Void Pointers

we are going to learn about ***Void pointers and their functionalities***. As we are already familiar, that void has no return type i.e., functions that are not returning anything are given the type void. So, in case of void pointers, they can be typecast into any data type whenever we want, meaning we do not have to decide a type for the pointer initially. In simple words, it is a general-purpose pointer variable.

We will try to understand the functionality of the void pointer with a couple of examples.

Example1:

```
int x = 1;
void *pointer = &x;
```

Now, in the above example, the data type of the void pointer is now int as we have stored an integer value in it.

Example2:

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

However, in this example, the void pointer's data type has shifted to char as we have stored a character value in it.

We saw the use of malloc() and calloc() for dynamic memory allocation. Here we will discuss the concept as we can allocate any data type to them if we request the memory from the heap using the void pointer. So void pointer comes in handy for these two functions because we can typecast them in any variable type.

Important points:

- C does not allow void pointers to be dereferenced.
- We cannot use pointer arithmetic with void pointers.

Let us understand the dereference concept in a little bit more detail with the help of an example.

Example 1:

```
int a = 1;
void * pointer;
pointer =&a;
printf("%d",* pointer);
```

This program will, through a compile-time error, as we cannot dereference a void pointer, meaning that we have to typecast the pointer every time it is being used.

Now let us take another example.

Example 2:

```
int a = 1;
void * pointer;
pointer =&a;
printf("%d",*(int*) pointer);
```

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

Null Pointer

We are going to learn about NULL pointers. The concept of the NULL pointer is very easy and simple to understand. A NULL pointer is a pointer that does not point to any memory location. It generally points to NULL or 0th memory location, so in simple words, no memory is allocated to a NULL pointer.

Dereference:

Here again, we will see the concept of dereferencing as its behaviour, in this case, is the same as a void pointer. We can say that the type of a NULL pointer is void. So we have to typecast it into any variable the same as in the void pointer case.

NULL pointer vs. Uninitialized pointer:

The two are different as the Null pointer points to the 0th memory location, which means that it does not occupy any memory location. In contrast, an uninitialized pointer means that the pointer occupies a garbage value. The garbage value can be any value the garbage collector assigns to the pointer, which may point to some memory location. So to be on the safe side, NULL pointers are preferred.

NULL pointer vs. Void pointer:

NULL pointer and void pointer may sound similar to their wordy meanings overlap too much, but they are very different as the NULL pointer is a value, and the void pointer is a type. We will see the meaning of the sentence with the help of an example.

```
int *ptr = NULL;
```

Here we have set ptr to NULL, which means it is not pointing to any memory location.

Now let us see another example:

```
void *ptr;
```

Now, this is a void pointer in which the value will set according to what sort of value we store in it. If we equal it to an integer, then its value will be int, and if we pass a character variable in it then, its value will be char and so on.

Advantages:

- We can initialize the pointer variable without allocating any specific address to it.
- 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.
- It is used for comparison with other pointers to check whether they are pointing to some memory address or not.
- We use it for error handling in the case of C programming.
- 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 freed or deleted.

Dangling pointers arise during object destruction, when an object with an incoming reference is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory. The system may reallocate the previously deleted memory; the unpredicted result may occur as the memory may now contain different data.

There are some causes of dangling pointers. The causes are explained below with examples.

- ### De-allocating or free variable memory:-

When memory is deallocated, the pointer keeps pointing to freed space.

```
int main() {
   int *ptrr=(int *)malloc(sizeof(int));
   int x=80;
   ptrr=&x;
   free(ptrr);
   return 0;
}
```

In the above code, we have created two variables *ptrr and a where the ptrr is a pointer, and x is the integer variable that contains a value 80. The *ptrr (pointer variable) is created with the help of the malloc() function. As we know that malloc() function returns the void, so we use int * for type conversion to convert void pointer into int pointer.

- Function call

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

```
#include
int *myvalue(){
int a=5;
return &a;
}
int main()
{
int *ptr=myvalue();
printf("%d", *ptr);
return 0;
}
```

Output:- Segmentation fault (core dumped).

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(). When the myvalue() is called, then the control moves to the context of the int *myvalue(), the myvalue() returns the address of the "a" variable. Now, when control comes back to the main() function, it means the variable a is no longer available. Therefore, we can say that the pointer is dangling as it points to a memory location that has been freed or deleted.

If the above code is written with a variable, 'a' is static. We know that static variable stores in global memory. So, in this case, the output will be 5.

How to Avoid the Dangling Pointer Errors?

The dangling pointer introduces nasty bugs in our C programming and these bugs frequently become security holes at a time. These dandling pointer errors can be avoided by initializing the pointer value to the NULL. If we assign the **NULL value** to the pointer, then the pointer will not point to the memory location that has been freed. By assigning the NULL value to the pointer means that the pointer is not pointing to any memory location.

Wild Pointer

We are going to learn about **wild pointers**. A wild pointer is a simple concept, but a separate tutorial had to be made to make you aware. So, let's start with the definition.

"Uninitialized pointers are known as void pointers."

For Example:

```
int *ptr;
```

In the above example, we created a pointer but didn't give it any value, so it becomes a wild pointer.

Its **disadvantage** is that it will store any garbage value in it, meaning it will hold some arbitrary memory location. Due to the storage of some random location, it can cause a lot of bugs in the program, and sometimes the programmer will not even be able to identify the cause.

Solution:

To avoid the bugs and errors it can cause in a program; we prefer to convert a void pointer to a **NULL pointer**. By doing so, our pointer will not point to any memory location, as it will point to 0 or NULL location. We can convert a wild pointer to a NULL pointer by merely placing it equal to NULL. Let us see it in C syntax.

```
int *ptr = NULL;
```

So, we will adopt this method if we are not using our pointer to point at some memory location.

Now the other way to save ourselves from such problems is to **initialize the pointer**.

For example:

```
int x = 3;
int *ptr;
ptr = &3;
```

If we only execute the first two lines in the above chunk of code, then it will be a wild pointer as no value has been initialized to it, but if we execute the third line too, it will point to some location, making it a normal pointer.

Dereferencing:

We cannot dereference a wild pointer as we can not be sure about the data in the memory it is pointing towards. Dereferencing a wild pointer can cause a lot of **bugs** and can also **crash** the program.

Conclusion:

So, guys, this tutorial was of a fundamental level. The reason for making it separately was to make you aware of a wild pointer to keep you from making errors that can create bugs you couldn't understand the reason for. I also want you to learn C programming concepts to an expert level, as I promised in my first tutorial.