

# 2.Pointer

## What is a Pointer in C?

*A pointer is defined as a derived data type that can store the address of other C variables or a memory location. We can access and manipulate the data stored in that memory location using pointers.*

## Syntax of C Pointers

The syntax of pointers is similar to the variable declaration in C, but we use the ( \* ) **dereferencing operator** in the pointer declaration.

```
datatype * ptr;
```

where

- **ptr** is the name of the pointer.
- **datatype** is the type of data it is pointing to.

## How to Use Pointers?

The use of pointers in C can be divided into three steps:

1. Pointer Declaration
  2. Pointer Initialization
  3. Pointer Dereferencing
1. Pointer Declaration

In pointer declaration, we only declare the pointer but do not initialize it. To declare a pointer, we use the ( \* ) **dereference operator** before its name.

### Example

```
int *ptr;
```

The pointer declared here will point to some random memory address as it is not initialized. Such pointers are called wild pointers.

### 2. Pointer Initialization

Pointer initialization is the process where we assign some initial value to the pointer variable. We generally use the ( & ) **addressof operator** to get the memory address of a variable and then store it in the pointer variable.

### Example

```
int var = 10;  
int * ptr;  
ptr = &var;
```

We can also declare and initialize the pointer in a single step. This method is called **pointer definition** as the pointer is declared and initialized at the same time.

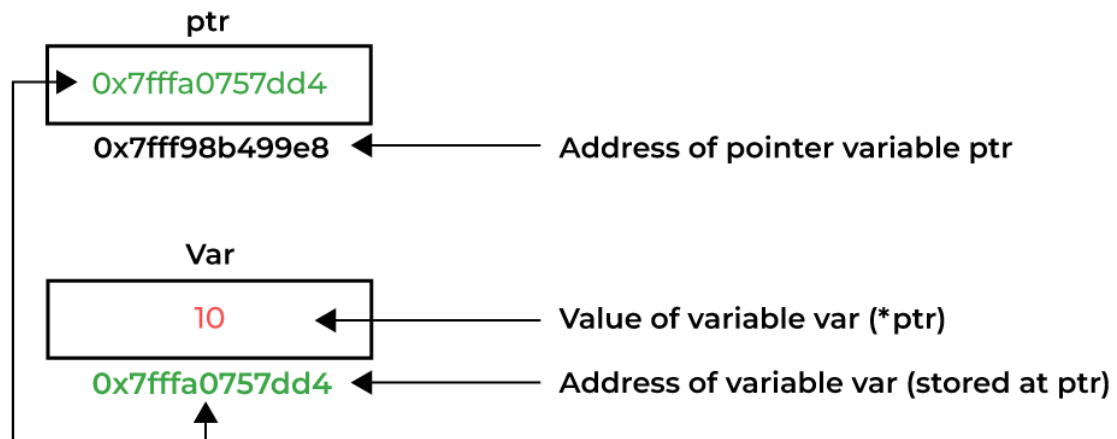
### Example

```
int *ptr = &var;
```

**Note:** It is recommended that the pointers should always be initialized to some value before starting using it. Otherwise, it may lead to number of errors.

### 3. Pointer Dereferencing

Dereferencing a pointer is the process of accessing the value stored in the memory address specified in the pointer. We use the same ( \* ) **dereferencing operator** that we used in the pointer declaration.



## C Pointer Example

// C program to illustrate Pointers

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

```
void geeks()
```

```
{
```

```
    int var = 10;
```

```
    // declare pointer variable
```

```
    int* ptr;
```

```
    // note that data type of ptr and var must be same
```

```
    ptr = &var;
```

```
    // assign the address of a variable to a pointer
```

```
    printf("Value at ptr = %p \n", ptr);
```

```

printf("Value at var = %d \n", var);

printf("Value at *ptr = %d \n", *ptr);

}

```

// Driver program

```

int main()
{
    geeks();

    return 0;
}

```

### Output

```

Value at ptr = 0x7fff1038675c
Value at var = 10
Value at *ptr = 10

```

## Types of Pointers in C

### Double Pointers

In C language, we can define a pointer that stores the memory address of another pointer. Such pointers are called double-pointers or [pointers-to-pointer](#). Instead of pointing to a data value, they point to another pointer.

#### Syntax

```
datatype ** pointer_name;
```

#### Dereferencing Double Pointer

```
*pointer_name; // get the address stored in the inner level pointer
**pointer_name; // get the value pointed by inner level pointer
```

**Note:** In C, we can create [multi-level pointers](#) with any number of levels such as – `***ptr3`, `****ptr4`, `*****ptr5` and so on.

### NULL Pointer

The [Null Pointers](#) are those pointers that do not point to any memory location. They can be created by assigning a NULL value to the pointer. A pointer of any type can be assigned the NULL value.

#### Syntax

```
data_type *pointer_name = NULL;
or
```

```
pointer_name = NULL
```

It is said to be good practice to assign NULL to the pointers currently not in use.

## Void Pointer

The [Void pointers](#) in C are the pointers of type void. It means that they do not have any associated data type. They are also called **generic pointers** as they can point to any type and can be typecasted to any type.

### Syntax

```
void * pointer_name;
```

One of the main properties of void pointers is that they cannot be dereferenced.

## Wild Pointers

The [Wild Pointers](#) are pointers that have not been initialized with something yet. These types of C-pointers can cause problems in our programs and can eventually cause them to crash.

### Example

```
int *ptr;  
char *str;
```

## Constant Pointers

In constant pointers, the memory address stored inside the pointer is constant and cannot be modified once it is defined. It will always point to the same memory address.

### Syntax

```
data_type * const pointer_name;
```

## Pointer to Constant

The pointers pointing to a constant value that cannot be modified are called pointers to a constant. Here we can only access the data pointed by the pointer, but cannot modify it. Although, we can change the address stored in the pointer to constant.

### Syntax

```
const data_type * pointer_name;
```

## Multiple Indirection:

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int –

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example –

[Live Demo](#)

```
#include <stdio.h>

int main () {

    int  var;
    int  *ptr;
    int  **pptr;

    var = 3000;

    /* take the address of var */
    ptr = &var;

    /* take the address of ptr using address of operator
    & */
    pptr = &ptr;

    /* take the value using pptr */
    printf("Value of var = %d\n", var );
    printf("Value available at *ptr = %d\n", *ptr );
    printf("Value available at **pptr = %d\n", **pptr);

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Value of var = 3000

Value available at \*ptr = 3000

Value available at \*\*pptr = 3000

**Parameter passing:**

## Call by value in C

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we can not modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.
- `#include<stdio.h>`
- `void change(int num) {`
- `printf("Before adding value inside function num=%d \n",num);`
- `num=num+100;`
- `printf("After adding value inside function num=%d \n", num);`
- `}`
- `int main() {`
- `int x=100;`
- `printf("Before function call x=%d \n", x);`
- `change(x);//passing value in function`
- `printf("After function call x=%d \n", x);`
- `return 0;`
- `}`

### Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

### Call by Value Example: Swapping the values of the two variables

1. `#include <stdio.h>`
2. `void swap(int , int); //prototype of the function`
3. `int main()`
4. `{`
5. `int a = 10;`
6. `int b = 20;`
7. `printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main`
8. `swap(a,b);`

```

9.     printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual param
    eters do not change by changing the formal parameters in call by value, a = 10, b = 20
10. }
11. void swap (int a, int b)
12. {
13.     int temp;
14.     temp = a;
15.     a=b;
16.     b=temp;
17.     printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a =
    20, b = 10
18. }

```

### Output

```

Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 10, b = 20

```

### Call by Value Example: Swapping the values of the two variables

```

1. #include <stdio.h>
2. void swap(int , int); //prototype of the function
3. int main()
4. {
5.     int a = 10;
6.     int b = 20;
7.     printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of
    a and b in main
8.     swap(a,b);
9.     printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual param
    eters do not change by changing the formal parameters in call by value, a = 10, b = 20
10. }
11. void swap (int a, int b)
12. {
13.     int temp;
14.     temp = a;
15.     a=b;
16.     b=temp;
17.     printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a =
    20, b = 10
18. }

```

### Output

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 10, b = 20
```

## Call by reference in C

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

### Consider the following example for the call by reference.

```
1. #include<stdio.h>
2. void change(int *num) {
3.     printf("Before adding value inside function num=%d \n",*num);
4.     (*num) += 100;
5.     printf("After adding value inside function num=%d \n", *num);
6. }
7. int main() {
8.     int x=100;
9.     printf("Before function call x=%d \n", x);
10.    change(&x); //passing reference in function
11.    printf("After function call x=%d \n", x);
12.    return 0;
13.}
```

### Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200
```

### Call by reference Example: Swapping the values of the two variables

```
1. #include <stdio.h>
2. void swap(int *, int *); //prototype of the function
```



```

3. int main()
4. {
5.     int a = 10;
6.     int b = 20;
7.     printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of
    a and b in main
8.     swap(&a,&b);
9.     printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual para
    meters do change in call by reference, a = 10, b = 20
10. }
11. void swap (int *a, int *b)
12. {
13.     int temp;
14.     temp = *a;
15.     *a=*b;
16.     *b=temp;
17.     printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Formal parameters, a
    = 20, b = 10
18. }

```

### Output

```

Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 20, b = 10

```

## Difference between call by value and call by reference in c

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function value is changed outside of the function also. The values of actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

**Pointers** are variables which stores the address of another variable. When we allocate memory to a variable, pointer points to the address of the variable. Unary operator (\*) is used to declare a variable and it returns the address of the allocated memory. Pointers to an array points the address of memory block of an array variable.

The following is the syntax of array pointers.

```
datatype *variable_name[size];
```

Here,

**datatype** – The datatype of variable like *int*, *char*, *float*, etc.

**variable\_name** – This is the name of variable given by user.

**size** – The size of array variable.

The following is an example of array pointers.

## Example

[Live Demo](#)

```
#include <stdio.h>
int main () {
    int *arr[3];
    int *a;
    printf( "Value of array pointer variable : %d
", arr);
    printf( "Value of pointer variable : %d
", &a);
    return 0;
}
```

## Output

Value of array pointer variable : 1481173888

Value of pointer variable : 1481173880

In the above program, an array pointer *\*arr* and an integer *\*a* are declared.

```
int *arr[3];
int *a;
```

The addresses of these pointers are printed as follows –

```
printf( "Value of array pointer variable : %d\n", arr);
printf( "Value of pointer variable : %d\n", &a);
```

# Array of Pointers in C

- 

In C, a pointer array is a homogeneous collection of indexed pointer variables that are references to a memory location. It is generally used in C Programming when we want to point at multiple memory locations of a similar data type in our C program. We can access the data by dereferencing the pointer pointing to it.

### Syntax:

```
pointer_type *array_name [array_size];
```

Here,

- **pointer\_type:** Type of data the pointer is pointing to.
- **array\_name:** Name of the array of pointers.
- **array\_size:** Size of the array of pointers.

**Note:** It is important to keep in mind the operator precedence and associativity in the array of pointers declarations of different type as a single change will mean the whole different thing. For example, enclosing `*array_name` in the parenthesis will mean that `array_name` is a pointer to an array.

**Example:**

- C

```
// C program to demonstrate the use of array of pointers

#include <stdio.h>

int main()
```

```
{

    // declaring some temp variables

    int var1 = 10;

    int var2 = 20;

    int var3 = 30;


    // array of pointers to integers

    int* ptr_arr[3] = { &var1, &var2, &var3 };


    // traversing using loop

    for (int i = 0; i < 3; i++) {

        printf("Value of var%d: %d\tAddress: %p\n", i + 1, *ptr_arr[i],
ptr_arr[i]);

    }

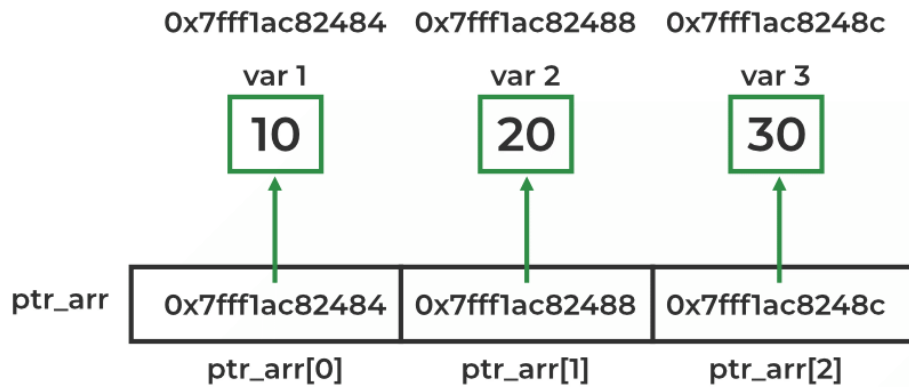

    return 0;

}
```

### Output

Value of var1: 10	Address: 0x7fff1ac82484
Value of var2: 20	Address: 0x7fff1ac82488
Value of var3: 30	Address: 0x7fff1ac8248c

### Explanation:



As shown in the above example, each element of the array is a pointer pointing to an integer. We can access the value of these integers by first selecting the array element and then dereferencing it to get the value.

## Array of Pointers to Character

One of the main applications of the array of pointers is to store multiple strings as an array of pointers to characters. Here, each pointer in the array is a character pointer that points to the first character of the string.

### Syntax:

```
char *array_name [array_size];
```

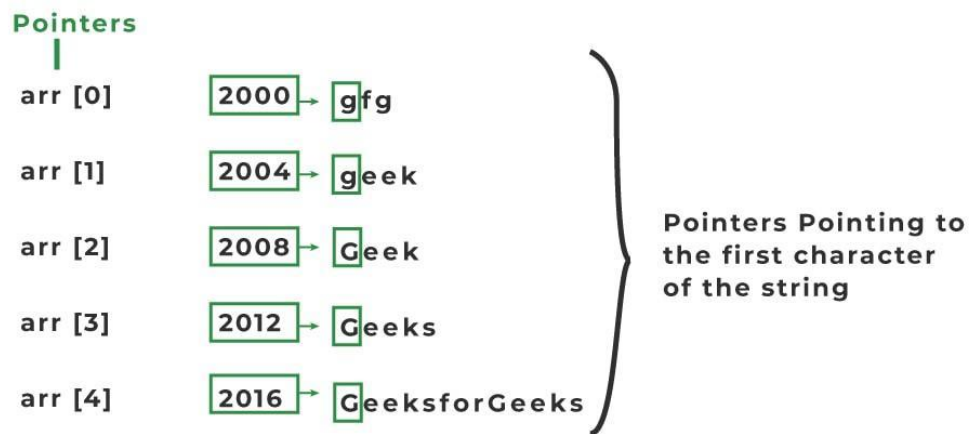
After that, we can assign a string of any length to these pointers.

### Example:

- C

```
char* arr[5]

= { "gfg", "geek", "Geek", "Geeks", "GeeksforGeeks" }
```



This method of storing strings has the advantage of the traditional array of strings. Consider the following two examples:

#### Example 1:

- C

```
// C Program to print Array of strings without array of pointers

#include <stdio.h>

int main()

{

    char str[3][10] = { "Geek", "Geeks", "Geekfor" };

    printf("String array Elements are:\n");

    for (int i = 0; i < 3; i++) {

        printf("%s\n", str[i]);

    }

}
```

```
    }

    return 0;

}
```

## Output

String array Elements are:

Geek

Geeks

Geekfor

In the above program, we have declared the 3 rows and 10 columns of our array of strings. But because of predefining the size of the array of strings the space consumption of the program increases if the memory is not utilized properly or left unused. Now let's try to store the same strings in an array of pointers.

## Example 2:

- C

```
// C program to illustrate the use of array of pointers to
// characters

#include <stdio.h>

int main()

{

    char* arr[3] = { "geek", "Geeks", "Geeksfor" };

}
```

```

for (int i = 0; i < 3; i++) {

    printf("%s\n", arr[i]);

}

return 0;

}

```

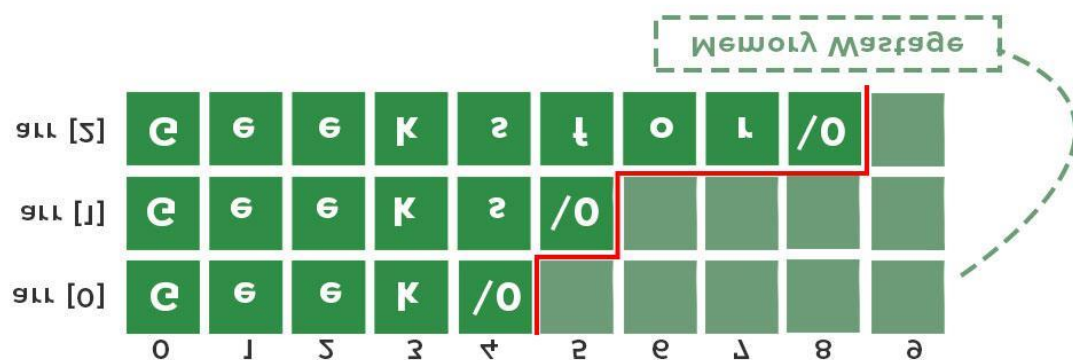
## Output

geek

Geeks

Geeksfor

Here, the total memory used is the memory required for storing the strings and pointers without leaving any empty space hence, saving a lot of wasted space. We can understand this using the image shown below.



## Memory Representation of an Array of Strings

The space occupied by the array of pointers to characters is shown by solid green blocks excluding the memory required for storing the pointer while the space occupied by the array of strings includes both solid and light green blocks.



## Array of Pointers to Different Types

Not only we can define the array of pointers for basic data types like int, char, float, etc. but we can also define them for derived and user-defined data types such as arrays, structures, etc. Let's consider the below example where we create an array of pointers pointing to a function for performing the different operations.

### Example:

- C

```
// C program to illustrate the use of array of pointers to

// function

#include <stdio.h>


// some basic arithmetic operations

void add(int a, int b) {

    printf("Sum : %d\n", a + b);

}


void subtract(int a, int b) {

    printf("Difference : %d\n", a - b);

}


void multiply(int a, int b) {

    printf("Product : %d\n", a * b);
```

```

}

void divide(int a, int b) {

    printf("Quotient : %d", a / b);

}

int main() {

    int x = 50, y = 5;

    // array of pointers to function of return type int

    void (*arr[4])(int, int)

        = { &add, &subtract, &multiply, ÷ };

    for (int i = 0; i < 4; i++) {

        arr[i](x, y);

    }

    return 0;

}

```

## Output

Sum : 55

Difference : 45

Product : 250

Quotient : 10

## Application of Array of Pointers

An array of pointers is useful in a wide range of cases. Some of these applications are listed below:

- It is most commonly used to store multiple strings.
- It is also used to implement LinkedHashMap in C and also in the Chaining technique of collision resolving in Hashing.
- It is used in sorting algorithms like bucket sort.
- It can be used with any pointer type so it is useful when we have separate declarations of multiple entities and we want to store them in a single place.

## Disadvantages of Array of Pointers

The array of pointers also has its fair share of disadvantages and should be used when the advantages outweigh the disadvantages. Some of the disadvantages of the array of pointers are:

- **Higher Memory Consumption:** An array of pointers requires more memory as compared to plain arrays because of the additional space required to store pointers.
- **Complexity:** An array of pointers might be complex to use as compared to a simple array.
- **Prone to Bugs:** As we use pointers, all the bugs associated with pointers come with it so we need to handle them carefully.

## Swap values by passing pointers

The following function receives the reference of two variables whose values are to be swapped.

```
/* function definition to swap the values */
int swap(int *x, int *y){
    int z;
    z = *x; /* save the value at address x */
    *x = *y; /* put y into x */
    *y = z; /* put z into y */

    return 0;
}
```

The main() function has two variables a and b, their addresses are passed as arguments to swap() function.

## Example

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

```

int swap(int *x, int *y){
    int z;
    z = *x;
    *x = *y;
    *y = z;
}
int main () {

    /* local variable definition */
    int a = 10;
    int b = 20;
    printf("Before swap, value of a : %d\n", a );
    printf("Before swap, value of b : %d\n", b );

    /* calling a function to swap the values */
    swap(&a, &b);
    printf("After swap, value of a : %d\n", a );
    printf("After swap, value of b : %d\n", b );

    return 0;
}

```

## Output

The result of the program when executed is –

```

Before swap, value of a : 10
Before swap, value of b : 20
After swap, value of a : 20
After swap, value of b : 10

```

## Pass array pointer to a function

In C, the name of array is the address of the first element of the array, or in other words, the pointer to array. In the following example, we declare an uninitialized array in main() and pass its pointer to a function, along with an integer. Inside the function, the array is filled with the square, cube and square root. The function returns the pointer of this array, using which the values are access and printed in main() function.

## Example

```
#include <stdio.h>
#include <math.h>
int arrfunction(int, float *);
int main(){
    int x=100;
    float arr[3];
    arrfunction(x, arr);
    printf("Square of %d: %f\n", x, arr[0]);
    printf("cube of %d: %f\n", x, arr[1]);
    printf("Square root of %d: %f\n", x, arr[2]);

    return 0;
}
int arrfunction(int x, float *arr){
    arr[0]=pow(x,2);
    arr[1]=pow(x, 3);
    arr[2]=pow(x, 0.5);
}
```

## Output

Square of 100: 10000.000000  
cube of 100: 1000000.000000  
Square root of 100: 10.000000

## Pass string pointers to Function to compare lengths

Let us have a look at another example, where pointers are passed to a function. In the following program, two strings are passed to compare() functions. In C, as string is an array of char data type. We use strlen() function to find the length of string which is the number of characters in it.

## Example

```
#include <stdio.h>
#include <string.h>
int compare( char *, char *);
int main() {
    char a[] = "BAT";
```

```

char b[] = "BALL";
int ret = compare(a, b);
return 0;
}
int compare (char *x, char *y){
    int val;
    if (strlen(x)>strlen(y)){
        printf("length of string a is greater than or equal to length of string b");
    }
    else{
        printf("length of string a is less than length of string b");
    }
}
}

```

## Output

length of string a is less than length of string b

## Returning pointer from function

[Pointers](#) in [C programming language](#) is a variable which is used to store the memory address of another variable. We can pass pointers to the function as well as return pointer from a function. But it is not recommended to return the address of a local variable outside the function as it goes out of scope after function returns.

```

// C program to illustrate the concept of

// returning pointer from a function

#include <stdio.h>


// Function that returns pointer

int* fun()

{

```

```
// Declare a static integer

static int A = 10;

return (&A);

}


// Driver Code

int main()

{

    // Declare a pointer

    int* p;


    // Function call

    p = fun();


    // Print Address

    printf("%p\n", p);


    // Print value at the above address

    printf("%d\n", *p);

    return 0;

}
```

**Output:**

0x601038

## Function Pointer:

# C Function Pointer

As we know that we can create a pointer of any data type such as int, char, float, we can also create a pointer pointing to a function. The code of a function always resides in memory, which means that the function has some address. We can get the address of memory by using the function pointer.

Let's see a simple example.

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `printf("Address of main() function is %p",main);`
5. `return 0;`
6. `}`

The above code prints the address of **main()** function.

### Output

```
Backward Skip 10sPlay VideoForward Skip 10s
✓ ↗ 📄
Address of main() function is 0x400536
...Program finished with exit code 0
Press ENTER to exit console.□
```

In the above output, we observe that the `main()` function has some address. Therefore, we conclude that every function has some address.

## Declaration of a function pointer

Till now, we have seen that the functions have addresses, so we can create pointers that can contain these addresses, and hence can point them.



## Syntax of function pointer

1. **return** type (\*ptr\_name)(type1, type2...);

For example:

1. **int** (\*ip) (**int**);

In the above declaration, \*ip is a pointer that points to a function which returns an int value and accepts an integer value as an argument.

1. **float** (\*fp) (**float**);

In the above declaration, \*fp is a pointer that points to a function that returns a float value and accepts a float value as an argument.

We can observe that the declaration of a function is similar to the declaration of a function pointer except that the pointer is preceded by a '\*'. So, in the above declaration, fp is declared as a function rather than a pointer.

Till now, we have learnt how to declare the function pointer. Our next step is to assign the address of a function to the function pointer.

1. **float** (\*fp) (**int** , **int**); // Declaration of a function pointer.
2. **float** func( **int** , **int** ); // Declaration of function.
3. fp = func; // Assigning address of func to the fp pointer.

In the above declaration, '**fp**' pointer contains the address of the '**func**' function.

**Note: Declaration of a function is necessary before assigning the address of a function to the function pointer.**

## Calling a function through a function pointer

We already know how to call a function in the usual way. Now, we will see how to call a function using a function pointer.

Suppose we declare a function as given below:

1. **float** func(**int** , **int**); // Declaration of a function.

Calling an above function using a usual way is given below:

1. result = func(a , b); // Calling a function using usual ways.

Calling a function using a function pointer is given below:

1. `result = (*fp)( a , b);` // Calling a function using function pointer.

Or

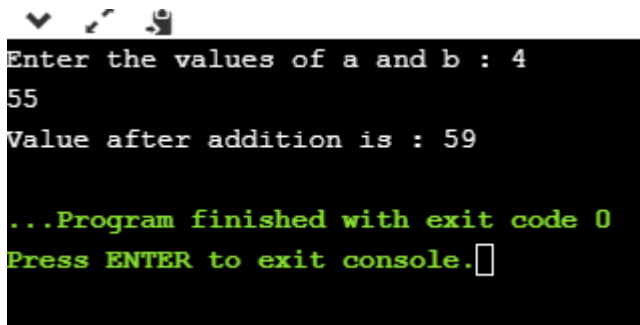
1. `result = fp(a , b);` // Calling a function using function pointer, and indirection operator can be removed.

The effect of calling a function by its name or function pointer is the same. If we are using the function pointer, we can omit the indirection operator as we did in the second case. Still, we use the indirection operator as it makes it clear to the user that we are using a function pointer.

**Let's understand the function pointer through an example.**

```
1. #include <stdio.h>
2. int add(int,int);
3. int main()
4. {
5.     int a,b;
6.     int (*ip)(int,int);
7.     int result;
8.     printf("Enter the values of a and b : ");
9.     scanf("%d %d",&a,&b);
10.    ip=add;
11.    result=(*ip)(a,b);
12.    printf("Value after addition is : %d",result);
13.    return 0;
14. }
15. int add(int a,int b)
16. {
17.     int c=a+b;
18.     return c;
19. }
```

**Output**



```
Enter the values of a and b : 4
55
Value after addition is : 59

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

## Pointer to Constant

A pointer to constant is a pointer through which the value of the variable that the pointer points cannot be changed. The address of these pointers can be changed, but the value of the variable that the pointer points cannot be changed.

### Syntax of Pointer to Constant

1. **const** <type of pointer>\* <name of pointer>

**Declaration of a pointer to constant is given below:**

1. **const int\*** ptr;

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

- **First, we write the code where we are changing the value of a pointer**

1. **#include <stdio.h>**
2. **int** main()
3. {
4.   **int** a=100;
5.   **int** b=200;
6.   **const int\*** ptr;
7.   ptr=&a;
8.   ptr=&b;
9.   printf("Value of ptr is :%u",ptr);
10. **return** 0;
11. }

**In the above code:**

- We declare two variables, i.e., a and b with the values 100 and 200 respectively.
- We declare a pointer to constant.

- First, we assign the address of variable 'a' to the pointer 'ptr'.
- Then, we assign the address of variable 'b' to the pointer 'ptr'.
- Lastly, we try to print the value of 'ptr'.

## Output

```
Value of ptr is :247760772
```

The above code runs successfully, and it shows the value of 'ptr' in the output.

- Now, we write the code in which we are changing the value of the variable to which the pointer points.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a=100;
5.     int b=200;
6.     const int* ptr;
7.     ptr=&b;
8.     *ptr=300;
9.     printf("Value of ptr is :%d",*ptr);
10.    return 0;
11. }
```

## In the above code:

- We declare two variables, i.e., 'a' and 'b' with the values 100 and 200 respectively.
- We declare a pointer to constant.
- We assign the address of the variable 'b' to the pointer 'ptr'.
- Then, we try to modify the value of the variable 'b' through the pointer 'ptr'.
- Lastly, we try to print the value of the variable which is pointed by the pointer 'ptr'.

## Output

```
main.c: In function 'main':
main.c:17:9: error: assignment of read-only location '*ptr'
    *ptr=300;
    ^
```

The above code shows the error "assignment of read-only location '\*ptr'". This error means that we cannot change the value of the variable to which the pointer is pointing.

## Constant Pointer to a Constant

A constant pointer to a constant is a pointer, which is a combination of the above two pointers. It can neither change the address of the variable to which it is pointing nor it can change the value placed at this address.

### Syntax

1. **const** <type of pointer> \* **const** <name of the pointer>;

**Declaration for a constant pointer to a constant is given below:**

1. **const int\* const** ptr;

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

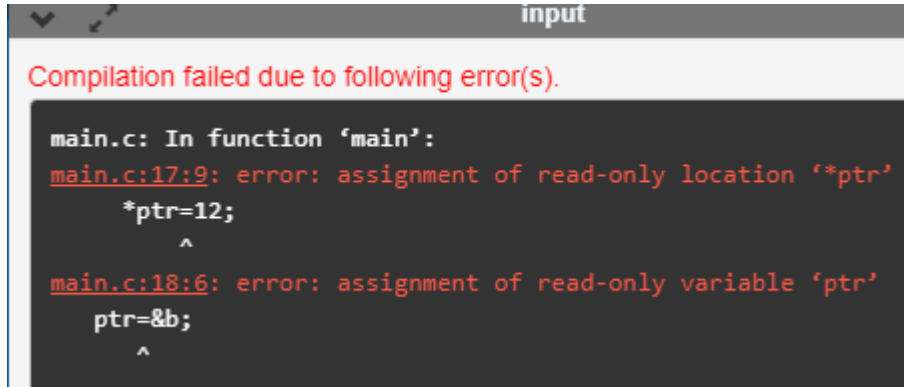
1. **#include <stdio.h>**
2. **int** main()
3. {
4.     **int** a=10;
5.     **int** b=90;
6.     **const int\* const** ptr=&a;
7.     \*ptr=12;
8.     ptr=&b;
9.     printf("Value of ptr is :%d",\*ptr);
10.    **return** 0;
11. }

**In the above code:**

- We declare two variables, i.e., 'a' and 'b' with the values 10 and 90, respectively.
- We declare a constant pointer to a constant and then assign the address of 'a'.
- We try to change the value of the variable 'a' through the pointer 'ptr'.

- Then we try to assign the address of variable 'b' to the pointer 'ptr'.
- Lastly, we print the value of the variable, which is pointed by the pointer 'ptr'.

## Output



```

input
Compilation failed due to following error(s).

main.c: In function 'main':
main.c:17:9: error: assignment of read-only location '*ptr'
    *ptr=12;
    ^
main.c:18:6: error: assignment of read-only variable 'ptr'
    ptr=&b;
    ^
  
```

The above code shows the error "assignment of read-only location '\*ptr'" and "assignment of read-only variable 'ptr'". Therefore, we conclude that the constant pointer to a constant can change neither address nor value, which is pointing by this pointer.

## const Pointer in C

### Constant Pointers

A constant pointer in C cannot change the address of the variable to which it is pointing, i.e., the address will remain constant. Therefore, we can say that if a constant pointer is pointing to some variable, then it cannot point to any other variable.

### Syntax of Constant Pointer

1. <type of pointer> \***const** <name of pointer>;

**Declaration of a constant pointer is given below:**

1. **int** \***const** ptr;

**Let's understand the constant pointer through an example.**

1. **#include** <stdio.h>
2. **int** main()
3. {
4.     **int** a=1;
5.     **int** b=2;

```
6.  int *const ptr;
7.  ptr=&a;
8.  ptr=&b;
9.  printf("Value of ptr is :%d",*ptr);
10. return 0;
11. }
```

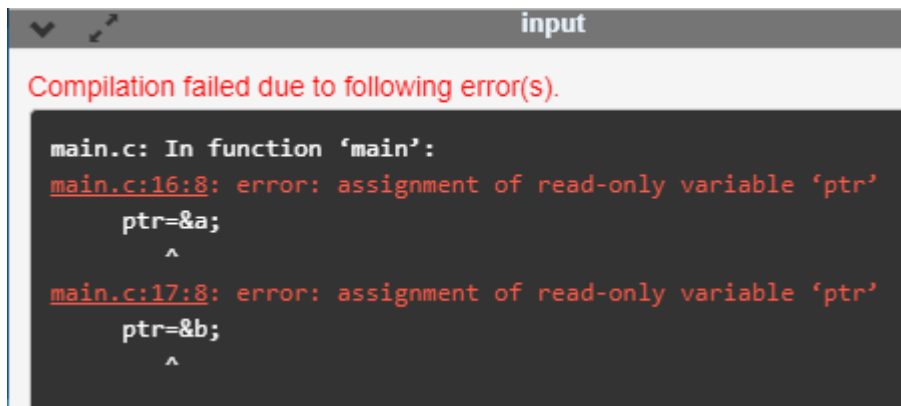
**In the above code:**

ADVERTISEMENT

ADVERTISEMENT

- We declare two variables, i.e., a and b with values 1 and 2, respectively.
- We declare a constant pointer.
- First, we assign the address of variable 'a' to the pointer 'ptr'.
- Then, we assign the address of variable 'b' to the pointer 'ptr'.
- Lastly, we try to print the value of the variable pointed by the 'ptr'.

## Output



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

main.c: In function 'main':
main.c:16:8: error: assignment of read-only variable 'ptr'
    ptr=&a;
    ^
main.c:17:8: error: assignment of read-only variable 'ptr'
    ptr=&b;
    ^
```

In the above output, we can observe that the above code produces the error "assignment of read-only variable 'ptr'". It means that the value of the variable 'ptr' which 'ptr' is holding cannot be changed. In the above code, we are changing the value of 'ptr' from &a to &b, which is not possible with constant pointers. Therefore, we can say that the constant pointer, which points to some variable, cannot point to another variable.

# C Dynamic Memory Allocation

As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

To allocate memory dynamically, library functions `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 free()

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`.

## Example 1: malloc() and free()

```
// Program to calculate the sum of n numbers entered by the user
```

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

int main() {
    int n, i, *ptr, sum = 0;

    printf("Enter number of elements: ");
    scanf("%d", &n);

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

    // if memory cannot be allocated
    if(ptr == NULL) {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements: ");
    for(i = 0; i < n; ++i) {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }
}
```

```

}

printf("Sum = %d", sum);

// deallocating the memory
free(ptr);

return 0;
}

```

[Run Code](#)

## Output

```

Enter number of elements: 3
Enter elements: 100
20
36
Sum = 156

```

Here, we have dynamically allocated the memory for `n` number of `int`.

## Example 2: calloc() and free()

```

// Program to calculate the sum of n numbers entered by the user

#include <stdio.h>
#include <stdlib.h>

int main() {
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) calloc(n, sizeof(int));
    if(ptr == NULL) {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements: ");
    for(i = 0; i < n; ++i) {
        scanf("%d", ptr + i);
    }
}

```

```
    sum += *(ptr + i);  
}  
  
printf("Sum = %d", sum);  
free(ptr);  
return 0;  
}  
Run Code
```

## Output

```
Enter number of elements: 3  
Enter elements: 100  
20  
36  
Sum = 156
```

## 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`.

### Example 3: realloc()

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

int main() {
    int *ptr, i , n1, n2;
    printf("Enter size: ");
    scanf("%d", &n1);

    ptr = (int*) malloc(n1 * sizeof(int));

    printf("Addresses of previously allocated memory:\n");
    for(i = 0; i < n1; ++i)
        printf("%pc\n", ptr + i);

    printf("\nEnter the new size: ");
    scanf("%d", &n2);

    // relocating the memory
    ptr = realloc(ptr, n2 * sizeof(int));

    printf("Addresses of newly allocated memory:\n");
    for(i = 0; i < n2; ++i)
        printf("%pc\n", ptr + i);

    free(ptr);

    return 0;
}
```

[Run Code](#)

### Output

```
Enter size: 2
Addresses of previously allocated memory:
26855472
26855476

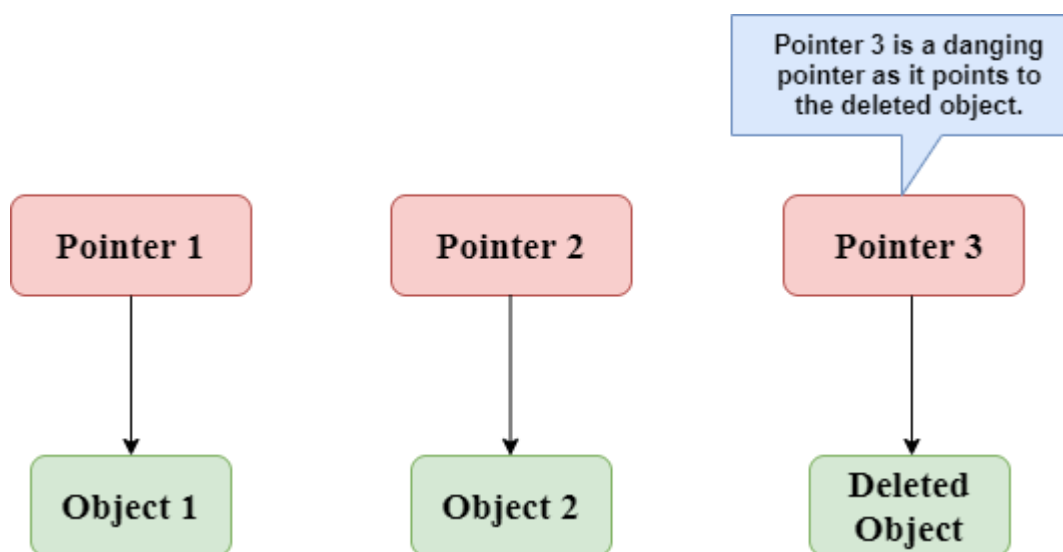
Enter the new size: 4
Addresses of newly allocated memory:
26855472
26855476
26855480
26855484
```

# 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.

**Let's understand the dangling pointer through some C programs.**

**Using free() function to de-allocate the memory.**

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

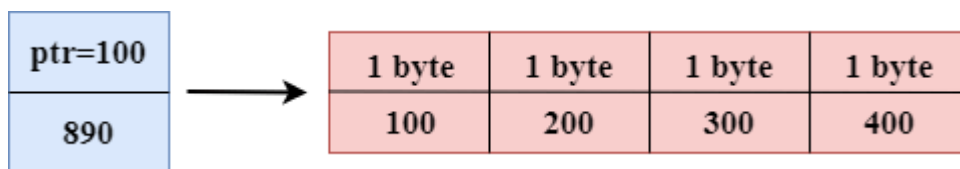
```

3. {
4.   int *ptr=(int *)malloc(sizeof(int));
5.   int a=560;
6.   ptr=&a;
7.   free(ptr);
8.   return 0;
9. }

```

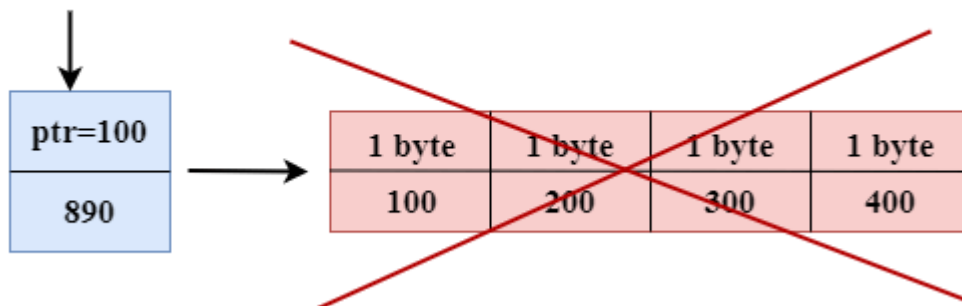
In the above code, we have created two variables, i.e., \*ptr and a where 'ptr' is a pointer and 'a' is a integer variable. The \*ptr is a pointer variable which is created with the help of **malloc()** function. As we know that malloc() function returns void, so we use int \* to convert void pointer into int pointer.

The statement **int \*ptr=(int \*)malloc(sizeof(int));** will allocate the memory with 4 bytes shown in the below image:



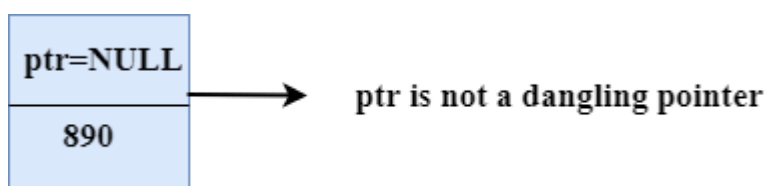
The statement **free(ptr)** de-allocates the memory as shown in the below image with a cross sign, and 'ptr' pointer becomes dangling as it is pointing to the de-allocated memory.

**Dangling pointer**



If we assign the NULL value to the 'ptr', then 'ptr' will not point to the deleted memory. Therefore, we can say that ptr is not a dangling pointer, as shown in the below image:

ADVERTISEMENT



## 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**.

```
1. #include<stdio.h>
2. int main()
3. {
4.     char *str;
5.     {
6.         char a = 'A';
7.         str = &a;
8.     }
9.     // a falls out of scope
10.    // str is now a dangling pointer
11.    printf("%s", *str);
12. }
```

**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.

## Function call

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

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

```
1. #include <stdio.h>
2. int *fun(){
3.     int y=10;
4.     return &y;
5. }
6. int main()
7. {
```



```

8. int *p=fun();
9. printf("%d", *p);
10. return 0;
11. }

```

**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, it means the variable '**y**' is no longer available. Therefore, we can say that the '**p**' pointer is a dangling pointer as it points to the de-allocated memory.

**Output**

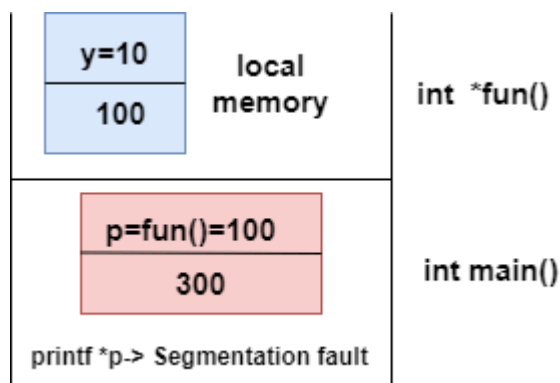
```

Segmentation fault

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

```

**Let's represent the working of the above code diagrammatically.**



**Let's consider another example of a dangling pointer.**

```

1. #include <stdio.h>
2. int *fun()
3. {
4.     static int y=10;
5.     return &y;
6. }
7. int main()
8. {
9.     int *p=fun();
10.    printf("%d", *p);
11.    return 0;
12. }

```

The above code is similar to the previous one but the only difference is that the variable 'y' is static. We know that static variable stores in the global memory.

## Output

```

10

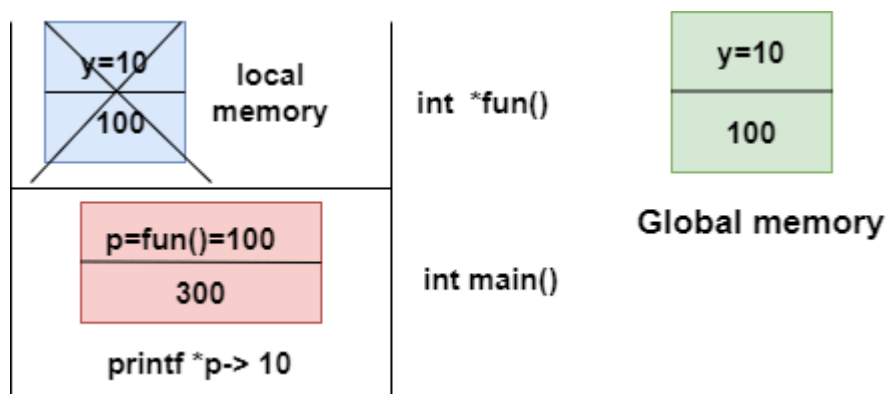
...Program finished with exit code 0
Press ENTER to exit console.

```

ADVERTISEMENT

ADVERTISEMENT

Now, we represent the working of the above code diagrammatically.



The above diagram shows the stack memory. 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.

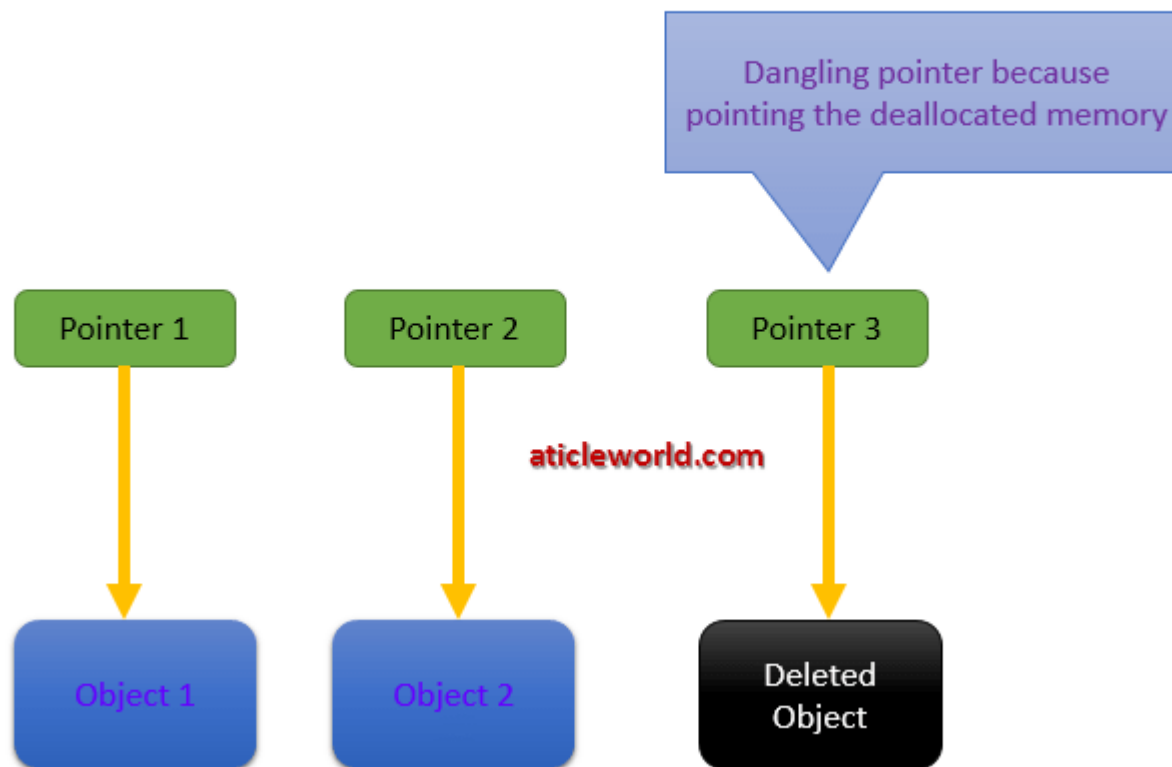
### Avoiding Dangling Pointer Errors

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.

## Dangling pointer in C

In the program, we should not use the dangling pointer. It can be the cause of memory faults. Here I am mentioning few forms of the dangling pointer, I think you should know.

1. If a pointer points to an address that not belongs to your process but knows by the OS, then this type of pointer is a dangling pointer. You should not use such type of pointers is because it leads to a memory fault.
2. Static and automated memory is handled by the compiler but if you allocate the memory in heap memory, the developer needs to free the allocated heap memory. The deleted pointer is another type of dangling pointer is because you have freed the memory but still its pointing to that memory.



**For example,**

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int *p= NULL;
    //Allocate memory for 5 integer
    p= malloc(sizeof(int)* 5);
    if(p == NULL)
    {
        return -1;
    }
    //free the allocated memory
    free(p);
    //p is dangling pointer
    *p= 2;
    printf("%d", *p);
    return 0;
}
```

## Output:

Undefined behavior.

After deallocating the memory, `p` becomes the dangling pointer and if you try to access the `p`, your program could be crash.

## What is a Memory leak?

A **memory leak** occurs when a program fails to release the memory it has allocated dynamically. It can happen when a program **loses track** of the memory it has allocated or when the program fails to **deallocate** the memory it has allocated. Over time, the program will consume more and more memory, which can eventually lead to running out of memory or, worse, system crashes.

## Example of a Memory Leak

Let's consider a simple program that allocates memory dynamically using the **`malloc()`** function:

```
1. #include <stdlib.h>
2. #include <stdio.h>
3.
4. int main() {
5.     int *ptr = (int *) malloc(sizeof(int));
6.     *ptr = 10;
7.     printf("Value of ptr: %d\n", *ptr);
8.     return 0;
9. }
```

## Output

```
Value of ptr: 10
```

## Explanation:

In this program, we allocate memory for an integer using the **`malloc()` function** and store the value **`10`** in it. After that, we print the value of **`ptr`** to the console. However, we have not **`deallocated`** the allocated memory using **`malloc()`**, which means we have a memory leak.

