Pointer

A pointer is a variable that stores the memory address of another variable, allowing direct memory manipulation by holding the location of the value rather than the value itself.

Syntax:

```
datatype *ptr;
```

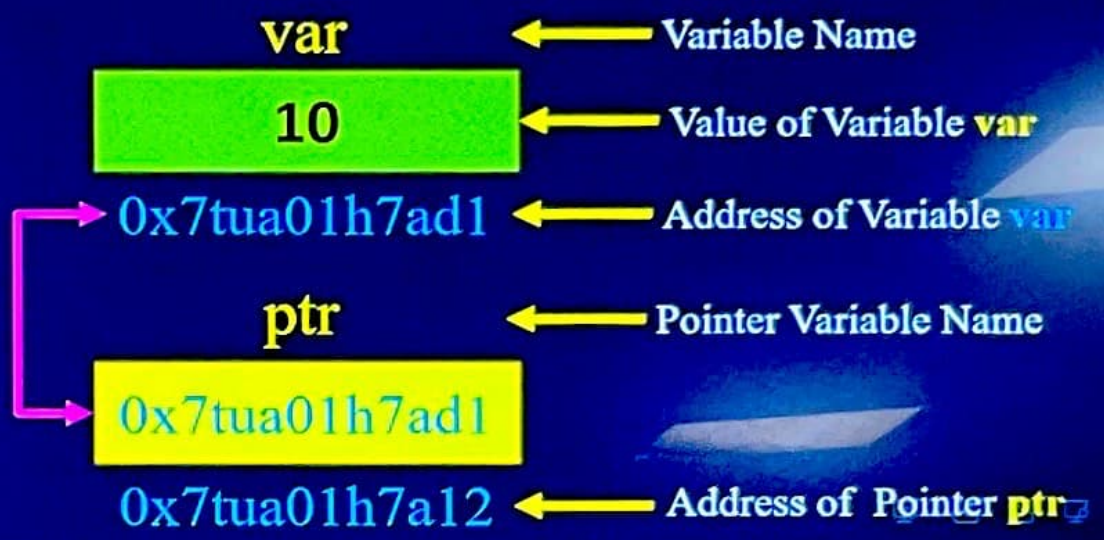
Here,

- *ptr is the chosen name for the pointer.*
- *datatype specifies the type of data the pointer is pointing to.*

```c
#include <stdio.h>

int main() {
    int var = 10;       // Normal variable
    int *ptr;           // Pointer variable declaration
    ptr = &var;         // ptr stores the address of var
    printf("%d\n", *ptr); // Prints the value at the address
    printf("%d\n", ptr); // Prints the the address of value

    return 0;
}
```



Pointer

Size of Pointers: The size of a pointer in C depends on the machine's architecture (bit system), not the data type it points to (e.g., int*, char*, float*), and remains constant across different data types.

- On a 32-bit system, all pointers typically occupy 4 bytes.
- On a 64-bit system, all pointers typically occupy 8 bytes.

Note: *We can check the size of a pointer using the sizeof operator* ***(e.g., sizeof(int*), sizeof(char*)).***

Pointer

Difference Between & and * :

- &(Address-of operator): Retrieves the memory address of a variable.
- *(Dereference operator): Accesses or modifies the value at that address.

There are two ways to declare pointer variables in C: (1) int* Number; (2) int *Number;

Advantage:

- Allow dynamic memory allocation and deallocation.
- Arrays and structures can be accessed efficiently using pointers.
- Enable direct access to memory locations.
- Used to create complex data structures like linked lists, trees, and graphs.
- Can reduce program length and execution time.

Disadvantage:

- Memory corruption may occur if pointers store invalid addresses.
- Difficult to understand compared to normal variables.
- Often cause memory leaks if memory is not freed.
- Slower access compared to direct variable access.
- Uninitialized pointers may lead to segmentation faults (program crash).

Pointer

Type of Pointer:

(1) Null Pointer: The Null Pointer is the pointer that does not point to any location but NULL.

Syntax:

*datatype *Pointer_Name = NULL;* **Eg:** *int *ptr = NULL; // ptr is a null pointer*

```
#include <stdio.h>
int main() {
    int *ptr = NULL;
    if (ptr == NULL) {
        printf("Pointer is null, safe to handle.\n");
    }
    printf("%d", *ptr); // Would cause segmentation fault
    return 0;
}
```

Pointer

Type of Pointer:

(2) Void Pointer: Versatile pointers that can be used to store the address of any data type. They are often used in dynamic memory allocation.

Syntax:

*void *Pointer_Name ;* **Eg:** *void *ptr;*

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    void *vp = malloc(sizeof(int)); // vp is a void pointer, it can point to any data type
    int *ip = (int*)vp; // Cast void pointer to int pointer so we can store integer values
    *ip = 42; // Assign value 42 to the memory location pointed by ip
    printf("Value: %d\n", *ip);
    free(vp); // Free the dynamically allocated memory to avoid memory leak
    return 0;
}
```

Output: Value: 42

Notes: Useful for generic functions or memory allocation, requiring type casting.

Pointer

Type of Pointer:

(3) Wild Pointer: **A wild pointer is a pointer that has not been initialized and therefore points to some unknown or random memory location.**

Syntax::

*data_type * pointer_name;*

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *wp; // Wild pointer
    printf("%d", *wp); // Segmentation fault
    wp = NULL; // Fixed by initialization
    printf("Now safe: %p\n", (void*)wp);
   return 0;
}
```

Output: -985026560 Now safe: (nil)

Notes: it's an error state that occurs if a pointer is declared but not initialized.

Pointer

Type of Pointer:

(4) Dangling Pointer: A pointer pointing to a memory location that has been deleted (e.g., after free) is called a dangling pointer.

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int* ptr = (int*)malloc(sizeof(int)); // After below free call, ptr becomes a dangling pointer
    free(ptr); // Freeing the allocated memory
    printf("Memory freed\n"); // After free(), ptr is now a dangling pointer (it points to freed memory)
    ptr = NULL; // To avoid issues, always set the pointer to NULL after free
    return 0;
}
```

Output: Memory freed

Notes: Avoid by setting to NULL after free to ensures it no longer points to an invalid location.

Pointer

Type of Pointer:

(6) Pointer to Constant: **A pointer that points to a constant value, which cannot be modified through the pointer, though the pointer itself can be reassigned.**

Syntax:

```
const data_type * pointer_name = & value;
```

Pointer

Type of Pointer:

(7) Pointer to Pointer **(Double Pointer): Pointers that store the address of another pointer.**

Syntax:

*data_type ** pointer_name;* eg: *int **doubleIntPtr;*

Pointer

Type of Pointer:

(7) Pointer to Pointer (Double Pointer): Pointers that store the address of another pointer.

Syntax:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int x = 50; // Base variable storing an integer value
    int *p = &x; // Single pointer pointing to variable x
    int **pp = &p; // Double pointer pointing to pointer p

    printf("Value of x: %d\n", x); // Display the value of x directly
    printf("Value via **pp: %d\n", **pp); // Access value of x indirectly using double pointer

    **pp = 100; // Modify x using double pointer and Changes the value of x to 100
    printf("New value of x: %d\n", x);
    return 0;
}
```

Output: Value of x: 50
Value via **pp: 50
New value of x: 100

Notes: These are useful in situations where you need to modify the original pointer.

Pointer

Type of Pointer:

(8) Integer Pointers: Pointers that are designed to store the memory addresses of integer variables.

Syntax: `int *intPtr;`

(9) Character Pointers: Pointers specifically intended for character data.

Syntax: `char *charPtr;`

(10) Floating Point Pointers: Pointers tailored for floating-point (decimal) data.

Syntax: `float *floatPtr;`

(11) Array Pointers: Pointers used to traverse arrays or store the base address of an array.

Syntax:
```
int arr[5];
int *arrPtr = arr;
```

(12) Function Pointers: Pointers that store the address of functions, allowing you to call functions indirectly.

Syntax: `int (*functionPtr)(int, int);`

Pointer Summery

Type	Description	Syntax Example	Insights
Null Pointer	A pointer that points to nothing. used for initialization and safety checks.	int *p = NULL;	Prevents accessing invalid memory. Check if(p != NULL) before dereferencing.
Void Pointer	Generic pointer that can point to any data type. Must be typecast before use.	void *vp; int x=10; vp=&x; printf("%d", *(int*)vp);	Often used in memory allocation functions like malloc().
Wild Pointer	Uninitialized pointer that may point to an unknown memory location.	int *p; *p = 5; // undefined	Causes crashes or segmentation faults. Always initialize before use.
Dangling Pointer	Points to memory that has been freed or deleted.	int *p=malloc(sizeof(int)); free(p); p=NULL;	Always set pointer to NULL after free() to avoid undefined behavior.
Constant Pointer	The pointer's address cannot be changed after initialization.	int x=5, y=10; int *const p=&x; *p=20;	Can modify value at the address, but not reassign the pointer.
Pointer to Constant	The data being pointed to cannot be modified through the pointer.	const int x=50; const int *p=&x;	Pointer can move, but value is read-only. Used for data protection.
Pointer to Pointer (Double Pointer)	A pointer that stores the address of another pointer.	int x=10; int *p=&x; int **pp=&p;	Used in dynamic 2D arrays, function arguments, and memory management.
Integer Pointer	Points to an integer variable and allows indirect access.	int x=7; int *p=&x;	Most commonly used pointer type in C.
Character Pointer	Points to a character variable or string.	char *p="Hello"; printf("%s", p);	Used for string manipulation and arrays of characters.
Floating Pointer	Points to a float variable.	float f=3.14; float *p=&f;	Same concept as int pointer, but for float data type.
Array Pointer	Points to the first element of an array.	int arr[5]={1,2,3,4,5}; int *p=arr;	Enables pointer arithmetic: *(p+2) gives arr[2].
Function Pointer	Points to a function and allows indirect function calls.	void func(){...} void (*fp)()=func; fp();	Used in callbacks, event handling, and dynamic dispatching.

Pointer

Q1. What is a pointer ? and why is it used?

A pointer is a variable that stores the memory address of another variable. Pointers are used for dynamic memory allocation, passing large structures efficiently to functions, building complex data structures (like linked lists, trees), and accessing arrays and functions indirectly.

Q2. How are pointers used in dynamic memory allocation (malloc, calloc, realloc, free)?

malloc() → allocates memory without initialization.
calloc() → allocates and initializes to zero.
realloc() → resizes allocated memory.
free() → releases memory.

Q3. What is the difference between const int *p and int *const p?

const int *p → pointer to constant value (value cannot change, pointer can move).
int *const p → constant pointer (pointer cannot move, value can change).

Pointer

Q4. What is a dangling pointer and how can it be prevented?

A dangling pointer points to memory that has been freed or deallocated. Prevent it by:

- Setting pointer to NULL after free().
- Avoid returning addresses of local variables.
- Carefully manage dynamic memory, ensuring each malloc/calloc has a corresponding free.

Key Point: A pointer is safe only if it points to valid memory or is NULL.

Q5. What is pointer arithmetic? Can you add two pointers together?

Pointer arithmetic allows **incrementing or decrementing pointers**, or finding the difference between two pointers pointing to the same array. You **cannot add two pointers together**, but you can subtract them to find the number of elements between them.

Q6. What happens if you dereference a NULL pointer?

Dereferencing a NULL pointer leads to **undefined behavior**, often causing a **segmentation fault** or program crash. Always check if a pointer is not NULL before dereferencing.

Pointer

Q27. Write a C program to sort an array using pointers.

```c
#include <stdio.h>
int main() {
    int n, i, j, temp;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];          // Array to hold elements
    int *ptr = arr;      // Pointer to the array

    printf("Enter %d elements:\n", n); // Input elements
    for(i = 0; i < n; i++) {
        scanf("%d", ptr + i);  // Using pointer arithmetic
    }

    // Sorting the array in ascending order using pointers (Bubble Sort)
    for(i = 0; i < n - 1; i++) {
        for(j = 0; j < n - i - 1; j++) {
            if(*(ptr + j) > *(ptr + j + 1)) {  // Compare elements
                temp = *(ptr + j);          // Swap elements
                *(ptr + j) = *(ptr + j + 1);
                *(ptr + j + 1) = temp;
            }
        }
    }

    // Display the sorted array
    printf("Sorted array in ascending order:\n");
    for(i = 0; i < n; i++) {
        printf("%d ", *(ptr + i));
    }
    printf("\n");

    return 0;
}
```

OUTPUT:

```
Enter the number of elements: 5
Enter 5 elements:
85
65
47
89
23
Sorted array in ascending order:23 47 65 85 89
```

Pointer

Q28. Write a C program to reverse a string using pointers.

```c
#include <stdio.h>
#include <string.h>
int main() {
    char str[100], *start, *end, temp;

    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);

    str[strcspn(str, "\n")] = '\0'; // Remove newline character if present

    start = str;                // Pointer to the first character
    end = str + strlen(str) - 1; // Pointer to the last character

    while(start < end) {     // Reverse the string using pointers
        temp = *start; // Swap characters
        *start = *end;
        *end = temp;

        start++; // Move pointers
        end--;
    }
    printf("Reversed string: %s\n", str);
    return 0;
```