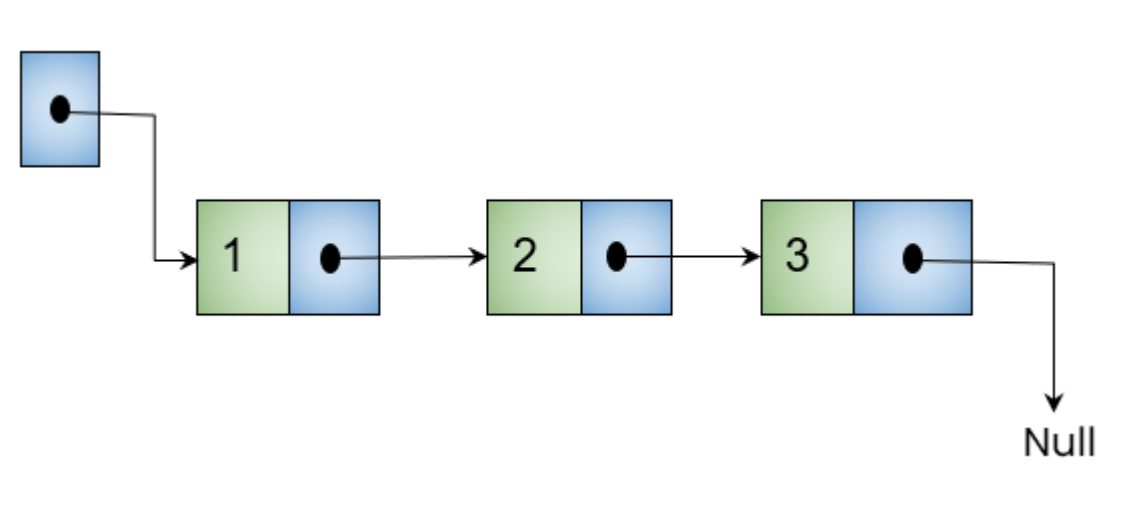
Types of pointers in c

In this tutorial, you will see the common types of pointers in C and their syntax and uses. Also, you will walk through certain issues that arise while using these pointers and how to solve them with the help of easy examples.

NULL POINTERS

Before learning about null pointer in programming let us grasp some insight about null pointer in the context of computer memory. In memory, a null pointer is a simple command used to instruct a program or OS to point to an empty location.

Null pointers in C are a unique category of pointers that do not point towards a specific address location. In the case of null pointers, we set a null value instead of a memory address. The below picture shows a linked list with a null pointer.



The syntax of null pointer is :

```
data_type * pointer_variable = NULL;
```

In the programming language, ' **NULL**' keyword is specially reserved for this purpose. For different data types they are declared like:

```
int *pntr = NULL;
char *pntr = '\0';
float *pntr = (float *)0;
```

Following program will give the idea of null pointer in C.

```
#include<stdio.h>
int main()
{
 int* vp = NULL;

  printf("vp contains value:%d\n",vp);
}
```

Output:

```
vp contains value:0
```

Unlike void pointers, Null pointers are type pointers as they specify the data type of pointer variable, but with a 'null' value. Hence it always has value **0**. In the program vp is a null pointer of integer type which contains the value 0.

Why null pointer is used?

1. **The first and foremost use of a null pointer is to initialize a pointer variable**
   Typically when a pointer variable is declared like below and is not initialized,

```
int * p; // uninitialized pointer
```

What actually happens is it will point to some random memory address and stores some garbage values. This garbage value can be a reason to crash the program when you attempt to use this pointer or to pass in the function as an argument. To avoid this always initialize the pointer with a Null value as shown below.

```
int * p = NULL; // null pointer
```

2. **Secondly to pass a null pointer as an argument in the function**
   In case if you don't want to pass a valid memory address to function you can use a null pointer as its argument

```
float funct(int *p)
{
:::::::::::
}
funct(NULL);
```

3. **Validate pointers with a null value**
   Before accessing a pointer always make sure that the pointer variable is initialized either to a valid memory address or null value. Otherwise unexpected errors may cause and will be a hassle.

```
#include<stdio.h>

void sum(int *p2)
{
    if(p2 == NULL)
    {
        //Handle NULL pointer
        return;
    }
    else
    {
        //function body
    }
}
void main()
{
    int *p1 = NULL;
    sum(p1);
}
```

In the above code, p1 is a null pointer that is passed as an argument in function sum. The function initially checks whether the argument passed is a null pointer or not. if it is a null pointer then the code will handle the null pointer. Under other conditions, the function body will be executed.

4. **To avoid dangling pointer cases while deallocating, you can use null pointers**
   Consider the case where you have a pointer that stores the memory address of the variable and contains data. If you wish to delete the data to free the memory what will happen to the pointer? The pointer will remain as it is and will point to the same memory location even after deleting the data. These types of pointers are known as dangling pointers. To circumvent this situation, the best solution is to assign the pointer to NULL.

```
#include<stdio.h>

void main()
{
    int *p = (int *)malloc(SIZE);
    //. . . . . .
    //. . . . . .

    free(p);
    //pointer p is now a dangling pointer

    p=NULL;
    //Now p is a null pointer not a dangling pointer
}
```

The above snippet tells that after the free() method is executed the data in the pointer is freed and the pointer becomes a dangling pointer. When the pointer is set to a null value it changes to a null pointer. You can view this while dealing with data structures like linked lists and trees.

VOID POINTERS

From our previous tutorial we have learned that the data type of a pointer must be equal to the corresponding data type of the variable whose address is stored. For example, an integer pointer must point to an integer variable. But what would happen when the programmer doesn't know the type of variable beforehand.

Void pointers come in handy in such situations. A void pointer, in C programming, also known as **generic pointers** can point to variables of any data type though it does not have any standard data type. Keyword void is used to create a **void** pointer. A void pointer can store addresses of any variable regardless of its data type.

Syntax of Void Pointer is :

```
void * pointer_variable ;
```

Example program of void pointer is given below;

```
#include <stdio.h>
int main()
{
  int x = 10;
  char c = 'C';
  void* vp;

  vp = &x;
  printf("vp stores address of integer variable x:%x\n",vp);
  printf("size of void pointer is : %d\n",sizeof(vp));

  vp = &c;
  printf("vp stores address of character variable c:%x\n",vp);
  printf("size of void pointer is : %d",sizeof(vp));

}
```

Output:

```
vp stores address of integer variable x:61fe14
size of void pointer is : 8

vp stores address of character variable c:61fe13
size of void pointer is : 8
```

When you examine the above code, we have two variables `x` of integer type and c of character type . `vp` is the pointer variable of type void. Hence `vp` has the ability to store addresses of variables irrespective of their datatype. Initially `vp` stores the address of the integer variable `x` and later it stores the address of a variable `c`. Thus `vp` enables the feature of reusability.

Dereferencing a void pointer

In our previous tutorial Pointers in C we have discussed dereferencing of a pointer. Now let see whether it is possible to dereference a void pointer. Observe the below example:

```
#include<stdio.h>
void main()
{
int x = 200;
void* vp ;

vp = &x;
printf("%d", *vp);

}
```

Output:

```
Invalid use of void expression
```

```
#include<stdio.h>
void main()
{
int x = 200;
int* p ;

p = &x;
printf("%d", *p);

}
```

Output:

```
200
```

The above 2 code snippets are a comparison of dereferencing of a void pointer and a typical pointer. From the comparison we could understand that void pointer cannot dereference like a typical pointer. Void pointers must be typecasted to proper data type prior to dereferencing as shown below.

```
#include<stdio.h>
void main()
{
int x = 200;
void* vp ;

vp = &x;
printf("%d", *(int *)vp); //type casting
```

Output:

```
200
```

Here,

- (int *) does the typecasting, where the void pointer `vp` is temporarily changed to an integer pointer and the life of typecasting exits when the evaluation of expression completes.
- *(int *) does the dereferencing of typecast pointers.

Arithmetic pointers with void pointers

C does not support arithmetic pointers with void pointers. The reason behind that is void is not a true type and so sizeof(void) does not have a proper meaning. Since pointer arithmetic changes the pointer value by multiples of sizeof the pointed object, a meaningful size is necessary. Here void fails to provide a correct size and thus is not appropriate for the arithmetic pointer.

WILD POINTERS

While learning null pointers we have come across the uninitialized pointers which are pointers pointing to some arbitrary location and causing a program to behave wrongly or to crash. This type of uninitialized pointers is known as wild pointers in C.

```
int * p; // Wild pointer
```

To convert a wild pointer to a pointer we need to initialize them before use. This can be done in two ways as given in the example

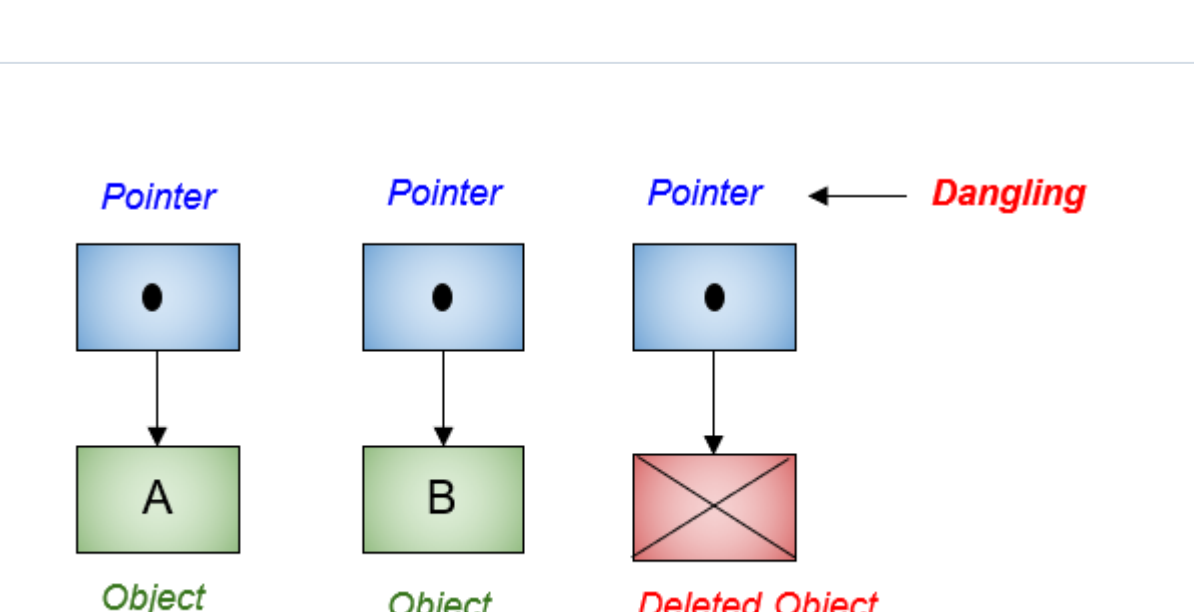
```
#include<stdio.h>

#include

int main()

{

    int * ptr; //wild pointer
    int
    var;

    // Method 1
    var = 100;
    ptr = &
        var; // Now ptr is no longer a wild pointer     printf("\n ptr c *(ptr));

    //Method 2 -creating memory allocation dynamically
    int * p = (int * ) malloc(sizeof(int));
    * p = 100;     printf("\n p c *(p));
    return 0;

}
```

Here ptr and p are two pointer variables that are initialized before their use. Initially, ptr was a wild pointer later on initialization changes to normal pointer as it points to memory location. Yet another way to avoid wild pointer is by dynamically allocating the pointers using calloc , malloc or realloc.

DANGLING POINTERS

The word Dangling means "hanging loosely" and we know pointers are references to memory location. So when a pointer points to an invalid or unreserved memory location it is called a dangling pointer. More precisely, it is a pointer that was active at some point of time in the program execution and currently is not pointing to an object.

Dangling pointers raise at the time of object destruction, specifically when an object is deleted or de-allocated from the memory without modifying the value of the pointer so that the pointer still references the original memory location which is deleted.



The above figure shows pointer A and pointer B points to the memory of allocated objects A and B respectively. On the other hand, the pointer C points to the memory of a deleted object, Hence it is named a dangling pointer.

The use of the dangling pointers can lead to a number of different types of problems, such as:

- Unpredictable behavior
- Segmentation faults / general protection faults
- Bugs due to silent corruption of unrelated data

CAUSE OF DANGLING POINTERS

1. **De- allocating or free variable memory**

```
#include<stdio.h>

#include

int main()

{

    int * ptr;

    //creating memory allocation dynamically
    ptr = (int * ) malloc(sizeof(int));
    printf("\n Memory allocated...");
    * ptr = 100;     printf("\n ptr c *(ptr));

    //Now ptr becomes a dangling pointer
    free(ptr);
    printf("\n Memory is freed ...\n ");     printf("ptr c *(ptr));

    //Now ptr is no longer a dangling pointer
    ptr = NULL

    return 0;

}
```

Output:

```
Memory allocated...  ptr c

Memory is freed  ptr c
```

In this example, after the **free()** function is executed the memory of `ptr` gets deallocated and hence becomes a dangling pointer.

2. **Variable goes out of Scope**

```
#include<stdio.h>

int main()
{
    char **StrPtr;
    {
        char *StrVar = "Hai";
        StrPtr = &StrVar;
    }
    // Since StrVar falls out of scope StrPtr is now a dangling pointer

    printf("%s", *StrPtr);
}
```

In this example, we have initially created a pointer variable `StrPtr`. Then we create another variable, `StrVar` whose visibility is constrained to local block and hence is non-visible in Outer Block. The StrPtr containing the address of `StrVar` becomes a dangling pointer when it comes out of the inner block as StrPtr is still pointing to an invalid memory location in the Outer block.

3. **Returning local variable in Function Call**

```
#include <stdio.h>

int * func() {
    int l = 10;
    return &l;
}
int main() {
    int * p = func();
    printf("%d", * p);
    return 0;
}
```