

What is a Pointer?

A variable that stores another variable's memory address is called a pointer. Normal variables store values. But pointers hold memory locations where values are kept. Using pointers lets you access and change system memory directly. This can make programs more efficient and powerful. C use pointers a lot. But other languages can also have pointers, though maybe not the same.

Types of Pointers in Programming:

1. Basic Pointer:

A primary pointer is a variable that is a memory address of another variable. Instead of having the data itself physically present, pointers only connect to the memory location where the data is kept. Answering that, the program will get a chance to use it to read and write with the value located at the given memory address.

```
#include <stdio.h>

int main()
{
    int var = 23;

    // Declare a pointer to an integer
    int* ptr;

    // Assign the pointer the address of var
    ptr = &var;

    // Output the value of var
    printf("Value of var: %d\n", var);

    // Output the memory address of var
    printf("Address of var: %p\n", &var);

    // Output the address stored in ptr
    printf("Value of ptr (Address it holds): %p\n", ptr);

    // Output the value at the memory address stored in ptr
    printf("Value pointed to by ptr: %d\n", *ptr);

    return 0;
}
```

Output

```
Value of var: 23
Address of var: 0x7ffc833c13b4
Value of ptr (Address it holds): 0x7ffc833c13b4
Value pointed to by ptr: 23
```

2. Null Pointer:

Null pointer is a specific component of pointers which doesn't point to any **legitimate memory location**. It is applied in programming to denote the particular case when a pointer intentionally not points on an object and is **uninitialized**. Such a pointer absolutely literally refers to the value **0** (or **NULL**, which in C terminology is generally set to **0**, which denotes a protected value reserved by the language rules that does not refer to anything).

```
#include <stdio.h>

int main() {
    int *ptr = NULL; // Declaration of a null pointer

    if (ptr == NULL) {
        printf("The pointer is null.\n");
    }

    return 0;
}
```

Output

```
The pointer is null.
```

3. Pointer to Pointer (Double Pointer):

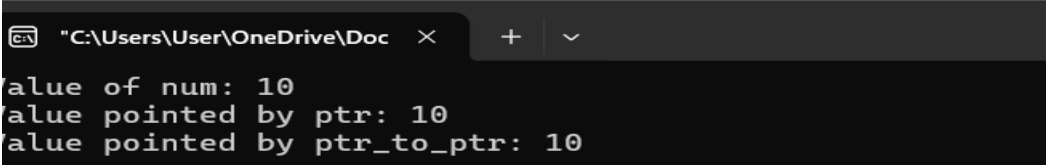
Pointer to a pointer sometimes gets referred as a **double pointer**. It is a type of pointer which stores the memory address of another pointer. Double pointers are used to achieve indirect reference of data via another pointer in the case of its required. As an important point, this C-like languages features support multiple indirection that can be used as extra layers to direct data structures which are the most useful for the **dynamic data structures**.

```
#include <stdio.h>

int main() {
    int num = 10;
    int *ptr = &num;
    int **ptr_to_ptr = &ptr; // Pointer to pointer

    printf("Value of num: %d\n", num);
    printf("Value pointed by ptr: %d\n", *ptr);
    printf("Value pointed by ptr_to_ptr: %d\n", **ptr_to_ptr);

    return 0;
}
```



The screenshot shows a terminal window with the following output:

```
Value of num: 10
Value pointed by ptr: 10
Value pointed by ptr_to_ptr: 10
```

4. Array Pointers:

An array pointer is a kind of pointer that can be used to point to the first element of the associated array in the memory. Such a pointer, besides storing the memory address of the first element of the array, is also usually tied to the element type of the array. The pointer will become aware of this and will facilitate pointer arithmetic making it possible to either increase it or decrease it and move through the array.

```
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int* p = arr; // Array pointer to the first element of arr

    for(int i = 0; i < 5; i++) {
        printf("%d ", *(p + i)); // Accessing array elements through pointer arithmetic
    }

    return 0;
}
```

Output

```
10 20 30 40 50
```

5. Function Pointer:

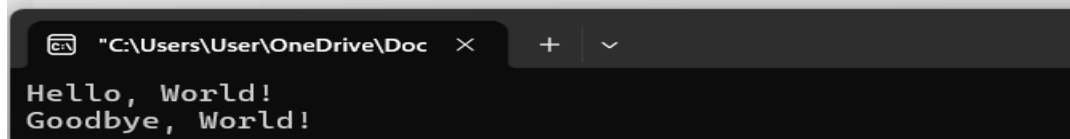
Function pointers are very powerful programming tools using which programmers can save function references that can be later called. It is the feature that is most visible for callback functions, implementing plugins, or for the simulation of polymorphism in C-like languages. Here is a short description of function pointers and their use in the several programming languages that allow this functionality.

```
#include <stdio.h>

void hello() {
    printf("Hello, World!\n");
}

void goodbye() {
    printf("Goodbye, World!\n");
}

int main() {
    void (*func_ptr)(); // Declaration of a function pointer
    func_ptr = hello;   // Pointing to hello function
    func_ptr();         // Calling hello function
    func_ptr = goodbye; // Now pointing to goodbye function
    func_ptr();         // Calling goodbye function
    return 0;
}
```



The screenshot shows a terminal window with a dark background. The title bar at the top reads "C:\Users\User\OneDrive\Doc" followed by a close button and a dropdown arrow. The terminal output consists of two lines: "Hello, World!" and "Goodbye, World!".

6. Pointer to Structure (Struct Pointer):

A Pointer to Structure (or Struct Pointer) is a concept present in some programming languages C. This construct enables a creation of pointers purposely directed to structures. Different from Structures, these structures are user-specified data types that permit the combination of different items of data. Pointer to structures are especially important for dynamic allocation of memory, function parameters passed to big structures with less overhead, and data structures which include linked lists and trees.

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

typedef struct {
    int id;
    char name[20];
} Employee;

int main() {
    Employee *ptr = malloc(sizeof(Employee)); // Dynamically allocate memory for one Employee struct
    if (ptr == NULL) {
        fprintf(stderr, "Malloc failed\n");
        return -1;
    }

    // Accessing struct members through pointer using arrow operator
    ptr->id = 123;
    strcpy(ptr->name, "Geeks For Geeks");

    printf("Employee ID: %d\n", ptr->id);
    printf("Employee Name: %s\n", ptr->name);

    free(ptr); // Free the allocated memory
    return 0;
}
```

Output

```
Employee ID: 123
Employee Name: Geeks For Geeks
```

pass by reference and pass by value :

Pass by Value:

- In the function `passByValue()`, we pass a copy of the variable `x`. The function works with the copy, so modifying it inside the function does **not** affect the original variable `x` in `main()`.

Pass by Reference:

- In the function `passByReference()`, we pass the address of the variable `x` (i.e., a pointer to `x`). This means the function modifies the original value of `x`, as it directly operates on the memory location of `x`.

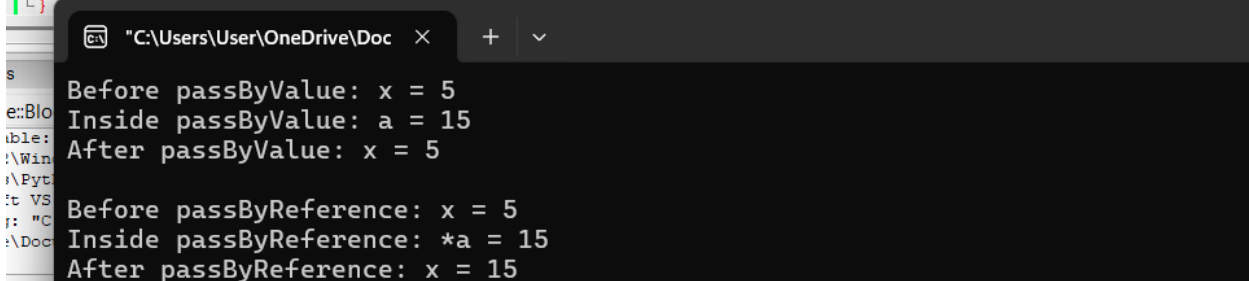
```
#include <stdio.h>
// Function to demonstrate pass by value
void passByValue(int a) {
    a = a + 10; // Modify the local copy of 'a'
    printf("Inside passByValue: a = %d\n", a); // This will print the modified value of 'a'
}

// Function to demonstrate pass by reference
void passByReference(int *a) {
    *a = *a + 10; // Modify the value of 'a' using pointer dereferencing
    printf("Inside passByReference: *a = %d\n", *a); // This will print the modified value of 'a'
}

int main() {
    int x = 5;
    // Call pass by value (does not modify the original 'x')
    printf("Before passByValue: x = %d\n", x);
    passByValue(x);
    printf("After passByValue: x = %d\n", x); // 'x' will remain the same

    // Call pass by reference (modifies the original 'x')
    printf("\nBefore passByReference: x = %d\n", x);
    passByReference(&x); // Pass the address of 'x'
    printf("After passByReference: x = %d\n", x); // 'x' will be modified

    return 0;
}
```



```
C:\Users\User\OneDrive\Doc >
Before passByValue: x = 5
Inside passByValue: a = 15
After passByValue: x = 5

Before passByReference: x = 5
Inside passByReference: *a = 15
After passByReference: x = 15
```