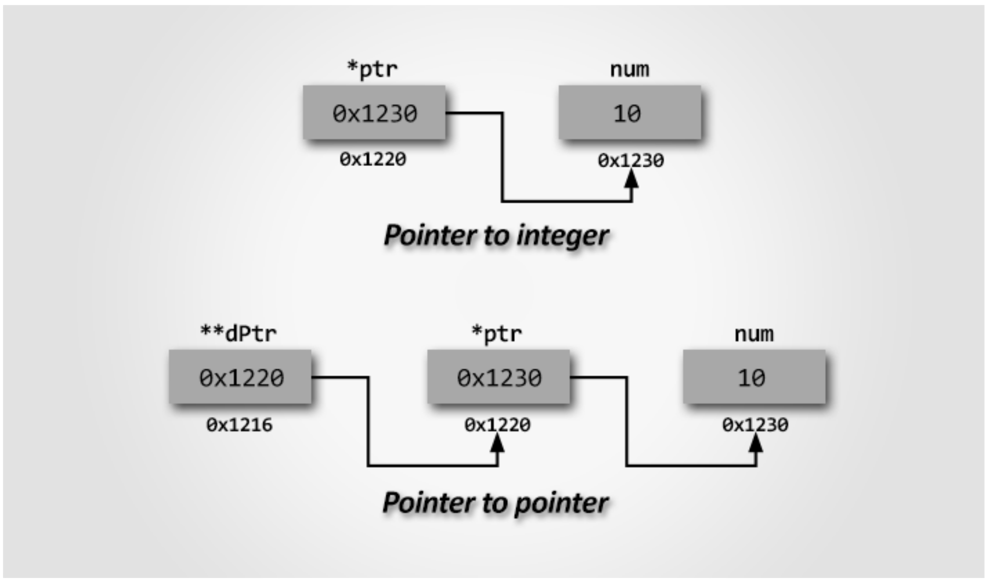
IMPQ (Interview Based)

I. What is pointer to pointer in C?

- In C, a pointer can also be used to store the address of another pointer. A *double pointer* or *pointer to pointer* is such a pointer.
- The address of a variable is stored in the first pointer, whereas the address of the first pointer is stored in the second pointer.
- The syntax of declaring a double pointer is given below:

```
int **p; // pointer to a pointer which is pointing to an integer
```



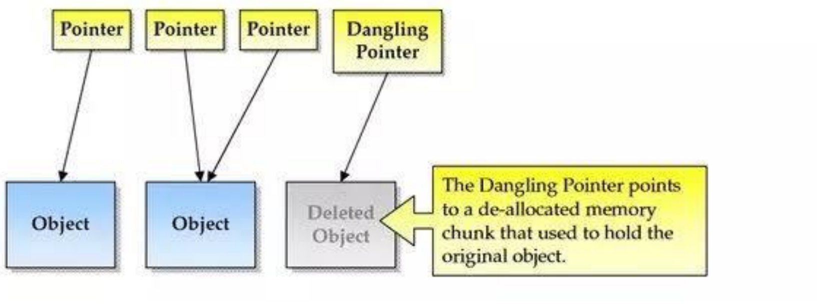
II. What is an R-value and L-value?

- **L-value (Left value)**: This is something that can be on the left side of an assignment. It's a location in memory where a value can be stored. Think of it like a box where you can put things in. For example, x = 10, x is an l-value because it's a place where we can store a value.

- **R-value (Right value)**: This is a value or expression that can't be assigned to. It's the value being stored into an l-value. For example, in x = 10, 10 is an r-value because it's just a value, not a location where we can store something.
- In short:
  - L-value = A place to store something (memory location).
  - R-value = The value you want to store (the value itself).

III. <u>What is a dangling pointer in C? How to solve problem of it.</u>

- A dangling pointer in C is a pointer that continues to reference to a memory location after the memory has been freed or deallocated.


Pointer
Pointer
Pointer
Dangling Pointer
Object
Object
Deleted Object
The Dangling Pointer points to a de-allocated memory chunk that used to hold the original object.

- Accessing such a pointer leads to undefined behaviour, which can cause program crashes, data corruption, or security vulnerabilities.
- ***Causes of Dangling Pointers:***

a) **Deallocation of Memory**

```
int ptr = (int)malloc(sizeof(int));
free(ptr);  // ptr is now dangling
```

b) Returning Addresses of Local Variables

```
int* func() {
    int x = 10;
    return &x; // Returning address of a local
variable
       }
```

c) Using Uninitialized Pointers

```
int *ptr;  // ptr is uninitialized (wild pointer)
*ptr = 10; // Undefined behaviour
```

d) Pointer Going Out of Scope

```
int* ptr;
{
    int x = 5;
    ptr = &x;
} // x goes out of scope, ptr is dangling
```

- ***Solutions to Avoid Dangling Pointers***

***a)* Set the Pointer to NULL after Freeing**

```
int ptr = (int)malloc(sizeof(int));
free(ptr);
ptr = NULL;
// Now it will not be a dangling pointer
```

b) Avoid Returning Local Addresses

```
int* func() {
    static int x = 10;
    // Use static to retain value
    return &x;
}
```

c) Use Smart Pointers (in C++ only)

If using C++, prefer *std::unique_ptr* or *std::shared_ptr* instead of raw pointers.

d) Initialize Pointers Before Use

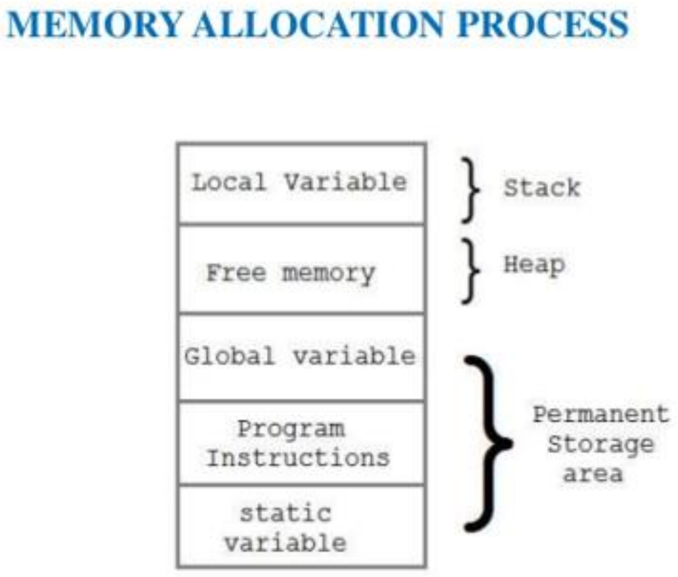
```
int *ptr = NULL;
if (ptr != NULL) {
    *ptr = 10;
// Safe access
}
```

e) Use Valgrind or Address Sanitizer

Tools like Valgrind or AddressSanitizer can help detect memory errors.

By following these best practices, we can prevent dangling pointer issues and make our C programs more robust.

IV. Memory Allocation Process:



The memory is divided into **3 main parts** when a program runs:

1) **Stack (Top):**
   - Stores local variables (like variables inside a function).
   - It *works* like a stack of plates—*last in, first out*.
   - Memory is *automatically managed* (allocated and freed).
2) **Heap (Middle):**
   - Stores dynamically allocated memory (like memory we request using malloc() or new).
   - We *need to manually free* it when done (e.g., using free() or delete).
   - This area grows and shrinks as needed.
3) **Permanent Storage Area (Bottom):**
   - Stores global variables, static variables, and program instructions.
   - These parts of memory *stay fixed* during the program's run.
   - *Used for things that do not change location* or get re-allocated.

V. <u>What is void Pointer?</u>

- A void pointer is a pointer that has no associated data type with it.
- A void pointer can hold an address of any type and can be typecasted to any type.

```
#include <stdio.h>
int main()
{
    int a = 10;
    char b = 'x';

    // void pointer holds address of int 'a'
    void* p = &a;
    // void pointer holds address of char 'b'
    p = &b;
}
```

- ***Properties of Void Pointer:***

  void pointers cannot be dereferenced.

```
Example:
#include <stdio.h>
int main()
{
    int a = 10;
    void* ptr = &a;
    printf("%d", *ptr);

    return 0;
}
```

Output:

```
Compiler Error: 'void*' is not a pointer-to-object type
```

Correct Code:

```
#include <stdio.h>
int main()
{
    int a = 10;
    void* ptr = &a;
    // The void pointer 'ptr' is cast to an integer pointer
    // using '(int*)ptr' Then, the value is dereferenced
    // with *(int*)ptr to get the value at that memory
    // location
    printf("%d", (int)ptr);
    return 0;
}
```

Output: 10

- ***Void Pointer Arithmetic Operation:***

```
#include <stdio.h>
int main()
{
    // Declare and initialize an integer array 'a' with two
    //elements
    int a[2] = { 1, 2 };
    // Declare a void pointer and assign the address of
    // array 'a' to it
    void* ptr = &a;

    // Increment the pointer by the size of an integer
    ptr = ptr + sizeof(int);

    // The void pointer 'ptr' is cast to an integer
    // pointer using '(int*)ptr' Then, the value is
    // dereferenced with *(int*)ptr to get the value at
    // that memory location
    printf("%d", (int)ptr);
    return 0;
}
Output: 2
```

- ***<u>Advantages of Void Pointers in C</u>***
  1. malloc() and calloc() return void * type and this allows these functions to be used to allocate memory of any data type (just because of void *).
  2. void pointers in C are used to implement generic functions (functions that work with any data type, so you can reuse the same code for different types) in C. For example, compare function which is used in qsort().
  3. void pointers used along with Function pointers of type void (*)(void) point to the functions that take any arguments and return any value.
  4. void pointers are mainly used in the implementation of data structures such as linked lists, trees, and queues i.e. dynamic data structures.
  5. void pointers are also commonly used for typecasting.

VI. <u>What is the difference between an array and a linked list?</u>

Array: A fixed-size, contiguous block of memory where elements are accessed using an index.

Linked List: A dynamic data structure where elements (nodes) are connected via pointers, allowing efficient insertion and deletion.

Key Difference: Arrays offer fast indexing, while linked lists provide better memory flexibility.

VII. <u>What is a Segmentation Fault in C? How do you fix it?</u>

A Segmentation Fault (Segfault) occurs when a program tries to access memory that it is not allowed to, such as dereferencing a NULL pointer or accessing an array out of bounds.

<u>Example 1: Dereferencing a NULL pointer</u>

```
#include <stdio.h>

int main() {
    int *ptr = NULL;   // pointer is not pointing to valid
memory
    *ptr = 10;         // trying to write to NULL – causes
segmentation fault
    return 0;
}
```

<u>Example 2: Accessing memory out of array bounds</u>

```
#include <stdio.h>

int main() {
    int arr[5];
    arr[10] = 50;   // out-of-bounds access
    return 0;
}
```

Example 3: Using freed memory

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

int main() {
    int *ptr = (int *)malloc(sizeof(int));
    free(ptr);        // memory is freed
    *ptr = 20;        // accessing freed memory
    return 0;
}
```

Example 4: Invalid memory access:

```
#include<stdio.h>

int main(){
    int num;
    printf("enter a number:\n");
    scanf("%d",num);
    printf("number is:%d",num); // trying to access memory
    that hasn't been correctly initialized
    return 0;
}
```

How to Fix:

- Always initialize pointers before using them.
- Check array indices before accessing elements.
- Use debugging tools like gdb to trace errors.

****HR Round mostly asked Question****

Tell me about yourself.

- "Good morning/afternoon! My name is [Drishti Sharma], and I am a final-year B.Tech Computer Science(AIML) student from [IPS Academy].

- I have a strong foundation in programming languages like C++, Java, and Python, along with expertise in Data Structures, Algorithms, and Database Management.

- I enjoy problem-solving and have participated in competitive coding challenges on platforms like HackerRank, LeetCode and CodeChef, where I have achieved a [mention ranking if applicable].

- During my academic journey, I have worked on several projects. One of my key projects was [Project Name], where I developed a [mention technology, e.g., full-stack web application, machine learning model] using [mention tools like React, Node.js, MySQL, etc.]. This project helped me gain hands-on experience in software development and database optimization.

- I also completed an internship at [Company Name], where I worked on [mention project or task], improving my teamwork and real-world problem-solving skills.

- Beyond academics, I am an active learner who stays updated with the latest technologies like AI, cloud computing, and cybersecurity. I am also a quick learner and a team player, which helps me adapt to new challenges efficiently.

- My career goal is to become a skilled software engineer, contributing to innovative solutions and working on projects that have a real-world impact. I am excited about this opportunity and eager to contribute my skills to [Company Name].