C Pointers

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

Consider the following example to define a pointer which stores the address of an integer.

```
1. int n = 10;
2. int* p = &n; // Variable p of type pointer is pointing to the address of the variable n
   of type integer.
```

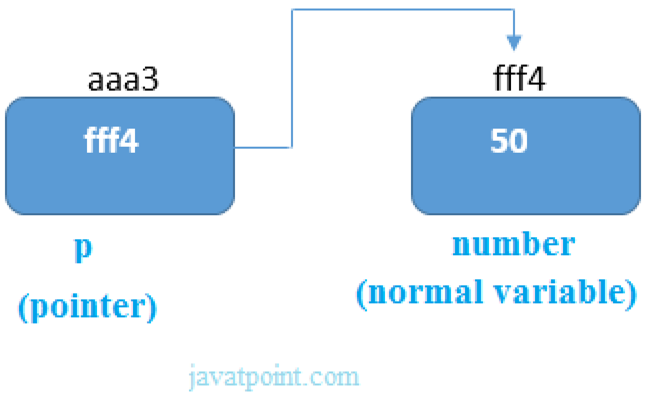
Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

```
1. int *a;//pointer to int
2. char *c;//pointer to char
```

Pointer Example

An example of using pointers to print the address and value is given below.



As you can see in the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (**indirection operator**), we can print the value of pointer variable p.

Let's see the pointer example as explained for the above figure.

```
#include<stdio.h>
int main(){
int number=50;
int *p;
p=&number;//stores the address of number variable
printf("Address of p variable is %x \n",p); // p contains the address of the number therefore printing p gives the address of number.
printf("Value of p variable is %d \n",*P); // As we know that * is used to dereference a pointer therefore if we print *p, we will get the value stored at the address contained by p.
return 0;
}
```

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```

Pointer to array

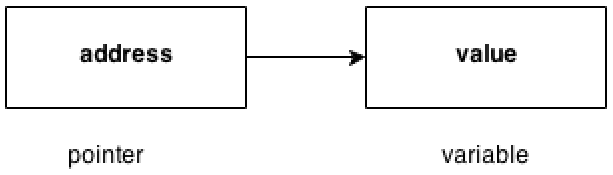
```
int arr[10];
int *p[10]=&arr; // Variable p of type pointer is pointing to the address of an integer array arr.
```

Pointer to a function

```
void show (int);
void(*p)(int) = &display; // Pointer p is pointing to the address of a function
```

Pointer to structure

```
struct st {
    int i;
    float f;
}ref;
struct st *p = &ref;
```



Advantage of pointer

1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.

2) We can **return multiple values from a function** using the pointer.

3) It makes you able to **access any memory location** in the computer's memory.

Usage of pointer

There are many applications of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

Address Of (&) Operator

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

```
#include<stdio.h>
int main(){
int number=50;
printf("value of number is %d, address of number is %u",number,&number);
return 0;
}
```

Output

```
value of number is 50, address of number is fff4
```

NULL Pointer

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will provide a better approach.

```
int *p=NULL;
```

In the most libraries, the value of the pointer is 0 (zero).

Pointer Program to swap two numbers without using the 3rd variable.

```
#include<stdio.h>
int main(){
int a=10,b=20,*p1=&a,*p2=&b;

printf("Before swap: *p1=%d *p2=%d",*p1,*p2);
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
printf("\nAfter swap: *p1=%d *p2=%d",*p1,*p2);

return 0;
}
```

Output

```
Before swap: *p1=10 *p2=20
After swap: *p1=20 *p2=10
```

Reading complex pointers

There are several things which must be taken into the consideration while reading the complex pointers in C. Lets see the precedence and associativity of the operators which are used regarding pointers.

Operator	Precedence	Associativity
(), []	1	Left to right
*, identifier	2	Right to left
Data type	3	-

Here,we must notice that,

- (): This operator is a bracket operator used to declare and define the function.
- []: This operator is an array subscript operator
- * : This operator is a pointer operator.
- Identifier: It is the name of the pointer. The priority will always be assigned to this.
- Data type: Data type is the type of the variable to which the pointer is intended to point. It also includes the modifier like signed int, long, etc).

How to read the pointer: int (*p)[10].

To read the pointer, we must see that () and [] have the equal precedence. Therefore, their associativity must be considered here. The associativity is left to right, so the priority goes to ().

Inside the bracket (), pointer operator * and pointer name (identifier) p have the same precedence. Therefore, their associativity must be considered here which is right to left, so the priority goes to p, and the second priority goes to *.

Assign the 3rd priority to [] since the data type has the last precedence. Therefore the pointer will look like following.

- char -> 4
- * -> 2
- p -> 1
- [10] -> 3

The pointer will be read as p is a pointer to an array of integers of size 10.

Example

How to read the following pointer?

```
1. int (*p)(int (*)[2], int (*)void))
```

Explanation

This pointer will be read as p is a pointer to such function which accepts the first parameter as the pointer to a one-dimensional array of integers of size two and the second parameter as the pointer to a function which parameter is void and return type is the integer.

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 **(&) 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.

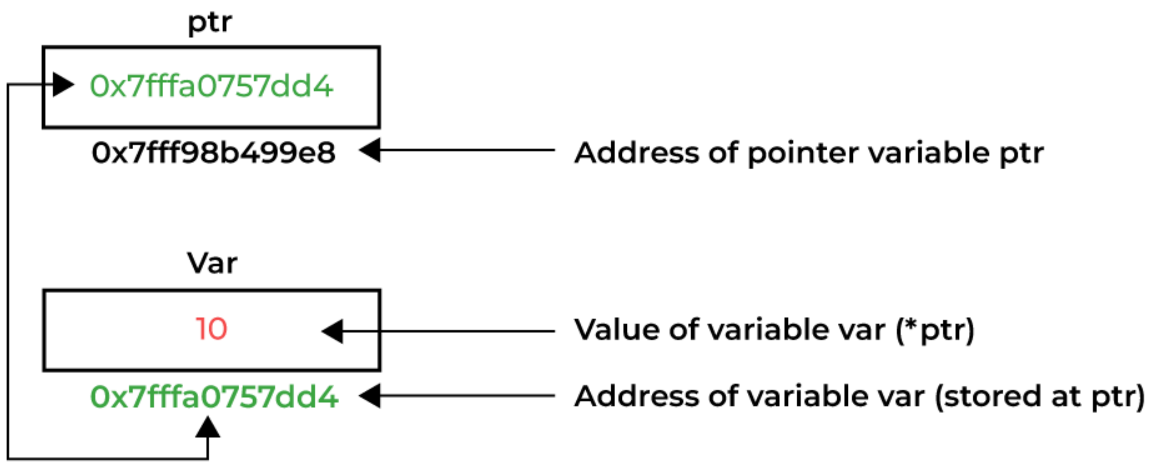
Example

```
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.



ptr
0x7fffa0757dd4
0x7fff98b499e8
Address of pointer variable ptr
Var
10
Value of variable var (*ptr)
0x7fffa0757dd4
Address of variable var (stored at ptr)

Dereferencing a Pointer in C

C Pointer Example

C

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

void geeks()
{
    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()
{
    geeks();
    return 0;
}
```

Output

```
Value at ptr = 0x7fff1038675c
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

```
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
```

Pointer Arithmetics in C with Examples

Last Updated : 22 Aug, 2023

Pointer Arithmetic is the set of valid arithmetic operations that can be performed on pointers. The pointer variables store the memory address of another variable. It doesn’t store any value.

Hence, there are only a few operations that are allowed to perform on Pointers in C language. The C pointer arithmetic operations are slightly different from the ones that we generally use for mathematical calculations. These operations are:

1. Increment/Decrement of a Pointer
2. Addition of integer to a pointer
3. Subtraction of integer to a pointer
4. Subtracting two pointers of the same type
5. Comparison of pointers

1. Increment/Decrement of a Pointer

Increment: It is a condition that also comes under addition. When a pointer is incremented, it actually increments by the number equal to the size of the data type for which it is a pointer.

For Example:

If an integer pointer that stores **address 1000** is incremented, then it will increment by 4(**size of an int**), and the new address will point to **1004.** While if a float type pointer is incremented then it will increment by 4(**size of a float**) and the new address will be **1004.**

Decrement: It is a condition that also comes under subtraction. When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.

For Example:

If an integer pointer that stores **address 1000** is decremented, then it will decrement by 4(**size of an int**), and the new address will point to **996.** While if a float type pointer is decremented then it will decrement by 4(**size of a float**) and the new address will be **996.**

Note: *It is assumed here that the architecture is 64-bit and all the data types are sized accordingly. For example, integer is of 4 bytes.*

Example of Pointer Increment and Decrement

Below is the program to illustrate pointer increment/decrement:

C

```c
#include <stdio.h>

// pointer increment and decrement

//pointers are incremented and decremented by the size of the data type they point to

int main()

{

    int a = 22;
    int *p = &a;

    printf("p = %u\n", p); // p = 6422288

    p++;

    printf("p++ = %u\n", p); //p++ = 6422292     +4    // 4 bytes

    p--;

    printf("p-- = %u\n", p); //p-- = 6422288     -4    // restored
to original value

    float b = 22.22;
    float *q = &b;



    printf("q = %u\n", q);  //q = 6422284



    q++;



    printf("q++ = %u\n", q); //q++ = 6422288    +4   // 4 bytes



    q--;



    printf("q-- = %u\n", q); //q-- = 6422284    -4  // restored to
original value





    char c = 'a';
    char *r = &c;

    printf("r = %u\n", r);   //r = 6422283

    r++;

    printf("r++ = %u\n", r);   //r++ = 6422284  +1   // 1 byte

    r--;

    printf("r-- = %u\n", r);   //r-- = 6422283  -1  // restored to
original value

    return 0;
```

```
}
```

Output

```
p = 1441900792

p++ = 1441900796

p-- = 1441900792

q = 1441900796

q++ = 1441900800

q-- = 1441900796

r = 1441900791

r++ = 1441900792

r-- = 1441900791
```

Note: *Pointers can be outputted using %p, since, most of the computers store the address value in hexadecimal form using %p gives the value in that form. But for simplicity and understanding we can also use %u to get the value in Unsigned int form.*

2. Addition of Integer to Pointer

When a pointer is added with an integer value, the value is first multiplied by the size of the data type and then added to the pointer.

For Example:

Consider the same example as above where the **ptr** is an **integer pointer** that stores **1000** as an address. If we add integer 5 to it using the expression, **ptr = ptr + 5**, then, the final address stored in the ptr will be **ptr = 1000 + sizeof(int) * 5 = 1020.**

Example of Addition of Integer to Pointer

C

```c
// C program to illustrate pointer Addition

#include <stdio.h>



// Driver Code

int main()

{

    // Integer variable

    int N = 4;
    // Pointer to an integer

    int *ptr1, *ptr2;

    // Pointer stores the address of N

    ptr1 = &N;

    ptr2 = &N;
    printf("Pointer ptr2 before Addition:
");

    printf("%p \n", ptr2);

    // Addition of 3 to ptr2

    ptr2 = ptr2 + 3;

    printf("Pointer ptr2 after Addition: ");

    printf("%p \n", ptr2);
    return 0;



}
```

Output

```
Pointer ptr2 before Addition: 0x7ffca373da9c

Pointer ptr2 after Addition: 0x7ffca373daa8
```

3. Subtraction of Integer to Pointer

When a pointer is subtracted with an integer value, the value is first multiplied by the size of the data type and then subtracted from the pointer similar to addition.

For Example:

Consider the same example as above where the **ptr** is an **integer pointer** that stores **1000** as an address. If we subtract integer 5 from it using the expression, **ptr = ptr – 5**, then, the final address stored in the ptr will be **ptr = 1000 – sizeof(int) * 5 = 980.**

Example of Subtraction of Integer from Pointer

Below is the program to illustrate pointer Subtraction:

C

```
// C program to illustrate pointer Subtraction

#include <stdio.h>



// Driver Code

int main()

{
    // Integer variable

    int N = 4;

    // Pointer to an integer

    int *ptr1, *ptr2;

    // Pointer stores the address of N

    ptr1 = &N;
    ptr2 = &N;

    printf("Pointer ptr2 before Subtraction:
");

    printf("%p \n", ptr2);

    // Subtraction of 3 to ptr2

    ptr2 = ptr2 - 3;

    printf("Pointer ptr2 after Subtraction: ");
    printf("%p \n", ptr2);




    return 0;



}
```

Output

```
Pointer ptr2 before Subtraction: 0x7ffd718ffebc

Pointer ptr2 after Subtraction: 0x7ffd718ffeb0
```

4. Subtraction of Two Pointers

The subtraction of two pointers is possible only when they have the same data type. The result is generated by calculating the difference between the addresses of the two pointers and calculating how many bits of data it is according to the pointer data type. The subtraction of two pointers gives the increments between the two pointers.

For Example:

Two integer pointers say **ptr1(address:1000)** and **ptr2(address:1004)** are subtracted. The difference between addresses is 4 bytes. Since the size of int is 4 bytes, therefore the **increment between ptr1 and ptr2** is given by **(4/4) = 1.**

Example of Subtraction of Two Pointer

Below is the implementation to illustrate the Subtraction of Two Pointers:

C

```c
// C program to illustrate Subtraction

// of two pointers

#include <stdio.h>



// Driver Code

int main()

{

    int x = 6; // Integer variable declaration
    int N = 4;

    // Pointer declaration

    int *ptr1, *ptr2;

    ptr1 = &N; // stores address of N

    ptr2 = &x; // stores address of x
    printf(" ptr1 = %u, ptr2 = %u\n", ptr1, ptr2);



    // %p gives an hexa-decimal value,



    // We convert it into an unsigned int value by using
%u







    // Subtraction of ptr2 and ptr1



    x = ptr1 - ptr2;







    // Print x to get the Increment
    // between ptr1 and ptr2

    printf("Subtraction of ptr1 "

           "& ptr2 is %d\n",

           x);

    return 0;

}
```

Output

```
ptr1 = 2715594428, ptr2 = 2715594424

Subtraction of ptr1 & ptr2 is 1
```

5. Comparison of Pointers

We can compare the two pointers by using the comparison operators in C. We can implement this by using all operators in C **>, >=, <, <=, ==, !=.** It returns true for the valid condition and returns false for the unsatisfied condition.

1. **Step 1:** Initialize the integer values and point these integer values to the pointer.
2. **Step 2:** Now, check the condition by using comparison or relational operators on pointer variables.
3. **Step 3:** Display the output.

Example of Pointer Comparision

C

```c
// C Program to illustrare pointer comparision

#include <stdio.h>



int main()

{

    // declaring array

    int arr[5];
    // declaring pointer to array name

    int* ptr1 = &arr;

    // declaring pointer to first element

    int* ptr2 = &arr[0];

    if (ptr1 == ptr2) {

        printf("Pointer to Array Name and First Element
"

               "are Equal.");
    }

    else {

        printf("Pointer to Array Name and First Element
"

                "are not Equal.");

    }

    return 0;

}
```

Output

```
Pointer to Array Name and First Element are Equal.
```

Comparison to NULL

A pointer can be compared or assigned a NULL value irrespective of what is the pointer type. Such pointers are called NULL pointers and are used in various pointer-related error-handling methods.

C

```c
// C Program to demonstrate the pointer comparison with
NULL

// value

#include <stdio.h>
```

```
int main()

{

    int* ptr = NULL;

    if (ptr == NULL) {

        printf("The pointer is NULL");

    }
    else {

        printf("The pointer is not NULL");

    }

    return 0;

}
```

Output

```
The pointer is NULL
```

Comparison operators on Pointers using an array

In the below approach, it results in the count of odd numbers and even numbers in an array. We are going to implement this by using a pointer.

1. **Step 1:** First, declare the length of an array and array elements.

2. **Step 2:** Declare the pointer variable and point it to the first element of an array.
3. **Step 3:** Initialize the count_even and count_odd. Iterate the for loop and check the conditions for the number of odd elements and even elements in an array
4. **Step 4:** Increment the pointer location ptr++ to the next element in an array for further iteration.
5. **Step 5:** Print the result.

Example of Pointer Comparison in Array

C

```
// Pointer Comparision in Array

#include <stdio.h>



int main()

{

    int n = 10; // length of an array



    int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int* ptr; // Declaration of pointer variable

    ptr = arr; // Pointer points the first (0th index)

                   // element in an array

    int count_even = 0;

    int count_odd = 0;

    for (int i = 0; i < n; i++) {
        if (*ptr % 2 == 0) {

            count_even++;

        }

        if (*ptr % 2 != 0) {

            count_odd++;

        }

        ptr++; // Pointing to the next element in an
array
```

```
    }

    printf("No of even elements in an array is : %d",

           count_even);

    printf("\nNo of odd elements in an array is : %d",

           count_odd);

}
```

Output

```
No of even elements in an array is : 5
No of odd elements in an array is : 5
```

Pointer Arithmetic on Arrays

Pointers contain addresses. Adding two addresses makes no sense because there is no idea what it would point to. Subtracting two addresses lets you compute the offset between the two addresses. An array name acts like a pointer constant. The value of this pointer constant is the address of the first element.

For Example: if an array is named **arr** then **arr and &arr[0]** can be used to reference the array as a pointer.

Below is the program to illustrate the Pointer Arithmetic on arrays:

Program 1:

C

```
// C program to illustrate the
array

// traversal using pointers

#include <stdio.h>



// Driver Code

int main()

{
    int N = 5;

    // An array

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

    // Declare pointer variable

    int* ptr;
    // Point the pointer to first

    // element in array arr[]

    ptr = arr;



    // Traverse array using ptr

    for (int i = 0; i < N; i++) {
        // Print element at which

        // ptr points

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

        ptr++;

    }

}
```

Output

```
1 2 3 4 5
```

Program 2:

```
    C
```

```
// C program to illustrate the array

// traversal using pointers in 2D array

#include <stdio.h>



// Function to traverse 2D array

// using pointers

void traverseArr(int* arr, int N, int M)

{
    int i, j;

    // Traverse rows of 2D matrix

    for (i = 0; i < N; i++) {

        // Traverse columns of 2D matrix

        for (j = 0; j < M; j++) {
            // Print the element
            printf("%d ", *((arr + i * M) + j));
        }
        printf("\n");
    }
}
```

```
// Driver Code

int main()

{

    int N = 3, M = 2;

    // A 2D array

    int arr[][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 }
};
```

```
    // Function Call

    traverseArr((int*)arr, N, M);

    return 0;

}
```

Output

```
1 2

3 4

5 6
```