**C Pointers**

A pointer is a variable that stores the memory address of another variable. Instead of holding a direct value, it holds the address where the value is stored in memory. It is the backbone of low-level memory manipulation in C. Accessing the pointer directly will just give us the address that is stored in the pointer. For example,

#include <stdio.h>

​

int main() {

// Normal Variable

int var = 10;

// Pointer Variable ptr that

// stores address of var

int\* ptr = &var;

// Directly accessing ptr will

// give us an address

printf("%d", ptr);

return 0;

}

**Output**

0x7fffa0757dd4

This hexadecimal integer (starting with 0x) is the memory address.

A diagram of a number

AI-generated content may be incorrect.

Let us understand different steps of the above program.

**Declare a Pointer**

A pointer is declared by specifying its name and type, just like simple variable declaration but with an **asterisk (\*)** symbol added before the pointer's name.

data\_type\* name

Here, **data\_type**defines the type of data that the pointer is pointing to. An integer type pointer can only point to an integer. Similarly, a pointer of float type can point to a floating-point data, and so on.

**Example:**

int \*ptr;

In the above statement, pointer **ptr**can store the address of an integer. It is pronounced as pointer to integer.

**Initialize the Pointer**

Pointer initialization means assigning some address to the pointer variable. In C, the [**(&) addressof operator**](https://www.geeksforgeeks.org/cpp/address-operator-in-c/)is used to get the memory address of any variable. This memory address is then stored in a pointer variable.

**Example:**

int var = 10;

*// Initializing ptr*

int \*ptr = &var;

In the above statement, pointer **ptr**store the address of variable **var**which was determined using address-of operator **(&).**

***Note:*** *We can also declare and initialize the pointer in a single step. This is called* ***pointer definition.***

**Dereference a Pointer**

We have to first [**dereference**](https://www.geeksforgeeks.org/cpp/dereference-pointer-in-c/)the pointer to access the value present at the memory address. This is done with the help of **dereferencing operator(\*)**(same operator used in declaration).

#include <stdio.h>

​

int main() {

int var = 10;

// Store address of var variable

int\* ptr = &var;

// Dereferencing ptr to access the value

printf("%d", \*ptr);

return 0;

}

**Output**

10

***Note****: Earlier, we used %d for printing pointers, but C provides a separate* [*format specifier*](https://www.geeksforgeeks.org/c/format-specifiers-in-c/)***%p*** *for printing pointers.*

**Size of Pointers**

The **size of a pointer in C** depends on the **architecture (bit system)** of the machine, **not the data type** it points to.

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

The size remains **constant regardless of the data type** (int\*, char\*, float\*, etc.). We can verify this using the [sizeof operator](https://www.geeksforgeeks.org/c/sizeof-operator-c/" \t "_blank).

#include <stdio.h>

​

int main() {

int \*ptr1;

char \*ptr2;

// Finding size using sizeof()

printf("%zu\n", sizeof(ptr1));

printf("%zu", sizeof(ptr2));

return 0;

}

**Output**

8

8

The reason for the same size is that the pointers store the memory addresses, no matter what type they are. As the space required to store the addresses of the different memory locations is the same, the memory required by one pointer type will be equal to the memory required by other pointer types.

***Note:*** *The actual size of the pointer may vary depending on the* ***compiler and system architecture****, but it is always* ***uniform across all data types*** *on the same system.*

**Special Types of Pointers**

There are 4 special types of pointers that used or referred to in different contexts:

**NULL Pointer**

The [NULL Pointers](https://www.geeksforgeeks.org/cpp/null-pointer-in-cpp/) are those pointers that do not point to any memory location. They can be created by assigning **NULL**value to the pointer. A pointer of any type can be assigned the NULL value.

#include <stdio.h>

​

int main() {

// Null pointer

int \*ptr = NULL;

return 0;

}

NULL pointers are generally used to represent the absence of any address. This allows us to check whether the pointer is pointing to any valid memory location by checking if it is equal to NULL.

**Void Pointer**

The [void pointers](https://www.geeksforgeeks.org/c/void-pointer-c-cpp/) in C are the pointers of type **void**. It means that they do not have any associated data type. They are also called **generic pointers** as they can point to any type and can be typecasted to any type.

#include <stdio.h>

​

int main() {

// Void pointer

void \*ptr;

return 0;

}

**Wild Pointers**

The [wild pointers](https://www.geeksforgeeks.org/dsa/what-are-wild-pointers-how-can-we-avoid/) are pointers that have not been initialized with something yet. These types of C-pointers can cause problems in our programs and can eventually cause them to crash. If values are updated using wild pointers, they could cause data abort or data corruption.

#include <stdio.h>

​

int main() {

​

// Wild Pointer

int \*ptr;

return 0;

}

**Dangling Pointer**

A pointer pointing to a memory location that has been deleted (or freed) is called a [dangling pointer](https://www.geeksforgeeks.org/c/dangling-void-null-wild-pointers/). Such a situation can lead to unexpected behavior in the program and also serve as a source of bugs in C programs.

#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);

printf("Memory freed\n");

​

// removing Dangling Pointer

ptr = NULL;

​

return 0;

}

**Output**

Memory freed

**C Pointer Arithmetic**

The [pointer arithmetic](https://www.geeksforgeeks.org/c/pointer-arithmetics-in-c-with-examples/) refers to the arithmetic operations that can be performed on a pointer. It is slightly different from the ones that we generally use for mathematical calculations as only a limited set of operations can be performed on pointers. These operations include:

* **Increment/Decrement**
* **Addition/Subtraction of Integer**
* **Subtracting Two Pointers of Same Type**
* **Comparing/Assigning Two Pointers of Same Type**
* **Comparing/Assigning with NULL**

**C Pointers and Arrays**

In C programming language,[pointers and arrays](https://www.geeksforgeeks.org/c/relationship-between-pointer-and-array-in-c/) are closely related. An array name acts like a pointer constant. The value of this pointer constant is the address of the first element. For example, if we have an array named **val,** then **val** and **&val[0]** can be used interchangeably.

If we assign this value to a non-constant [pointer to array](https://www.geeksforgeeks.org/c/pointer-array-array-pointer/) of the same type, then we can access the elements of the array using this pointer. Not only that, as the array elements are stored continuously, we can use pointer arithmetic operations such as increment, decrement, addition, and subtraction of integers on pointer to move between array elements.

This concept is not limited to the one-dimensional array, we can refer to a multidimensional array element as well using pointers.

**Constant Pointers**

In **constant pointers**, the memory address stored inside the pointer is constant and cannot be modified once it is defined. It will always point to the same memory address.

**Example**:

#include <stdio.h>

​

int main() {

int a = 90;

int b = 50;

​

// Creating a constant pointer

int\* const ptr = &a;

// Trying to reassign it to b

ptr = &b;

​

return 0;

}

**Output**

solution.c: In function ‘main’:  
solution.c:11:9: error: assignment of read-only variable ‘ptr’  
 11 | ptr = &b;  
 | ^

We can also create a pointer to constant or even constant pointer to constant. Refer to this article to know more - [Constant pointer, Pointers to Constant and Constant Pointers to Constant](https://www.geeksforgeeks.org/cpp/difference-between-constant-pointer-pointers-to-constant-and-constant-pointers-to-constants/)

**Pointer to Function**

A [**function pointer**](https://www.geeksforgeeks.org/c/function-pointer-in-c/) is a type of pointer that stores the address of a function, allowing functions to be passed as arguments and invoked dynamically. It is useful in techniques such as callback functions, event-driven programs.

**Example**:

#include <stdio.h>

​

int add(int a, int b) {

return a + b;

}

​

int main() {

// Declare a function pointer that matches

// the signature of add() fuction

int (\*fptr)(int, int);

​

// Assign address of add()

fptr = &add;

​

// Call the function via ptr

printf("%d", fptr(10, 5));

​

return 0;

}

**Output**

15

**Multilevel Pointers**

In C, we can create [**multi-level pointers**](https://www.geeksforgeeks.org/c/chain-of-pointers-in-c-with-examples/)with any number of levels such as – \*\*\*ptr3, \*\*\*\*ptr4, \*\*\*\*\*\*ptr5 and so on. Most popular of them is [**double pointer**](https://www.geeksforgeeks.org/c/c-pointer-to-pointer-double-pointer/) (pointer to pointer). It stores the memory address of another pointer. Instead of pointing to a data value, they point to another pointer.

**Example**:

#include <stdio.h>

​

int main() {

int var = 10;

// Pointer to int

int \*ptr1 = &var;

// Pointer to pointer (double pointer)

int \*\*ptr2 = &ptr1;

​

// Accessing values using all three

printf("var: %d\n", var);

printf("\*ptr1: %d\n", \*ptr1);

printf("\*\*ptr2: %d", \*\*ptr2);

​

return 0;

}

**Output**

var: 10

\*ptr1: 10

\*\*ptr2: 10

**Uses of Pointers in C**

The C pointer is a very powerful tool that is widely used in C programming to perform various useful operations. It is used to achieve the following functionalities in C:

* [Pass Arguments by Pointers](https://www.geeksforgeeks.org/c/passing-pointers-to-functions-in-c/)
* Accessing Array Elements
* [Return Multiple Values from Function](https://www.geeksforgeeks.org/c/how-to-return-a-pointer-from-a-function-in-c/)
* [Dynamic Memory Allocation](https://www.geeksforgeeks.org/c/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/)
* [Implementing Data Structures](https://www.geeksforgeeks.org/dsa/dsa-tutorial-learn-data-structures-and-algorithms/)
* In System-Level Programming where memory addresses are useful.
* To use in Control Tables.

**Advantages of Pointers**

Following are the major advantages of pointers in C:

* Pointers are used for dynamic memory allocation and deallocation.
* An Array or a structure can be accessed efficiently with pointers
* Pointers are useful for accessing memory locations.
* Pointers are used to form complex data structures such as linked lists, graphs, trees, etc.
* Pointers reduce the length of the program and its execution time as well.

**Issues with Pointers**

Pointers are vulnerable to errors and have following disadvantages:

* Memory corruption can occur if an incorrect value is provided to pointers.
* Pointers are a little bit complex to understand.
* Pointers are majorly responsible for [memory leaks in C](https://www.geeksforgeeks.org/c/what-is-memory-leak-how-can-we-avoid/).
* Accessing using pointers are comparatively slower than variables in C.
* Uninitialized pointers might cause a[segmentation fault.](https://www.geeksforgeeks.org/c/segmentation-fault-sigsegv-vs-bus-error-sigbus/)