer). | uses of more memory per node (two pointers). |
| Example. | Implementation of simple queue. (FIFO) | Implementation of navigation systems, undo functionality in applications more complex Queues, Stacks. |

48. Node definition with typedef.

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

#typedef allows you to use Node directly instead of writing struct node every time

49. Free all nodes in list.

```
void free_list()
{
    struct node *temp;
    while(head != NULL){
        temp = head;
        head = head -> next;
```

```
        free(temp);
    }
    printf("All node have been freed");
}
```

50). Why use linked list over array:-

We can

when we need dynamic memory allocation, efficient insertions, deletions and don't know the size of data in advance.

Arrays are good for index access, but they require contiguous memory and fixed size, which flexiability.

linked list don't waste the memory.

25.07.25

51. What is FIFO?

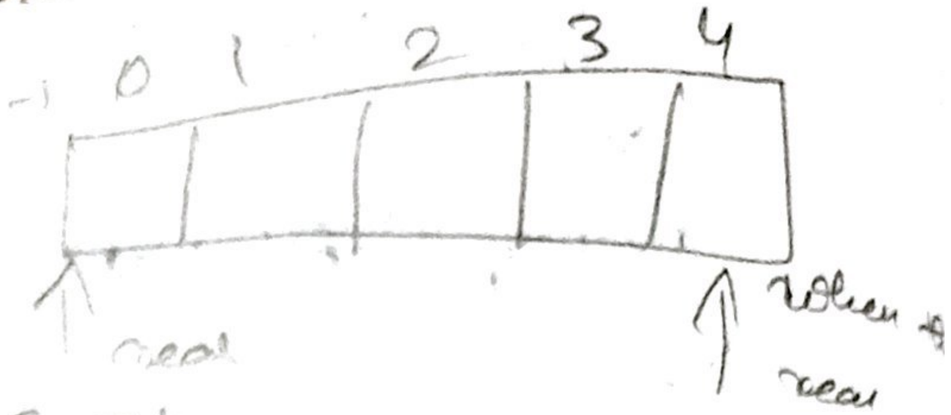
→ FIFO stands for First in First Out.

→ A queue is a linear data structure that follows the FIFO principle. this means that element are added to one end ("rear" or "tail") and removed from the other end ("front" or "head") the element that was inserted first into the queue will be the first one to be removed.

Ex:- Ticket counter.

The first person in a line is the first to be served.

52). Array based implementation in Queue :-



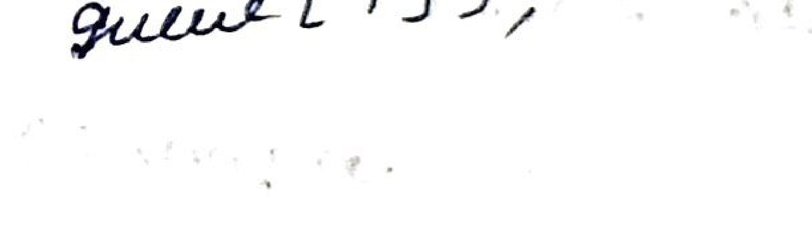

TWO Ends :-

Front → To remove the element in front

Rear → To add the element in the rear.

```
#define N 5;
int queue [N];
int front = -1;
int rear = -1;
void enqueue (int x)
{
    if (rear == N-1)
    {
        printf ("Queue is full");
    }
else if ( front == -1 && rear == -1)
    {
        front = rear = 0;
        queue [rear] = x;
    }
    else {
        rear ++;
        queue [rear] = x;
    }
}
void dequeue ()
{
    if ( front == -1 && rear == -1 )
    {
        printf ("Queue is Empty");
    }
```

```
else if ( front == real )
  {
    front = real = -1;
  }
else
  {
printf (" Dequeue element is : %d ", queue [front]);
front ++;
  }
}
void display()
{ if (front == -1 && real == -1)
    { printf (" Queue is Empty ");
      }
else
    { for ( int i = front ; i <= real; i ++ )
      { printf ( " %d ", queue [ i ]);
      }
    }
}
void peak ()
{ if (front == -1 && real == -1)
    { printf (" Queue is Empty ");
    }
else
    { printf ( " %d ", queue [front]);
    }
}
int main()
{ dequeue ()  // Queue is Empty.
  enqueue (10);
  enqueue (20);
  enqueue (30);
  enqueue (40);
  display()      dequeue (); // 20
```



53. What cause queue overflow?

Queue Overflow happens when the try to insert the new value in the full Queue - meaning there's is no more space left to add a new element.

Queue overflow occurs when an attempt is made to add an element to a queue that is already full. This means the queue has reached its maximum capacity and cannot accept any more elements until some are removed.

54. Enqueue and dequeue opera functions :-

The enqueue() function is used to add to element to the real of the queue.

The dequeue() function is used to remove the element from the front of the queue.

55. Circular Queue logic :-

In the Circular Queue. that uses a fixed-size array in circular way - so once the end is reached, it wraps around to the beginning (if space is available).

```
#define n 5;
int queue[n];
int front = -1;
int rear = -1;
```

```
void enqueue (int x)
{ if (front == -1 && real == -1)
    { front = real = 0;
      queue [real] = x;
    }
  else if ((( real + 1) % N) == front) //front
    { printf ("Queue is full \n");
    }
  else
    { real = (real + 1) % N;
      queue [real] = x;
    }
}
void dequeue()
{ if (front == -1 && real == -1)
    { printf ("Queue is empty");
    }
  else if((( front + 1) % N == front))
  else if ( front == real)
    { printf ("Deleted Queue : %d \n", queue [Front]);
      front = real = -1;
    }
  else
    { printf ("deleted Queue : %d", queue [Front]);
      front = (front + 1) % n;
  } }
void display ()
{ int i = front;
  if ( front == -1 && real == -1)
    { printf ("Queue is Empty");
  } }
```

0+1 1+5 1 == 3

(0+1)+1.5 == 0

(4+1)%5 == 3

5+1.5 == 3

0 == 3 X

2+1.5 == 0

2 == 0 X

(0+1)+1.5

1+1.5 = 1

real = 0

queue [real] = 20 real = 1

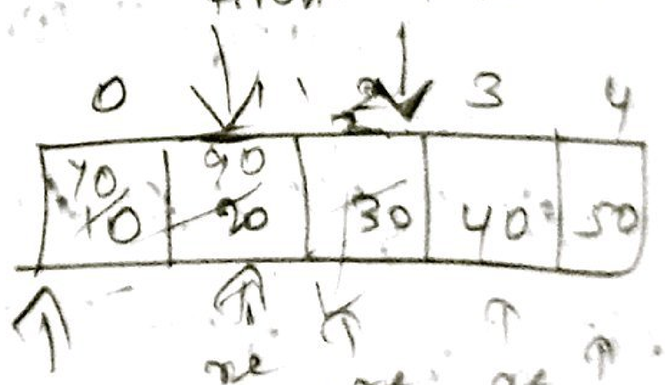
queue [1] = 20

0+1+1.5 =

2+1.5 = 2 1

2

```
else
{ Printf("Queue Contains ");
  while (i != rear)
  { Printf("%d", queue[i]);
    i = (i+1)%n;
  }
Printf("%d\n", queue[rear]);
}}
void peek()
{
if (front == -1 && rear == -1)
  { printf("Queue is Empty");
  }
else
  { Printf("The peek element is: %d\n", queue[front]);
  }}
int main()
{
  enqueue(10);
  enqueue(20);
  enqueue(30);
  enqueue(50);
  enqueue(60);
  dequeue();
  dequeue();
  display();
  enqueue(70);
  enqueue(90);
  display(); enqueue(100);
  dequeue();
  peek();
```




(1+1)%5 == 2

2%5 == 2

2 == 2 front = 0.

0+1 %5 = front

1%5 = 1

re = 4

F = 2.

F = 2.

re = 1

Queue is full.

re = 4

display(); → 70, 90, 30, 40, 50

(5.55) Circular Queue logic :-

```
rear = (rear + 1) % size;
front = (front + 1) % size;
```

These two logic helps to prevent the spaces after dequeue ..

Difference : queue vs Stacks.

| Queue | Stack. |
|---|---|
| → Queue using First In First out principle. | → the Stack using Last In First out principle. |
| → Elements are added and removed from the same end | → Elements are added at the rear End remove from the Front End. |
| → Primary operations in the Enqueue (to add an item), dequeue (remove an item), Front (view the first item, Rear view the last item). | → Push (add an item), pop remove an item), peek (view the top item). |
| → can be implemented using array, Linked lists, Circular buffer. | → can be implemented using array or Linked list. |
| Push : O(1), Pop O(1), Peek O(1) | enqueue : O(1), dequeue : O(1), Peek O(1). |
| Ex:- Reversing a word | front : O(1) |

58. Use queue to reverse a String :-

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define Max 100
char queue[Max];
int front = -1;
int real = -1;
void enqueue(char ch)
{if(real == = Max-1)
   { printf("Queue is overflow");
   }
 if (front == -1)
    { front = 0; real = 0;
      queue[ real] = ch;
    }
}
void dequeue()
  {
   if( front == -1 && front > real)
       {
          Printf ("Queue is Empty");
          return '\0';
       }
      return queue[ front++];
  }
char Stack[Max];
int top=-1;
void push(char ch)
  { if (top == Max-1)
          {
```

```
        printf("Stack Overflow");
    }
    else
    {
        top++;
        Stack[top] = ch;
    }
}
char pop()
{
    if(top == -1)
    {
        printf("Stack is Empty");
        return '\0';
    }
    else
    {
        return Stack[top--];
    }
}
int main()
{
    char str[Max];
    Printf("Enter a string");
    scanf("%s", str);
    for(int i=0; str[i] != '\0'; i++){
        enqueue(str[i]);
    }
    while(front <= real)
    {
        char ch = dequeue();
        push(ch);
    }
Printf("Reverse a string:");
    while(top != -1){
        printf("%c", pop());
    }
    printf("\n");
    return 0;
}
```

59. Queue using Linked list :-

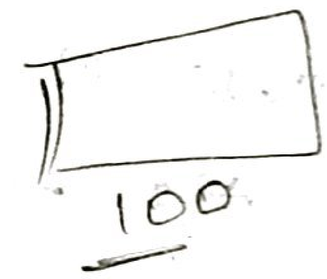
enqueue => O(1)
dequeue => O(1)

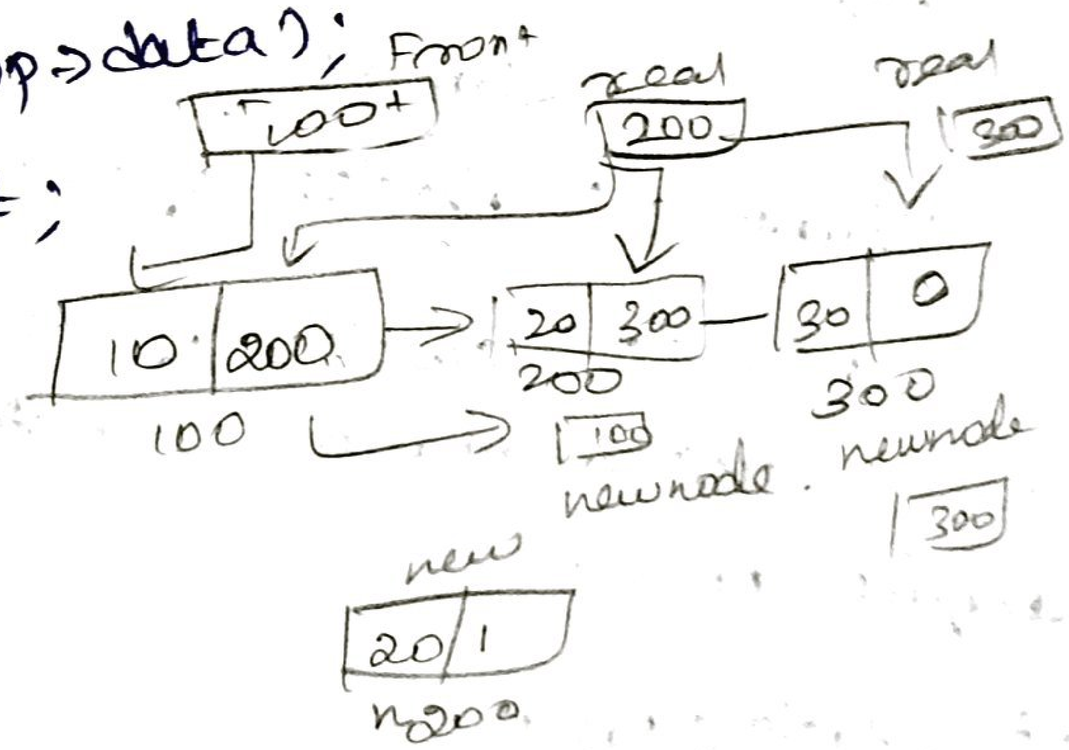
```
Struct node {
    int data;
    Struct node * next;
};

Struct Node* front = NULL;
Struct Node* real = NULL;

void enqueue (int value)
{
    struct node * newnode = (struct Node * ) malloc
                            (Sizeof (struct Node));

    newnode -> data = value;
    newnode -> next = NULL;

    if ( real == NULL)
        { front = real = newnode;
        }
    else
        {
            real -> next = Newnode;
            real = newnode;
        }
Printf (" Inserted : %d \n", value);
}
void dequeue ( )
{ if (front == NULL)
    {
        Printf (" Queue is empty, Can't be delete");
        return;
    }
else
```

```
struct Node *temp = front;
printf("Deleted : %d", temp->data);
front = front->next;
if(Front == NULL)
{ rear = NULL;
  free(temp);
}
void display(){ ⇒ temp=front
struct Node * temp = front;
if(temp == NULL)
  { printf("Queue is empty");
  }
printf("Queue:");
while(temp != NULL)
  { printf("%d", temp->data);
    temp = temp->next;
  }
printf("\n");
}
int main()
{
 enqueue(10);
 enqueue(20);
 enqueue(30);
 display();
 enqueue(50);
 display();
 dequeue();
 display();
}
```

front
100
temp=100

100





bo. complexity of enqueue?

The complexity of enqueue in linear array based queue implementation is

→ O(1) → constant time.

Stack :-

It follow the LIFO (Last in First out) Principle.

Insert and deletion at the same End.

push() → This operation to used to add the element in the stack.

pop() → This operation is used to remove the element.

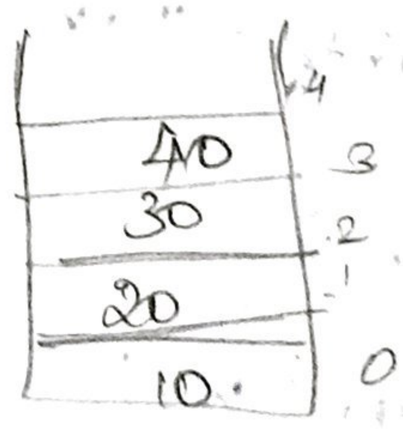
peek() → It is used to return the top of the element.

isEmpty() → checks if the stack is empty.

isFull() → checks if the stack has full or not.

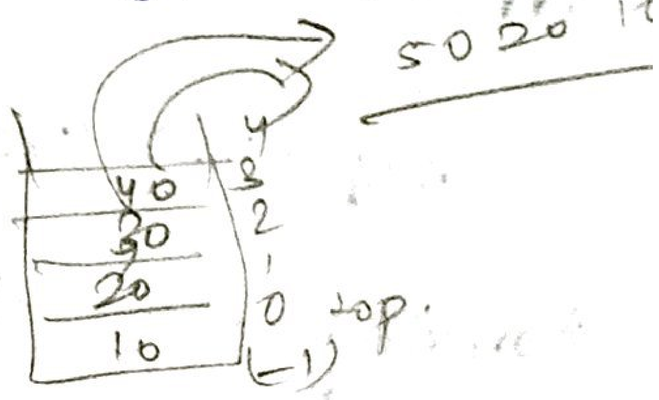
72). Stack implementation using Array.

```
#define N 5;
int stack[N];
int top = -1;
void push(int x)
{ int x; //
  Printf("Enter data "); //
  Scanf("%d", &x); //
  if(top == N-1)
  {
    Printf("Overflow");
```



```
else
{
  top++;
  stack[top] = x;
}
}
void pop()
{
  int temp;
  if(top == -1)
  { printf("underflow");
  }
else {
   temp = stack[top];
    top--;
   printf("%d", temp);
}
void peak()
{
  if(top == -1)
    { printf("Stack is Empty");
      }
  else
    {
      printf("%d", stack[top]);
    }
}
void display()
{
  int i;
  for(i=top; i>=0; i--)
    {
      printf("%d", stack[i]);
  }  }
```

```
void main()
{
 push(10);
 push(20);
 push(30);
 push(40);
 pop(); // 40
 display(); 30, 20, 10
 pop(); 30 → out
 push(50); 50
 Display();
}
```



11). What is LIFO :-

LIFO stands for "Last In First Out". It is a principle where the last element added inserted into a collection is the first one to be removed. This is commonly used in stack data structure in C Programming.

12. What is Stack overflow?

When the stack is full and we try to add another element to get the stack overflow.

if (top == n-1) // when this condition true to get the stack overflow

13. Recursive function and Stack?

When a recursive function is called, the system uses a call stack to manage function calls. Each time the function call itself, a new stack frame is created to store.

→ parameters
→ local variables
→ Return address.

```
int fact_in (int n)
{
  if (n==0) {
    return 1;
  else
    return n* factorial (n-1);
}.
```

14. Push and pop operations :-

Push () → the push operation is used to add the element to the top of the stack

pop () → The pop operation is used to Removing the top of the element from the stack.

76). Stacks in expression evaluation?

Stacks are used in expression evaluation to manage operators and operands efficiently. They help convert and evaluate Infix, postfix, prefix expressions.

Infix :- A+B * C
postfix :- ABC * +
prefix :- + A * BC.

} These are the stack expression evaluation.

77. Call stack frame in recursion:

A call stack frame is a block of memory created for each function call.

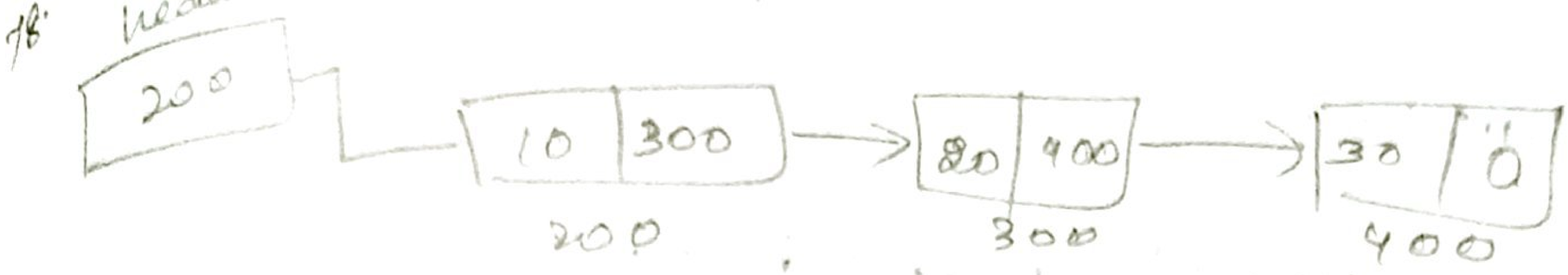
→ every time a function calls itself, the system creates a new frame on the call stack

store → function parameters

→ local variables.

→ the return address.

# stack using linked list.

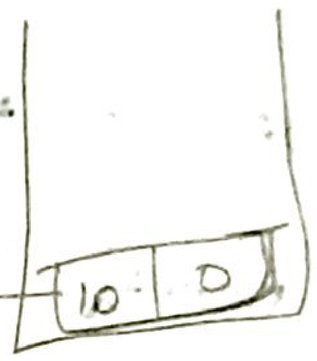



Stack in linked list,

Pop & Push time complexity

O(n). because to

traverse the list and to add a last node and delete also same way. that why the time complexity is O(n).




top=0
head=0.

```
struct node {
    int data;
    struct node *next;
};
struct node *top = NULL;
void push (int x) {
    struct node *newnode;
    newnode = (struct node*) malloc (sizeof (struct node
    newnode->data = x;
    newnode->next = top;
        top= newnode;
}
void display()
{ struct node *temp;
    temp = top;
    if (top==0)
```

```
else
  { while (temp != 0)
      { printf ("%d", temp->data);
        temp = temp->next;
      }
  }
void peek()
{
  if (top == NULL)
    { printf ("Stack is Empty");
    }
  else
    { printf ("%d ", top->data);
    }
}
void pop()
  {
    struct node *temp;
    temp = top;
    if (top == NULL)
      { printf("stack is empty");
      }
    else
      { printf ("poped element : %d", top->data);
        top = top->next;
        free (temp);
      }
  }
void main()
  { push (20);    display();
    push (30);    peek();
```

79. check stack underflow?

Stack underflow occurs when try to pop or peek from an empty stack.

top == NULL. When the condition is true stack underflow happen.

80. complexity of push / pop?

The time complexity of both push() and pop() operations in stack. is O(1).

Heap and Memory Errors :-

81. malloc vs calloc :-

| Malloc | calloc. |
|---|---|
| → malloc() function creates a single block of memory of a specific size | → calloc() function assigns multiple block of memory to a single variable. |
| → The number of arguments in malloc() is 1. | → The number of coguments in calloc is 2. |
| → malloc is fastel. | → This is slowel |
| → This is high time efficiency | → This is low time efficiency. |
| → The memory block allocated by malloc() has a garbage value. | → the memory block allocated by calloc() is initialized by zero. |
| → the malloc indicated the memory allocation. | → the calloc() indicated contiguous allocation. |

83). What is segmentation fault?

A segmentation fault is a runtime error that occurs when a program tries to access memory that it's not allowed to access, such as dereferencing a NULL or uninitialized pointer, accessing memory outside the bounds of an array, or using freed memory.

84). Accessing freed memory?

Accessing freed memory is a form of undefined behaviour. It happens when a pointer is used after its memory is deallocated.

-> Accessing freed memory means using a pointer after the memory it points to has been released. This is undefined behaviour, and cause segmentation fault, data corruption, or unexpected crashes.

84). Double free error?

A double free error occurs when a program calls free() more than once on the same memory address. This leads to undefined behavior, memory corruption, or segmentation faults.

Ex:-

```
int main()
{
    int *ptr = (int*)malloc(sizeof(int));
    free(ptr); // First free O/C to the free ptr memory.
    free(ptr); // already free the memory again the
    return 0;      Program call it show double free
}.                 error.
```

85. Memory leak example

A memory leak occurs when a program allocates the memory on the heap using malloc() or calloc() but never frees it.

Ex:-

```
int main()
{
    int *ptr = (int *) malloc (sizeof(int));
    *ptr = 10;
    return;
}
```

→ here memory is not free - this is memory leak.
→ free(ptr); → now is correct.

86. Dangling pointer. usage?

A dangling pointer that still points to a memory location that has been freed, deleted or gone out of scope.

```
int main() {
    int *ptr = (int *) malloc (sizeof(int));
    *ptr = 100;
    free(ptr);
    *ptr = 200; → It points the freed memory.
}
```

87) Heap vs Stack Memory.

| Feature | Stack | Heap. |
|---|---|---|
| Memory Allocation | Automatically by compiler | Memory allocation manually by the programmer. |
| Size | Smaller | larger. |
| Lifetime | limited of function/ block scope. | lives until manually freed. |
| Access Speed | very fast | slower (requires pointer derefrencing). |

| | | |
|---|---|---|
| Storage | local variable, function parameters | dynamically allocated memory. |
| freeing memory | Automatically when the function End. | must be done manually (free()). |

88. Segfault on NULL pointer dereference.

The Segmentation fault occurs when a program tries to access memory it's not allowed to, such as a dereferencing a NULL pointer.

89. use-after-free Vulnerability:-

→ A use-after-free occurs when a program continue to use a pointer to memory that has already been freed.

→ This leads of security vulnerabilities such as code injection or exploitation.

```
int main(){
  int *ptr = (int*)malloc(sizeof(int));
  *ptr = 40;
  free(ptr);
  *ptr = 50;
  printf("%d", *ptr);
}
```

90. Tools to detect memory errors?

To detec memory errors like leaks, buffer overflows, or use-after-free bugs.

Tools:-

Valgrind → Detects memory leaks, invalid reads/writes, use-after-free, uninitialized memory.

AddressSanitizer (ASan).

Detects out-of-bounds access, use-after-free, stack/heap buffer overflows.

compiler support: Built into GCC/clang.

They help catch issues early during testing and ensure memory safe code.

Function pointers and callbacks:-

61. Declare pointer to function returning int.

```
int add(int x, int y){
    return x+y;
}
int main()
{ int (*ptr)(int, int);
  ptr = add;
  int result = ptr(10,20);
  Printf("Result : %d\n", result);
}.
```

62. How to pass function pointer to another function?

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

63. Call back function in C :-

```
#include<stdio.h>
void sum(int a, int b)
  {
    printf("%d\n", a+b);
  }
void sub(int a, int b);
  { printf("%d\n", a-b);
  }
void display( void (*ptr)(int, int))
  {
    (*ptr).(5,5);
  }
void main()
  { display(sum);
    display(sum);
  }
```

64. Difference :- Normal vs Call back function.

| Normal Function | callback Function. |
|---|---|
| → A normal function is directly called by your code | → A callback function is passed as an argument to another function. |
| → the normal function calls it explicitly. | → the call back function called inside of another function. |
| → It perform specific task only. | → allow flexible / custom behaviour. |

65- Array of function pointer :-

```
#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 (*arr)(int, int) = {add, sub, mul};
 int x=10, y=20;
 Printf("Sum: %d",
  for(int i=0; i<3; i++)
  { int res= arr[i](x,y);
     Printf("Result of Opelation: %d ", res);
} }
```

66. use function pointer to call add (int, int).

```
#include<stdio.h>
int add (int a, int b){
    return a+b;
}
int main()
{ int (*ptr)(int, int);
  ptr = add;
  int res = ptr(10,20);
  printf("Res: %d", res);
  return 0;
}.
```

67. can function pointers point to static functions?

```
#include <stdio.h>
static int multiply (int a, int b)
{
    return a*b;
}
int main()
{
    int (*ptr)(int,int) = multiply;
    int res;
    res = ptr(10, 5);
    printf("%d\n", res);
    return 0;
}
```

68. what is typedef for function pointer?

The typedef allows you to give a name (alias) to a function pointer type, making your code cleaner and easier to read.

```
#include<stdio.h>
typedef int (*funptr)(int, int);
int add (int a, int b){
    return a+b;
}
int main(){
    funptr ptr = add;
    int res = ptr(10,30);
    Printf("%d", res);
    return 0;
}.
```

69. How is function address stored in memory?

functions are stored in the code (text) segment of memory. The compiler places each function in the .text (code) segment. Each function starts at a specific memory address. This address is used when calling function.

70. void (*fptr)(); what is this?

void (*fptr)() means fptr is a pointer to a function returning void. this allows indirect calling, dynamic behaviour and is the basis of for callbacks.