C Pointers

[redacted]

Pointers are one of the core components of the C programming language. A pointer can be used to store the **memory address** of other variables, functions, or even other pointers. The use of pointers allows low-level memory access, dynamic memory allocation, and many other functionality in C.
In this article, we will discuss C pointers in detail, their types, uses, advantages, and disadvantages with examples.

What is a Pointer in C?

A pointer is defined as a derived data type that can store the address of other C variables or a memory location. We can access and manipulate the data stored in that memory location using pointers.

As the pointers in C store the memory addresses, their size is independent of the type of data they are pointing to. This size of pointers in C only depends on the system architecture.

Syntax of C Pointers

The syntax of pointers is similar to the variable declaration in C, but we use the **(*) dereferencing operator** in the pointer declaration.

```
datatype * ptr;
```

where

- **ptr** is the name of the pointer.
- **datatype** is the type of data it is pointing to.

The above syntax is used to define a pointer to a variable. We can also define pointers to functions, structures, etc.

How to Use Pointers?

The use of pointers in C can be divided into three steps:

1. **Pointer Declaration**
2. **Pointer Initialization**
3. **Pointer Dereferencing**

1. Pointer Declaration

In pointer declaration, we only declare the pointer but do not initialize it. To declare a pointer, we use the **(*) dereference operator** before its name.

Example

```
int *ptr;
```

The pointer declared here will point to some random memory address as it is not initialized. Such pointers are called wild pointers.

2. Pointer Initialization

Pointer initialization is the process where we assign some initial value to the pointer variable. We generally use the **(&: ampersand) addressof operator** to get the memory address of a variable and then store it in the pointer variable.

Example

```
int var = 10;
int * ptr;
ptr = &var;
```

We can also declare and initialize the pointer in a single step. This method is called **pointer definition** as the pointer is declared and initialized at the same time.

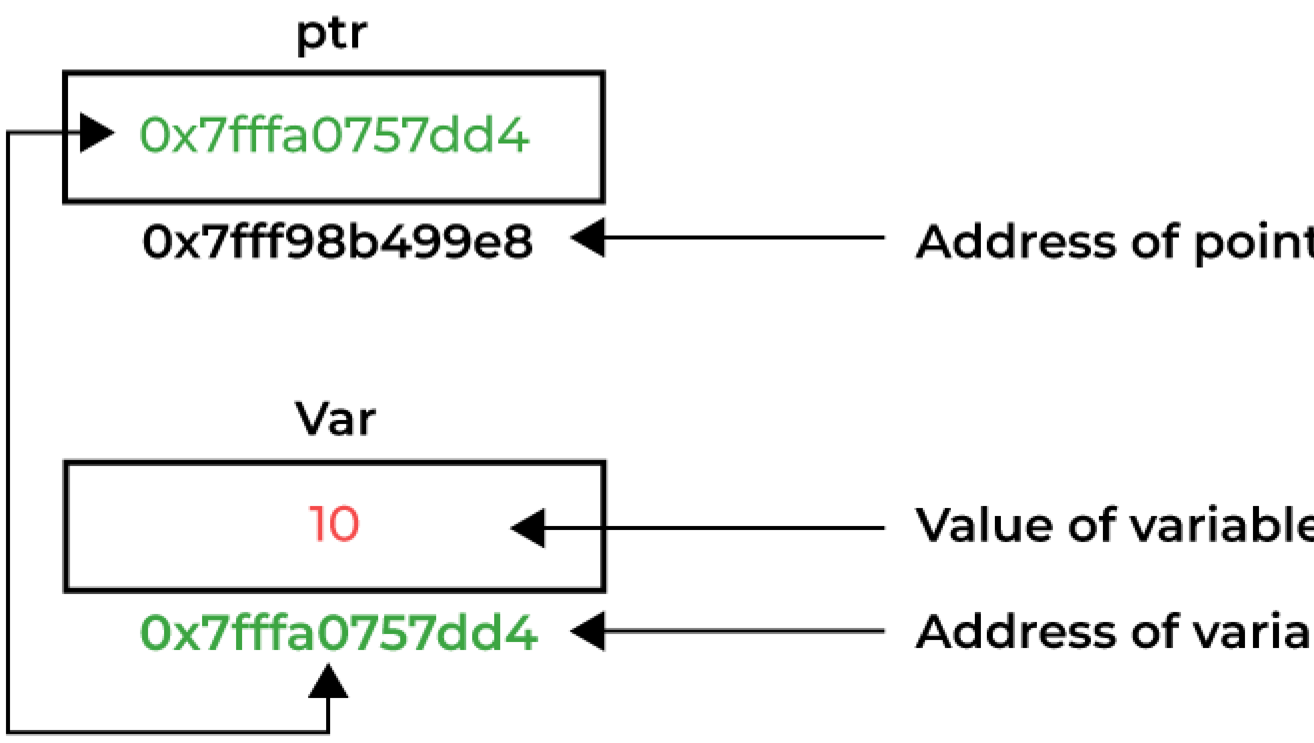
Example

```
int *ptr = &var;
```

Note: *It is recommended that the pointers should always be initialized to some value before starting using it. Otherwise, it may lead to number of errors.*

3. Pointer Dereferencing

Dereferencing a pointer is the process of accessing the value stored in the memory address specified in the pointer. We use the same **(*) dereferencing operator** that we used in the pointer declaration.



Dereferencing a Pointer in C

C Pointer Example

C

```c
// C program to illustrate Pointers
#include <stdio.h>

void Intro_Pointer ()
{
    int var = 10;

    // declare pointer variable
    int* ptr;

    // note that data type of ptr and var must be same
    ptr = &var;

    // assign the address of a variable to a pointer
    printf("Value at ptr = %p \n", ptr);
    printf("Value at var = %d \n", var);
    printf("Value at *ptr = %d \n", *ptr);
}

// Driver program
int main()
{
    Intro_Pointer ();
    return 0;
}
```

Output

```
Value at ptr = 0x7ffca84068dc

Value at var = 10

Value at *ptr = 10
```

Types of Pointers in C

Pointers in C can be classified into many different types based on the parameter on which we are defining their types. If we consider the type of variable stored in the memory location pointed by the pointer, then the pointers can be classified into the following types:

1. Integer Pointers

As the name suggests, these are the pointers that point to the integer values.

Syntax

```c
int *ptr;
```

These pointers are pronounced as **Pointer to Integer.**

Similarly, a pointer can point to any primitive data type. It can point also point to derived data types such as arrays and user-defined data types such as structures.

2. Array Pointer

Pointers and Array are closely related to each other. Even the array name is the pointer to its first element. They are also known as Pointer to Arrays. We can create a pointer to an array using the given syntax.

Syntax

```
char *ptr = &array_name;
```

Pointer to Arrays exhibits some interesting properties which we discussed later in this article.

3. Structure Pointer

The pointer pointing to the structure type is called Structure Pointer or Pointer to Structure. It can be declared in the same way as we declare the other primitive data types.

Syntax

```
struct struct_name *ptr;
```

In C, structure pointers are used in data structures such as linked lists, trees, etc.

4. Function Pointers

Function pointers point to the functions. They are different from the rest of the pointers in the sense that instead of pointing to the data, they point to the code. Let's consider a function prototype – **int func (int, char)**, the function pointer for this function will be

Syntax

```
int (*ptr)(int, char);
```

Note: The syntax of the function pointers changes according to the function prototype.

5. Double Pointers

In C language, we can define a pointer that stores the memory address of another pointer. Such pointers are called double-pointers or pointers-to-pointer. Instead of pointing to a data value, they point to another pointer.

Syntax

```
datatype ** pointer_name;
```

Dereferencing Double Pointer

```
*pointer_name; // get the address stored in the inner level pointer
**pointer_name; // get the value pointed by inner level pointer
```

Note: *In C, we can create multi-level pointers with any number of levels such as – ***ptr3, ****ptr4, ******ptr5 and so on.*

6. NULL Pointer

The Null Pointers are those pointers that do not point to any memory location. They can be created by assigning a NULL value to the pointer. A pointer of any type can be assigned the NULL value.

Syntax

```
data_type *pointer_name = NULL;
        or
pointer_name = NULL
```

It is said to be good practice to assign NULL to the pointers currently not in use.

7. Void Pointer

The Void pointers 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.

Syntax

```
void * pointer_name;
```

One of the main properties of void pointers is that they cannot be dereferenced.

8. Wild Pointers

The Wild Pointers 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 is updated using wild pointers, they could cause data abort or data corruption.

Example

```
int *ptr;
char *str;
```

9. 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.

Syntax

```
data_type * const pointer_name;
```

10. Pointer to Constant

The pointers pointing to a constant value that cannot be modified are called pointers to a constant. Here we can only access the data pointed by the pointer, but cannot modify it. Although, we can change the address stored in the pointer to constant.

Syntax

```
const data_type * pointer_name;
```

Size of Pointers in C

The size of the pointers in C is equal for every pointer type. The size of the pointer does not depend on the type it is pointing to. It only depends on the operating system and CPU architecture. The size of pointers in C is

- **8 bytes** for a **64-bit System**
- **4 bytes** for a **32-bit System**

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.

How to find the size of pointers in C?

We can find the size of pointers using the **sizeof operator** as shown in the following program:

Example: C Program to find the size of different pointer types.

C

```c
// C Program to find the size of different pointers types
#include <stdio.h>

// dummy structure
struct str {
};

// dummy function
void func(int a, int b){};

int main()
{
    // dummy variables definitions
    int a = 10;
    char c = 'G';
    struct str x;

    // pointer definitions of different types
    int* ptr_int = &a;
    char* ptr_char = &c;
    struct str* ptr_str = &x;
    void (*ptr_func)(int, int) = &func;
    void* ptr_vn = NULL;

    // printing sizes
    printf("Size of Integer Pointer  \t:\t%d bytes\n",
           sizeof(ptr_int));
    printf("Size of Character Pointer\t:\t%d bytes\n",
           sizeof(ptr_char));
    printf("Size of Structure Pointer\t:\t%d bytes\n",
           sizeof(ptr_str));
    printf("Size of Function Pointer\t:\t%d bytes\n",
           sizeof(ptr_func));
    printf("Size of NULL Void Pointer\t:\t%d bytes",
           sizeof(ptr_vn));

    return 0;
}
```

Output

```
Size of Integer Pointer      :    8 bytes

Size of Character Pointer    :    8 bytes

Size of Structure Pointer    :    8 bytes

Size of Function Pointer    :    8 bytes

Size of NULL Void Pointer    :    8 bytes
```

As we can see, no matter what the type of pointer it is, the size of each and every pointer is the same.
Now, one may wonder that if the size of all the pointers is the same, then why do we need to declare the pointer type in the declaration? **The type declaration is needed in the pointer for dereferencing and pointer arithmetic purposes.**

C Pointer Arithmetic

The Pointer Arithmetic refers to the legal or valid 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 in a Pointer
- Decrement in a Pointer
- Addition of integer to a pointer
- Subtraction of integer to a pointer
- Subtracting two pointers of the same type
- Comparison of pointers of the same type.
- Assignment of pointers of the same type.

C

```
// C program to illustrate Pointer Arithmetic

#include <stdio.h>

int main()
{
    // Declare an array
    int v[3] = { 10, 100, 200 };

    // Declare pointer variable
    int* ptr;

    // Assign the address of v[0] to ptr
    ptr = v;

    for (int i = 0; i < 3; i++) {

        // print value at address which is stored in ptr
        printf("Value of *ptr = %d\n", *ptr);

        // print value of ptr
        printf("Value of ptr = %p\n\n", ptr);

        // Increment pointer ptr by 1
        ptr++;
    }
    return 0;
}
```

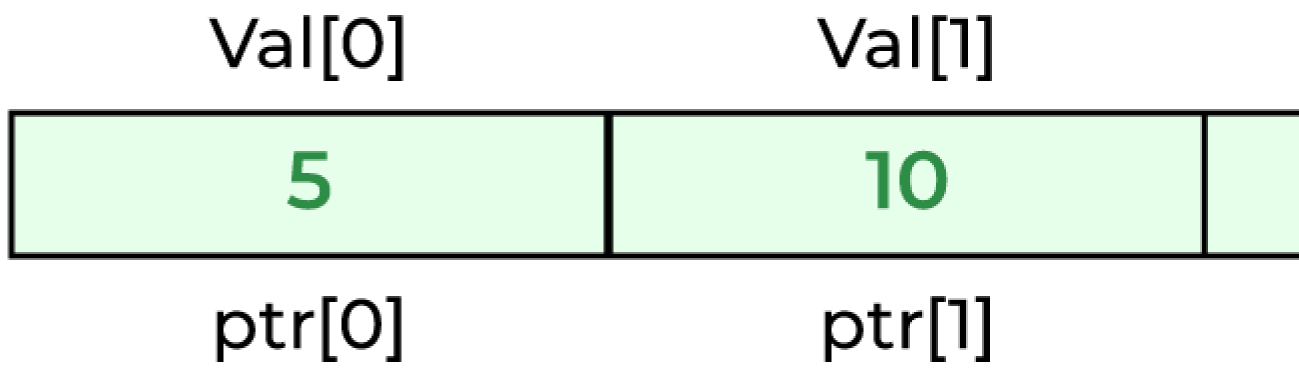
Output

```
Value of *ptr = 10

Value of ptr = 0x7ffcfe7a77a0


Value of *ptr = 100

Value of ptr = 0x7ffcfe7a77a4


Value of *ptr = 200

Value of ptr = 0x7ffcfe7a77a8
```

C Pointers and Arrays

In C programming language, pointers and arrays 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 of the same type, then we can access the elements of the array using this pointer.

Example 1: Accessing Array Elements using Pointer with Array Subscript



C

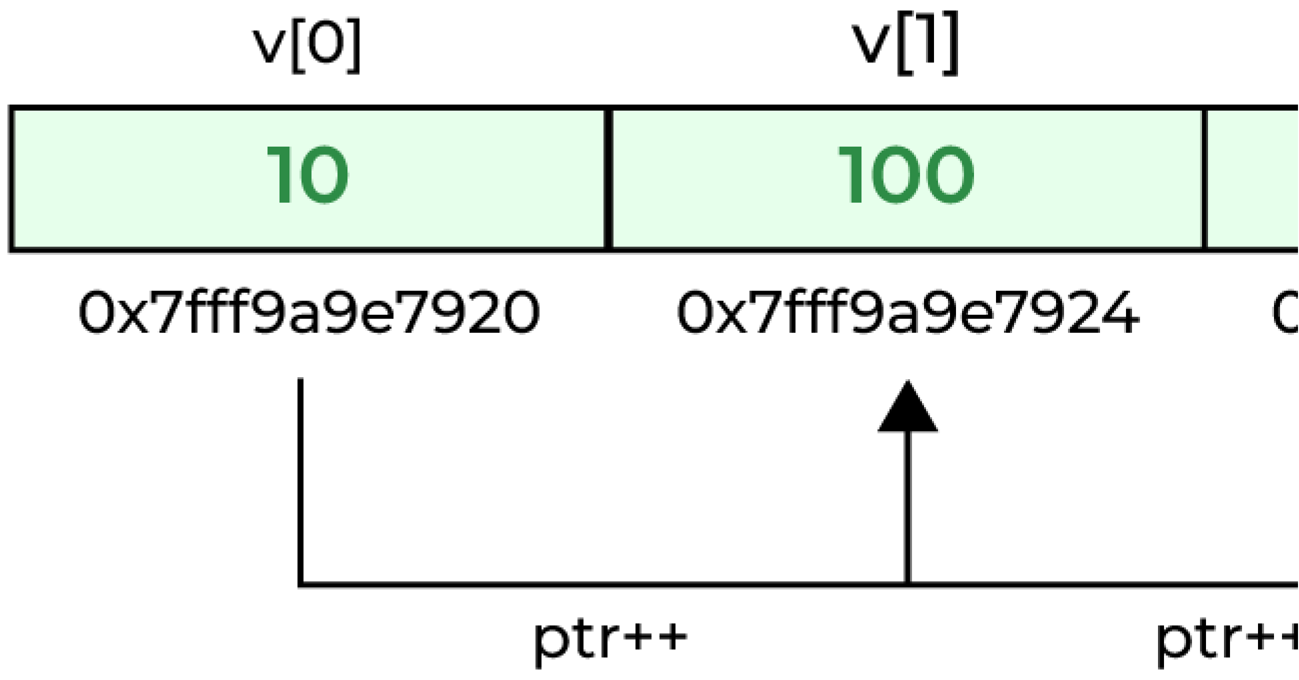
```c
// C Program to access array elements using pointer
#include <stdio.h>

void geeks()
{
    // Declare an array
    int val[3] = { 5, 10, 15 };

    // Declare pointer variable
    int* ptr;

    // Assign address of val[0] to ptr.
    // We can use ptr=&val[0];(both are same)
    ptr = val;

    printf("Elements of the array are: ");

    printf("%d, %d, %d", ptr[0], ptr[1], ptr[2]);

    return;
}

// Driver program
int main()
{
    geeks();
    return 0;
}
```

Output

```
Elements of the array are: 5, 10, 15
```

Not only that, as the array elements are stored continuously, we can pointer arithmetic operations such as increment, decrement, addition, and subtraction of integers on pointer to move between array elements.

Example 2: Accessing Array Elements using Pointer Arithmetic



C

```c
// C Program to access array elements using pointers
#include <stdio.h>

int main()
{

    // defining array
    int arr[5] = { 1, 2, 3, 4, 5 };

    // defining the pointer to array
    int* ptr_arr = arr;

    // traversing array using pointer arithmetic
    for (int i = 0; i < 5; i++) {
        printf("%d ", *ptr_arr++);
    }
    return 0;
}
```

Output

```
1 2 3 4 5
```

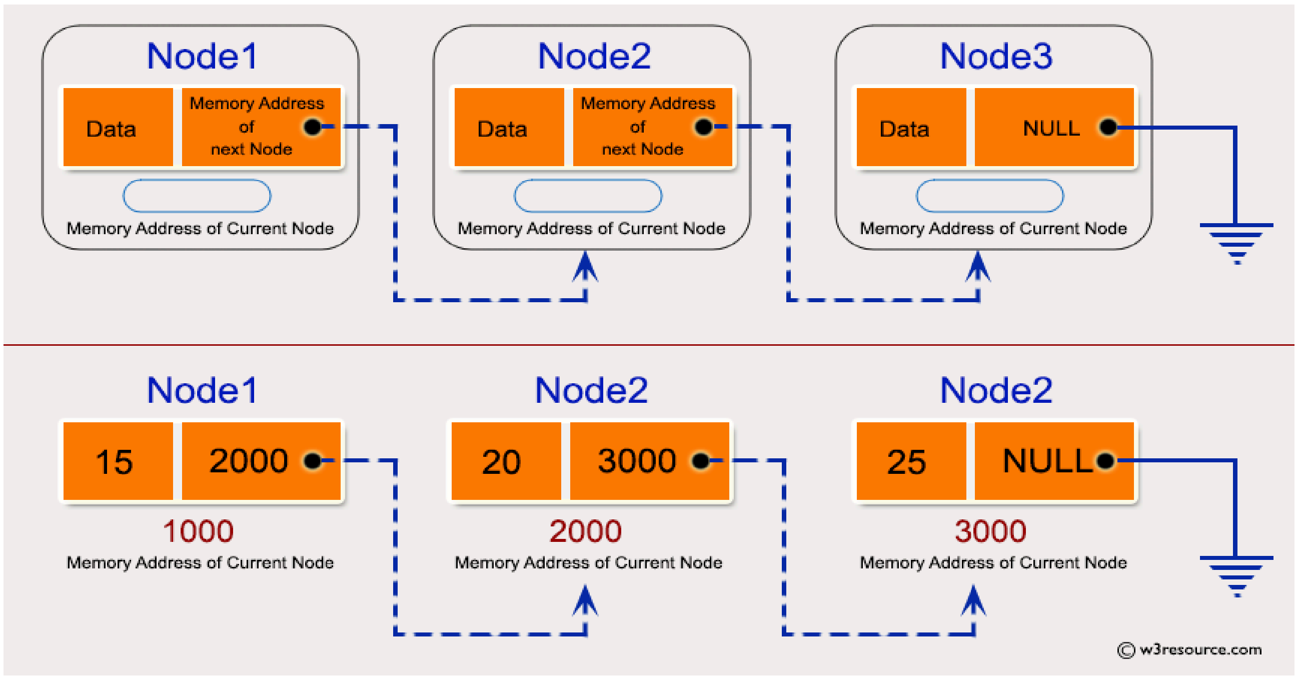
This concept is not limited to the one-dimensional array, we can refer to a multidimensional array element as well using pointers.
To know more about pointers to an array, refer to this article – Pointer to an Array

C Exercises: To create and display Singly Linked List

C Singly Linked List :

Write a program in C to create and display a Singly Linked List.

Visual Presentation:


Node1
Data
Memory Address of next Node
Memory Address of Current Node
Node2
Data
Memory Address of next Node
Memory Address of Current Node
Node3
Data
NULL
Memory Address of Current Node
Node1
15
2000
1000
Memory Address of Current Node
Node2
20
3000
2000
Memory Address of Current Node
Node2
25
NULL
3000
Memory Address of Current Node
© w3resource.com

```c
#include <stdio.h>

#include <stdlib.h>


// Structure for a node in a linked list
```

```
struct node {
    int num;                    // Data of the node
    struct node *nextptr;       // Address of the next node
} *stnode;                      // Pointer to the starting node

// Function prototypes
void createNodeList(int n); // Function to create the linked list
void displayList();         // Function to display the linked list

// Main function
int main() {
    int n;

    // Displaying the purpose of the program
    printf("\n\n Linked List : To create and display Singly Linked
List :\n");

printf("------------------------------------------------------------
-\n");

    // Inputting the number of nodes for the linked list
    printf(" Input the number of nodes : ");
    scanf("%d", &n);

    // Creating the linked list with n nodes
    createNodeList(n);

    // Displaying the data entered in the linked list
```

```c
    printf("\n Data entered in the list : \n");
    displayList();

    return 0;
}

// Function to create a linked list with n nodes
void createNodeList(int n) {
    struct node *fnNode, *tmp;
    int num, i;

    // Allocating memory for the starting node
    stnode = (struct node *)malloc(sizeof(struct node));

    // Checking if memory allocation is successful
    if(stnode == NULL) {
        printf(" Memory can not be allocated.");
    } else {
        // Reading data for the starting node from user input
        printf(" Input data for node 1 : ");
        scanf("%d", &num);
        stnode->num = num;
        stnode->nextptr = NULL; // Setting the next pointer to NULL
        tmp = stnode;

        // Creating n nodes and adding them to the linked list
        for(i = 2; i <= n; i++) {
            fnNode = (struct node *)malloc(sizeof(struct node));

            // Checking if memory allocation is successful
            if(fnNode == NULL) {
                printf(" Memory can not be allocated.");
                break;
            } else {
                // Reading data for each node from user input
                printf(" Input data for node %d : ", i);
                scanf(" %d", &num);

                fnNode->num = num;      // Setting the data for
fnNode
                fnNode->nextptr = NULL; // Setting the next pointer
to NULL

                tmp->nextptr = fnNode;  // Linking the current node
to fnNode
                tmp = tmp->nextptr;     // Moving tmp to the next
node
            }
        }
    }
}

// Function to display the linked list
void displayList() {
    struct node *tmp;
```

```c
    if(stnode == NULL) {
        printf(" List is empty.");
    } else {
        tmp = stnode;

        // Traversing the linked list and printing each node's data
        while(tmp != NULL) {
            printf(" Data = %d\n", tmp->num); // Printing the data of the current node
            tmp = tmp->nextptr;                // Moving to the next node in the list
        }
    }
}
```

Copy
Sample Output:

```
 Linked List : To create and display Singly Linked List :
-------------------------------------------------------------
 Input the number of nodes : 3
 Input data for node 1 : 5
 Input data for node 2 : 6
 Input data for node 3 : 7

 Data entered in the list :
 Data = 5
 Data = 6
 Data = 7
```