C Pointers

The pointer in C language is a variable which stores the address of another variable i.e., direct address of the memory location. This variable can be of type int, char, array, function, or any other pointer. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is –

```
type *var-name;
```

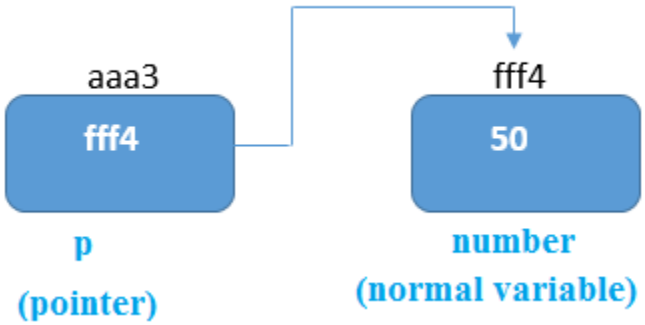
Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations –

int *a;//pointer to int

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.

```
#include <stdio.h>

int main () {

   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );
```

```
   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

OUTPUT

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program −

```
#include <stdio.h>

int main () {

   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;
}
```

OUPUT

```
The value of ptr is 0
```

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows –

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

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.

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.

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

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

Pointer Arithmetic in C

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

- Increment
- Decrement
- Addition
- Subtraction
- Comparison

Incrementing Pointer in C

If we increment a pointer by 1, the pointer will start pointing to the immediate next location. This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.

We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

The Rule to increment the pointer is given below:

```
new_address= current_address + i * size_of(data type)
```

Where i is the number by which the pointer get increased.

32-bit

For 32-bit int variable, it will be incremented by 2 bytes.

64-bit

For 64-bit int variable, it will be incremented by 4 bytes.

Let's see the example of incrementing pointer variable on 64-bit architecture.

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p+1;
8. printf("After increment: Address of p variable is %u \n",p); // in our case, p will get incre
   mented by 4 bytes.
9. return 0;
10. }
```

OUTPUT

```
Address of p variable is 3214864300
After increment: Address of p variable is 3214864304
```

Traversing an array by using pointer

```
1. #include<stdio.h>
2. void main ()
3. {
4.     int arr[5] = {1, 2, 3, 4, 5};
5.     int *p = arr;
6.     int i;
7.     printf("printing array elements...\n");
8.     for(i = 0; i< 5; i++)
9.     {
10.        printf("%d  ",*(p+i));
11.    }
12. }
```

OUPUT

```
printing array elements...
1  2  3  4  5
```

Decrementing Pointer in C

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formula of decrementing the pointer is given below:

new_address= current_address - i * size_of(data type)

32-bit

For 32-bit int variable, it will be decremented by 2 bytes.

64-bit

For 64-bit int variable, it will be decremented by 4 bytes.

Let's see the example of decrementing pointer variable on 64-bit OS.

```
1. #include <stdio.h>
2. void main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p-1;
8. printf("After decrement: Address of p variable is %u \n",p); // P will now point to the im
   midiate previous location.
9. }
```

OUTPUT

```
Address of p variable is 3214864300
After decrement: Address of p variable is 3214864296
```

C Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

1. new_address= current_address + (number * size_of(data type))

32-bit

For 32-bit int variable, it will add 2 * number.

64-bit

For 64-bit int variable, it will add 4 * number.

Let's see the example of adding value to pointer variable on 64-bit architecture.

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p+3;   //adding 3 to pointer variable
8. printf("After adding 3: Address of p variable is %u \n",p);
9. return 0;
10. }
```

OUTPUT

```
Address of p variable is 3214864300
After adding 3: Address of p variable is 3214864312
```

As you can see, the address of p is 3214864300. But after adding 3 with p variable, it is 3214864312, i.e., 4*3=12 increment. Since we are using 64-bit architecture, it increments 12. But if we were using 32-bit architecture, it was incrementing to 6 only, i.e., 2*3=6. As integer value occupies 2-byte memory in 32-bit OS.

C Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given below:

new_address= current_address - (number * size_of(data type))

32-bit

For 32-bit int variable, it will subtract 2 * number.

64-bit

For 64-bit int variable, it will subtract 4 * number.

Let's see the example of subtracting value from the pointer variable on 64-bit architecture.

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

```
2. int main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p-3; //subtracting 3 from pointer variable
8. printf("After subtracting 3: Address of p variable is %u \n",p);
9. return 0;
10. }
```

OUTPUT

```
Address of p variable is 3214864300
After subtracting 3: Address of p variable is 3214864288
```

You can see after subtracting 3 from the pointer variable, it is 12 (4*3) less than the previous address value.

However, instead of subtracting a number, we can also subtract an address from another address (pointer). This will result in a number. It will not be a simple arithmetic operation, but it will follow the following rule.

If two pointers are of the same type,

Address2 - Address1 = (Subtraction of two addresses)/size of data type which pointer points

```
1. #include<stdio.h>
2. void main ()
3. {
4.     int i = 100;
5.     int *p = &i;
6.     int *temp;
7.     temp = p;
8.     p = p + 3;
9.     printf("Pointer Subtraction: %d - %d = %d",p, temp, p-temp);
10. }
```

OUTPUT

```
Pointer Subtraction: 1030585080 - 1030585068 = 3
```

Illegal arithmetic with pointers

There are various operations which can not be performed on pointers. Since, pointer stores address hence we must ignore the operations which may lead to an illegal address, for example, addition, and multiplication. A list of such operations is given below.

- Address + Address = illegal
- Address * Address = illegal
- Address % Address = illegal
- Address / Address = illegal
- Address & Address = illegal
- Address ^ Address = illegal
- Address | Address = illegal
- ~Address = illegal