C Programming Pointers

A **pointer** is a variable that stores the address of another variable.

Unlike other variables that hold values of a certain type, pointer holds the address of a variable.

For example, an integer variable holds (or you can say stores) an integer value, however an integer pointer holds the address of a integer variable.

Pointer Syntax

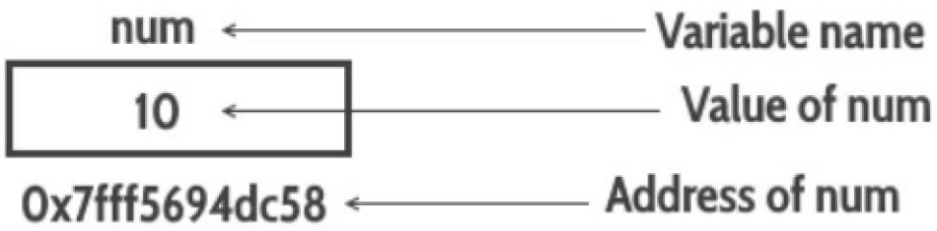
Here is how we can declare pointers.

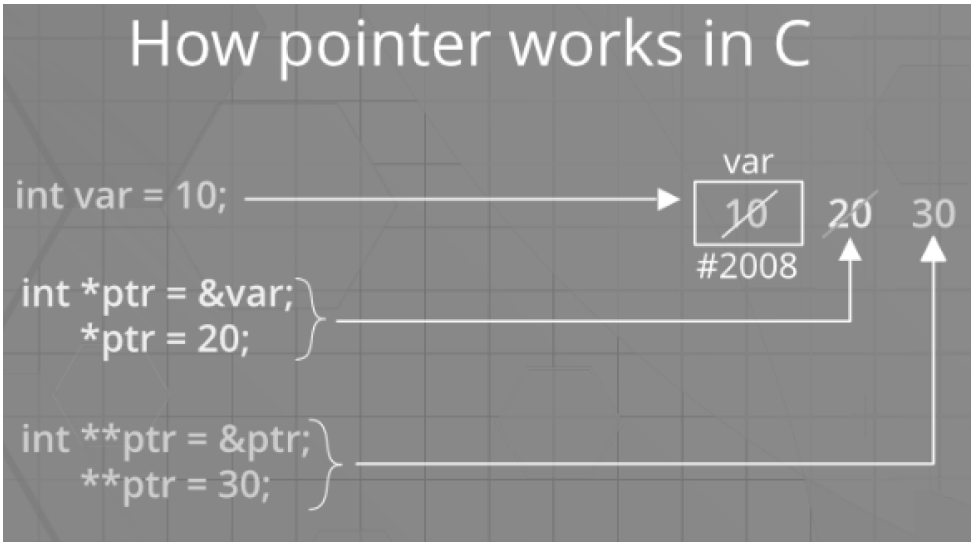
```
int* p;
```

Here, we have declared a pointer `p` of `int` type.

You can also declare pointers in these ways.

```
int *p1;
int * p2;
```





```
// General syntax
datatype *var_name;

// An example pointer "ptr" that holds
// address of an integer variable or holds
// address of a memory whose value(s) can
// be accessed as integer values through "ptr"
int *ptr;
```

```
#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
```

Operators that are used with Pointers

Lets discuss the operators & and * that are used with Pointers in C.

"Address of"(&) Operator

We have already seen in the first example that we can display the address of a variable using ampersand sign. I have used &num to access the address of variable num. The **& operator** is also known as "**Address of**" Operator.

printf("Address of var is: %p", &num);

Point to note: %p is a format specifier which is used for displaying the address in hex format. Now that you know how to get the address of a variable but **how to store that address in some other variable?** That's where pointers comes into picture. As mentioned in the beginning of this guide, pointers in C programming are used for holding the address of another variables.

Pointer is just like another variable, the main difference is that it stores address of another variable rather than a value.

"Value at Address"(*) Operator

The * Operator is also known as **Value at address** operator.

By using * operator we can access the value of a variable through a pointer. For example:

double a = 10;

double *p;

p = &a;

*p would give us the value of the variable a. The following statement would display 10 as output.

printf("%d", *p);

Similarly if we assign a value to *pointer like this:

*p = 200;

It would change the value of variable a. The statement above will change the value of a from 10 to 200.

Example of Pointer demonstrating the use of & and *

```
#include <stdio.h>
int main()
{
   /* Pointer of integer type, this can hold the
    * address of a integer type variable.
    */
   int *p;

   int var = 10;

   /* Assigning the address of variable var to the pointer
    * p. The p can hold the address of var because var is
    * an integer type variable.
    */
   p= &var;

   printf("Value of variable var is: %d", var);
   printf("\nValue of variable var is: %d", *p);
   printf("\nAddress of variable var is: %p", &var);
   printf("\nAddress of variable var is: %p", p);
   printf("\nAddress of pointer p is: %p", &p);
   return 0;
}
```

Output:

```
Value of variable var is: 10
Value of variable var is: 10
Address of variable var is: 0x7fff5ed98c4c
Address of variable var is: 0x7fff5ed98c4c
Address of pointer p is: 0x7fff5ed98c50
```

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 –

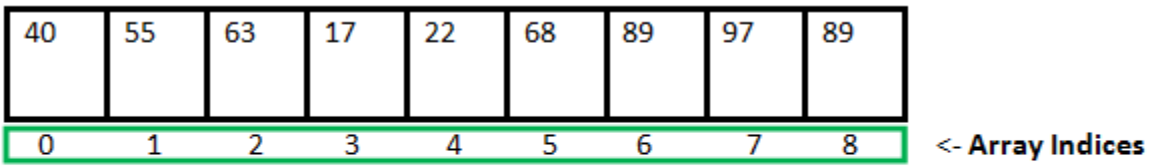
```
#include <stdio.h>

int main () {

   int  *ptr = NULL;

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

   return 0;
}
```

OUTPUT:

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

Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()

Since C is a structured language, it has some fixed rules for programming. One of it includes changing the size of an array. An array is collection of items stored at continuous memory locations.



Array Length = 9
First Index = 0
Last Index = 8

As it can be seen that the length (size) of the array above made is 9. But what if there is a requirement to change this length (size). For Example,

- If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.
- Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

This procedure is referred to as **Dynamic Memory Allocation in C**.

C **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

To allocate memory dynamically, library functions are `malloc()`, `calloc()`, `realloc()` and `free()` are used. These functions are defined in the `<stdlib.h>` header file.

C malloc()

- The name "malloc" stands for memory allocation.
- The `malloc()` function reserves a block of memory of the specified number of bytes. And, it returns a pointer of `void` which can be casted into pointers of any form.

Syntax of malloc()

```
ptr = (castType*) malloc(size);
```

Example

```
ptr = (float*) malloc(100 * sizeof(float));
```

- The above statement allocates 400 bytes of memory. It's because the size of `float` is 4 bytes. And, the pointer `ptr` holds the address of the first byte in the allocated memory.
- The expression results in a `NULL` pointer if the memory cannot be allocated.

C calloc()

- The name "calloc" stands for contiguous allocation.
- The `malloc()` function allocates memory and leaves the memory uninitialized. Whereas, the `calloc()` function allocates memory and initializes all bits to zero.

Syntax of calloc()

```
ptr = (castType*)calloc(n, size);
```

Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

- The above statement allocates contiguous space in memory for 25 elements of type `float`.

C realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the `realloc()` function.

Syntax of realloc()

```
ptr = realloc(ptr, x);
```

Here, `ptr` is reallocated with a new size `x`.

- Dynamically allocated memory created with either `calloc()` or `malloc()` doesn't get freed on their own. You must explicitly use `free()` to release the space.

Syntax of free()

```
free(ptr);
```

- This statement frees the space allocated in the memory pointed by `ptr`.