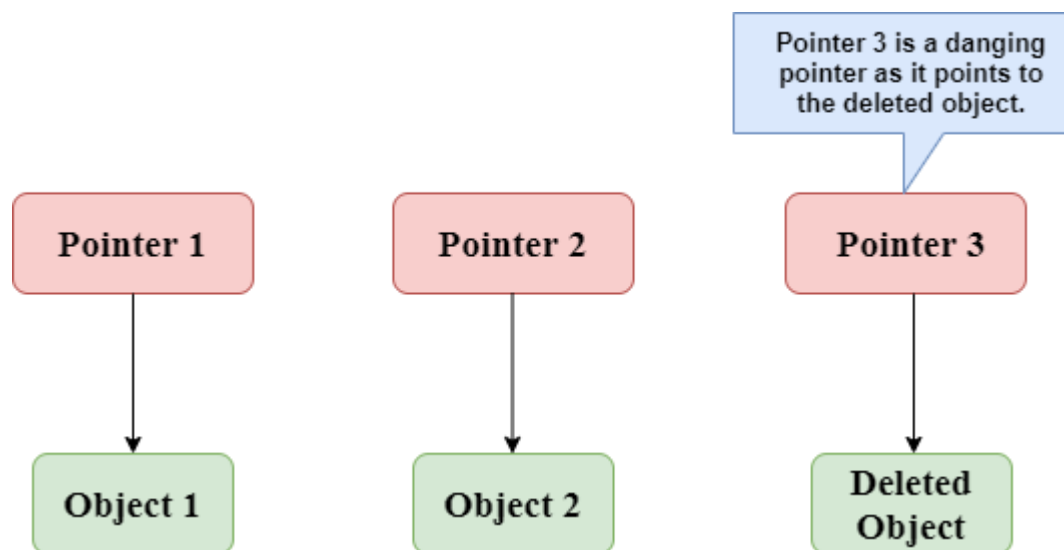


# Dangling Pointers in C

The most common bugs related to pointers and memory management is **dangling/wild pointers**. Sometimes the programmer fails to initialize the pointer with a valid address, then this type of initialized pointer is known as a dangling pointer in C.

Dangling pointer occurs at the time of the object destruction when the object is deleted or de-allocated from memory without modifying the value of the pointer. In this case, the pointer is pointing to the memory, which is de-allocated. The dangling pointer can point to the memory, which contains either the program code or the code of the operating system. If we assign the value to this pointer, then it overwrites the value of the program code or operating system instructions; in such cases, the program will show the undesirable result or may even crash. If the memory is re-allocated to some other process, then we dereference the dangling pointer will cause the segmentation faults.

Let's observe the following examples.



In the above figure, we can observe that the **Pointer 3** is a dangling pointer. **Pointer 1** and **Pointer 2** are the pointers that point to the allocated objects, i.e., Object 1 and Object 2, respectively. **Pointer 3** is a dangling pointer as it points to the de-allocated object.

There are **three** different ways where Pointer acts as dangling pointer

## 1. De-allocation of memory

```
// Deallocating a memory pointed by ptr causes
// dangling pointer
#include <stdlib.h>
#include <stdio.h>
int main()
{
```

```

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

    // After below free call, ptr becomes a
    // dangling pointer
    free(ptr);

    // No more a dangling pointer
    ptr = NULL;
}

```

## OUTPUT

No output

## 2. Variable goes out of the scope

When the variable goes out of the scope then the pointer pointing to the variable becomes a **dangling pointer**.

```

#include<stdio.h>
int main()
{
    char *str;
    {
        char a = 'A';
        str = &a;
    }
    // a falls out of scope
    // str is now a dangling pointer
    printf("%s", *str);
}

```

## OUTPUT

```

main.c:22:14: warning: format '%s' expects argument of type 'char *', but argument 2 has type 'int' [-Wformat=]
Segmentation fault (core dumped)

```

**In the above code, we did the following steps:**

- First, we declare the pointer variable named 'str'.
- In the inner scope, we declare a character variable. The str pointer contains the address of the variable 'a'.
- When the control comes out of the inner scope, 'a' variable will no longer be available, so str points to the de-allocated memory. It means that the str pointer becomes the dangling pointer.

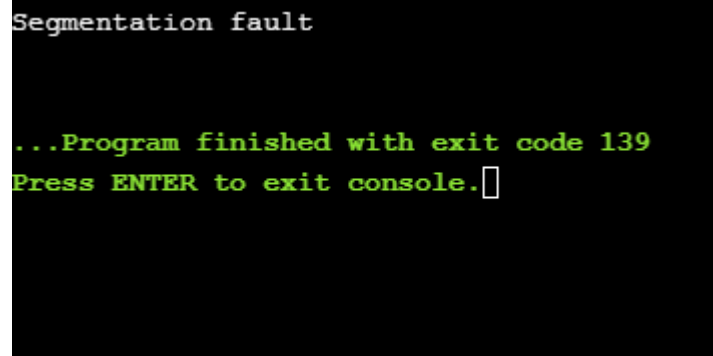
## 3. Function call

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

**Let's understand through an example.**

```
#include <stdio.h>
int *fun(){
    int y=10;
    return &y;
}
int main()
{
    int *p=fun();
    printf("%d", *p);
    return 0;
}
```

OUTPUT



```
Segmentation fault

...Program finished with exit code 139
Press ENTER to exit console.
```

**In the above code, we did the following steps:**

- First, we create the **main()** function in which we have declared '**p**' pointer that contains the return value of the **fun()**.
- When the **fun()** is called, then the control moves to the context of the **int \*fun()**, the **fun()** returns the address of the 'y' variable.
- When control comes back to the context of the **main()** function, the variable 'y' is no longer available as y is local variable and goes out of scope after an execution of fun() is over. Therefore, we can say that the '**p**' pointer is a dangling pointer as it points to the de-allocated memory.

The above problem doesn't appear (or p doesn't become dangling) if y is a static variable.

```
#include <stdio.h>
int *fun(){
    static int y=10;
    return &y;
}
```

```

}
int main()
{
int *p=fun();
printf("%d", *p);
return 0;

}

```

OUTPUT

10

First, the **fun()** function is called, then the control moves to the context of the **int \*fun()**. As 'y' is a static variable, so it stores in the global memory; Its scope is available throughout the program. When the address value is returned, then the control comes back to the context of the **main()**. The pointer 'p' contains the address of 'y', i.e., 100. When we print the value of '\*p', then it prints the value of 'y', i.e., 10. Therefore, we can say that the pointer 'p' is not a dangling pointer as it contains the address of the variable which is stored in the global memory.

### Avoiding Dangling Pointer Errors

The dangling pointer errors can be avoided by initializing the pointer to the **NULL** value. If we assign the **NULL** value to the pointer, then the pointer will not point to the de-allocated memory. Assigning **NULL** value to the pointer means that the pointer is not pointing to any memory location.

## sizeof() operator in C

The **sizeof()** operator is commonly used in C. It determines the size of the expression or the data type specified in the number of char-sized storage units. The **sizeof()** operator contains a single operand which can be either an expression or a data typecast where the cast is data type enclosed within parenthesis. The data type cannot only be primitive data types such as integer or floating data types, but it can also be pointer data types and compound data types such as unions and structs.

### Need of sizeof() operator

Mainly, programs know the storage size of the primitive data types. Though the storage size of the data type is constant, it varies when implemented in different platforms. **sizeof** is greatly used in dynamic memory allocation. For example, if we want to allocate memory for which is sufficient to hold 10 integers and we don't know the **sizeof(int)** in that particular machine. We can allocate with the help of **sizeof**.

```
int* ptr = (int*)malloc(10 * sizeof(int));
```

The **sizeof()** operator behaves differently according to the type of the operand.

- **Operand is a data type**

- **Operand is an expression**

When operand is a data type.

```
#include <stdio.h>
int main() {
    int a = 16;
    printf("Size of variable a : %d\n",sizeof(a));
    printf("Size of int data type : %d\n",sizeof(int));
    printf("Size of char data type : %d\n",sizeof(char));
    printf("Size of float data type : %d\n",sizeof(float));
    printf("Size of double data type : %d\n",sizeof(double));
    return 0;
}
```

OUTPUT

```
Size of variable a : 4
Size of int data type : 4
Size of char data type : 1
Size of float data type : 4
Size of double data type : 8
```

When operand is an expression

```
#include <stdio.h>
int main()
{
    double i=78.0; //variable initialization.
    float j=6.78; //variable initialization.
    printf("size of (i+j) expression is : %d",sizeof(i+j)); //Displaying the size of the expression (
i+j).
    return 0;
}
```

OUTPUT

```
size of (i+j) expression is : 8
```

```
#include <stdio.h>
int main() {
    char a = 'S';
    double b = 4.65;
    printf("Size of variable a : %d\n",sizeof(a));
    printf("Size of an expression : %d\n",sizeof(a+b));
    int s = (int) (a+b);
    printf("Size of explicitly converted expression : %d\n",sizeof(s));
    return 0;
}
```

OUTPUT

```
Size of variable a : 1
Size of an expression : 8
Size of explicitly converted expression : 4
```

## void pointer in C

Till now, we have studied that the address assigned to a pointer should be of the same type as specified in the pointer declaration. For example, if we declare the int pointer, then this int pointer cannot point to the float variable or some other type of variable, i.e., it can point to only int type variable. To overcome this problem, we use a pointer to void. A pointer to void means a generic pointer that can point to any data type. We can assign the address of any data type to the void pointer, and a void pointer can be assigned to any type of the pointer without performing any explicit typecasting.

#### Syntax of void pointer

```
void *pointer name;
```

**Declaration of the void pointer is given below:**

```
void *ptr;
```

**Let us consider some examples:**

```
int i=9;    // integer variable initialization.
```

```
int *p;     // integer pointer declaration.
```

```
float *fp;  // floating pointer declaration.
```

```
void *ptr;  // void pointer declaration.
```

```
p=fp;      // incorrect.
```

```
fp=&i;      // incorrect
```

```
ptr=p;     // correct
```

```
ptr=fp;    // correct
```

```
ptr=&i;     // correct
```

#### Size of the void pointer in C

The size of the void pointer in C is the same as the size of the pointer of character type. According to C perception, the representation of a pointer to void is the same as the pointer of character type. The size of the pointer will vary depending on the platform that you are using.

**Let's look at the below example:**

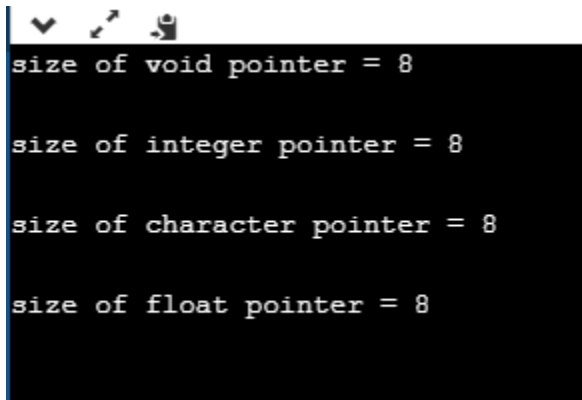
```
#include <stdio.h>
int main()
{
```

```

void *ptr = NULL; //void pointer
int *p = NULL; // integer pointer
char *cp = NULL; //character pointer
float *fp = NULL; //float pointer
//size of void pointer
printf("size of void pointer = %d\n\n", sizeof(ptr));
//size of integer pointer
printf("size of integer pointer = %d\n\n", sizeof(p));
//size of character pointer
printf("size of character pointer = %d\n\n", sizeof(cp));
//size of float pointer
printf("size of float pointer = %d\n\n", sizeof(fp));
return 0;
}

```

## OUTPUT



```

size of void pointer = 8

size of integer pointer = 8

size of character pointer = 8

size of float pointer = 8

```

## Advantages of void pointer

### Following are the advantages of a void pointer:

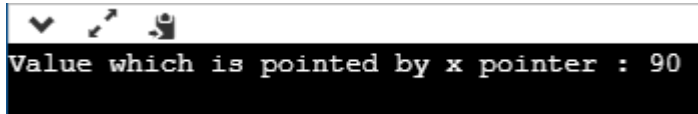
- The malloc() and calloc() function return the void pointer, so these functions can be used to allocate the memory of any data type.

```

#include <stdio.h>
#include <malloc.h>
int main()
{
int a=90;
int *x = (int*)malloc(sizeof(int)) ;
x=&a;
printf("Value which is pointed by x pointer : %d",*x);
return 0;
}

```

## OUTPUT



```
Value which is pointed by x pointer : 90
```

- The void pointer in C can also be used to implement the generic functions in C.

### Some important points related to void pointer are:

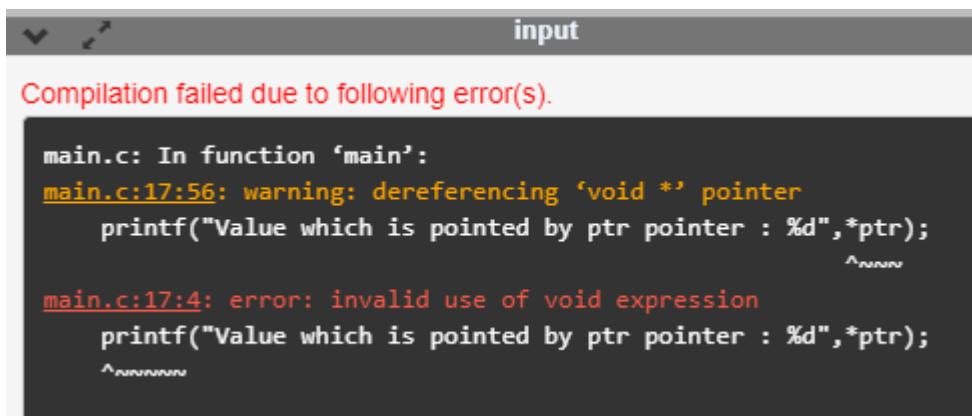
- **Dereferencing a void pointer in C**

The void pointer in C cannot be dereferenced directly. Let's see the below example.

```
#include <stdio.h>
int main()
{
    int a=90;
    void *ptr;
    ptr=&a;
    printf("Value which is pointed by ptr pointer : %d",*ptr);
    return 0;
}
```

In the above code, \*ptr is a void pointer which is pointing to the integer variable 'a'. As we already know that the void pointer cannot be dereferenced, so the above code will give the compile-time error because we are printing the value of the variable pointed by the pointer 'ptr' directly.

## OUTPUT



```
input
Compilation failed due to following error(s).

main.c: In function 'main':
main.c:17:56: warning: dereferencing 'void *' pointer
    printf("Value which is pointed by ptr pointer : %d",*ptr);
                                                         ^~~~~~
main.c:17:4: error: invalid use of void expression
    printf("Value which is pointed by ptr pointer : %d",*ptr);
    ^~~~~~
```

Now, we rewrite the above code to remove the error.

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

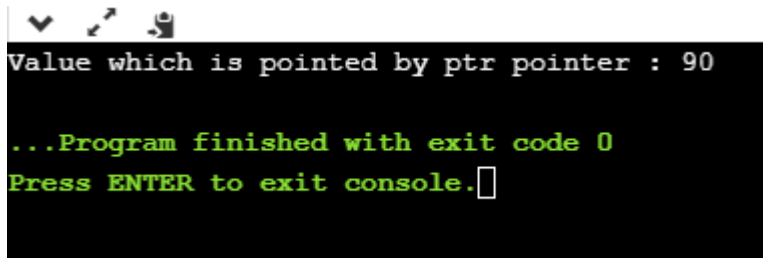


```

int main()
{
    int a=90;
    void *ptr;
    ptr=&a;
    printf("Value which is pointed by ptr pointer : %d",*(int*)ptr);
    return 0;
}

```

OUTPUT



```

Value which is pointed by ptr pointer : 90

...Program finished with exit code 0
Press ENTER to exit console.

```

In the above code, we typecast the void pointer to the integer pointer by using the statement given below:

**(int\*)ptr;**

Then, we print the value of the variable which is pointed by the void pointer 'ptr' by using the statement given below:

**\*(int\*)ptr;**

### Arithmetic operation on void pointers

We cannot apply the arithmetic operations on void pointers in C directly. We need to apply the proper typecasting so that we can perform the arithmetic operations on the void pointers.

**Let's see the below example:**

```

#include<stdio.h>
int main()
{
    float a[4]={ 6.1,2.3,7.8,9.0};
    void *ptr;
    ptr=a;
    for(int i=0;i<4;i++)
    {
        printf("%f,",*ptr);
        ptr=ptr+1;    // Incorrect.
    }
}

```

```
}}
```

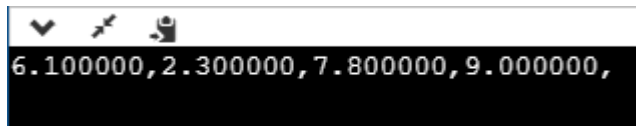
The above code shows the compile-time error that **"invalid use of void expression"** as we cannot apply the arithmetic operations on void pointer directly, i.e., `ptr=ptr+1`.

**Let's rewrite the above code to remove the error.**

```
#include<stdio.h>
int main()
{
    float a[4]={ 6.1,2.3,7.8,9.0};
    void *ptr;
    ptr=a;
    for(int i=0;i<4;i++)
    {
        printf("%f,",*((float*)ptr+i));
    }
}
```

The above code runs successfully as we applied the proper casting to the void pointer, i.e., `(float*)ptr` and then we apply the arithmetic operation, i.e., `*((float*)ptr+i)`.

OUTPUT



```
6.100000, 2.300000, 7.800000, 9.000000,
```

Why we use void pointers?

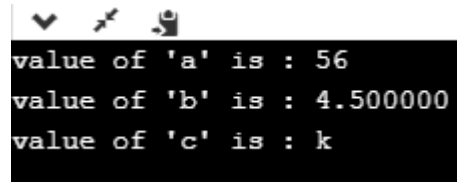
We use void pointers because of its reusability. Void pointers can store the object of any type, and we can retrieve the object of any type by using the indirection operator with proper typecasting.

**Let's understand through an example.**

```
#include<stdio.h>
int main()
{
    int a=56; // initialization of a integer variable 'a'.
    float b=4.5; // initialization of a float variable 'b'.
    char c='k'; // initialization of a char variable 'c'.
    void *ptr; // declaration of void pointer.
    // assigning the address of variable 'a'.
    ptr=&a;
    printf("value of 'a' is : %d",*((int*)ptr));
}
```

```
// assigning the address of variable 'b'.
ptr=&b;
printf("\nvalue of 'b' is : %f",*((float*)ptr));
// assigning the address of variable 'c'.
ptr=&c;
printf("\nvalue of 'c' is : %c",*((char*)ptr));
return 0;
}
```

## OUTPUT

A terminal window with a black background and white text. At the top left, there are three small icons: a downward arrow, a cursor, and a document. The output text is as follows:

```
value of 'a' is : 56
value of 'b' is : 4.500000
value of 'c' is : k
```