C Pointers

When we initialize an array, we usually come to know about the:

- Memory block which is the space a variable gets in RAM. We can think of that space as a block.
- Name of the memory block which is the variable's name itself.
- Content of that block that is the value stored in that variable.
- Address of the memory block assigned to the variable which is a unique address that allows us to access that variable.

What is a Pointer?

- A pointer is a variable that contains the address of another variable. It means it is a variable that points to any other variable.
- Although this is itself a variable, it contains the address or memory address of any other variable.
- It can be of type int, char, array, function, or even any other pointer.
- Its size depends on the architecture.
- Pointers in C Language can be declared using the * (asterisk symbol).

So, pointers are nothing but variables that store addresses of other variables, and by using pointers, we can access other variables too and can even manipulate them.

Applications of Pointers

- Pointers are used to dynamically allocate or deallocate memory using methods such as malloc(), realloc(), calloc(), and free().
- Pointers are used to point to several containers such as arrays, or structs, and also for passing addresses of containers to functions.
- Return multiple values from a function.
- Rather than passing a copy of a container to a function, we can simply pass its pointer. This helps reduce the memory usage of the program.
- Pointers reduce the code and improve the performance.

Operations on Pointers

Address of Operator (&):

- It is a unary operator.
- Operand must be the name of an already defined variable.
- & operator gives the address number of the variable.
- & is also known as the “Referencing Operator”.

Here’s one example to demonstrate the use of the address of the operator.

```
#include <stdio.h>


int main()
{
   int a = 100;
   printf("%d\n", a);
   printf("Address of variable a is %d", &a);
   return 0;
}
```

Output:

100

Address of variable a is 6422220

Indirection Operator (*):

- * is an indirection operator.
- It is also known as the “Dereferencing Operator”.
- It is also a unary operator.
- It takes an address as an argument.
- * returns the content/container whose address is its argument.

Here’s one example to demonstrate the use of the indirection operator.

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

```
int main()
{
   int a = 100;
   printf("Value of variable a stored at address %d is %d.", &a, *(&a));
   return 0;
}
```

Output:

Value of variable a stored at address 6422220 is 100.

C VOID Pointer

After a brief discussion about pointers, it's time to start with a very important type of pointers: void pointers and their functionalities. We already know that a void function has no return type, i.e., functions that are not returning anything are given the type void. Now, in the case of pointers that are given the datatype of a void, they can be typecasted into any other data type according to the necessity. This means we do not have to decide on a data type for the pointer initially.

Void pointers can also be addressed as **general-purpose pointer variables.**

Let's see a few examples that will demonstrate the functionalities of a void pointer.

Example:

```
int var = 1;void *voidPointer = &var;
```

Here, the data type of the void pointer gets typecasted into int as we have stored the address of an integer value in it.

```
char x = 'a';void *voidPointer = &x;
```

In this example, the void pointer's data type gets typecasted to char as we have stored the address of a character value in it.

Type casting a void pointer must also remind you of the way we used to type cast a void pointer returned by the functions malloc() and calloc() for dynamic memory allocation. There also, the heap returns a void pointer to the memory requested. We could type cast it to any other data type, and that is where a void pointer comes in handy.

Two important features of a VOID pointer are:

Void pointers cannot be dereferenced.

This can be demonstrated with the help of an example.

int a = 10;void *voidPointer;voidPointer = &a;printf("%d", *voidPointer);

Output:

Compiler Error!

This program will throw a compile-time error, as we cannot dereference a void pointer, meaning that we would compulsorily have to typecast the pointer every time it is being used. Here's how it should be done.

int a = 10;void *voidPointer;voidPointer = &a;printf("%d", *(int *)voidPointer);

Output:

10

The compiler will not throw any error and will directly output the result because we are using the type along with the pointer.

Pointer arithmetics cannot be used with void pointers since it is not holding any address to be able to increment or decrement its value.

C NULL Pointer

A pointer that is not assigned any value or memory address but NULL is known as a NULL pointer. A NULL pointer does not point to any object, variable, or function. NULL pointers are often used to initialize a pointer variable, where we wish to represent that the pointer variable isn't currently assigned to any valid memory address yet.

This is how we define a NULL pointer:

int *ptr = NULL;

A NULL pointer generally points to a NULL or 0th memory location, so in simple words, no memory is allocated to a NULL pointer.

Dereferencing a NULL pointer

The dereferencing behavior of a NULL pointer is very much similar to that of a void pointer. A NULL pointer itself is a kind of a VOID pointer and hence, we have to typecast it into any data type the way we do to a void pointer before dereferencing. Failing to do so results in an error at compile time.

NULL pointer vs. Uninitialized pointer

NULL pointers and uninitialized pointers are different, as a NULL pointer does not occupy any memory location. That means, it points to nowhere but to a zeroth location. In contrast, an uninitialized pointer means that the pointer occupies a garbage value

address. The garbage value address is still a real memory location and hence not a NULL value. So to be on the safe side, NULL pointers are preferred.

NULL pointer vs. Void pointer

NULL pointers and void pointers very much sound similar just because of their nomenclatures, but they are very different as a NULL pointer is a pointer with a NULL value address, and a void pointer is a pointer of void data type. Their significances are contrasting.

An example of a NULL pointer is as follows:

int *ptr = NULL;

Here, an integer pointer variable is declared with a value NULL, which means it is not pointing to any memory location.

An example of a VOID pointer is as follows:

void *ptr;

Now, this is a void pointer. This pointer will typecast itself to any other data type as per the datatype of the value stored in it.

Advantages of a NULL pointer

1. We can initialize a pointer variable without allocating any specific memory location to it.
2. We can use it to check whether a pointer is legitimate or not. We can check that by making the pointer a NULL pointer, after which it cannot be dereferenced.
3. A NULL pointer is used for comparison with other pointers to check whether that other pointer itself is pointing to some memory address or not.
4. We use it for error handling in the case of C programming.
5. We can pass a NULL pointer at places where we do not want to pass a pointer with a valid memory address.

Dangling Pointer

Dangling pointers are pointers that are pointing to a memory location that has been already freed or deleted.

Dangling pointers often come into existence during object destruction. It happens when an object with an incoming reference is deleted or de-allocated, without

modifying the value of the pointer. The pointer still points to the memory location of the deallocated memory.

The system may itself reallocate the previously deleted memory and several unpredicted results may occur as the memory may now contain different data.

Dangling pointers are caused by the following factors:

De-allocating or free variable memory

When memory is deallocated, the pointer keeps pointing to freed space. An example
#include <stdio.h>

int main()

{

    int a = 80;

    int *ptr = (int *)malloc(sizeof(int));

    ptr = &a;

    free(ptr);

    return 0;

}

The above code demonstrates how a variable pointer `*ptr` and an integer variable `a` containing a value 80 was created. The pointer variable `*ptr` is created with the help of the `malloc()` function. As we know that `malloc()` function returns a void pointer, so we use `int *` for type conversion to convert the void pointer into an int pointer.

Function Call

Now, we will see how the pointer becomes dangling with the function call.

```
#include <stdio.h>

int *myvalue()

{

  int a = 10;

  return &a;
```

```
}


int main()

{

   int *ptr = myvalue();

   printf("%d", *ptr);

   return 0;

}
```

Output:

```
Segmentation Fault!
```

In the above code, first, we create the `main()` function in which we have declared `ptr` pointer, which contains the return value of the `myvalue()` function. When the function `myvalue()` is called, the program control moves to the context of the `int *myvalue()`. Then, the function `myvalue()` returns the address of the integer variable `a`.

This is where the program control comes back to the `main()` function and the integer variable `a` becomes unavailable for the rest of the program execution. And the pointer `ptr` becomes dangling as it points to a memory location that has been freed or deleted from the stack. Hence, the program results in a segmentation fault.

Had this code been updated, and the integer variable been declared globally which is static, and as we know, any static variable stores in global memory, the output would have been 10.

How to avoid the Dangling pointer errors

The dangling pointer introduces nasty bugs into our programs, and these bugs often result in security holes. By merely initializing the pointer value to `NULL`, these errors following the creation of dangling pointer can be avoided. After that, the pointer will no longer point to the freed memory location. The reason behind assigning the `NULL` value to the pointer is to have the pointer not point to any random or previously assigned memory location.

Wild Pointer

Uninitialized pointers are known as wild pointers because they point to some arbitrary memory location while they are not assigned to any other memory location. This may even cause a program to crash or behave unpredictably at times.

For example:

int *ptr;

Here, a pointer named ptr is created but was not given any value. This makes the pointer ptr a wild pointer.

Declaring a pointer and not initializing it has its own disadvantages. One such disadvantage is that it will store any garbage value in it. A random location in memory will be held in it arbitrarily. This random allocation often becomes tough for a programmer to debug, causing a lot of problems in the execution of the program.

A. Avoiding problems due to WILD pointers

Dereferencing a wild pointer becomes problematic at times, and to avoid them, we often prefer to convert a wild pointer to a NULL pointer. By doing so, our pointer will not point to any garbage memory location; rather, it will point to a NULL location. We can convert a wild pointer to a NULL pointer by merely setting it equal to NULL.

B. Dereferencing

We cannot dereference a wild pointer as we are not sure about the data in the memory it is pointing towards. In addition to causing a lot of bugs, dereferencing a wild pointer can also cause the program to crash.